Python code for
Artificial Intelligence
Foundations of Computational Agents

David L. Poole and Alan K. Mackworth

Contents

2.2.3 Plotting ................................................. 29
2.3 Hierarchical Controller ........................... 31
  2.3.1 Environment .................................... 31
  2.3.2 Body ............................................. 31
  2.3.3 Middle Layer ................................... 34
  2.3.4 Top Layer ....................................... 35
  2.3.5 Plotting .......................................... 36

3 Searching for Solutions .......................... 41
  3.1 Representing Search Problems ................. 41
    3.1.1 Explicit Representation of Search Graph 43
    3.1.2 Paths ......................................... 45
    3.1.3 Example Search Problems ................... 47
  3.2 Generic Searcher and Variants ................... 53
    3.2.1 Searcher ...................................... 53
    3.2.2 GUI for Tracing Search ......................... 55
    3.2.3 Frontier as a Priority Queue ................. 59
    3.2.4 A∗ Search ..................................... 60
    3.2.5 Multiple Path Pruning ......................... 62
  3.3 Branch-and-bound Search ......................... 64

4 Reasoning with Constraints ..................... 69
  4.1 Constraint Satisfaction Problems ............. 69
    4.1.1 Variables .................................... 69
    4.1.2 Constraints ................................... 70
    4.1.3 CSPs .......................................... 71
    4.1.4 Examples ..................................... 74
  4.2 A Simple Depth-first Solver ....................... 83
  4.3 Converting CSPs to Search Problems .......... 84
  4.4 Consistency Algorithms .......................... 86
    4.4.1 Direct Implementation of Domain Splitting 89
    4.4.2 Consistency GUI ................................ 91
    4.4.3 Domain Splitting as an interface to graph searching 93
  4.5 Solving CSPs using Stochastic Local Search ... 95
    4.5.1 Any-conflict .................................. 97
    4.5.2 Two-Stage Choice ................................ 98
    4.5.3 Updatable Priority Queues .................... 100
    4.5.4 Plotting Run-Time Distributions ............... 102
    4.5.5 Testing ...................................... 104
  4.6 Discrete Optimization ............................ 104
    4.6.1 Branch-and-bound Search ....................... 106

5 Propositions and Inference ...................... 109
  5.1 Representing Knowledge Bases .................. 109
  5.2 Bottom-up Proofs (with askables) ............. 112
## 5.3 Top-down Proofs (with askables) ............................................ 114
## 5.4 Debugging and Explanation .............................................. 115
## 5.5 Assumables ...................................................................... 119
## 5.6 Negation-as-failure .......................................................... 122

### 6 Deterministic Planning ......................................................... 125

#### 6.1 Representing Actions and Planning Problems ......................... 125

- **6.1.1 Robot Delivery Domain** .................................................. 126
- **6.1.2 Blocks World** ................................................................. 128

#### 6.2 Forward Planning ............................................................ 130

- **6.2.1 Defining Heuristics for a Planner** ................................. 132

#### 6.3 Regression Planning ......................................................... 135

- **6.3.1 Defining Heuristics for a Regression Planner** .................. 137

#### 6.4 Planning as a CSP ............................................................ 138

#### 6.5 Partial-Order Planning ....................................................... 142

### 7 Supervised Machine Learning .............................................. 149

#### 7.1 Representations of Data and Predictions ............................... 150

- **7.1.1 Creating Boolean Conditions from Features** .................. 153
- **7.1.2 Evaluating Predictions** ................................................ 155
- **7.1.3 Creating Test and Training Sets** ...................................... 157
- **7.1.4 Importing Data From File** ............................................. 157
- **7.1.5 Augmented Features** .................................................... 160

#### 7.2 Generic Learner Interface .................................................. 162

#### 7.3 Learning With No Input Features ...................................... 163

- **7.3.1 Evaluation** ................................................................. 165

#### 7.4 Decision Tree Learning ..................................................... 167

#### 7.5 Cross Validation and Parameter Tuning ................................ 171

#### 7.6 Linear Regression and Classification .................................. 175

#### 7.7 Boosting ........................................................................ 181

- **7.7.1 Gradient Tree Boosting** ............................................... 184

### 8 Neural Networks and Deep Learning ................................... 187

#### 8.1 Layers .......................................................................... 187

#### 8.2 Feedforward Networks .................................................... 190

#### 8.3 Improved Optimization ..................................................... 192

- **8.3.1 Momentum** .............................................................. 192
- **8.3.2 RMS-Prop** ............................................................... 193

#### 8.4 Dropout ........................................................................ 194

- **8.4.1 Examples** .......................................................... 195

### 9 Reasoning with Uncertainty ............................................... 201

#### 9.1 Representing Probabilistic Models .................................... 201

#### 9.2 Representing Factors ....................................................... 201

#### 9.3 Conditional Probability Distributions ................................. 203
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1.4 Variable elimination for decision networks</td>
<td>285</td>
</tr>
<tr>
<td>12.2 Markov Decision Processes</td>
<td>287</td>
</tr>
<tr>
<td>12.2.1 Problem Domains</td>
<td>289</td>
</tr>
<tr>
<td>12.2.2 Value Iteration</td>
<td>297</td>
</tr>
<tr>
<td>12.2.3 Value Iteration GUI for Grid Domains</td>
<td>298</td>
</tr>
<tr>
<td>12.2.4 Asynchronous Value Iteration</td>
<td>300</td>
</tr>
<tr>
<td>13 Reinforcement Learning</td>
<td>305</td>
</tr>
<tr>
<td>13.1 Representing Agents and Environments</td>
<td>305</td>
</tr>
<tr>
<td>13.1.1 Environments</td>
<td>305</td>
</tr>
<tr>
<td>13.1.2 Agents</td>
<td>306</td>
</tr>
<tr>
<td>13.1.3 Simulating an Environment-Agent Interaction</td>
<td>307</td>
</tr>
<tr>
<td>13.1.4 Party Environment</td>
<td>308</td>
</tr>
<tr>
<td>13.1.5 Environment from a Problem Domain</td>
<td>309</td>
</tr>
<tr>
<td>13.1.6 Monster Game Environment</td>
<td>310</td>
</tr>
<tr>
<td>13.2 Q Learning</td>
<td>313</td>
</tr>
<tr>
<td>13.2.1 Exploration Strategies</td>
<td>315</td>
</tr>
<tr>
<td>13.2.2 Testing Q-learning</td>
<td>316</td>
</tr>
<tr>
<td>13.3 Q-learning with Experience Replay</td>
<td>318</td>
</tr>
<tr>
<td>13.4 Stochastic Policy Learning Agent</td>
<td>320</td>
</tr>
<tr>
<td>13.5 Model-based Reinforcement Learner</td>
<td>322</td>
</tr>
<tr>
<td>13.6 Reinforcement Learning with Features</td>
<td>325</td>
</tr>
<tr>
<td>13.6.1 Representing Features</td>
<td>325</td>
</tr>
<tr>
<td>13.6.2 Feature-based RL learner</td>
<td>329</td>
</tr>
<tr>
<td>13.7 GUI for RL</td>
<td>332</td>
</tr>
<tr>
<td>14 Multiagent Systems</td>
<td>337</td>
</tr>
<tr>
<td>14.1 Minimax</td>
<td>337</td>
</tr>
<tr>
<td>14.1.1 Creating a two-player game</td>
<td>337</td>
</tr>
<tr>
<td>14.1.2 Minimax and α-β Pruning</td>
<td>340</td>
</tr>
<tr>
<td>14.2 Multiagent Learning</td>
<td>342</td>
</tr>
<tr>
<td>14.2.1 Simulating Multiagent Interaction with an Environment</td>
<td>342</td>
</tr>
<tr>
<td>14.2.2 Example Games</td>
<td>344</td>
</tr>
<tr>
<td>14.2.3 Testing Games and Environments</td>
<td>345</td>
</tr>
<tr>
<td>15 Individuals and Relations</td>
<td>347</td>
</tr>
<tr>
<td>15.1 Representing Datalog and Logic Programs</td>
<td>347</td>
</tr>
<tr>
<td>15.2 Unification</td>
<td>349</td>
</tr>
<tr>
<td>15.3 Knowledge Bases</td>
<td>350</td>
</tr>
<tr>
<td>15.4 Top-down Proof Procedure</td>
<td>352</td>
</tr>
<tr>
<td>15.5 Logic Program Example</td>
<td>354</td>
</tr>
<tr>
<td>16 Knowledge Graphs and Ontologies</td>
<td>357</td>
</tr>
<tr>
<td>16.1 Triple Store</td>
<td>357</td>
</tr>
<tr>
<td>16.2 Integrating Datalog and Triple Store</td>
<td>360</td>
</tr>
</tbody>
</table>

https://aipython.org  Version 0.9.12  December 22, 2023
Chapter 1

Python for Artificial Intelligence

AIPython contains runnable code for the book *Artificial Intelligence, foundations of computational agents, 3rd Edition*. It has the following design goals:

- Readability is more important than efficiency, although the asymptotic complexity is not compromised. AIPython is not a replacement for well-designed libraries, or optimized tools. Think of it like a model of an engine made of glass, so you can see the inner workings; don’t expect it to power a big truck, but it lets you see how a metal engine can power a truck.

- It uses as few libraries as possible. A reader only needs to understand Python. Libraries hide details that we make explicit. The only library used is matplotlib for plotting and drawing.

1.1 Why Python?

We use Python because Python programs can be close to pseudo-code. It is designed for humans to read.

Python is reasonably efficient. Efficiency is usually not a problem for small examples. If your Python code is not efficient enough, a general procedure to improve it is to find out what is taking most of the time, and implement just that part more efficiently in some lower-level language. Most of these lower-level languages interoperate with Python nicely. This will result in much less programming and more efficient code (because you will have more time to optimize) than writing everything in a low-level language. You will not have to do that for the code here if you are using it for larger projects.
1.2 Getting Python

You need Python 3.9 or later (https://python.org/) and a compatible version of matplotlib (https://matplotlib.org/). This code is not compatible with Python 2 (e.g., with Python 2.7).

Download and install the latest Python 3 release from https://python.org/ or https://www.anaconda.com/download. This should also install pip3. You can install matplotlib using

```
pip3 install matplotlib
```

in a terminal shell (not in Python). That should “just work”. If not, try using pip instead of pip3.

The command python or python3 should then start the interactive Python shell. You can quit Python with a control-D or with quit().

To upgrade matplotlib to the latest version (which you should do if you install a new version of Python) do:

```
pip3 install --upgrade matplotlib
```

We recommend using the enhanced interactive python ipython (https://ipython.org/) [?]. To install ipython after you have installed python do:

```
pip3 install ipython
```

1.3 Running Python

We assume that everything is done with an interactive Python shell. You can either do this with an IDE, such as IDLE that comes with standard Python distributions, or just running ipython3 or python3 (or perhaps just ipython or python) from a shell.

Here we describe the most simple version that uses no IDE. If you download the zip file, and cd to the “aipython” folder where the .py files are, you should be able to do the following, with user input in bold. The first python command is in the operating system shell; the -i is important to enter interactive mode.

```
python -i searchGeneric.py
```

Testing problem 1:

7 paths have been expanded and 4 paths remain in the frontier
Path found: A --> C --> B --> D --> G
Passed unit test

```python
>>> searcher2 = AStarSearcher(searchProblem.acyclic_delivery_problem) #A*
>>> searcher2.search() # find first path
16 paths have been expanded and 5 paths remain in the frontier
o103 --> o109 --> o119 --> o123 --> r123
>>> searcher2.search() # find next path
21 paths have been expanded and 6 paths remain in the frontier
```

https://aipython.org Version 0.9.12 December 22, 2023
1.4 Pitfalls

It is important to know when side effects occur. Often AI programs consider what would/might happen given certain conditions. In many such cases, we don’t want side effects. When an agent acts in the world, side effects are appropriate.

In Python, you need to be careful to understand side effects. For example, the inexpensive function to add an element to a list, namely append, changes the list. In a functional language like Haskell or Lisp, adding a new element to a list, without changing the original list, is a cheap operation. For example if $x$ is a list containing $n$ elements, adding an extra element to the list in Python (using append) is fast, but it has the side effect of changing the list $x$. To construct a new list that contains the elements of $x$ plus a new element, without changing the value of $x$, entails copying the list, or using a different representation for lists. In the searching code, we will use a different representation for lists for this reason.

1.5 Features of Python

1.5.1 f-strings

Python can use matching ‘, ”, ’’ or “”, the latter two respecting line breaks in the string. We use the convention that when the string denotes a unique symbol, we use single quotes, and when it is designed to be for printing, we use double quotes.
We make extensive use of f-strings [https://docs.python.org/3/tutorial/inputoutput.html] In its simplest form

\[
\text{f"str1\{e1\}str2\{e2\}str3"}
\]

where e1 and e2 are expressions, is an abbreviation for

"str1"+str(e2)+"str2"+str(e2)+"str3"

where + is string concatenation, and str is the function that returns a string representation of its expression argument.

1.5.2 Lists, Tuples, Sets, Dictionaries and Comprehensions

We make extensive uses of lists, tuples, sets and dictionaries (dicts). See [https://docs.python.org/3/library/stdtypes.html]

One of the nice features of Python is the use of [https://docs.python.org/3/reference/expressions.html#displays-for-lists-sets-and-dictionaries](https://docs.python.org/3/reference/expressions.html#displays-for-lists-sets-and-dictionaries) (and also list, tuple, set and dictionary comprehensions). A generator expression is of the form

\[(fe \text{ for } e \text{ in } \text{iter} \text{ if } \text{cond})\]

enumerates the values fe for each e in iter for which cond is true. The “if cond” part is optional, but the “for” and “in” are not optional. Here e is a variable (or a pattern that can be on the left side of =), iter is an iterator, which can generate a stream of data, such as a list, a set, a range object (to enumerate integers between ranges) or a file. cond is an expression that evaluates to either True or False for each e, and fe is an expression that will be evaluated for each value of e for which cond returns True.

The result can go in a list or used in another iteration, or can be called directly using next. The procedure next takes an iterator and returns the next element (advancing the iterator); it raises a StopIteration exception if there is no next element. The following shows a simple example, where user input is prepended with >>>

```python
>>> [e*e for e in range(20) if e%2==0]
[0, 4, 16, 36, 100, 144, 196, 256, 324]
>>> a = (e*e for e in range(20) if e%2==0)
>>> next(a)
0
>>> next(a)
4
>>> next(a)
16
>>> list(a)
[36, 64, 100, 144, 196, 256, 324]
```

[https://docs.python.org/3/reference/expressions.html#displays-for-lists-sets-and-dictionaries]
1.5. Features of Python

```python
>>> next(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Notice how `list(a)` continued on the enumeration, and got to the end of it.

Comprehensions can also be used for dictionaries. The following code creates an index for list `a`:

```python
>>> a = ["a","f","bar","b","a","aaaa"]
>>> ind = {a[i]:i for i in range(len(a))}
>>> ind
{'a': 4, 'f': 1, 'bar': 2, 'b': 3, 'aaaa': 5}
```

which means that 'b' is the 3rd element of the list.

The assignment of `ind` could have also be written as:

```python
>>> ind = {val:i for (i,val) in enumerate(a)}
```

where `enumerate` is a built-in function that, given a dictionary, returns an iterator of `(index, value)` pairs.

1.5.3 Functions as first-class objects

Python can create lists and other data structures that contain functions. There is an issue that tricks many newcomers to Python. For a local variable in a function, the function uses the last value of the variable when the function is called, not the value of the variable when the function was defined (this is called “late binding”). This means if you want to use the value a variable has when the function is created, you need to save the current value of that variable. Whereas Python uses “late binding” by default, the alternative that newcomers often expect is “early binding”, where a function uses the value a variable had when the function was defined. The following examples show how early binding can be implemented.

Consider the following programs designed to create a list of 5 functions, where the `i`th function in the list is meant to add `i` to its argument:

```python
fun_list1 = []
for i in range(5):
    def fun1(e):
        return e+i
    fun_list1.append(fun1)
```

2Numbered lines are Python code available in the code-directory, aipython. The name of the file is given in the gray text above the listing. The numbers correspond to the line numbers in that file.
fun_list2 = []
for i in range(5):
    def fun2(e, iv=i):
        return e + iv
    fun_list2.append(fun2)

fun_list3 = [lambda e: e + i for i in range(5)]
fun_list4 = [lambda e, iv=i: e + iv for i in range(5)]

Try to predict, and then test to see the output, of the output of the following calls, remembering that the function uses the latest value of any variable that is not bound in the function call:

# in Shell do
## ipython -i pythonDemo.py
# Try these (copy text after the comment symbol and paste in the Python prompt):
# print([f(10) for f in fun_list1])
# print([f(10) for f in fun_list2])
# print([f(10) for f in fun_list3])
# print([f(10) for f in fun_list4])

In the first for-loop, the function fun1 uses i, whose value is the last value it was assigned. In the second loop, the function fun2 uses iv. There is a separate iv variable for each function, and its value is the value of i when the function was defined. Thus fun1 uses late binding, and fun2 uses early binding. fun_list3 and fun_list4 are equivalent to the first two (except fun_list4 uses a different i variable).

One of the advantages of using the embedded definitions (as in fun1 and fun2 above) over the lambda is that it is possible to add a __doc__ string, which is the standard for documenting functions in Python, to the embedded definitions.

### 1.5.4 Generators

Python has generators which can be used for a form of lazy evaluation – only computing values when needed.

The yield command returns a value that is obtained with next. It is typically used to enumerate the values for a for loop or in generators. (The yield command can also be used for coroutines, but AIPython only uses it for generators.)

A version of the built-in range, with 2 or 3 arguments (and positive steps) can be implemented as:
1.5. Features of Python

```python
def myrange(start, stop, step=1):
    '''enumerates the values from start in steps of size step that are
    less than stop.'''
    assert step>0, f'only positive steps implemented in myrange: {step}'
    i = start
    while i<stop:
        yield i
        i += step

print("list(myrange(2,30,3)): ", list(myrange(2,30,3)))
```

Note that the built-in range is unconventional in how it handles a single argument, as the single argument acts as the second argument of the function. Note also that the built-in range also allows for indexing (e.g., range(2,30,3)[2] returns 8), but the above implementation does not. However myrange also works for floats, whereas the built-in range does not.

**Exercise 1.1** Implement a version of myrange that acts like the built-in version when there is a single argument. (Hint: make the second argument have a default value that can be recognized in the function.) There is no need to make it with indexing.

Yield can be used to generate the same sequence of values as in the example of Section 1.5.2:

```python
def ga(n):
    '''generates square of even nonnegative integers less than n'''
    for e in range(n):
        if e%2==0:
            yield e*e

a = ga(20)
```

The sequence of next(a), and list(a) gives exactly the same results as the comprehension in Section 1.5.2.

It is straightforward to write a version of the built-in enumerate called myenumerate:

```python
def myenumerate(enum):
    for i in range(len(enum)):
        yield i, enum[i]
```

**Exercise 1.2** Write a version of enumerate where the only iteration is “for val in enum”. Hint: keep track of the index.
1.6 Useful Libraries

1.6.1 Timing Code

In order to compare algorithms, we often want to compute how long a program takes; this is called the run time of the program. The most straightforward way to compute run time is to use `time.perf_counter()`, as in:

```python
import time
start_time = time.perf_counter()
compute_for_a_while()
end_time = time.perf_counter()
print("Time: ", end_time - start_time, "seconds")
```

Note that `time.perf_counter()` measures clock time; so this should be done without user interaction between the calls. On the interactive python shell, you should do:

```python
start_time = time.perf_counter(); compute_for_a_while(); end_time = time.perf_counter()
```

If this time is very small (say less than 0.2 second), it is probably very inaccurate, and it may be better to run your code many times to get a more accurate count. For this you can use `timeit` ([https://docs.python.org/3/library/timeit.html](https://docs.python.org/3/library/timeit.html)). To use `timeit` to time the call to `foo.bar(aaa)` use:

```python
import timeit
time = timeit.timeit("foo.bar(aaa)",
                    setup="from __main__ import foo,aaa", number=100)
```

The setup is needed so that Python can find the meaning of the names in the string that is called. This returns the number of seconds to execute `foo.bar(aaa)` 100 times. The variable `number` should be set so that the run time is at least 0.2 seconds.

You should not trust a single measurement as that can be confounded by interference from other processes. `timeit.repeat` can be used for running `timeit` a few (say 3) times. When reporting the time of any computation, you should be explicit and explain what you are reporting. Usually the minimum time is the one to report.

1.6.2 Plotting: Matplotlib

The standard plotting for Python is matplotlib ([https://matplotlib.org/](https://matplotlib.org/)). We will use the most basic plotting using the pyplot interface.

Here is a simple example that uses everything we will use. The output is shown in Figure 1.1.

```python
import matplotlib.pyplot as plt
```

https://aipython.org  Version 0.9.12  December 22, 2023
1.6. Useful Libraries

```python
import matplotlib.pyplot as plt

def myplot(minv, maxv, step, fun1, fun2):
    plt.ion()  # make it interactive
    plt.xlabel("The x axis")
    plt.ylabel("The y axis")
    plt.xscale('linear')  # Makes a 'log' or 'linear' scale
    xvalues = range(minv, maxv, step)
    plt.plot(xvalues, [fun1(x) for x in xvalues],
             label="The first fun")
    plt.plot(xvalues, [fun2(x) for x in xvalues], linestyle='--', color='k',
             label=fun2.__doc__)  # use the doc string of the function
    plt.legend(loc="upper right")  # display the legend

def slin(x):
    """y=2x+7""
    return 2*x+7

def sqfun(x):
    """y=(x-40)^2/10-20""
    return (x-40)**2/10-20

# Try the following:
# from pythonDemo import myplot, slin, sqfun
# import matplotlib.pyplot as plt
# myplot(0,100,1,slin,sqfun)
# plt.legend(loc="best")
# import math
# plt.plot([41+40*math.cos(th/10) for th in range(50)],
#           [100+100*math.sin(th/10) for th in range(50)])
```

Figure 1.1: Result of pythonDemo code
At the end of the code are some commented-out commands you should try in interactive mode. Cut from the file and paste into Python (and remember to remove the comments symbol and leading space).

## 1.7 Utilities

### 1.7.1 Display

In this distribution, to keep things simple, using only standard Python, we use a text-oriented tracing of the code. A graphical depiction of the code can override the definition of display (e.g., see SearcherGUI in Section 3.2.2 and ConsistencyGUI in Section 4.4.2).

The method `self.display` is used to trace the program. Any call

```python
self.display(level, to_print ...)
```

where the `level` is less than or equal to the value for `max_display_level` will be printed. The `to_print ...` can be anything that is accepted by the built-in `print` (including any keyword arguments).

The definition of `display` is:

```python
class Displayable(object):
    """Class that uses 'display'.
    The amount of detail is controlled by max_display_level
    """
    max_display_level = 1  # can be overridden in subclasses or instances

def display(self, level, *args, **nargs):
    """print the arguments if level is less than or equal to the current max_display_level.
    level is an integer.
    the other arguments are whatever arguments print can take.
    """
    if level <= self.max_display_level:
        print(*args, **nargs)  # if error you are using Python2 not Python3
```

(Note that `args` gets a tuple of the positional arguments, and `nargs` gets a dictionary of the keyword arguments). This will not work in Python 2, and will give an error.

Any class that wants to use `display` can be made a subclass of `Displayable`. To change the maximum display level to 3 for a class do:

```
Classname.max_display_level = 3
```

https://aipython.org
which will make calls to display in that class print when the value of level is less-than-or-equal to 3. The default display level is 1. It can also be changed for individual objects (the object value overrides the class value).

The value of max_display_level by convention is:

0  display nothing
1  display solutions (nothing that happens repeatedly)
2  also display the values as they change (little detail through a loop)
3  also display more details
4 and above  even more detail

1.7.2 Argmax

Python has a built-in max function that takes a generator (or a list or set) and returns the maximum value. The argmax method returns the index of an element that has the maximum value. If there are multiple elements with the maximum value, one of the indexes to that value is returned at random. argmaxe assumes an enumeration; a generator of (element, value) pairs, as for example is generated by the built-in enumerate(list) for lists or dict.items() for dictionaries.

```python
import random
import math

def argmaxall(gen):
    """gen is a generator of (element,value) pairs, where value is a real.
    argmaxall returns a list of all of the elements with maximal value.
    """
    maxv = -math.inf  # negative infinity
    maxvals = []  # list of maximal elements
    for (e,v) in gen:
        if v>maxv:
            maxvals,maxv = [e], v
        elif v==maxv:
            maxvals.append(e)
    return maxvals

def argmaxe(gen):
    """gen is a generator of (element,value) pairs, where value is a real.
    argmaxe returns an element with maximal value.
    If there are multiple elements with the max value, one is returned at random.
    """
    return random.choice(argmaxall(gen))

def argmax(lst):
```

https://aipython.org
Exercise 1.3 Change argmaxall to have an optional argument that specifies whether you want the “first”, “last” or a “random” index of the maximum value returned. If you want the first or the last, you don’t need to keep a list of the maximum elements. Enable the other methods to have this optional argument.

1.7.3 Probability

For many of the simulations, we want to make a variable True with some probability. flip(p) returns True with probability p, and otherwise returns False.

Utilities.py — (continued)

```python
def flip(prob):
    """return true with probability prob""
    return random.random() < prob

The select_from_dist method takes in an item : probability dictionary, and returns one of the items in proportion to its probability. The probabilities should sum to 1 or more. If they sum to more than one, the excess is ignored.

Utilities.py — (continued)
```

```python
def select_from_dist(item_prob_dist):
    """returns a value from a distribution.
    item_prob_dist is an item:probability dictionary, where the
    probabilities sum to 1.
    returns an item chosen in proportion to its probability
    ""
    ranreal = random.random()
    for (it,prob) in item_prob_dist.items:
        if ranreal < prob:
            return it
        else:
            ranreal -= prob
    raise RuntimeError(f"{item_prob_dist} is not a probability distribution")
```
1.8 Testing Code

It is important to test code early and test it often. We include a simple form of **unit test**. The value of the current module is in __name__ and if the module is run at the top-level, its value is "__main__". See [https://docs.python.org/3/library/__main__.html](https://docs.python.org/3/library/__main__.html).

The following code tests argmax and dict_union, but only when if utilities is loaded in the top-level. If it is loaded in a module the test code is not run.

In your code, you should do more substantial testing than done here. In particular, you should also test boundary cases.

```python
63 def test():
    """Test part of utilities""
    assert argmax([1, 6, 55, 3, 55, 23]) in [2, 4]
    print("Passed unit test in utilities")
    print("run test_aipython() to test (almost) everything")
    if __name__ == "__main__":
        test()
```

The following does a simple check of all of AIPython that has automatic checks. If you develop new algorithms or tests, add them here!

```python
72 def test_aipython():
    # Agents: currently no tests
    # Search:
    print("***** testing Search *****")
    import searchGeneric, searchBranchAndBound, searchExample, searchTest
    searchGeneric.test(searchGeneric.AStarSearcher)
    searchBranchAndBound.test(searchBranchAndBound.DF_branch_and_bound)
    searchTest.run(searchExample.problem1, "Problem 1")
    # CSP
    print("\n***** testing CSP *****")
    import cspExamples, cspDFS, cspSearch, cspConsistency, cspSLS
    cspExamples.test_csp(cspDFS.dfs_solve1)
    cspExamples.test_csp(cspSearch.solver_from_searcher)
    cspExamples.test_csp(cspConsistency.ac_solver)
    cspExamples.test_csp(cspConsistency.ac_search_solver)
    cspExamples.test_csp(cspSLS.sls_solver)
    cspExamples.test_csp(cspSLS.any_conflict_solver)
    # Propositions
    print("\n***** testing Propositional Logic *****")
    import logicBottomUp, logicTopDown, logicExplain, logicNegation
    logicBottomUp.test()
    logicTopDown.test()
    logicExplain.test()
    logicNegation.test()
    # Planning
```
print("\n***** testing Planning *****")
import stripsHeuristic
stripsHeuristic.test_forward_heuristic()
stripsHeuristic.test_regression_heuristic()

# Learning
print("\n***** testing Learning *****")
import learnProblem, learnNoInputs, learnDT, learnLinear
learnNoInputs.test_no_inputs(training_sizes=[4])
data = learnProblem.Data_from_file('data/carbool.csv', target_index=-1, seed=123)
learnDT.testDT(data, print_tree=False)
learnLinear.test()

# Deep Learning: currently no tests
# Uncertainty
print("\n***** testing Uncertainty *****")
import probGraphicalModels, probRC, probVE, probStochSim
probGraphicalModels.InferenceMethod.testIM(probRC.ProbSearch)
probGraphicalModels.InferenceMethod.testIM(probRC.ProbRC)
probGraphicalModels.InferenceMethod.testIM(probVE.VE)
probGraphicalModels.InferenceMethod.testIM(probStochSim.RejectionSampling, threshold=0.1)
probGraphicalModels.InferenceMethod.testIM(probStochSim.LikelihoodWeighting, threshold=0.1)
probGraphicalModels.InferenceMethod.testIM(probStochSim.ParticleFiltering, threshold=0.1)
probGraphicalModels.InferenceMethod.testIM(probStochSim.GibbsSampling, threshold=0.1)

# Learning under uncertainty: currently no tests
# Causality: currently no tests
# Planning under uncertainty
print("\n***** testing Planning under Uncertainty *****")
import decnNetworks
decnNetworks.test(decnNetworks.fire_dn)
import mdpExamples
mdpExamples.test_MDP(mdpExamples.partyMDP)

# Reinforcement Learning:
print("\n***** testing Reinforcement Learning *****")
import rlQLearner
rlQLearner.test_RL(rlQLearner.Q_learner, alpha_fun=lambda k:10/(9+k))
import rlQExperienceReplay
rlQLearner.test_RL(rlQExperienceReplay.Q_ER_learner, alpha_fun=lambda k:10/(9+k))
import rlStochasticPolicy
rlQLearner.test_RL(rlStochasticPolicy.StochasticPIAgent, alpha_fun=lambda k:10/(9+k))
import rlModelLearner
rlQLearner.test_RL(rlModelLearner.Model_based_reinforcement_learner)
import rlFeatures
rlQLearner.test_RL(rlFeatures.SARSA_LFA_learner, es_kwargs={'epsilon':1}, eps=4)
1.8. Testing Code

```python
# Multiagent systems: currently no tests
# Individuals and Relations
print("\n***** testing Datalog and Logic Programming *****")
import relnExamples
relnExamples.test_ask_all()
# Knowledge Graphs and Onologies
print("\n***** testing Knowledge Graphs and Onologies *****")
import knowledgeGraph
knowledgeGraph.test_kg()
# Relational Learning: currently no tests
```
Agent Architectures and Hierarchical Control

This implements the controllers described in Chapter 2 of ?.

These provide sequential implementations of the control. More sophisticated version may have them run concurrently (either as coroutines or in parallel).

In this version the higher-levels call the lower-levels. The higher-levels calling the lower-level works in simulated environments when there is a single agent, and where the lower-level are written to make sure they return (and don’t go on forever), and the higher level doesn’t take too long (as the lower-levels will wait until called again).

2.1 Representing Agents and Environments

In the initial implementation, both agents and the environment are treated as objects in the send of object-oriented programs: they can have an internal state they maintain, and can evaluate methods that can provide answers. This is the same representation used for the reinforcement learning algorithms (Chapter 13).

An environment takes in actions of the agents, updates its internal state and returns the next percept, using the method do.

An agent takes the precept, updates its internal state, and output it next action. An agent implements the method select_action that takes percept and returns its next action.

The methods do and select_action are chained together to build a simulator. In order to start this, we need either an action or a percept. There are two variants used:
An agent implements the `initial_action()` method which is used initially. This is the method used in the reinforcement learning chapter (page 305).

The environment implements the `initial_percept()` method which gives the initial percept. This is the method used in this chapter.

In this implementation, the state of the agent and the state of the environment are represented using standard Python variables, which are updated as the state changes. The percept and the actions are represented as variable-value dictionaries. When agent has only a limited number of actions, the action can be a single value.

In the following code `raise NotImplementedError()` is a way to specify an abstract method that needs to be overridden in any implemented agent or environment.

```python
from display import Displayable
class Agent(Displayable):
    def initial_action(self, percept):
        """return the initial action.""
        return self.select_action(percept) # same as select_action
    def select_action(self, percept):
        """return the next action (and update internal state) given percept
        percept is variable:value dictionary"
        raise NotImplementedError("go") # abstract method

class Environment(Displayable):
    def initial_percept(self):
        """returns the initial percept for the agent"
        raise NotImplementedError("initial_percept") # abstract method
    def do(self, action):
        """does the action in the environment
        returns the next percept"
        raise NotImplementedError("Environment.do") # abstract method
```

The simulator lets the agent and the environment take turns in updating their states and returning the action and the percept.
2.2. Paper buying agent and environment

The first implementation is a simple procedure to carry out \( n \) steps of the simulation and return the agent state and the environment state at the end.

```python
class Simulate(Displayable):
    """simulate the interaction between the agent and the environment for \( n \) time steps.
    Returns a pair of the agent state and the environment state.
    ""
    def __init__(self, agent, environment):
        self.agent = agent
        self.env = environment
        self.percept = self.env.initial_percept()
        self.percept_history = [self.percept]
        self.action_history = []

    def go(self, n):
        for i in range(n):
            action = self.agent.select_action(self.percept)
            self.display(2, f"i={i} action={action}"
            self.percept = self.env.do(action)
            self.display(2, f"percept={self.percept}""
```

2.2 Paper buying agent and environment

To run the demo, in folder "aipython", load "agents.py", using e.g., `ipython -i agentBuying.py`, and copy and paste the commented-out commands at the bottom of that file.

This is an implementation of Example 2.1 of ?. You might get different plots to Figures 2.2 and 2.3 as there is randomness in the environment.

2.2.1 The Environment

The environment state is given in terms of the time and the amount of paper in stock. It also remembers the in-stock history and the price history. The percept consists of the price and the amount of paper in stock. The action of the agent is the number to buy.

Here we assume that the prices are obtained from the prices list (which cycles) plus a random integer in range \([0, \max\_price\_addon]\) plus a linear "inflation". The agent cannot access the price model; it just observes the prices and the amount in stock.
2. Agent Architectures and Hierarchical Control

```python
class TP_env(Environment):
    price_delta = [0, 0, 0, 21, 0, 20, 0, -64, 0, 0, 23, 0, 0, 0, -35,
                   0, 76, 0, -41, 0, 0, 21, 0, 5, 0, 5, 0, 0, 5, 0, -15, 0, 5,
                   0, 5, 0, -115, 0, 115, 0, 5, 0, -15, 0, 5, 0, 0, 0, 5, 0,
                   -59, 0, 44, 0, 5, 0, 5, 0, 5, 0, 0, 5, 0, -65, 50, 0, 5, 0,
                   5, 0, 0, 5, 0, -59, 0, 44, 0, 5, 0, 5, 0, 5, 0, 0, 5, 0]
    sd = 5 # noise standard deviation

def __init__(self):
    """paper buying agent""
    self.time=0
    self.stock=20
    self.stock_history = [] # memory of the stock history
    self.price_history = [] # memory of the price history

    def initial_percept(self):
        """return initial percept"
        self.stock_history.append(self.stock)
        self.price = round(234+self.sd*random.gauss(0,1))
        self.price_history.append(self.price)
        return {'price': self.price,
                'instock': self.stock}

    def do(self, action):
        """does action (buy) and returns percept consisting of price and instock"
        used = select_from_dist({6:0.1, 5:0.1, 4:0.1, 3:0.3, 2:0.2, 1:0.2})
        # used = select_from_dist({7:0.1, 6:0.2, 5:0.2, 4:0.3, 3:0.1,
        #                          2:0.1}) # uses more paper
        bought = action['buy']
        self.stock = self.stock+bought-used
        self.stock_history.append(self.stock)
        self.time += 1
        self.price = round(self.price
                           + self.price_delta[self.time%len(self.price_delta)] # repeating pattern
                           + self.sd*random.gauss(0,1)) # plus randomness
        self.price_history.append(self.price)
        return {'price': self.price,
                'instock': self.stock}
```

2.2.2 The Agent

The agent does not have access to the price model but can only observe the current price and the amount in stock. It has to decide how much to buy.

The belief state of the agent is an estimate of the average price of the paper, and the total amount of money the agent has spent.

https://aipython.org
2.2. Paper buying agent and environment

```python
class TP_agent(Agent):
    def __init__(self):
        self.spent = 0
        percept = env.initial_percept()
        self.ave = self.last_price = percept['price']
        self.instock = percept['instock']
        self.buy_history = []

    def select_action(self, percept):
        """return next action to carry out
        """
        self.last_price = percept['price']
        self.ave = self.ave + (self.last_price - self.ave) * 0.05
        self.instock = percept['instock']
        if self.last_price < 0.9 * self.ave and self.instock < 60:
            tobuy = 48
        elif self.instock < 12:
            tobuy = 12
        else:
            tobuy = 0
        self.spent += tobuy * self.last_price
        self.buy_history.append(tobuy)
        return {'buy': tobuy}
```

Set up an environment and an agent. Uncomment the last lines to run the agent for 90 steps, and determine the average amount spent.

```python
env = TP_env()
ag = TP_agent()
sim = Simulate(ag, env)
# sim.go(90)
# ag.spent/env.time ## average spent per time period
```

2.2.3 Plotting

The following plots the price and number in stock history:
Figure 2.1 shows the result of the plotting in the previous code.

**Exercise 2.1** Design a better controller for a paper-buying agent.

- Justify a performance measure that is a fair comparison. Note that minimizing the total amount of money spent may be unfair to agents who have built up a stockpile, and favors agents that end up with no paper.

- Give a controller that can work for many different price histories. An agent
can use other local state variables, but does not have access to the environment model.

• Is it worthwhile trying to infer the amount of paper that the home uses? (Try your controller with the different paper consumption commented out in TP_env.do.)

2.3 Hierarchical Controller

To run the hierarchical controller, in folder "aipython", load "agentTop.py", using e.g., ipython -i agentTop.py, and copy and paste the commands near the bottom of that file.

In this implementation, each layer, including the top layer, implements the environment class, because each layer is seen as an environment from the layer above.

We arbitrarily divide the environment and the body, so that the environment just defines the walls, and the body includes everything to do with the agent. Note that the named locations are part of the (top-level of the) agent, not part of the environment, although they could have been.

2.3.1 Environment

The environment defines the walls.

```python
import math
from display import Displayable
from agents import Environment

class Rob_env(Environment):
    def __init__(self, walls = {}):
        """walls is a set of line segments where each line segment is of the form ((x0,y0),(x1,y1))""
        self.walls = walls
```

2.3.2 Body

The body defines everything about the agent body.

```python
import math
from agents import Environment
import matplotlib.pyplot as plt
import time

https://aipython.org
```
class Rob_body(Environment):
    def __init__(self, env, init_pos=(0,0,90)):
        """ env is the current environment
        init_pos is a triple of (x-position, y-position, direction)
        direction is in degrees; 0 is to right, 90 is straight-up, etc
        ""
        self.env = env
        self.rob_x, self.rob_y, self.rob_dir = init_pos
        self.turning_angle = 18 # degrees that a left makes
        self.whisker_length = 6 # length of the whisker
        self.whisker_angle = 30 # angle of whisker relative to robot
        self.crashed = False
        # The following control how it is plotted
        self.plotting = True # whether the trace is being plotted
        self.sleep_time = 0.05 # time between actions (for real-time plotting)
        # The following are data structures maintained:
        self.history = [(self.rob_x, self.rob_y)] # history of (x,y) positions
        self.wall_history = [] # history of hitting the wall
        def percept(self):
            return {'rob_x_pos':self.rob_x, 'rob_y_pos':self.rob_y,
                    'rob_dir':self.rob_dir, 'whisker':self.whisker(),
                    'crashed':self.crashed}
        initial_percept = percept # use percept function for initial percept too
        def do(self,action):
            """ action is {'steer':direction}
            direction is 'left', 'right' or 'straight'
            ""
            if self.crashed:
                return self.percept()
            direction = action['steer']
            compass_deriv =
            {'left':1,'straight':0,'right':-1}[direction]*self.turning_angle
            self.rob_dir = (self.rob_dir + compass_deriv + 360)%360 # make in range [0,360)
            rob_x_new = self.rob_x + math.cos(self.rob_dir*math.pi/180)
            rob_y_new = self.rob_y + math.sin(self.rob_dir*math.pi/180)
            path = ((self.rob_x,self.rob_y),(rob_x_new,rob_y_new))
            if any(line_segments_intersect(path,wall) for wall
                   in self.env.walls):
                self.crashed = True
            if self.plotting:
                plt.plot([self.rob_x],[self.rob_y],"r*",markersize=20.0)
            plt.draw()
            self.rob_x, self.rob_y = rob_x_new, rob_y_new
            self.history.append((self.rob_x, self.rob_y))
            if self.plotting and not self.crashed:
2.3. Hierarchical Controller

```python
plt.plot([self.rob_x],[self.rob_y],"go")
plt.draw()
plt.pause(self.sleep_time)
return self.percept()
```

The Boolean whisker method returns True when the whisker and the wall intersect.

```python
def whisker(self):
    """returns true whenever the whisker sensor intersects with a wall
    """
    whisk_ang_world = (self.rob_dir-self.whisker_angle)*math.pi/180
    # angle in radians in world coordinates
    wx = self.rob_x + self.whisker_length * math.cos(whisk_ang_world)
    wy = self.rob_y + self.whisker_length * math.sin(whisk_ang_world)
    whisker_line = ((self.rob_x,self.rob_y),(wx,wy))
    hit = any(line_segments_intersect(whisker_line,wall)
              for wall in self.env.walls)
    if hit:
        self.wall_history.append((self.rob_x, self.rob_y))
        if self.plotting:
            plt.plot([self.rob_x],[self.rob_y],"ro")
            plt.draw()
    return hit

def line_segments_intersect(linea,lineb):
    """returns true if the line segments, linea and lineb intersect.
    A line segment is represented as a pair of points.
    A point is represented as a (x,y) pair.
    """
    ((x0a,y0a),(x1a,y1a)) = linea
    ((x0b,y0b),(x1b,y1b)) = lineb
    da, db = x1a-x0a, x1b-x0b
    ea, eb = y1a-y0a, y1b-y0b
    denom = db*ea-eb*da
    if denom==0:  # line segments are parallel
        return False
    cb = (da*(y0b-y0a)-ea*(x0b-x0a))/denom # position along line b
    if cb<0 or cb>1:
        return False
    ca = (db*(y0b-y0a)-eb*(x0b-x0a))/denom # position along line a
    return 0<=ca<=1

    # Test cases:
    # assert line_segments_intersect(((0,0),(1,1)),((1,0),(0,1)))
    # assert not line_segments_intersect(((0,0),(1,1)),((1,0),(0.6,0.4)))
    # assert line_segments_intersect(((0,0),(1,1)),((1,0),(0.4,0.6)))
```

https://aipython.org
2.3.3 Middle Layer

The middle layer acts like both a controller (for the environment layer) and an environment for the upper layer. It has to tell the environment how to steer. Thus it calls env.do(·). It also is told the position to go to and the timeout. Thus it also has to implement do(·).

```python
def __init__(self, env):
    self.env = env
    self.percept = env.initial_percept()
    self.straight_angle = 11  # angle that is close enough to straight ahead
    self.close_threshold = 2  # distance that is close enough to arrived
    self.close_threshold_squared = self.close_threshold**2  # just compute it once

def initial_percept(self):
    return {}

def do(self, action):
    """action is {'go_to':target_pos,'timeout':timeout}
    target_pos is (x,y) pair
    timeout is the number of steps to try
    returns {'arrived':True} when arrived is true
    or {'arrived':False} if it reached the timeout
    """
    if 'timeout' in action:
        remaining = action['timeout']
    else:
        remaining = -1  # will never reach 0
    target_pos = action['go_to']
    arrived = self.close_enough(target_pos)
    while not arrived and remaining != 0:
        self.percept = self.env.do({'steer': self.steer(target_pos)})
        remaining -= 1
        arrived = self.close_enough(target_pos)
    return {'arrived': arrived}
```

The following method determines how to steer depending on whether the goal is to the right or the left of where the robot is facing.

```python
def steer(self, target_pos):
    if self.percept['whisker']:
        self.display(3, 'whisker on', self.percept)
    return "left"
```

https://aipython.org
2.3. Hierarchical Controller

```python
else:
    return self.head_towards(target_pos)

def head_towards(self, target_pos):
    """ given a target position, return the action that heads
    towards that position
    """
    gx, gy = target_pos
    rx, ry = self.percept['rob_x_pos'], self.percept['rob_y_pos']
    goal_dir = math.acos((gx-rx)/math.sqrt((gx-rx)*(gx-rx)
                        + (gy-ry)*(gy-ry)))*180/math.pi
    if ry>gy:
        goal_dir = -goal_dir
    goal_from_rob = (goal_dir - self.percept['rob_dir'] + 540)%360-180
    assert -180 < goal_from_rob <= 180
    if goal_from_rob > self.straight_angle:
        return "left"
    elif goal_from_rob < -self.straight_angle:
        return "right"
    else:
        return "straight"

def close_enough(self, target_pos):
    gx, gy = target_pos
    rx, ry = self.percept['rob_x_pos'], self.percept['rob_y_pos']
    return (gx-rx)**2 + (gy-ry)**2 <= self.close_threshold_squared
```

### 2.3.4 Top Layer

The top layer treats the middle layer as its environment. Note that the top layer
is an environment for us to tell it what to visit.

```python
from display import Displayable
from agentMiddle import Rob_middle_layer
from agents import Environment

class Rob_top_layer(Environment):
    def __init__(self, middle, timeout=200, locations = {
        'mail':(-5,10),
        'o103':(50,10),
        'o109':(100,10),
        'storage':(101,51)
    ):
        """middle is the middle layer
        timeout is the number of steps the middle layer goes before giving
        up
        locations is a loc:pos dictionary
        where loc is a named location, and pos is an (x,y) position.
        """
        self.middle = middle
        self.timeout = timeout # number of steps before the middle layer
        should give up
```

https://aipython.org
2. Agent Architectures and Hierarchical Control

```python
self.locations = locations

def do(self, plan):
    """carry out actions.  
actions is of the form {'visit': list_of_locations}
It visits the locations in turn.
    """
    to_do = plan['visit']
    for loc in to_do:
        position = self.locations[loc]
        arrived = self.middle.do({'go_to': position,
                                  'timeout': self.timeout})
        self.display(1, "Arrived at", loc, arrived)

2.3.5 Plotting

The following is used to plot the locations, the walls and (eventually) the movement of the robot. It can either plot the movement if the robot as it is going (with the default `env.plotting = True`), or not plot it as it is going (setting `env.plotting = False`; in this case the trace can be plotted using `pl.plot_run`).

```
2.3. Hierarchical Controller

Figure 2.2: A trace of the trajectory of the agent. Red dots correspond to the whisker sensor being on; the green dot to the whisker sensor being off. The agent starts at position \((0,0)\) facing up.

```python
def plot_run(self):
    
    if self.body.history:
        xs, ys = zip(*self.body.history)
        plt.plot(xs, ys, "go")

    if self.body.wall_history:
        wxs, wys = zip(*self.body.wall_history)
        plt.plot(wxs, wys, "ro")
```

The following code plots the agent as it acts in the world. Figure 2.2 shows the result of the top.do

```python
from agentEnv import Rob_body, Rob_env

env = Rob_env(((20,0),(30,20)), ((70,-5),(70,25)))
body = Rob_body(env)
middle = Rob_middle_layer(body)
top = Rob_top_layer(middle)

# try:
# pl=Plot_env(body,top)
# top.do({'visit':['o109','storage','o109','o103']})
# You can directly control the middle layer:
# middle.do({'go_to':(30,-10), 'timeout':200})
# Can you make it crash?
```

Exercise 2.2 The following code implements a robot trap (Figure 2.3). Write a
controller that can escape the “trap” and get to the goal. See Exercise 2.4 in the textbook for hints.

```python
# Robot Trap for which the current controller cannot escape:
trap_env = Rob_env(((10,-21),(10,0)), ((10,10),(10,31)),
                  ((30,-10),(30,0)),
                  ((30,10),(30,20)), ((50,-21),(50,31)),
                  ((10,-21),(50,-21)),
                  ((10,0),(30,0)), ((10,10),(30,10)), ((10,31),(50,31))])
trap_body = Rob_body(trap_env,init_pos=(-1,0,90))
trap_middle = Rob_middle_layer(trap_body)
trap_top = Rob_top_layer(trap_middle,locations={"goal":(71,0)})
```

Plotting for Moving Targets

Exercise 2.5 refers to targets that can move. The following implements targets than can be moved by the user (using the mouse).

```python
import matplotlib.pyplot as plt
from agentTop import Plot_env, body, top

class Plot_follow(Plot_env):
    def __init__(self, body, top, epsilon=2.5):
        """plot the agent in the environment.
        epsilon is the threshold how how close someone needs to click to select a location.
        ""
```
2.3. Hierarchical Controller

```python
Plot_env.__init__(self, body, top)
self.epsilon = epsilon
self.canvas = plt.gca().figure.canvas
self.canvas.mpl_connect('button_press_event', self.on_press)
self.canvas.mpl_connect('button_release_event', self.on_release)
self.canvas.mpl_connect('motion_notify_event', self.on_move)
self.pressloc = None
self.pressevent = None
for loc in self.top.locations:
    self.display(2, f' loc {loc} at {self.top.locations[loc]}')

def on_press(self, event):
    self.display(2, 'v', end='')
    self.display(2, f'Press at ({event.xdata}, {event.ydata})')
    for loc in self.top.locations:
        lx, ly = self.top.locations[loc]
        if abs(event.xdata - lx) <= self.epsilon and abs(event.ydata - ly) <= self.epsilon:
            self.pressloc = loc
            self.pressevent = event
            self.display(2, 'moving', loc)

def on_release(self, event):
    self.display(2, '^', end='')
    if self.pressloc is not None: # and event.inaxes == self.pressevent.inaxes:
        self.top.locations[self.pressloc] = (event.xdata, event.ydata)
        self.display(1, f'Placing {self.pressloc} at ({event.xdata},
        event.ydata})
        self.pressloc = None
        self.pressevent = None

def on_move(self, event):
    if self.pressloc is not None: # and event.inaxes ==
        self.pressevent.inaxes:
            self.display(2, '-', end='')
        self.top.locations[self.pressloc] = (event.xdata, event.ydata)
        self.redraw()
else:
    self.display(2, '.', end='')

# try:
# pl=Plot_follow(body,top)
# top.do(['visit':['o109','storage','o109','o103']])
```

Exercise 2.3 Change the code to also allow walls to move.
Chapter 3

Searching for Solutions

3.1 Representing Search Problems

A search problem consists of:

- a start node

- a `neighbors` function that given a node, returns an enumeration of the arcs from the node

- a specification of a goal in terms of a Boolean function that takes a node and returns true if the node is a goal

- a (optional) heuristic function that, given a node, returns a non-negative real number. The heuristic function defaults to zero.

As far as the searcher is concerned a node can be anything. If multiple-path pruning is used, a node must be hashable. In the simple examples, it is a string, but in more complicated examples (in later chapters) it can be a tuple, a frozen set, or a Python object.

In the following code, “raise Not Implemented Error()” is a way to specify that this is an abstract method that needs to be overridden to define an actual search problem.

```python
from display import Displayable
import matplotlib.pyplot as plt
import random

class Search_problem(Displayable):
    """A search problem consists of:
```
3. Searching for Solutions

* a start node
* a neighbors function that gives the neighbors of a node
* a specification of a goal
* a (optional) heuristic function.

The methods must be overridden to define a search problem.

```python
def start_node(self):
    """returns start node""
    raise NotImplementedError("start_node")  # abstract method

def is_goal(self, node):
    """is True if node is a goal""
    raise NotImplementedError("is_goal")  # abstract method

def neighbors(self, node):
    """returns a list (or enumeration) of the arcs for the neighbors of node""
    raise NotImplementedError("neighbors")  # abstract method

def heuristic(self, n):
    """Gives the heuristic value of node n.
    Returns 0 if not overridden."
    return 0
```

The neighbors is a list of arcs. A (directed) arc consists of a `from_node` node and a `to_node` node. The arc is the pair `from_node, to_node`, but can also contain a non-negative cost (which defaults to 1) and can be labeled with an action.

```
class Arc(object):
    """An arc has a from_node and a to_node node and a (non-negative) cost""
    def __init__(self, from_node, to_node, cost=1, action=None):
        self.from_node = from_node
        self.to_node = to_node
        self.action = action
        self.cost = cost
        assert cost >= 0, (f"Cost cannot be negative: {self}, cost={cost}")

def __repr__(self):
    """string representation of an arc"
    if self.action:
        return f"{self.from_node} --{self.action}--> {self.to_node}"
    else:
        return f"{self.from_node} --> {self.to_node}"
```
3.1. Representing Search Problems

3.1.1 Explicit Representation of Search Graph

The first representation of a search problem is from an explicit graph (as opposed to one that is generated as needed).

An explicit graph consists of

- a list or set of nodes
- a list or set of arcs
- a start node
- a list or set of goal nodes
- (optionally) a dictionary that maps a node to a heuristic value for that node

To define a search problem, we need to define the start node, the goal predicate, the neighbors function and the heuristic function.

```python
class Search_problem_from_explicit_graph(Search_problem):
    """A search problem from an explicit graph."
    ""

def __init__(self, title, nodes, arcs, start=None, goals=set(), hmap={},
             positions=None, show_costs = True):
    """A search problem consists of:
* list or set of nodes
* list or set of arcs
* start node
* list or set of goal nodes
* hmap: dictionary that maps each node into its heuristic value.
* positions: dictionary that maps each node into its (x,y) position
* show_costs is used for show()
""

    self.title = title
    self.neighs = {}
    self.nodes = nodes
    for node in nodes:
        self.neighs[node]=[]
    self.arcs = arcs
    for arc in arcs:
        self.neighs[arc.from_node].append(arc)
    self.start = start
    self.goals = goals
    self.hmap = hmap
    if positions is None:
        self.positions = {node:(random.random(),random.random()) for node in nodes}
    else:
```

[https://aipython.org](https://aipython.org)
44

3. Searching for Solutions

```python
    self.positions = positions
    self.show_costs = show_costs

    def start_node(self):
        """returns start node""
        return self.start

    def is_goal(self, node):
        """is True if node is a goal""
        return node in self.goals

    def neighbors(self, node):
        """returns the neighbors of node (a list of arcs)""
        return self.neighs[node]

    def heuristic(self, node):
        """Gives the heuristic value of node n.
        Returns 0 if not overridden in the hmap.""
        if node in self.hmap:
            return self.hmap[node]
        else:
            return 0

    def __repr__(self):
        """returns a string representation of the search problem""
        res ="
        for arc in self.arcs:
            res += f"{(arc). "
        return res
```

Graphical Display of a Search Graph

```python
    def show(self, fontsize=10, node_color='orange', show_costs=None):
        """Show the graph as a figure
        ""
        self.fontsize = fontsize
        if show_costs is not None: # override default definition
            self.show_costs = show_costs
        plt.ion() # interactive
        ax = plt.figure().gca()
        ax.set_axis_off()
        plt.title(self.title, fontsize=fontsize)
        self.show_graph(ax, node_color)

    def show_graph(self, self, ax, node_color='orange'):
        bbox =
        dict(boxstyle="round4", pad=1.0, rounding_size=0.5", facecolor=node_color)
```

https://aipython.org
Version 0.9.12 December 22, 2023
3.1. Representing Search Problems

```python
for arc in self.arcs:
    self.show_arc(ax, arc)
for node in self.nodes:
    self.show_node(ax, node, node_color = node_color)

def show_node(self, ax, node, node_color):
    x,y = self.positions[node]
    ax.text(x,y,node,bbox=dict(boxstyle="round4,pad=1.0,rounding_size=0.5",
                               facecolor=node_color),
                               ha='center',va='center',
                               fontsize=self.fontsize)

def show_arc(self, ax, arc, arc_color='black', node_color='white'):
    from_pos = self.positions[arc.from_node]
    to_pos = self.positions[arc.to_node]
    ax.annotate(arc.to_node, from_pos, xytext=to_pos,
                arrowprops=dict(facecolor='black',
                                 shrink=0.1, width=2),
                arrowprops={'arrowstyle':'<|-', 'linewidth':2, 'color':arc_color},
                bbox=dict(boxstyle="round4,pad=1.0,rounding_size=0.5",
                          facecolor=node_color),
                ha='center',va='center',
                fontsize=self.fontsize)

    # Add costs to middle of arcs:
    if self.show_costs:
        ax.text(((from_pos[0]+to_pos[0])/2, (from_pos[1]+to_pos[1])/2,
                 arc.cost, bbox=dict(pad=1,fc='w',ec='w'),
                 ha='center',va='center',fontsize=self.fontsize)
```

3.1.2 Paths

A searcher will return a path from the start node to a goal node. A Python list is not a suitable representation for a path, as many search algorithms consider multiple paths at once, and these paths should share initial parts of the path. If we wanted to do this with Python lists, we would need to keep copying the list, which can be expensive if the list is long. An alternative representation is used here in terms of a recursive data structure that can share subparts.

A path is either:

- a node (representing a path of length 0) or
- a path, `initial` and an arc, where the `from_node` of the arc is the node at the end of `initial`.

These cases are distinguished in the following code by having `arc=None` if the path has length 0, in which case `initial` is the node of the path. Note that we only use the most basic form of Python's `yield` for enumerations (Section 1.5.4).

[https://aipython.org](https://aipython.org)  
Version 0.9.12  
December 22, 2023
class Path(object):
    """A path is either a node or a path followed by an arc""

    def __init__(self, initial, arc=None):
        """initial is either a node (in which case arc is None) or
        a path (in which case arc is an object of type Arc)""
        self.initial = initial
        self.arc = arc
        if arc is None:
            self.cost = 0
        else:
            self.cost = initial.cost + arc.cost

    def end(self):
        """returns the node at the end of the path""
        if self.arc is None:
            return self.initial
        else:
            return self.arc.to_node

    def nodes(self):
        """enumerates the nodes for the path.
        This enumerates the nodes in the path from the last elements
        backwards.""
        current = self
        while current.arc is not None:
            yield current.arc.to_node
            current = current.initial
        yield current.initial

    def initial_nodes(self):
        """enumerates the nodes for the path before the end node.
        This calls nodes() for the initial part of the path.""
        if self.arc is not None:
            yield from self.initial.nodes()

    def __repr__(self):
        """returns a string representation of a path""
        if self.arc is None:
            return str(self.initial)
        elif self.arc.action:
            return f"""{self.initial}
            --{self.arc.action}-->
            {self.arc.to_node}"
        else:
            return f"""{self.initial} --> {self.arc.to_node}"""
3.1.3 Example Search Problems

The first search problem is one with 5 nodes where the least-cost path is one with many arcs. See Figure 3.1. Note that this example is used for the unit tests, so the test (in searchGeneric) will need to be changed if this is changed.

```python
from searchProblem import Arc, Search_problem_from_explicit_graph,
Search_problem

problem1 = Search_problem_from_explicit_graph('Problem 1',
   {'A','B','C','D','G'},
   [Arc('A','B',3), Arc('A','C',1), Arc('B','D',1), Arc('B','G',3),
    Arc('C','B',1), Arc('C','D',3), Arc('D','G',1)],
   start = 'A',
   goals = {'G'},
   positions={'A': (0, 1), 'B': (0.5, 0.5), 'C': (0,0.5), 'D': (0.5,0),
    'G': (1,0))
```

The second search problem is one with 8 nodes where many paths do not lead to the goal. See Figure 3.2.

```python
problem2 = Search_problem_from_explicit_graph('Problem 2',
   {'A','B','C','D','E','G','H','J'},
   [Arc('A','B',1), Arc('B','C',3), Arc('B','D',1), Arc('D','E',3),
    Arc('D','G',1), Arc('A','H',3), Arc('H','J',1)],
   start = 'A',
   goals = {'G'},
   positions={'A': (0, 1), 'B': (0, 3/4), 'C': (0,0), 'D': (1/4,3/4), 'E':
    (1/4,0),
```

The third search problem is a disconnected graph (contains no arcs), where the start node is a goal node. This is a boundary case to make sure that weird cases work.

https://aipython.org

Version 0.9.12 December 22, 2023
Figure 3.2: problem2

Figure 3.3: simp_delivery_graph with arc costs and h values of nodes

```python
searchExample.py — (continued)

problem3 = Search_problem_from_explicit_graph('Problem 3',
{ 'a', 'b', 'c', 'd', 'e', 'g', 'h', 'j' },
[],
start = 'g',
goals = {'k', 'g'})

The simp_delivery_graph is the graph shown Figure 3.3. This is Figure 3.3 with the heuristics of Figure 3.1 as shown in Figure 3.13 of [?].

searchExample.py — (continued)
simp_delivery_graph = Search_problem_from_explicit_graph("Acyclic Delivery Graph",
[ Arc('A', 'B', 2),
  Arc('A', 'C', 3),
  Arc('A', 'D', 4),
  Arc('B', 'E', 2),
  Arc('B', 'F', 3),
```
cyclic_simp_delivery_graph is the graph shown Figure 3.4. This is the graph of Figure 3.10 of [?]. The heuristic values are the same as in simp_delivery_graph.
3. Searching for Solutions

```
cyclic_simp_delivery_graph = Search_problem_from_explicit_graph("Cyclic
Delivery Graph",
    {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J'},
    [  
        Arc('A', 'B', 2),
        Arc('A', 'C', 3),
        Arc('A', 'D', 4),
        Arc('B', 'E', 2),
        Arc('B', 'F', 3),
        Arc('C', 'A', 3),
        Arc('C', 'J', 6),
        Arc('D', 'A', 4),
        Arc('D', 'H', 4),
        Arc('F', 'A', 4),
        Arc('F', 'D', 2),
        Arc('G', 'B', 3),
        Arc('G', 'J', 4),
        Arc('H', 'D', 4),
        Arc('H', 'G', 3),
        Arc('J', 'C', 6),
        Arc('J', 'G', 4)],
    start = 'A',
    goals = {'G'},
    hmap = {
        'A': 7,
        'B': 5,
        'C': 9,
        'D': 6,
        'E': 3,
        'F': 5,
        'G': 0,
        'H': 3,
        'J': 4,
    },
    positions = {
        'A': (0.4,0.1),
        'B': (0.4,0.4),
        'C': (0.1,0.1),
        'D': (0.7,0.1),
        'E': (0.6,0.7),
        'F': (0.7,0.4),
        'G': (0.7,0.9),
        'H': (0.9,0.6),
        'J': (0.3,0.9)
    })
```

The next problem is the tree graph shown in Figure 3.6 and is Figure 3.15 in ?.
3.1. Representing Search Problems

Figure 3.5: tree_graph.show(show_costs = False)

```python
[
  Arc('A', 'B', 1),
  Arc('A', 'C', 1),
  Arc('B', 'D', 1),
  Arc('B', 'E', 1),
  Arc('C', 'F', 1),
  Arc('C', 'G', 1),
  Arc('D', 'H', 1),
  Arc('D', 'I', 1),
  Arc('E', 'J', 1),
  Arc('E', 'K', 1),
  Arc('F', 'L', 1),
  Arc('G', 'M', 1),
  Arc('G', 'N', 1),
```
Arc('H', 'O', 1),
Arc('H', 'P', 1),
Arc('J', 'Q', 1),
Arc('J', 'R', 1),
Arc('L', 'S', 1),
Arc('L', 'T', 1),
Arc('N', 'U', 1),
Arc('N', 'V', 1),
Arc('O', 'W', 1),
Arc('P', 'X', 1),
Arc('P', 'Y', 1),
Arc('R', 'Z', 1),
Arc('R', 'AA', 1),
Arc('T', 'BB', 1),
Arc('T', 'CC', 1),
Arc('V', 'DD', 1),
Arc('V', 'EE', 1),
Arc('W', 'FF', 1),
Arc('X', 'GG', 1),
Arc('Y', 'HH', 1),
Arc('AA', 'II', 1),
Arc('CC', 'JJ', 1),
Arc('CC', 'KK', 1)
],
start = 'A',
goals = {'K', 'M', 'T', 'X', 'Z', 'HH'},
positions = {
    'A': (0.5, 0.95),
    'B': (0.3, 0.8),
    'C': (0.7, 0.8),
    'D': (0.2, 0.65),
    'E': (0.4, 0.65),
    'F': (0.6, 0.65),
    'G': (0.8, 0.65),
    'H': (0.2, 0.5),
    'I': (0.3, 0.5),
    'J': (0.4, 0.5),
    'K': (0.5, 0.5),
    'L': (0.6, 0.5),
    'M': (0.7, 0.5),
    'N': (0.8, 0.5),
    'O': (0.1, 0.35),
    'P': (0.2, 0.35),
    'Q': (0.3, 0.35),
    'R': (0.4, 0.35),
    'S': (0.5, 0.35),
    'T': (0.6, 0.35),
    'U': (0.7, 0.35),
    'V': (0.8, 0.35),
    'W': (0.1, 0.2),
    'X': (0.2, 0.2),
    'Y': (0.4, 0.2),
    'Z': (0.5, 0.2),
    'AA': (0.6, 0.2),
    'BB': (0.7, 0.2),
    'CC': (0.8, 0.2),
    'DD': (0.1, 0.1),
    'EE': (0.2, 0.1),
    'FF': (0.3, 0.1),
    'GG': (0.4, 0.1),
    'HH': (0.5, 0.1),
    'II': (0.6, 0.1),
    'JJ': (0.7, 0.1),
    'KK': (0.8, 0.1),
    'LL': (0.1, 0.05),
    'MM': (0.2, 0.05),
    'NN': (0.3, 0.05),
    'OO': (0.4, 0.05),
    'PP': (0.5, 0.05),
    'QQ': (0.6, 0.05),
    'RR': (0.7, 0.05),
    'SS': (0.8, 0.05),
    'TT': (0.1, 0.1),
    'UU': (0.2, 0.1),
    'VV': (0.3, 0.1),
    'WW': (0.4, 0.1),
    'XX': (0.5, 0.1),
    'YY': (0.6, 0.1),
    'ZZ': (0.7, 0.1)
}
3.2. Generic Searcher and Variants

To run the search demos, in folder “aipython”, load “searchGeneric.py”, using e.g., ipython -i searchGeneric.py, and copy and paste the example queries at the bottom of that file.

3.2.1 Searcher

A Searcher for a problem can be asked repeatedly for the next path. To solve a problem, you can construct a Searcher object for the problem and then repeatedly ask for the next path using search. If there are no more paths, None is returned.

---

```python
class Searcher(Displayable):
    """
    returns a searcher for a problem.
    Paths can be found by repeatedly calling search().
    This does depth-first search unless overridden
    """
    def __init__(self, problem):
        """
        creates a searcher from a problem 
        """
        self.problem = problem
        self.initialize_frontier()
        self.num_expanded = 0
```

---
self.add_to_frontier(Path(problem.start_node()))
super().__init__()

def initialize_frontier(self):
    self.frontier = []

def empty_frontier(self):
    return self.frontier == []

def add_to_frontier(self,path):
    self.frontier.append(path)

def search(self):
    """returns (next) path from the problem's start node to a goal node.
    Returns None if no path exists.
    """
    while not self.empty_frontier():
        self.path = self.frontier.pop()
        self.num_expanded += 1
        if self.problem.is_goal(self.path.end()): # solution found
            self.solution = self.path # store the solution found
            self.display(1, f"Solution: {self.path} (cost: {self.path.cost})\n",
                         self.num_expanded, "paths have been expanded and",
                         len(self.frontier), "paths remain in the frontier")
            return self.path
        else:
            self.display(4,f"Expanding: {self.path} (cost: {self.path.cost})")
            neighs = self.problem.neighbors(self.path.end())
            self.display(2,f"Expanding: {self.path} with neighbors (neighs)")
            for arc in reversed(list(neighs)):
                self.add_to_frontier(Path(self.path,arc))
            self.display(3, f"New frontier: {[p.end() for p in self.frontier]}")
    self.display(0,"No (more) solutions. Total of",
                 self.num_expanded,"paths expanded.")

Note that this reverses the neighbors so that it implements depth-first search in an intuitive manner (expanding the first neighbor first). The call to list is for the case when the neighbors are generated (and not already in a list). Reversing the neighbors might not be required for other methods. The calls to reversed and list can be removed, and the algorithm still implements depth-first search.

To use depth-first search to find multiple paths for problem1 and simp_delivery_graph, copy and paste the following into Python's read-evaluate-print loop; keep finding next solutions until there are no more:

https://aipython.org
3.2. Generic Searcher and Variants

Expanding: A -> B

Exercise 3.1 Implement breadth-first search. Only add_to_frontier and/or pop need to be modified to implement a first-in first-out queue.

3.2.2 GUI for Tracing Search

This GUI implements most of the functionality of the AISpace.org search app. Figure 3.6 shows the GUI to step through various algorithms. Here the path A -> B is being expanded, and the neighbors are E and F. The other nodes at the end of paths of the frontier are C and D. Thus the frontier contains paths to C and D, used to also contain A -> B, and now will contain A -> B -> E and A -> B -> F.

SearcherGUI takes a search class and a problem, and lets one explore the search space after calling go(). A GUI can only be used for one search; at the end of the search the loop ends and the buttons no longer work.

https://aipython.org
This is implemented by redefining \textit{display}. The search algorithms don’t need to be modified. If you modify them (or create your own), you just have to be careful to use the appropriate number for the display. The first argument to \textit{display} has the following meanings:

1. a solution has been found

2. what is shown for a “step” on a GUI; here it is assumed to be the path, the neighbors of the end of the path, and the other nodes at the end of paths on the frontier

3. (shown with “fine step” but not with “step”) the frontier and the path selected

4. (shown with “fine step” but not with “step”) the frontier.

It is also useful to look at the Python console, as the display information is printed there.

```
import matplotlib.pyplot as plt
from matplotlib.widgets import Button
import time

class SearcherGUI(object):
    def __init__(self, SearchClass, problem, fontsize=10,
                 colors = {'selected': 'red', 'neighbors': 'blue',
                           'frontier': 'green', 'goal': 'yellow'}):
        self.problem = problem
        self.searcher = SearchClass(problem)
        self.problem.fontsize = fontsize
        self.colors = colors
        #self.go()

    def go(self):
        fig, self.ax = plt.subplots()
        plt.ion()  # interactive
        self.ax.set_axis_off()
        plt.subplots_adjust(bottom=0.15)
        step_butt = Button(plt.axes([0.05, 0.02, 0.15, 0.05]), "step")
        step_butt.on_clicked(self.step)
        fine_butt = Button(plt.axes([0.25, 0.02, 0.15, 0.05]), "fine step")
        fine_butt.on_clicked(self.finestep)
        auto_butt = Button(plt.axes([0.45, 0.02, 0.15, 0.05]), "auto search")
        auto_butt.on_clicked(self.auto)
        quit_butt = Button(plt.axes([0.65, 0.02, 0.15, 0.05]), "quit")
        quit_butt.on_clicked(self.quit)
        self.ax.text(0.85, 0,
                     '\n'.join(self.colors[a]+": "+a for a in self.colors))
        self.problem.show_graph(self.ax, node_color='white')
```
3.2. Generic Searcher and Variants

```python
def display(self, level,*args,**nargs):
    if level <= self.click: # step
        print(*args, **nargs)
        self.ax.set_title(f"Expanding: {self.searcher.path}",
                          fontsize=self.problem.fontsize)
        if level == 1:
            self.show_frontier(self.colors['frontier'])
            self.show_path(self.colors['selected'])
            self.ax.set_title(f"Solution Found: {self.searcher.path}",
                              fontsize=self.problem.fontsize)
        elif level == 2: # what should be shown if a node is in all three?
            self.show_frontier(self.colors['frontier'])
            self.show_path(self.colors['selected'])
            self.show_neighbors(self.colors['neighbors'])
        elif level == 3:
            self.show_frontier(self.colors['frontier'])
            self.show_path(self.colors['selected'])
            path_show = self.searcher.path
        elif level == 4:
            self.show_frontier(self.colors['frontier'])

        # wait for a button click
        self.click = 0
        plt.draw()
        while self.click == 0:
            plt.pause(0.1)
        self.ax.set_title(""")
        self.show_frontier('white')
        self.show_neighbors('white')
        self.searcher.display = self.display
        try:
            while self.searcher.frontier:
                path = self.searcher.search()
        except ExitToPython:
            print("Exited")
        else:
            print("No more solutions")
```

https://aipython.org  Version 0.9.12  December 22, 2023
while path_show.arc:
    self.problem.show_arc(self.ax, path_show.arc, 'black')
    self.problem.show_node(self.ax, path_show.end(), 'white')
    path_show = path_show.initial
    self.problem.show_node(self.ax, path_show.end(), 'white')
    if self.problem.is_goal(self.searcher.path.end()):
        self.problem.show_node(self.ax, self.searcher.path.end(),
                               self.colors['goal'])
plt.draw()

def show_frontier(self, color):
    for path in self.searcher.frontier:
        self.problem.show_node(self.ax, path.end(), color)

def show_path(self, color):
    """color selected path""
    path_show = self.searcher.path
    while path_show.arc:
        self.problem.show_arc(self.ax, path_show.arc, color)
        self.problem.show_node(self.ax, path_show.end(), color)
        path_show = path_show.initial
        self.problem.show_node(self.ax, path_show.end(), color)

def show_neighbors(self, color):
    for neigh in self.problem.neighbors(self.searcher.path.end()):
        self.problem.show_node(self.ax, neigh.to_node, color)

def auto(self, event):
    self.click = 1

def step(self, event):
    self.click = 2

def finestep(self, event):
    self.click = 3

def quit(self, event):
    quit()

class ExitToPython(Exception):
    pass

from searchGeneric import Searcher, AStarSearcher
from searchMPP import SearcherMPP
import searchExample
from searchBranchAndBound import DF_branch_and_bound

# to demonstrate depth-first search:
# sdfs = SearcherGUI(Searcher, searchExample.tree_graph); sdfs.go()

# delivery graph examples:
# sh = SearcherGUI(Searcher, searchExample.simp_delivery_graph); sh.go()
3.2. Generic Searcher and Variants

```python
# sha = SearcherGUI(AStarSearcher, searchExample.simp_delivery_graph);
   sha.go()
# shac = SearcherGUI(AStarSearcher,
   searchExample.cyclic_simp_delivery_graph); shac.go()
# shm = SearcherGUI(SearcherMPP,
   searchExample.cyclic_simp_delivery_graph); shm.go()
# shb = SearcherGUI(DF_branch_and_bound,
   searchExample.simp_delivery_graph); shb.go()

# The following is AI:FCA figure 3.15, and is useful to show branch&bound:
# shbt = SearcherGUI(DF_branch_and_bound, searchExample.tree_graph);
   shbt.go()
```

3.2.3 Frontier as a Priority Queue

In many of the search algorithms, such as $A^*$ and other best-first searchers, the frontier is implemented as a priority queue. The following code uses the Python’s built-in priority queue implementations, `heapq`.

Following the lead of the Python documentation, [https://docs.python.org/3/library/heapq.html](https://docs.python.org/3/library/heapq.html), a frontier is a list of triples. The first element of each triple is the value to be minimized. The second element is a unique index which specifies the order that the elements were added to the queue, and the third element is the path that is on the queue. The use of the unique index ensures that the priority queue implementation does not compare paths; whether one path is less than another is not defined. It also lets us control what sort of search (e.g., depth-first or breadth-first) occurs when the value to be minimized does not give a unique next path.

The variable `frontier_index` is the total number of elements of the frontier that have been created. As well as being used as the unique index, it is useful for statistics, particularly in conjunction with the current size of the frontier.

```python
import heapq # part of the Python standard library
from searchProblem import Path

class FrontierPQ(object):
    """A frontier consists of a priority queue (heap), frontierpq, of
    (value, index, path) triples, where
    * value is the value we want to minimize (e.g., path cost + h).
    * index is a unique index for each element
    * path is the path on the queue
    Note that the priority queue always returns the smallest element.
    """
    def __init__(self):
        """constructs the frontier, initially an empty priority queue
        """
        self.frontier_index = 0 # the number of items added to the frontier
```
The following methods are used for finding and printing information about the frontier.

### 3.2.4 $A^*$ Search

For an $A^*$ Search the frontier is implemented using the FrontierPQ class.

```python
class AStarSearcher(Searcher):
    """returns a searcher for a problem.

    Paths can be found by repeatedly calling search().
    """

    def __init__(self, problem):
        self.frontierpq = [] # the frontier priority queue
        def empty(self):
            """is True if the priority queue is empty""
            return self.frontierpq == []
        def add(self, path, value):
            """add a path to the priority queue
    value is the value to be minimized""
            self.frontier_index += 1 # get a new unique index
            heapq.heappush(self.frontierpq,(value, -self.frontier_index, path))
        def pop(self):
            """returns and removes the path of the frontier with minimum value.
    """
            (_,_,path) = heapq.heappop(self.frontierpq)
            return path
        def count(self,val):
            """returns the number of elements of the frontier with value=val""
            return sum(1 for e in self.frontierpq if e[0]==val)
        def __repr__(self):
            """string representation of the frontier""
            return str([(n,c,str(p)) for (n,c,p) in self.frontierpq])
        def __len__(self):
            """length of the frontier""
            return len(self.frontierpq)
        def __iter__(self):
            """iterate through the paths in the frontier""
            for (_,_,path) in self.frontierpq:
                yield path

https://aipython.org
```
3.2. Generic Searcher and Variants

```python
super().__init__(problem)

def initialize_frontier(self):
    self.frontier = FrontierPQ()

def empty_frontier(self):
    return self.frontier.empty()

def add_to_frontier(self,path):
    """add path to the frontier with the appropriate cost""
    value = path.cost+self.problem.heuristic(path.end())
    self.frontier.add(path, value)

Code should always be tested. The following provides a simple unit test, using problem1 as the default problem.

```python
import searchExample

def test(SearchClass, problem=searchExample.problem1,
          solutions=[['G','D','B','C','A']]):
    """Unit test for aipython searching algorithms. SearchClass is a class that takes a problem and implements search()
    problem is a search problem
    solutions is a list of optimal solutions
    """
    print("Testing problem 1:")
    schr1 = SearchClass(problem)
    path1 = schr1.search()
    print("Path found:",path1)
    assert path1 is not None, "No path is found in problem1"
    assert list(path1.nodes()) in solutions, "Shortest path not found in problem1"
    print("Passed unit test")

if __name__ == "__main__":
    test(Searcher)  # what needs to be changed to make this succeed?
    test(AStarSearcher)

# example queries:
# searcher1 = Searcher(searchExample.simp_delivery_graph) # DFS
# searcher1.search() # find first path
# searcher1.search() # find next path
# searcher2 = AStarSearcher(searchExample.simp_delivery_graph) # A*
# searcher2.search() # find first path
# searcher2.search() # find next path
# searcher3 = Searcher(searchExample.cyclic_simp_delivery_graph) # DFS
# searcher3.search() # find first path with DFS. What do you expect to happen?
# searcher4 = AStarSearcher(searchExample.cyclic_simp_delivery_graph) # A*
# searcher4.search() # find first path
```

https://aipython.org

Version 0.9.12

December 22, 2023
Exercise 3.2  Change the code so that it implements (i) best-first search and (ii) lowest-cost-first search. For each of these methods compare it to $A^*$ in terms of the number of paths expanded, and the path found.

Exercise 3.3  The searcher acts like a Python iterator, in that it returns one value (here a path) and then returns other values (paths) on demand, but does not implement the iterator interface. Change the code so it implements the iterator interface. What does this enable us to do?

3.2.5  Multiple Path Pruning

The following implements $A^*$ with multiple-path pruning. It overrides `search()` in `Searcher`.

```python
from searchGeneric import AStarSearcher
from searchProblem import Path

class SearcherMPP(AStarSearcher):
    r"""returns a searcher for a problem. Paths can be found by repeatedly calling search()."
    r""
    def __init__(self, problem):
        super().__init__(problem)
        self.explored = set()

    def search(self):
        r"""returns next path from an element of problem's start nodes to a goal node. Returns None if no path exists."
        while not self.empty_frontier():
            self.path = self.frontier.pop()
            if self.path.end() not in self.explored:
                self.explored.add(self.path.end())
                self.num_expanded += 1
                if self.problem.is_goal(self.path.end()):
                    self.solution = self.path # store the solution found
                    self.display(1, f"Solution: {self.path} (cost: {self.path.cost})\n",
                    self.num_expanded, "paths have been expanded and",
                    len(self.frontier), "paths remain in the frontier")
                    return self.path
            else:
```

To run the multiple-path pruning demo, in folder "aipython", load "searchMPP.py", using e.g., `ipython -i searchMPP.py`, and copy and paste the example queries at the bottom of that file.
3.2. Generic Searcher and Variants

```python
self.display(4,f"Expanding: {self.path} (cost: {self.path.cost})")
neighs = self.problem.neighbors(self.path.end())
self.display(2,f"Expanding: {self.path} with neighbors {neighs}"
for arc in neighs:
    self.add_to_frontier(Path(self.path,arc))
self.display(3, f"New frontier: {[p.end() for p in self.frontier]}")
self.display(0,"No (more) solutions. Total of",
self.num_expanded,"paths expanded.")

from searchGeneric import test
if __name__ == "__main__":
test(SearcherMPP)

import searchExample
# searcherMPPcdp = SearcherMPP(searchExample.cyclic_simp_delivery_graph)
# searcherMPPcdp.search() # find first path
```

Exercise 3.4 Chris was very puzzled as to why there was a minus ("−") in the second element of the tuple added to the heap in the add method in FrontierPQ in searchGeneric.py.

Sam suggested the following example would demonstrate the importance of the minus. Consider an infinite integer grid, where the states are pairs of integers, the start is (0,0), and the goal is (10,10). The neighbors of \((i,j)\) are \((i+1,j)\) and \((i,j+1)\). Consider the heuristic function \(h((i,j)) = |10 - i| + |10 - j|\). Sam suggested you compare how many paths are expanded with the minus and without the minus. searchGrid is a representation of Sam’s graph. If something takes too long, you might consider changing the size.

```
from searchProblem import Search_problem, Arc

class GridProblem(Search_problem):
    """a node is a pair (x,y)""
    def __init__(self, size=10):
        self.size = size
    
    def start_node(self):
        """returns the start node""
        return (0,0)
    
    def is_goal(self,node):
        """returns True when node is a goal node""
        return node == (self.size,self.size)
    
    def neighbors(self,node):
        """returns a list of the neighbors of node""
        (x,y) = node
```

[https://aipython.org](https://aipython.org)  Version 0.9.12  December 22, 2023
3. Searching for Solutions

```python
return [Arc(node,(x+1,y)), Arc(node,(x,y+1))]

def heuristic(self,node):
    (x,y) = node
    return abs(x-self.size)+abs(y-self.size)

class GridProblemNH(GridProblem):
    """Grid problem with a heuristic of 0""
    def heuristic(self,node):
        return 0

from searchGeneric import Searcher, AStarSearcher
from searchMPP import SearcherMPP
from searchBranchAndBound import DF_branch_and_bound

def testGrid(size = 10):
    print("\nWith MPP")
    gridsearchermpp = SearcherMPP(GridProblem(size))
    print(gridsearchermpp.search())
    print("\nWithout MPP")
    gridsearchera = AStarSearcher(GridProblem(size))
    print(gridsearchera.search())
    print("\nWith MPP and a heuristic = 0 (Dijkstra's algorithm)")
    gridsearchermppnh = SearcherMPP(GridProblemNH(size))
    print(gridsearchermppnh.search())
```

Explain to Chris what the minus does and why it is there. Give evidence for your claims. It might be useful to refer to other search strategies in your explanation. As part of your explanation, explain what is special about Sam’s example.

**Exercise 3.5** Implement a searcher that implements cycle pruning instead of multiple-path pruning. You need to decide whether to check for cycles when paths are added to the frontier or when they are removed. (Hint: either method can be implemented by only changing one or two lines in SearcherMPP. Hint: there is a cycle if path.end() in path.initial_nodes()) Compare no pruning, multiple path pruning and cycle pruning for the cyclic delivery problem. Which works better in terms of number of paths expanded, computational time or space?

### 3.3 Branch-and-bound Search

To run the demo, in folder “aipython”, load “searchBranchAndBound.py”, and copy and paste the example queries at the bottom of that file.

Depth-first search methods do not need a priority queue, but can use a list as a stack. In this implementation of branch-and-bound search, we call `search` to find an optimal solution with cost less than bound. This uses depth-first search to find a path to a goal that extends `path` with cost less than the bound.
Once a path to a goal has been found, that path is remembered as the best path, the bound is reduced, and the search continues.

```python
from searchProblem import Path
from searchGeneric import Searcher
from display import Displayable

class DF_branch_and_bound(Searcher):
    """returns a branch and bound searcher for a problem. An optimal path with cost less than bound can be found by calling search()""
    def __init__(self, problem, bound=float("inf")):  # bound gives the initial bound. By default this is infinite - meaning there is no initial pruning due to depth bound
        super().__init__(problem)
        self.best_path = None
        self.bound = bound

    def search(self):
        """returns an optimal solution to a problem with cost less than bound.
        returns None if there is no solution with cost less than bound.""
        self.frontier = [Path(self.problem.start_node())]
        self.num_expanded = 0
        while self.frontier:
            self.path = self.frontier.pop()
            if self.path.cost+self.problem.heuristic(self.path.end()) < self.bound:
                # if self.path.end() not in self.path.initial_nodes(): # for cycle pruning
                self.num_expanded += 1
                if self.problem.is_goal(self.path.end()):
                    self.best_path = self.path
                    self.bound = self.path.cost
                    self.display(1,"New best path:",self.path,"cost:",self.path.cost)
            else:
                neighs = self.problem.neighbors(self.path.end())
                self.display(4,"Neighbors are", neighs)
                for arc in reversed(list(neighs)):
                    self.add_to_frontier(Path(self.path, arc))
                self.display(3,f"New frontier: {[p.end() for p in self.frontier]}")
        self.path = self.best_path
```
self.solution = self.best_path
self.display(1, "Optimal solution is {self.best_path}" if self.best_path
else "No solution found.",
f"Number of paths expanded: {self.num_expanded}")
return self.best_path

Note that this code used reversed in order to expand the neighbors of a node in the left-to-right order one might expect. It does this because pop() removes the rightmost element of the list. The call to list is there because reversed only works on lists and tuples, but the neighbors can be generated.

Here is a unit test and some queries:

```python
from searchGeneric import test
if __name__ == "__main__":
    test(DF_branch_and_bound)

# Example queries:
import searchExample
# searcherb1 = DF_branch_and_bound(searchExample.simp_delivery_graph)
# searcherb1.search() # find optimal path
# searcherb2 =
#    DF_branch_and_bound(searchExample.cyclic_simp_delivery_graph,
#        bound=100)
# searcherb2.search() # find optimal path
```

**Exercise 3.6** In searcherb2, in the code above, what happens if the bound is smaller, say 10? What if it is larger, say 1000?

**Exercise 3.7** Implement a branch-and-bound search using recursion. Hint: you don’t need an explicit frontier, but can do a recursive call for the children.

**Exercise 3.8** After the branch-and-bound search found a solution, Sam ran search again, and noticed a different count. Sam hypothesized that this count was related to the number of nodes that an $A^*$ search would use (either expand or be added to the frontier). Or maybe, Sam thought, the count for a number of nodes when the bound is slightly above the optimal path case is related to how $A^*$ would work. Is there a relationship between these counts? Are there different things that it could count so they are related? Try to find the most specific statement that is true, and explain why it is true.

To test the hypothesis, Sam wrote the following code, but isn’t sure it is helpful:

```python
from searchGeneric import Searcher, AStarSearcher
from searchBranchAndBound import DF_branch_and_bound
from searchMPP import SearcherMPP
DF_branch_and_bound.max_display_level = 1
Searcher.max_display_level = 1
```

**https://aipython.org**
3.3. Branch-and-bound Search

```python
def run(problem, name):
    print("\n\n*******", name)
    print("\nA*:")
    asearcher = AStarSearcher(problem)
    print("Path found:", asearcher.search(), " cost=", asearcher.solution.cost)
    print("there are", asearcher.frontier.count(asearcher.solution.cost),
          "elements remaining on the queue with
          f-value=", asearcher.solution.cost)

    print("\nA* with MPP:")
    msearcher = SearcherMPP(problem)
    print("Path found:", msearcher.search(), " cost=", msearcher.solution.cost)
    print("there are", msearcher.frontier.count(msearcher.solution.cost),
          "elements remaining on the queue with
          f-value=", msearcher.solution.cost)

    bound = asearcher.solution.cost+0.01
    print("\nBranch and bound (with too-good initial bound of", bound,")")
    tbb = DF_branch_and_bound(problem, bound) # cheating!!!!
    print("Path found:", tbb.search(), " cost=", tbb.solution.cost)
    print("Rerunning B&B")
    print("Path found:", tbb.search())

    bbound = asearcher.solution.cost*2+10
    print("\nBranch and bound (with not-very-good initial bound of", bbound,")")
    tbb2 = DF_branch_and_bound(problem, bbound)
    print("Path found:", tbb2.search(), " cost=", tbb2.solution.cost)
    print("Rerunning B&B")
    print("Path found:", tbb2.search())

    print("\nDepth-first search: (Use ˆC if it goes on forever")
    tsearcher = Searcher(problem)
    print("Path found:", tsearcher.search(), " cost=", tsearcher.solution.cost)

import searchExample
from searchTest import run
if __name__ == "__main__":
    run(searchExample.problem1, "Problem 1")
    # run(searchExample.simp_delivery_graph, "Acyclic Delivery")
    # run(searchExample.cyclic_simp_delivery_graph, "Cyclic Delivery")
    # also test some graphs with cycles, and some with multiple least-cost paths
```

https://aipython.org  Version 0.9.12  December 22, 2023
Chapter 4

Reasoning with Constraints

4.1 Constraint Satisfaction Problems

4.1.1 Variables

A variable consists of a name, a domain and an optional (x,y) position (for displaying). The domain of a variable is a list or a tuple, as the ordering will matter in the representation of constraints.

```python
import random

class Variable(object):
    """A random variable.
    name (string) - name of the variable
domain (list) - a list of the values for the variable.
Variables are ordered according to their name.
""

def __init__(self, name, domain, position=None):
    """Variable
    name a string
domain a list of printable values
position of form (x,y)
""
    self.name = name # string
    self.domain = domain # list of values
    self.position = position if position else (random.random(), random.random())
    self.size = len(domain)

def __str__(self):
```

---

variable.py — Representations of a variable in CSPs and probabilistic models

---
4. Reasoning with Constraints

4.1.2 Constraints

A constraint consists of:

- A tuple (or list) of variables called the **scope**.

- A **condition**, a Boolean function that takes the same number of arguments as there are variables in the scope. The condition must have a `__name__` property that gives a printable name of the function; built-in functions and functions that are defined using `def` have such a property; for other functions you may need to define this property.

- An optional name

- An optional \((x, y)\) position

```python
return self.name

def __repr__(self):
    return self.name # f"Variable({self.name})"
```

---

```python
from variable import Variable

# for showing csps:
import matplotlib.pyplot as plt
import matplotlib.lines as lines

class Constraint(object):
    """A Constraint consists of
    * scope: a tuple of variables
    * condition: a Boolean function that can applied to a tuple of values
    for variables in scope
    * string: a string for printing the constraints. All of the strings
    must be unique.
    for the variables
    """
    def __init__(self, scope, condition, string=None, position=None):
        self.scope = scope
        self.condition = condition
        if string is None:
            self.string = f"{self.condition.__name__}({self.scope})"
        else:
            self.string = string
        self.position = position

    def __repr__(self):
        return self.string
```

---

https://aipython.org  Version 0.9.12    December 22, 2023
4.1. Constraint Satisfaction Problems

An assignment is a variable: value dictionary.

If con is a constraint, con.holds(assignment) returns True or False depending on whether the condition is true or false for that assignment. The assignment assignment must assign a value to every variable in the scope of the constraint con (and could also assign values to other variables); con.holds gives an error if not all variables in the scope of con are assigned in the assignment. It ignores variables in assignment that are not in the scope of the constraint.

In Python, the * notation is used for unpacking a tuple. For example, F(*(1,2,3)) is the same as F(1,2,3). So if t has value (1,2,3), then F(*t) is the same as F(1,2,3).

4.1.3 CSPs

A constraint satisfaction problem (CSP) requires:

- **variables**: a list or set of variables
- **constraints**: a set or list of constraints.

Other properties are inferred from these:

- **var_to_const** is a mapping from variables to set of constraints, such that var_to_const[var] is the set of constraints with var in the scope.
def __init__(self, title, variables, constraints):
    """title is a string
    variables is set of variables
    constraints is a list of constraints
    """
    self.title = title
    self.variables = variables
    self.constraints = constraints
    self.var_to_const = {var:set() for var in self.variables}
    for con in constraints:
        for var in con.scope:
            self.var_to_const[var].add(con)

    def __str__(self):
        """string representation of CSP""
        return str(self.title)

    def __repr__(self):
        """more detailed string representation of CSP""
        return f"""CSP({self.title}, {self.variables}, {([str(c) for c in self.constraints])})""

csp.consistent(assignment) returns true if the assignment is consistent with each
of the constraints in csp (i.e., all of the constraints that can be evaluated
to true). Note that this is a local consistency with each constraint; it does not
imply the CSP is consistent or has a solution.

def consistent(self, assignment):
    """assignment is a variable:value dictionary
    returns True if all of the constraints that can be evaluated
    evaluate to True given assignment.
    """
    return all(con.holds(assignment)
                for con in self.constraints
                if con.can_evaluate(assignment))

The show method uses matplotlib to show the graphical structure of a con-
straint network. If the node positions are not specified, this gives different
positions each time it is run; if you don’t like the graph, try again.
4.1. Constraint Satisfaction Problems

```python
for var in self.variables:
    if var.position is None:
        var.position = (random.random(), random.random())
self.showAutoAC = showAutoAC # used for consistency GUI
self.autoAC = False
domains = {var: var.domain for var in self.variables} if showDomains else {}
self.draw_graph(domains=domains)

def draw_graph(self, domains={}, to_do={}, title=None, fontsize=10):
    self.ax.clear()
    self.ax.set_axis_off()
    if title:
        plt.title(title, fontsize=fontsize)
    else:
        plt.title(self.title, fontsize=fontsize)
    var_bbox = dict(boxstyle="round4",pad=1.0, rounding_size=0.5")
    con_bbox = dict(boxstyle="square", pad=1.0", color="green")
    self.autoACtext = plt.text(0, 0,"Auto AC" if self.showAutoAC else ",
    bbox={"boxstyle":'square', "color":"yellow"},
    picker=True, fontsize=fontsize)
    for con in self.constraints:
        if con.position is None:
            con.position = tuple(sum(var.position[i] for var in con.scope)/len(con.scope)
            for i in range(2))
        cx, cy = con.position
        bbox = dict(boxstyle="square", pad=1.0", color="green")
        for var in con.scope:
            vx, vy = var.position
            if (var, con) in to_do:
                color = 'blue'
            else:
                color = 'limegreen'
            line = lines.Line2D([cx, vx], [cy, vy], axes=self.ax,
                color=color,
                picker=True, pickradius=10,
                linewidth=self.linewidth)
            self.arcs[line] = (var, con)
            self.thelines[(var, con)] = line
            self.ax.add_line(line)
p1.text(cx, cy, con.string,
                bbox=con_bbox,
                ha='center', va='center', fontsize=fontsize)
    for var in self.variables:
        x, y = var.position
        if domains:
            node_label = f"{var.name}\n{domains[var]}"
        else:
            node_label = var.name
```

https://aipython.org  Version 0.9.12  December 22, 2023
4.4 Examples

In the following code $ne_\_\_$, when given a number, returns a function that is true when its argument is not that number. For example, if $f = ne_\_(3)$, then $f(2)$ is True and $f(3)$ is False. That is, $ne_\_(x)(y)$ is true when $x \neq y$. Allowing a function of multiple arguments to use its arguments one at a time is called currying, after the logician Haskell Curry. Functions used as conditions in constraints require names (so they can be printed).

```python
def ne_(val):
    """not equal value""
    # nev = lambda x: x != val # alternative definition
    # nev = partial(neq,val) # another alternative definition
    def nev(x):
        return val != x
    nev.__name__ = f"{val} != " # name of the function
    return nev
```

Similarly $is_\_(x)(y)$ is true when $x = y$.

```python
def is_(val):
    """is a value""
    # isv = lambda x: x == val # alternative definition
```

4.4.4 Examples

In the following code $ne_\_\_$, when given a number, returns a function that is true when its argument is not that number. For example, if $f = ne_\_(3)$, then $f(2)$ is True and $f(3)$ is False. That is, $ne_\_(x)(y)$ is true when $x \neq y$. Allowing a function of multiple arguments to use its arguments one at a time is called currying, after the logician Haskell Curry. Functions used as conditions in constraints require names (so they can be printed).

```python
def ne_(val):
    """not equal value""
    # nev = lambda x: x != val # alternative definition
    # nev = partial(neq,val) # another alternative definition
    def nev(x):
        return val != x
    nev.__name__ = f"{val} != " # name of the function
    return nev
```

Similarly $is_\_(x)(y)$ is true when $x = y$.

```python
def is_(val):
    """is a value""
    # isv = lambda x: x == val # alternative definition
```
The CSP, \( csp_0 \) has variables \( X \), \( Y \) and \( Z \), each with domain \( \{1, 2, 3\} \). The constraints are \( X < Y \) and \( Y < Z \).

The CSP, \( csp_1 \) has variables \( A \), \( B \) and \( C \), each with domain \( \{1, 2, 3, 4\} \). The constraints are \( A < B \), \( B \neq 2 \), and \( B < C \). This is slightly more interesting than \( csp_0 \) as it has more solutions. This example is used in the unit tests, and so if it is changed, the unit tests need to be changed. The CSP \( csp_1s \) is the same, but with only the constraints \( A < B \) and \( B < C \).
The next CSP, \textit{csp2} is Example 4.9 of \textit{?}; the domain consistent network (after applying the unary constraints) is shown in Figure 4.2. Note that we use the same variables as the previous example and add two more.

```python
D = Variable('D', {1,2,3,4}, position=(0.0,0.4))
E = Variable('E', {1,2,3,4}, position=(0.5,0))
csp2 = CSP("csp2", {A,B,C,D,E},
[ Constraint([B], ne_(3), "B != 3", position=(1,0.9)),
  Constraint([C], ne_(2), "C != 2", position=(1,0.2)),
  Constraint([A,B], ne, "A != B"),
  Constraint([B,C], ne, "A != C"),
  Constraint([C,D], lt, "C < D"),
  Constraint([A,D], eq, "A = D"),
  Constraint([E,A], lt, "E < A"),
  Constraint([E,B], lt, "E < B"),
  Constraint([E,C], lt, "E < C"),
  Constraint([E,D], lt, "E < D"),
  Constraint([B,D], ne, "B != D")])
```
The following example is another scheduling problem (but with multiple answers). This is the same as “scheduling 2” in the original AIspace.org consistency app.

```python
csp3 = CSP("csp3", {A,B,C,D,E},
    [Constraint([A,B], ne, "A != B"),
     Constraint([A,D], lt, "A < D"),
     Constraint([A,E], lambda a,e: (a-e)%2 == 1, "A-E is odd"),
     Constraint([B,E], lt, "B < E"),
     Constraint([D,C], lt, "D < C"),
     Constraint([C,E], ne, "C != E"),
     Constraint([D,E], ne, "D != E")])
```

The following example is another abstract scheduling problem. What are the solutions?

```python
def adjacent(x,y):
    """True when x and y are adjacent numbers""
    return abs(x-y) == 1

csp4 = CSP("csp4", {A,B,C,D},
```

Figure 4.3: csp3.show()
The following examples represent the crossword shown in Figure 4.5.

In the first representation, the variables represent words. The constraint imposed by the crossword is that where two words intersect, the letter at the intersection must be the same. The method `meet_at` is used to test whether two words intersect with the same letter. For example, the constraint `meet_at(2,0)` means that the third letter (at position 2) of the first argument is the same as the first letter of the second argument. This is shown in Figure 4.6.

```python
def meet_at(p1, p2):
    """returns a function of two words that is true
    when the words intersect at positions p1, p2.
    The positions are relative to the words; starting at position 0.
    meet_at(p1,p2)(w1,w2) is true if the same letter is at position p1 of
    word w1
    and at position p2 of word w2.
    ""
    def meets(w1,w2):
        return w1[p1] == w2[p2]
    meets.__name__ = f"meet_at({p1},{p2})"
    return meets
```

https://aipython.org
4.1. Constraint Satisfaction Problems

Figure 4.5: crossword1: a crossword puzzle to be solved

Words:
ant, big, bus, car, has, book, buys, hold, lane, year, ginger, search, symbol, syntax.

Figure 4.6: crossword1.show()
In an alternative representation of a crossword (the “dual” representation), the variables represent letters, and the constraints are that adjacent sequences of letters form words. This is shown in Figure 4.7.
4.1. Constraint Satisfaction Problems

```python
words = {'ant', 'big', 'bus', 'car', 'has', 'book', 'buys', 'hold',
         'lane', 'year', 'ginger', 'search', 'symbol', 'syntax'}

def is_word(*letters, words=words):
    """is true if the letters concatenated form a word in words""
    return ''.join(letters) in words

letters = {"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l",
           "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y",
           "z"}

# pij is the variable representing the letter i from the left and j down
# (starting from 0)
p00 = Variable('p00', letters, position=(0.1,0.85))
p10 = Variable('p10', letters, position=(0.3,0.85))
p20 = Variable('p20', letters, position=(0.5,0.85))
p01 = Variable('p01', letters, position=(0.1,0.7))
p21 = Variable('p21', letters, position=(0.5,0.7))
p02 = Variable('p02', letters, position=(0.1,0.55))
p12 = Variable('p12', letters, position=(0.3,0.55))
p22 = Variable('p22', letters, position=(0.5,0.55))
p32 = Variable('p32', letters, position=(0.7,0.55))
p03 = Variable('p03', letters, position=(0.1,0.4))
p23 = Variable('p23', letters, position=(0.5,0.4))
p24 = Variable('p24', letters, position=(0.5,0.25))
p34 = Variable('p34', letters, position=(0.7,0.25))
p44 = Variable('p44', letters, position=(0.9,0.25))
p25 = Variable('p25', letters, position=(0.5,0.1))

crossword1d = CSP("crossword1d",
                   {p00, p10, p20, # first row
                    p01, p21, # second row
                    p02, p12, p22, p32, # third row
                    p03, p23, # fourth row
                    p24, p34, p44, # fifth row
                    p25 # sixth row
                   },
                   [Constraint([p00, p10, p20], is_word,
                                position=(0.3,0.95)), #1-across
                    Constraint([p00, p01, p02, p03], is_word,
                                position=(0,0.625)), # 1-down
                    Constraint([p02, p12, p22, p32], is_word,
                                position=(0.3,0.625)), # 3-across
                    Constraint([p20, p21, p22, p23, p24, p25], is_word,
                                position=(0.45,0.475)), # 2-down
                    Constraint([p24, p34, p44], is_word,
                                position=(0.7,0.325)) # 4-across
                   ])
```
Exercise 4.1  How many assignments of a value to each variable are there for each of the representations of the above crossword? Do you think an exhaustive enumeration will work for either one?

The queens problem is a puzzle on a chess board, where the idea is to place a queen on each column so the queens cannot take each other: there are no two queens on the same row, column or diagonal. The n-queens problem is a generalization where the size of the board is an $n \times n$, and $n$ queens have to be placed.

Here is a representation of the n-queens problem, where the variables are the columns and the values are the rows in which the queen is placed. The original queens problem on a standard $(8 \times 8)$ chess board is $n\text{\_queens}(8)$

```python
def queens(ri,rj):
    """ri and rj are different rows, return the condition that the queens cannot take each other""
    def no_take(ci,cj):
        """is true if queen at (ri,ci) cannot take a queen at (rj,cj)""
        return ci != cj and abs(ri-ci) != abs(rj-cj)
    return no_take

def n_queens(n):
    """returns a CSP for n-queens""
    columns = list(range(n))
    variables = [Variable(f"R{i}",columns) for i in range(n)]
    return CSP("n-queens",
                variables,
                [Constraint([variables[i], variables[j]], queens(i,j))
                 for i in range(n) for j in range(n) if i != j])
```

Exercise 4.2  How many constraints does this representation of the n-queens problem produce? Can it be done with fewer constraints? Either explain why it can't be done with fewer constraints, or give a solution using fewer constraints.

Unit tests

The following defines a unit test for csp solvers, by default using example csp1.

```python
def test_csp(CSP_solver, csp=csp1,
            solutions=[[A: 1, B: 3, C: 4], [A: 2, B: 3, C: 4]]):
    """CSP_solver is a solver that takes a csp and returns a solution
    csp is a constraint satisfaction problem
    solutions is the list of all solutions to csp
    This tests whether the solution returned by CSP_solver is a solution.
    """
```
4.2. A Simple Depth-first Solver

The first solver carries out a depth-first search through the space of partial assignments. This takes in a CSP problem and an optional variable ordering (a list of the variables in the CSP). It returns a generator of the solutions (see Section 1.5.4 on yield for enumerations).

```python
import cspExamples

def dfs_solver(constraints, context, var_order):
    """generator for all solutions to csp.
    context is an assignment of values to some of the variables.
    var_order is a list of the variables in csp that are not in context.
    ""
    to_eval = {c for c in constraints if c.can_evaluate(context)}
    if all(c.holds(context) for c in to_eval):
        if var_order == []:
            yield context
        else:
            rem_cons = [c for c in constraints if c not in to_eval]
            var = var_order[0]
            for val in var.domain:
                yield from dfs_solver(rem_cons, context|{var:val}, var_order[1:])

def dfs_solve_all(csp, var_order=None):
    """depth-first CSP solver to return a list of all solutions to csp.
    ""
    if var_order == None: # use an arbitrary variable order
        var_order = list(csp.variables)
    return list(dfs_solver(csp.constraints, {}, var_order))
```

Exercise 4.3 Modify `test` so that instead of taking in a list of solutions, it checks whether the returned solution actually is a solution.

Exercise 4.4 Propose a test that is appropriate for CSPs with no solutions. Assume that the test designer knows there are no solutions. Consider what a CSP solver should return if there are no solutions to the CSP.

Exercise 4.5 Write a unit test that checks whether all solutions (e.g., for the search algorithms that can return multiple solutions) are correct, and whether all solutions can be found.
def dfs_solve1(csp, var_order=None):
    """depth-first CSP solver""
    if var_order == None: # use an arbitrary variable order
        var_order = list(csp.variables)
    for sol in dfs_solver(csp.constraints, {}, var_order):
        return sol #return first one

if __name__ == "__main__":
    cspExamples.test_csp(dfs_solve1)

#Try:
# dfs_solve_all(cspExamples.csp1)
#dfs_solve_all(cspExamples.csp2)
#dfs_solve_all(cspExamples.crossword1)
#dfs_solve_all(cspExamples.crossword1d) # warning: may take a *very* long time!

Exercise 4.6 Instead of testing all constraints at every node, change it so each constraint is only tested when all of its variables are assigned. Given an elimination ordering, it is possible to determine when each constraint needs to be tested. Implement this. Hint: create a parallel list of sets of constraints, where at each position $i$ in the list, the constraints at position $i$ can be evaluated when the variable at position $i$ has been assigned.

Exercise 4.7 Estimate how long dfs_solve_all(crossword1d) will take on your computer. To do this, reduce the number of variables that need to be assigned, so that the simplified problem can be solved in a reasonable time (between 0.1 second and 10 seconds). This can be done by reducing the number of variables in var_order, as the program only splits on these. How much more time will it take if the number of variables is increased by 1? (Try it!) Then extrapolate to all of the variables. See Section 1.6.1 for how to time your code. Would making the code 100 times faster or using a computer 100 times faster help?

4.3 Converting CSPs to Search Problems

To run the demo, in folder "aipython", load "cspSearch.py", and copy and paste the example queries at the bottom of that file.

The next solver constructs a search space that can be solved using the search methods of the previous chapter. This takes in a CSP problem and an optional variable ordering, which is a list of the variables in the CSP. In this search space:

- A node is a variable : value dictionary which does not violate any constraints (so that dictionaries that violate any constraints are not added).

- An arc corresponds to an assignment of a value to the next variable. This assumes a static ordering; the next variable chosen to split does not depend on the context. If no variable ordering is given, this makes no attempt to choose a good ordering.

https://aipython.org

Version 0.9.12

December 22, 2023
from cspProblem import CSP, Constraint
from searchProblem import Arc, Search_problem

class Search_from_CSP(Search_problem):
    """A search problem directly from the CSP."
    """
    A node is a variable:value dictionary"
    def __init__(self, csp, variable_order=None):
        self.csp=csp
        if variable_order:
            assert set(variable_order) == set(csp.variables)
            assert len(variable_order) == len(csp.variables)
        else:
            self.variables = list(csp.variables)

    def is_goal(self, node):
        """returns whether the current node is a goal for the search
        """
        return len(node)==len(self.csp.variables)

    def start_node(self):
        """returns the start node for the search
        """
        return {}

    def neighbors(self, node):
        """returns a list of the neighboring nodes of node.
        """
        var = self.variables[len(node)] # the next variable
        res = []
        for val in var.domain:
            new_env = node|{var:val} #dictionary union
            if self.csp.consistent(new_env):
                res.append(Arc(node,new_env))
        return res

The neighbors(node) method uses the fact that the length of the node, which is the number of variables already assigned, is the index of the next variable to split on. Note that we do not need to check whether there are no more variables to split on, as the nodes are all consistent, by construction, and so when there are no more variables we have a solution, and so don’t need the neighbors.

import cspExamples
from searchGeneric import Searcher
def solver_from_searcher(csp):
    """depth-first search solver""
    path = Searcher(Search_from_CSP(csp)).search()
    if path is not None:
        return path.end()
    else:
        return None

if __name__ == "__main__":
    test_csp(solver_from_searcher)

## Test Solving CSPs with Search:
searcher1 = Searcher(Search_from_CSP(cspExamples.csp1))
# print(searcher1.search()) # get next solution
searcher2 = Searcher(Search_from_CSP(cspExamples.csp2))
# print(searcher2.search()) # get next solution
searcher3 = Searcher(Search_from_CSP(cspExamples.crossword1))
# print(searcher3.search()) # get next solution
searcher4 = Searcher(Search_from_CSP(cspExamples.crossword1d))
# print(searcher4.search()) # get next solution (warning: slow)

Exercise 4.8 What would happen if we constructed the new assignment by assigning `node[var] = val` (with side effects) instead of using dictionary union? Give an example of where this could give a wrong answer. How could the algorithm be changed to work with side effects? (Hint: think about what information needs to be in a node).

Exercise 4.9 Change neighbors so that it returns an iterator of values rather than a list. (Hint: use `yield`.)

4.4 Consistency Algorithms

To run the demo, in folder "aipython", load "cspConsistency.py", and copy and paste the commented-out example queries at the bottom of that file.

A `Con Solver` is used to simplify a CSP using arc consistency.

```python
from display import Displayable
class Con Solver(Displayable):
    """Solves a CSP with arc consistency and domain splitting ""
    def __init__(self, csp):
        """a CSP solver that uses arc consistency
        * csp is the CSP to be solved ""
```

self.csp = csp
super().__init__()  # Or Displayable.__init__(self)

The following implementation of arc consistency maintains the set to_do of (variable, constraint) pairs that are to be checked. It takes in a domain dictionary and returns a new domain dictionary. It needs to be careful to avoid side effects (by copying the domains dictionary and the to_do set).

def make_arc_consistent(self, domains=None, to_do=None):
    """Makes this CSP arc-consistent using generalized arc consistency
    domains is a variable:domain dictionary
    to_do is a set of (variable,constraint) pairs
    returns the reduced domains (an arc-consistent variable:domain
dictionary)
    """
    if domains is None:
        self.domains = {var:var.domain for var in self.csp.variables}
    else:
        self.domains = domains.copy()  # use a copy of domains
    if to_do is None:
        to_do = {(var, const) for const in self.csp.constraints
                 for var in const.scope}
    else:
        to_do = to_do.copy()  # use a copy of to_do
    self.display(5,"Performing AC with domains", self.domains)
    while to_do:
        self.arc_selected = (var, const) = self.select_arc(to_do)
        self.display(5, "Processing arc (", var, ",", const, ")")
        other_vars = [ov for ov in const.scope if ov != var]
        new_domain = {val for val in self.domains[var]
                      if self.any_holds(self.domains, const, {var: val}, other_vars)}
        if new_domain != self.domains[var]:
            self.add_to_do = self.new_to_do(var, const) - to_do
            self.display(3, f"Arc: {{var}, {const}} is inconsistent\n            f"Domain pruned, dom({var}) ={new_domain} due to
            {const}""
            self.domains[var] = new_domain
        self.display(4, " adding", self.add_to_do if self.add_to_do
            else "nothing", "to to_do.")
        to_do |= self.add_to_do  # set union
        self.display(5, f"Arc: {{var},{const}} now consistent")
    self.display(5, "AC done. Reduced domains", self.domains)
    return self.domains

def new_to_do(self, var, const):
    """returns new elements to be added to to_do after assigning
    variable var in constraint const.
    """
    return ((nvar, nconst) for nconst in self.csp.var_to_const[var]
if nconst != const
for nvar in nconst.scope
    if nvar != var

The following selects an arc. Any element of to-do can be selected. The selected element needs to be removed from to-do. The default implementation just selects which ever element pop method for sets returns. The graphical user interface below allows the user to select an arc. Alternatively, a more sophisticated selection could be employed.

def select_arc(self, to_do):
    """Selects the arc to be taken from to_do.
    * to_do is a set of arcs, where an arc is a (variable,constraint) pair
    the element selected must be removed from to_do.
    """
    return to_do.pop()

The value of new_domain is the subset of the domain of var that is consistent with the assignment to the other variables. To make it easier to understand, the following treats unary (with no other variables in the constraint) and binary (with one other variables in the constraint) constraints as special cases. These cases are not strictly necessary; the last case covers the first two cases, but is more difficult to understand without seeing the first two cases. Note that this case analysis is not in the code distribution, but can replace the assignment to new_domain above.

if len(other_vars)==0:  # unary constraint
    new_domain = {val for val in self.domains[var]
        if const.holds({var:val})}
elif len(other_vars)==1:  # binary constraint
    other = other_vars[0]
    new_domain = {val for val in self.domains[var]
        if any(const.holds({var: val,other:other_val})
            for other_val in self.domains[other])}
else:  # general case
    new_domain = {val for val in self.domains[var]
        if self.any_holds(self.domains, const, {var: val}, other_vars)}

any_holds is a recursive function that tries to finds an assignment of values to the other variables (other_vars) that satisfies constraint const given the assignment in env. The integer variable ind specifies which index to other_vars needs to be checked next. As soon as one assignment returns True, the algorithm returns True.

def any_holds(self, domains, const, env, other_vars, ind=0):
    """returns True if Constraint const holds for an assignment
4.4. Consistency Algorithms

```python
that extends env with the variables in other_vars[ind:]
env is a dictionary
"
if ind == len(other_vars):
    return const.holds(env)
else:
    var = other_vars[ind]
    for val in domains[var]:
        if self.any_holds(domains, const, env|{var:val}, other_vars, ind + 1):
            return True
    return False
```

4.4.1 Direct Implementation of Domain Splitting

The following is a direct implementation of domain splitting with arc consistency. It implements the generator interface of Python (see Section 1.5.4). When it has found a solution it yields the result; otherwise it recursively splits a domain (using `yield from`).

```python
def generate_sols(self, domains=None, to_do=None, context=dict()):
    """return list of all solution to the current CSP
    to_do is the list of arcs to check
    context is a dictionary of splits made (used for display)
    """
    new_domains = self.make_arc_consistent(domains, to_do)
    if any(len(new_domains[var]) == 0 for var in new_domains):
        self.display(1,f"No solutions for context {context}"
    elif all(len(new_domains[var]) == 1 for var in new_domains):
        self.display(1, "solution: ", str({var: select(new_domains[var]) for var in new_domains}))
        yield {var: select(new_domains[var]) for var in new_domains}
    else:
        var = self.select_var(x for x in self.csp.variables if len(new_domains[var]) > 1)
        dom1, dom2 = partition_domain(new_domains[var])
        self.display(4, "...splitting", var, " into", dom1, "and", dom2)
        new_doms1 = new_domains | {var:dom1}
        new_doms2 = new_domains | {var:dom2}
        to_do = self.new_to_do(var, None)
        self.display(4, " adding", to_do if to_do else "nothing", " to"
        to_do.")
        yield from self.generate_sols(new_doms1, to_do, context|{var:dom1})
        yield from self.generate_sols(new_doms2, to_do, context|{var:dom1})

def solve_all(self, domains=None, to_do=None):
```

[https://aipython.org](https://aipython.org)
def solve_one(self, domains=None, to_do=None):
    return select(self.generate_sols())

def select_var(self, iter_vars):
    """return the next variable to split""
    return select(iter_vars)

def partition_domain(dom):
    """partitions domain dom into two.
    ""
    split = len(dom) // 2
    dom1 = set(list(dom)[:split])
    dom2 = dom - dom1
    return dom1, dom2

def select(iterable):
    """select an element of iterable. Returns None if there is no such
    element.
    ""
    This implementation just picks the first element.
    For many of the uses, which element is selected does not affect
    correctness,
    but may affect efficiency.
    ""
    for e in iterable:
        return e # returns first element found

Exercise 4.10  Implement solve_all that returns the set of all solutions without
using yield. Hint: it can be like generate_sols but returns a set of solutions; the
recursive calls can be unioned; | is Python's union.

Exercise 4.11 Implement solve_one that returns one solution if one exists, or False
otherwise, without using yield. Hint: Python's "or" has the behaviour A or B
will return the value of A unless it is None or False, in which case the value of B is
returned.

Unit test:

import cspExamples
def ac_solver(csp):
    "arc consistency (ac_solver)"
    for sol in Con_solver(csp).generate_sols():
        return sol

if __name__ == "__main__":
    cspExamples.test_csp(ac_solver)
4.4. Consistency Algorithms

4.4.2 Consistency GUI

The consistency GUI allows students to step through the algorithm, choosing which arc to process next, and which variable to split.

Figure 4.8 shows the state of the GUI after two arcs have been made arc consistent. The arcs on the to_do list are colored blue. The green arcs are those that have been made arc consistent. The user can click on a blue arc to process that arc. If the arc selected is not arc consistent, it is made red, the domain is reduced, and then the arc becomes green. If the arc was already arc consistent it turns green.

This is implemented by overriding select_arc and select_var to allow the user to pick the arcs and the variables, and overriding display to allow for the animation. Note that the first argument of display (the number) in the code above is interpreted with a special meaning by the GUI and should only be changed with care.

Clicking AutoAC automates arc selection until the network is arc consistent.

```python
from cspConsistency import Con_solver
import matplotlib.pyplot as plt

class ConsistencyGUI(Con_solver):
    def __init__(self, csp, fontsize=10, speed=1, **kwargs):
        """
        csp is the csp to show
        fontsize is the size of the text
        speed is the number of animations per second (controls delay_time)
        1 (slow) and 4 (fast) seem like good values
        """
```
self.fontsize = fontsize
self.delay_time = 1/speed
Con_solver.__init__(self, csp, **kwargs)
csp.show(showAutoAC = True)

def go(self):
    res = self.solve_all()
    self.csp.draw_graph(domains=self.domains,
                        title="No more solutions. GUI finished. ",
                        fontsize=self.fontsize)
    return res

def select_arc(self, to_do):
    while True:
        self.csp.draw_graph(domains=self.domains, to_do=to_do,
                            title="click on to_do (blue) arc",
                            fontsize=self.fontsize)
        while self.csp.picked == None and not self.csp.autoAC:
            plt.pause(0.01) # controls reaction time of GUI
            if self.csp.autoAC:
                break
            picked = self.csp.picked
            self.csp.picked = None
            if picked in to_do:
                to_do.remove(picked)
                print(f"{picked} picked")
                return picked
            else:
                print(f"{picked} not in to_do")
        if self.csp.autoAC:
            self.csp.draw_graph(domains=self.domains, to_do=to_do,
                                title="Auto AC", fontsize=self.fontsize)
            plt.pause(self.delay_time)
            return to_do.pop()

def select_var(self, iter_vars):
    vars = list(iter_vars)
    while True:
        self.csp.draw_graph(domains=self.domains,
                            title="Arc consistent. Click node to split",
                            fontsize=self.fontsize)
        while self.csp.picked == None:
            plt.pause(0.01) # controls reaction time of GUI
            picked = self.csp.picked
            self.csp.picked = None
            self.csp.autoAC = False
            if picked in vars:
                #print("splitting",picked)
4.4. Consistency Algorithms

```python
    return picked
else:
    print(picked,"not in",vars)

def display(self,n,*args,**nargs):
    if n <= self.max_display_level: # default display
        print(*args, **nargs)
    if n==1: # solution found or no solutions"
        self.csp.draw_graph(domains=self.domains, to_do=set(),
            title= ''.join(args)+": click any node or
            arc to continue",
            fontsize=self.fontsize)
        self.csp.autoAC = False
        while self.csp.picked == None and not self.csp.autoAC:
            plt.pause(0.01) # controls reaction time of GUI
            self.csp.picked = None
    elif n==2: # backtracking
        plt.title("backtracking: click any node or arc to continue")
        self.csp.autoAC = False
        while self.csp.picked == None and not self.csp.autoAC:
            plt.pause(0.01)
            self.csp.picked = None
    elif n==3: # inconsistent arc
        line = self.csp.thelines[self.arc_selected]
        line.set_color('#red')
        line.set_linewidth(10)
        plt.pause(self.delay_time)
        line.set_color('#limegreen')
        line.set_linewidth(self.csp.linewidth)
        #elif n==4 and self.add_to_do: # adding to to_do
        #    print("adding to to_do",self.add_to_do) ## highlight these arc

import cspExamples
# Try:
# ConsistencyGUI(cspExamples.csp1).go()
# ConsistencyGUI(cspExamples.csp3).go()
# ConsistencyGUI(cspExamples.csp3, speed=4, fontsize=15).go()
```

4.4.3 Domain Splitting as an interface to graph searching

An alternative implementation is to implement domain splitting in terms of
the search abstraction of Chapter\[3\]

A node is a dictionary that maps the variables to their (pruned) domains.
4. Reasoning with Constraints

A node is a CSP ""

def __init__(self, csp):
    self.cons = Con_solver(csp) #copy of the CSP
    self.domains = self.cons.make_arc_consistent()

def is_goal(self, node):
    """node is a goal if all domains have 1 element""
    return all(len(node[var])==1 for var in node)

def start_node(self):
    return self.domains

def neighbors(self,node):
    """returns the neighboring nodes of node."
    """
    neighs = []
    var = select(x for x in node if len(node[x])>1)
    if var:
        dom1, dom2 = partition_domain(node[var])
        self.display(2,"Splitting", var, "into", dom1, "and", dom2)
        to_do = self.cons.new_to_do(var,None)
        for dom in [dom1,dom2]:
            newdoms = node | {var:dom}
            cons_doms = self.cons.make_arc_consistent(newdoms,to_do)
            if all(len(cons_doms[v])>0 for v in cons_doms):
                # all domains are non-empty
                neighs.append(Arc(node,cons_doms))
            else:
                self.display(2,"...",var,"in",dom,"has no solution")
    return neighs

Exercise 4.12 When splitting a domain, this code splits the domain into half, approximately in half (without any effort to make a sensible choice). Does it work better to split one element from a domain?

Unit test:

from searchGeneric import Searcher

def ac_search_solver(csp):
    """arc consistency (search interface)""
    sol = Searcher(Search_with_AC_from_CSP(csp)).search()
    if sol:
        return {v:select(d) for (v,d) in sol.end().items()}

if __name__ == "__main__":
cspExamples.test_csp(ac_search_solver)

Testing:

https://aipython.org
4.5 Solving CSPs using Stochastic Local Search

To run the demo, in folder “aipython”, load “cspSLS.py”, and copy and paste the commented-out example queries at the bottom of that file. This assumes Python 3. Some of the queries require matplotlib.

The following code implements the two-stage choice (select one of the variables that are involved in the most constraints that are violated, then a value), the any-conflict algorithm (select a variable that participates in a violated constraint) and a random choice of variable, as well as a probabilistic mix of the three.

Given a CSP, the stochastic local searcher (SLSearcher) creates the data structures:

- variables_to_select is the set of all of the variables with domain-size greater than one. For a variable not in this set, we cannot pick another value from that variable.

- var_to_constraints maps from a variable into the set of constraints it is involved in. Note that the inverse mapping from constraints into variables is part of the definition of a constraint.
A node is a variable:value dictionary

```python
def __init__(self, csp):
    self.csp = csp
    self.variables_to_select = {var for var in self.csp.variables  
                                if len(var.domain) > 1}
    # Create assignment and conflicts set
    self.current_assignment = None  # this will trigger a random restart
    self.number_of_steps = 0  # number of steps after the initialization
```

`restart` creates a new total assignment, and constructs the set of conflicts (the constraints that are false in this assignment).

```python
def restart(self):
    """creates a new total assignment and the conflict set
    """
    self.current_assignment = {var: random_choice(var.domain) for  
                               var in self.csp.variables}
    self.display(2, "Initial assignment", self.current_assignment)
    self.conflicts = set()
    for con in self.csp.constraints:
        if not con.holds(self.current_assignment):
            self.conflicts.add(con)
    self.display(2, "Number of conflicts", len(self.conflicts))
    self.variable_pq = None
```

The `search` method is the top-level searching algorithm. It can either be used to start the search or to continue searching. If there is no current assignment, it must create one. Note that, when counting steps, a restart is counted as one step, which is not appropriate for CSPs with many variables, as it is a relatively expensive operation for these cases.

This method selects one of two implementations. The argument `prob_best` is the probability of selecting a best variable (one involving the most conflicts). When the value of `prob_best` is positive, the algorithm needs to maintain a priority queue of variables and the number of conflicts (using `search_with_var_pq`). If the probability of selecting a best variable is zero, it does not need to maintain this priority queue (as implemented in `search_with_any_conflict`).

The argument `prob_anycon` is the probability that the any-conflict strategy is used (which selects a variable at random that is in a conflict), assuming that it is not picking a best variable. Note that for the probability parameters, any value less that zero acts like probability zero and any value greater than 1 acts like probability 1. This means that when `prob_anycon = 1.0`, a best variable is chosen with probability `prob_best`, otherwise a variable in any conflict is chosen. A variable is chosen at random with probability `1 – prob_anycon – prob_best` as long as that is positive.

This returns the number of steps needed to find a solution, or `None` if no solution is found. If there is a solution, it is in `self.current_assignment`.https://aipython.org
def search(self, max_steps, prob_best=0, prob_anycon=1.0):
    """
    returns the number of steps or None if these is no solution.
    If there is a solution, it can be found in self.current_assignment
    max_steps is the maximum number of steps it will try before giving
    up
    prob_best is the probability that a best variable (one in most
    conflict) is selected
    prob_anycon is the probability that a variable in any conflict is
    selected
    (otherwise a variable is chosen at random)
    """
    if self.current_assignment is None:
        self.restart()
        self.number_of_steps += 1
        if not self.conflicts:
            self.display(1, "Solution found:", self.current_assignment,
                         "after restart")
            return self.number_of_steps
    if prob_best > 0: # we need to maintain a variable priority queue
        return self.search_with_var_pq(max_steps, prob_best,
                                        prob_anycon)
    else:
        return self.search_with_any_conflict(max_steps, prob_anycon)

Exercise 4.13 This does an initial random assignment but does not do any ran-
dom restarts. Implement a searcher that takes in the maximum number of walk
steps (corresponding to existing max_steps) and the maximum number of restarts,
and returns the total number of steps for the first solution found. (As in search, the
solution found can be extracted from the variable self.current_assignment).

4.5.1 Any-conflict

In the any-conflict heuristic a variable that participates in a violated constraint
is picked at random. The implementation need to keeps track of which vari-
ables are in conflicts. This is can avoid the need for a priority queue that is
needed when the probability of picking a best variable is greter than zero.
if random.random() < prob_anycon:
    con = random_choice(self.conflicts)  # pick random conflict
    var = random_choice(con.scope)      # pick variable in conflict
else:
    var = random_choice(self.variables_to_select)
if len(var.domain) > 1:
    val = random_choice([val for val in var.domain
                          if val is not self.current_assignment[var]])
    self.display(2,self.number_of_steps,"Assigning",var,"=",val)
    self.current_assignment[var]=val
    for varcon in self.csp.var_to_const[var]:
        if varcon.holds(self.current_assignment):
            if varcon in self.conflicts:
                self.conflicts.remove(varcon)
            else:
                if varcon not in self.conflicts:
                    self.conflicts.add(varcon)
        self.display(2," Number of conflicts",len(self.conflicts))
if not self.conflicts:
    self.display(1,"Solution found:", self.current_assignment,
                 "in", self.number_of_steps,"steps")
    return self.number_of_steps
self.display(1,"No solution in",self.number_of_steps,"steps",
             len(self.conflicts),"conflicts remain")
return None

Exercise 4.14 This makes no attempt to find the best value for the variable selected. Modify the code to include an option selects a value for the selected variable that reduces the number of conflicts the most. Have a parameter that specifies the probability that the best value is chosen, and otherwise chooses a value at random.

4.5.2 Two-Stage Choice

This is the top-level searching algorithm that maintains a priority queue of variables ordered by the number of conflicts, so that the variable with the most conflicts is selected first. If there is no current priority queue of variables, one is created.

The main complexity here is to maintain the priority queue. When a variable var is assigned a value val, for each constraint that has become satisfied or unsatisfied, each variable involved in the constraint need to have its count updated. The change is recorded in the dictionary var.differential, which is used to update the priority queue (see Section 4.5.3).

```python
cspSLS.py — (continued)
def search_with_var_pq(self,max_steps, prob_best=1.0, prob_anycon=1.0):
    """search with a priority queue of variables.

https://aipython.org  
Version 0.9.12  
December 22, 2023"""
This is used to select a variable with the most conflicts.

```python
if not self.variable_pq:
    self.create_pq()

pick_best_or_con = prob_best + prob_anycon
for i in range(max_steps):
    self.number_of_steps += 1
    randnum = random.random()
    # Pick a variable
    if randnum < prob_best: # pick best variable
        var, oldval = self.variable_pq.top()
    elif randnum < pick_best_or_con: # pick a variable in a conflict
        con = random_choice(self.conflicts)
        var = random_choice(con.scope)
    else: # pick any variable that can be selected
        var = random_choice(self.variables_to_select)
    if len(var.domain) > 1: # var has other values
        # Pick a value
        val = random_choice([val for val in var.domain if val is not
                              self.current_assignment[var]])
        self.display(2, "Assigning", var, val)
        # Update the priority queue
        var_differential = {}
        self.current_assignment[var] = val
        for varcon in self.csp.var_to_const[var]:
            self.display(3, "Checking", varcon)
            if varcon.holds(self.current_assignment):
                if varcon in self.conflicts: # was incons, now consis
                    self.conflicts.remove(varcon)
                    for v in varcon.scope: # v is in one fewer conflicts
                        var_differential[v] =
                            var_differential.get(v, 0) - 1
                else:
                    if varcon not in self.conflicts: # was consis, not now
                        self.conflicts.add(varcon)
                        for v in varcon.scope: # v is in one more conflicts
                            var_differential[v] =
                                var_differential.get(v, 0) + 1
            self.variable_pq.update_each_priority(var_differential)
        self.display(2, "Number of conflicts", len(self.conflicts))
    if not self.conflicts: # no conflicts, so solution found
        self.display(1, "Solution found:",
                    self.current_assignment, "in",
                    self.number_of_steps, "steps")
        return self.number_of_steps
    self.display(1, "No solution in", self.number_of_steps, "steps",
```
create_pq creates an updatable priority queue of the variables, ordered by the number of conflicts they participate in. The priority queue only includes variables in conflicts and the value of a variable is the negative of the number of conflicts the variable is in. This ensures that the priority queue, which picks the minimum value, picks a variable with the most conflicts.

```python
self.variable_pq = Updatable_priority_queue()
var_to_number_conflicts = {}
for con in self.conflicts:
    for var in con.scope:
        var_to_number_conflicts[var] = var_to_number_conflicts.get(var, 0) + 1
for var, num in var_to_number_conflicts.items():
    if num > 0:
        self.variable_pq.add(var, -num)
```

Exercise 4.15 These implementations always select a value for the variable selected that is different from its current value (if that is possible). Change the code so that it does not have this restriction (so it can leave the value the same). Would you expect this code to be faster? Does it work worse (or better)?

4.5.3 Updatable Priority Queues

An updatable priority queue is a priority queue, where key-value pairs can be stored, and the pair with the smallest key can be found and removed quickly, and where the values can be updated. This implementation follows the idea of [http://docs.python.org/3.9/library/heapq.html](http://docs.python.org/3.9/library/heapq.html), where the updated elements are marked as removed. This means that the priority queue can be used unmodified. However, this might be expensive if changes are more common than popping (as might happen if the probability of choosing the best is close to zero).
In this implementation, the equal values are sorted randomly. This is achieved by having the elements of the heap being \([\text{val}, \text{rand}, \text{elt}]\) triples, where the second element is a random number. Note that Python requires this to be a list, not a tuple, as the tuple cannot be modified.

```python
class Updatable_priority_queue(object):
    """A priority queue where the values can be updated.
    Elements with the same value are ordered randomly.
    """

    This code is based on the ideas described in
    http://docs.python.org/3.3/library/heapq.html
    It could probably be done more efficiently by
    shuffling the modified element in the heap.
    ""
    def __init__(self):
        self.pq = []  # priority queue of [val,rand,elt] triples
        self.elt_map = {}  # map from elt to [val,rand,elt] triple in pq
        self.REMOVED = "*removed*
                        # a string that won't be a legal element
        self.max_size = 0

    def add(self, elt, val):
        """adds elt to the priority queue with priority=val.
        ""
        assert val <= 0, val
        assert elt not in self.elt_map, elt
        new_triple = [val, random.random(), elt]
        heapq.heappush(self.pq, new_triple)
        self.elt_map[elt] = new_triple

    def remove(self, elt):
        """remove the element from the priority queue""
        if elt in self.elt_map:
            self.elt_map[elt][2] = self.REMOVED
            del self.elt_map[elt]

    def update_each_priority(self, update_dict):
        """update values in the priority queue by subtracting the values in
        update_dict from the priority of those elements in priority queue.
        ""
        for elt, incr in update_dict.items():
            if incr != 0:
                newval = self.elt_map.get(elt, [0])[0] - incr
                assert newval <= 0, f""""{elt}:{newval+incr}-{incr}"
                self.remove(elt)
                if newval != 0:
                    self.add(elt, newval)

    def pop(self):
        """Removes and returns the (elt,value) pair with minimal value."""
```

[https://aipython.org](https://aipython.org)
4. Reasoning with Constraints

```python
If the priority queue is empty, IndexError is raised.

""
self.max_size = max(self.max_size, len(self.pq)) # keep statistics
triple = heapq.heappop(self.pq)
while triple[2] == self.REMOVED:
    triple = heapq.heappop(self.pq)
self.elt_map[triple[2]]
return triple[2], triple[0] # elt, value

def top(self):
    """Returns the (elt,value) pair with minimal value, without
    removing it.
    If the priority queue is empty, IndexError is raised.
    ""
    self.max_size = max(self.max_size, len(self.pq)) # keep statistics
    triple = self.pq[0]
    while triple[2] == self.REMOVED:
        heapq.heappop(self.pq)
    triple = self.pq[0]
    return triple[2], triple[0] # elt, value

def empty(self):
    """returns True iff the priority queue is empty"
    return all(triple[2] == self.REMOVED for triple in self.pq)
```

4.5.4 Plotting Run-Time Distributions

`Runtime_distribution` uses matplotlib to plot run time distributions. Here the run time is a misnomer as we are only plotting the number of steps, not the time. Computing the run time is non-trivial as many of the runs have a very short run time. To compute the time accurately would require running the same code, with the same random seed, multiple times to get a good estimate of the run time. This is left as an exercise.

```python
import matplotlib.pyplot as plt
# plt.style.use('grayscale')

class Runtime_distribution(object):
    def __init__(self, csp, xscale='log'):
        """Sets up plotting for csp
        xscale is either 'linear' or 'log'
        ""
        self.csp = csp
        plt.ion()
        plt.xlabel("Number of Steps")
        plt.ylabel("Cumulative Number of Runs")
        plt.xscale(xscale) # Makes a 'log' or 'linear' scale
```

[https://aipython.org](https://aipython.org) Version 0.9.12 December 22, 2023
4.5. Solving CSPs using Stochastic Local Search

Figure 4.9 gives run-time distributions for 3 algorithms. It is also useful to compare the distributions of different runs of the same algorithms and settings.
4.5.5 Testing

```python
import cspExamples

def sls_solver(csp, prob_best=0.7):
    """stochastic local searcher (prob_best=0.7)""
    se0 = SLSearcher(csp)
    se0.search(1000, prob_best)
    return se0.current_assignment

def any_conflict_solver(csp):
    """stochastic local searcher (any-conflict)""
    return sls_solver(csp, 0)

if __name__ == '__main__':
    cspExamples.test_csp(sls_solver)
    cspExamples.test_csp(any_conflict_solver)

## Test Solving CSPs with Search:
# se1 = SLSearcher(cspExamples.csp1); print(se1.search(100))
# se2 = SLSearcher(cspExamples.csp2); print(se2.search(1000, 1.0)) # greedy
# se2 = SLSearcher(cspExamples.csp2); print(se2.search(1000, 0)) # any_conflict
# se2 = SLSearcher(cspExamples.csp2); print(se2.search(1000, 0.7)) # 70% greedy; 30% any_conflict
# SLSearcher.max_display_level=2 # more detailed display
# se3 = SLSearcher(cspExamples.crossword1); print(se3.search(100), 0.7)
# p = Runtime_distribution(cspExamples.csp2)
# p.plot_runs(1000, 1000, 0) # any_conflict
# p.plot_runs(1000, 1000, 1.0) # greedy
# p.plot_runs(1000, 1000, 0.7) # 70% greedy; 30% any_conflict
```

**Exercise 4.16** Modify this to plot the run time, instead of the number of steps. To measure run time use `timeit` (https://docs.python.org/3.9/library/timeit.html). Small run times are inaccurate, so `timeit` can run the same code multiple times. Stochastic local algorithms give different run times each time called. To make the timing meaningful, you need to make sure the random seed is the same for each repeated call (see `random.getstate` and `random.setstate` in https://docs.python.org/3.9/library/random.html). Because the run time for different seeds can vary a great deal, for each seed, you should start with 1 iteration and multiplying it by, say 10, until the time is greater than 0.2 seconds. Make sure you plot the average time for each run. Before you start, try to estimate the total run time, so you will be able to tell if there is a problem with the algorithm stopping.

4.6 Discrete Optimization

A SoftConstraint is a constraint, but where the condition is a real-valued function. Because the definition of the constraint class did not force the condition to be Boolean, you can use the Constraint class for soft constraints too.
4.6. Discrete Optimization

```python
from cspProblem import Variable, Constraint, CSP

class SoftConstraint(Constraint):
    """A Constraint consists of
    * scope: a tuple of variables
    * function: a real-valued function that can applied to a tuple of values
    * string: a string for printing the constraints. All of the strings
      must be unique.
    for the variables
    """
    def __init__(self, scope, function, string=None, position=None):
        Constraint.__init__(self, scope, function, string, position)

    def value(self, assignment):
        return self.holds(assignment)

A = Variable('A', {1,2}, position=(0.2,0.9))
B = Variable('B', {1,2,3}, position=(0.8,0.9))
C = Variable('C', {1,2}, position=(0.5,0.5))
D = Variable('D', {1,2}, position=(0.8,0.1))

def c1fun(a,b):
    if a==1: return 5 if b==1 else 2
    else: return 0 if b==1 else 4 if b==2 else 3

A = Variable('A', {1,2}, position=(0.2,0.9))
B = Variable('B', {1,2,3}, position=(0.8,0.9))
C = Variable('C', {1,2}, position=(0.5,0.5))
D = Variable('D', {1,2}, position=(0.8,0.1))

def c1fun(a,b):
    if a==1: return 5 if b==1 else 2
    else: return 0 if b==1 else 4 if b==2 else 3

c1 = SoftConstraint([A,B],c1fun,"c1")

def c2fun(b,c):
    if b==1: return 5 if c==1 else 2
    elif b==2: return 0 if c==1 else 4
    else: return 2 if c==1 else 0

c2 = SoftConstraint([B,C],c2fun,"c2")

def c3fun(b,d):
    if b==1: return 3 if d==1 else 0
    elif b==2: return 2
    else: return 2 if d==1 else 4

c3 = SoftConstraint([B,D],c3fun,"c3")

def penalty_if_same(pen):
    """returns a function that gives a penalty of pen if the arguments are
    the same"
    return lambda x,y: (pen if (x==y) else 0)

c4 = SoftConstraint([C,A],penalty_if_same(3),"c4")

scsp1 = CSP("scsp1", {A,B,C,D}, [c1,c2,c3,c4])

### The second soft CSP has an extra variable, and 2 constraints
E = Variable('E', {1,2}, position=(0.1,0.1))

c5 = SoftConstraint([C,E],penalty_if_same(3),"c5")
```
4. Reasoning with Constraints

4.6.1 Branch-and-bound Search

Here we specialize the branch-and-bound algorithm (Section 3.3 on page 64) to solve soft CSP problems.

```python
import math

class DF_branch_and_bound_opt(Displayable):
    """returns a branch and bound searcher for a problem.
    An optimal assignment with cost less than bound can be found by calling
    search()
    """
    def __init__(self, csp, bound=math.inf):
        """creates a searcher than can be used with search() to find an
        optimal path.
        bound gives the initial bound. By default this is infinite -
        meaning there
        is no initial pruning due to depth bound
        """
        super().__init__()
        self.csp = csp
        self.best_asst = None
        self.bound = bound

    def optimize(self):
        """returns an optimal solution to a problem with cost less than
        bound.
        returns None if there is no solution with cost less than bound."""
        self.num_expanded=0
        self.cbsearch({}, 0, self.csp.constraints)
        self.display(1,"Number of paths expanded:",self.num_expanded)
        return self.best_asst, self.bound

    def cbsearch(self, asst, cost, constraints):
        """finds the optimal solution that extends path and is less the
        bound"
        self.display(2,"cbsearch:",asst,cost,constraints)
        can_eval = [c for c in constraints if c.can_evaluate(asst)]
        rem_cons = [c for c in constraints if c not in can_eval]
        newcost = cost + sum(c.value(asst) for c in can_eval)
        self.display(2,"Evaluating:",can_eval,"cost:",newcost)
        if newcost < self.bound:
            self.num_expanded += 1
            if rem_cons==[]:
                self.best_asst = asst
```

https://aipython.org

Version 0.9.12 December 22, 2023
4.6. Discrete Optimization

self.bound = newcost
self.display(1,"New best assignment:",asst," cost:",newcost)

else:
    var = next(var for var in self.csp.variables if var not in asst)
    for val in var.domain:
        self.cbsearch({var:val}|asst, newcost, rem_cons)

# bnb = DF_branch_and_bound_opt(scsp)
# bnb.max_display_level=3 # show more detail
# bnb.optimize()

Exercise 4.17  Change the stochastic-local search algorithms to work for soft constraints. Hint: The analog of a conflict is a soft constraint that is not at its lowest value. Instead of the number of constraints violated, consider how much a change in a variable affects the objective function. Instead of returning a solution, return the best assignment found.
Propositions and Inference

5.1 Representing Knowledge Bases

A clause consists of a head (an atom) and a body. A body is represented as a list of atoms. Atoms are represented as strings.

```python
class Clause(object):
    '''A definite clause'''
    def __init__(self, head, body=[]):
        '''clause with atom head and lost of atoms body'''
        self.head = head
        self.body = body

    def __repr__(self):
        '''returns the string representation of a clause.
        '''
        if self.body:
            return f"{self.head} <- {'.join(str(a) for a in self.body)}."
        else:
            return f"{self.head}".
```

An askable atom can be asked of the user. The user can respond in English or French or just with a "y".

```python
class Askable(object):
    '''An askable atom'''
    def __init__(self, atom):
```

```
5. Propositions and Inference

A knowledge base is a list of clauses and askables. In order to make top-down inference faster, this creates a dictionary that maps each atom into the set of clauses with that atom in the head.

```python
from display import Displayable
class KB(Displayable):
    """A knowledge base consists of a set of clauses. This also creates a dictionary to give fast access to the clauses with an atom in head."
    ""
    def __init__(self, statements=[]):
        self.statements = statements
        self.clauses = [c for c in statements if isinstance(c, Clause)]
        self.askables = [c.atom for c in statements if isinstance(c, Askable)]
        self.atom_to_clauses = {}  # dictionary giving clauses with atom as head
        for c in self.clauses:
            self.add_clause(c)

    def add_clause(self, c):
        if c.head in self.atom_to_clauses:
            self.atom_to_clauses[c.head].append(c)
        else:
            self.atom_to_clauses[c.head] = [c]

    def clauses_for_atom(self, a):
        """returns list of clauses with atom a as the head""
        if a in self.atom_to_clauses:
            return self.atom_to_clauses[a]
        else:
            return []

    def __str__(self):
        """returns a string representation of this knowledge base."
        ""
        return '\n'.join([str(c) for c in self.statements])
```

https://aipython.org
5.1. Representing Knowledge Bases

Here is a trivial example (I think therefore I am) used in the unit tests:

```
triv_KB = KB([  
    Clause('i_am', ['i_think']),  
    Clause('i_think'),  
    Clause('i_smell', ['i_exist'])  
])
```

Here is a representation of the electrical domain of the textbook:

```
elect = KB([  
    Clause('light_l1'),  
    Clause('light_l2'),  
    Clause('ok_l1'),  
    Clause('ok_l2'),  
    Clause('ok_cb1'),  
    Clause('ok_cb2'),  
    Clause('live_outside'),  
    Clause('live_l1', ['live_w0']),  
    Clause('live_w0', ['up_s2','live_w1']),  
    Clause('live_w0', ['down_s2','live_w2']),  
    Clause('live_w1', ['up_s1','live_w3']),  
    Clause('live_w2', ['down_s1','live_w3'] ),  
    Clause('live_l2', ['live_w4']),  
    Clause('live_w4', ['up_s3','live_w3'] ),  
    Clause('live_p_1', ['live_w3']),  
    Clause('live_w3', ['live_w5', 'ok_cb1']),  
    Clause('live_p_2', ['live_w6']),  
    Clause('live_w6', ['live_w5', 'ok_cb2']),  
    Clause('live_w5', ['live_outside']),  
    Clause('lit_l1', ['light_l1', 'live_l1', 'ok_l1']),  
    Clause('lit_l2', ['light_l2', 'live_l2', 'ok_l2']),  
    Askable('up_s1'),  
    Askable('down_s1'),  
    Askable('up_s2'),  
    Askable('down_s2'),  
    Askable('up_s3'),  
    Askable('down_s2')  
])
```

The following knowledge base is false in the intended interpretation. One of the clauses is wrong; can you see which one? We will show how to debug it.

```
elect_bug = KB([  
    Clause('light_l2'),  
    Clause('ok_l1'),  
    Clause('ok_l2'),  
])
```
Clause('ok_cb1'),
Clause('ok_cb2'),
Clause('live_outside'),
Clause('live_p_2', ['live_w6']),
Clause('live_w6', ['live_w5', 'ok_cb2']),
Clause('light_l1'),
Clause('live_w5', ['live_outside']),
Clause('lit_l1', ['light_l1', 'live_l1', 'ok_l1']),
Clause('lit_l2', ['light_l2', 'live_l2', 'ok_l2']),
Clause('live_l1', ['live_w0']),
Clause('live_w0', ['up_s2', 'live_w1']),
Clause('live_w0', ['down_s2', 'live_w2']),
Clause('live_w1', ['up_s3', 'live_w3']),
Clause('live_w2', ['down_s1', 'live_w3']),
Clause('live_2', ['live_w4']),
Clause('live_w4', ['up_s3', 'live_w3']),
Clause('live_p_1', ['live_w3']),
Clause('live_w3', ['live_w5', 'ok_cb1']),
Askable('up_s1'),
Askable('up_s2'),
Clause('light_l2'),
Clause('ok_l1'),
Clause('light_l2'),
Clause('ok_l1'),
Clause('ok_cb1'),
Clause('ok_cb2'),
Clause('live_outside'),
Clause('live_p_2', ['live_w6']),
Clause('live_w6', ['live_w5', 'ok_cb2']),
Clause('ok_l2'),
Clause('ok_cb1'),
Clause('ok_cb2'),
Clause('live_outside'),
Clause('live_p_2', ['live_w6']),
Clause('live_w6', ['live_w5', 'ok_cb2']),
Askable('down_s2'),
Askable('up_s3'),
Askable('down_s2'))

# print(kb)

## 5.2 Bottom-up Proofs (with askables)

*fixed_point* computes the fixed point of the knowledge base *kb*.
from logicProblem import yes

def fixed_point(kb):
    """Returns the fixed point of knowledge base kb."
    ""
    fp = ask_askables(kb)
    added = True
    while added:
        added = False # added is true when an atom was added to fp this iteration
        for c in kb.clauses:
            if c.head not in fp and all(b in fp for b in c.body):
                fp.add(c.head)
                added = True
                kb.display(2,c.head,"added to fp due to clause",c)
    return fp

def ask_askables(kb):
    return {at for at in kb.askables if yes(input("Is " + at + " true? "))}

The following provides a trivial unittest, by default using the knowledge base triv_KB:

```python
from logicProblem import triv_KB
def test(kb=triv_KB, fixedpt = {'i_am','i_think'}):
    fp = fixed_point(kb)
    assert fp == fixedpt, "kb gave result {fp}"
    print("Passed unit test")
if __name__ == "__main__":
    test()
from logicProblem import elect
# elect.max_display_level=3 # give detailed trace
# fixed_point(elect)
```

**Exercise 5.1** It is not very user-friendly to ask all of the askables up-front. Implement ask-the-user so that questions are only asked if useful, and are not re-asked. For example, if there is a clause \( h \leftarrow a \land b \land c \land d \land e \), where \( c \) and \( e \) are askable, \( c \) and \( e \) only need to be asked if \( a, b, d \) are all in \( fp \) and they have not been asked before. Askable \( e \) only needs to be asked if the user says "yes" to \( c \). Askable \( c \) doesn't need to be asked if the user previously replied "no" to \( e \).

This form of ask-the-user can ask a different set of questions than the top-down interpreter that asks questions when encountered. Give an example where they ask different questions (neither set of questions asked is a subset of the other).

**Exercise 5.2** This algorithm runs in time \( O(n^2) \), where \( n \) is the number of clauses, for a bounded number of elements in the body; each iteration goes through each of the clauses, and in the worst case, it will do an iteration for each clause. It is possible to implement this in time \( O(n) \) time by creating an index that maps an atom to the set of clauses with that atom in the body. Implement this. What is its
complexity as a function of $n$ and $b$, the maximum number of atoms in the body of a clause?

**Exercise 5.3** It is possible to be asymptotically more efficient (in terms of the number of elements in a body) than the method in the previous question by noticing that each element of the body of clause only needs to be checked once. For example, the clause $a \leftarrow b \land c \land d$, needs only be considered when $b$ is added to $fp$. Once $b$ is added to $fp$, if $c$ is already in $fp$, we know that $a$ can be added as soon as $d$ is added. Implement this. What is its complexity as a function of $n$ and $b$, the maximum number of atoms in the body of a clause?

### 5.3 Top-down Proofs (with askables)

The following implements the top-down proof procedure for propositional definite clauses, as described in Section 5.3.2 and Figure 5.4 of ?.

> **prove(kb, goal)** is used to prove goal from a knowledge base, $kb$, where a goal is a list of atoms. It returns True if $kb \vdash goal$. The indent is used when displaying the code (and doesn’t need to be called initially with a non-default value).

```python
def prove(kb, ans_body, indent=""):    
    """returns True if kb |- ans_body
    ans_body is a list of atoms to be proved"
    
    kb.display(2,indent,'yes <-', ' & '.join(ans_body))
    if ans_body:
        selected = ans_body[0] # select first atom from ans_body
        if selected in kb.askables:
            return (yes(input("Is " + selected+" true? "))
                and prove(kb,ans_body[1:],indent+" "))
        else:
            return any(prove(kb,cl.body+ans_body[1:],indent+" ")
                for cl in kb.clauses_for_atom(selected))
    else:
        return True # empty body is true
```

The following provides a simple unit test that is hard wired for triv_KB:

```python
from logicProblem import yes

def prove(kb, ans_body, indent=""):    
    """returns True if kb |- ans_body
    ans_body is a list of atoms to be proved"
    
    kb.display(2,indent,'yes <-', ' & '.join(ans_body))
    if ans_body:
        selected = ans_body[0] # select first atom from ans_body
        if selected in kb.askables:
            return (yes(input("Is " + selected+" true? "))
                and prove(kb,ans_body[1:],indent+" "))
        else:
            return any(prove(kb,cl.body+ans_body[1:],indent+" ")
                for cl in kb.clauses_for_atom(selected))
    else:
        return True # empty body is true

def test():
    a1 = prove(triv_KB,['i_am'])
    assert a1, f"triv_KB proving i_am gave {a1}"
    a2 = prove(triv_KB,['i_smell'])
    assert not a2, f"triv_KB proving i_smell gave {a2}"```
5.4. Debugging and Explanation

Here we modify the top-down procedure to build a proof tree than can be traversed for explanation and debugging.

prove_atom(kb, atom) returns a proof for atom from a knowledge base kb, where a proof is a pair of the atom and the proofs for the elements of the body of the clause used to prove the atom. prove_body(kb, body) returns a list of proofs for list body from a knowledge base, kb. The indent is used when displaying the code (and doesn’t need to have a non-default value).

Exercise 5.4 This code can re-ask a question multiple times. Implement this code so that it only asks a question once and remembers the answer. Also implement a function to forget the answers.

Exercise 5.5 What search method is this using? Implement the search interface so that it can use \( A^* \) or other searching methods. Define an admissible heuristic that is not always 0.

```python
print("Passed unit tests")
if __name__ == "__main__":
    test()
    # try
from logicProblem import elect
# elect.max_display_level=3 # give detailed trace
# prove(elect,['live_w6'])
# prove(elect,['lit_l1'])
```

```python
Exercise 5.4 This code can re-ask a question multiple times. Implement this code so that it only asks a question once and remembers the answer. Also implement a function to forget the answers.

Exercise 5.5 What search method is this using? Implement the search interface so that it can use \( A^* \) or other searching methods. Define an admissible heuristic that is not always 0.

5.4 Debugging and Explanation

Here we modify the top-down procedure to build a proof tree than can be traversed for explanation and debugging.

prove_atom(kb, atom) returns a proof for atom from a knowledge base kb, where a proof is a pair of the atom and the proofs for the elements of the body of the clause used to prove the atom. prove_body(kb, body) returns a list of proofs for list body from a knowledge base, kb. The indent is used when displaying the code (and doesn’t need to have a non-default value).

```python
from logicProblem import yes # for asking the user
def prove_atom(kb, atom, indent=""):
    """returns a pair (atom,proofs) where proofs is the list of proofs
    of the elements of a body of a clause used to prove atom.
    """
    kb.display(2,indent,'proving',atom)
    if atom in kb.askables:
        if yes(input("Is "+atom+" true? ")):
            return (atom,"answered")
        else:
            return "fail"
    else:
        for cl in kb.clauses_for_atom(atom):
            kb.display(2,indent,"trying",atom,'<-'," ".join(cl.body))
            pr_body = prove_body(kb, cl.body, indent)
            if pr_body != "fail":
                return (atom, pr_body)
            return "fail"
```
5. Propositions and Inference

returns proof tree if \( \text{kb} \vdash \text{ans\_body} \) or "fail" if there is no proof

\( \text{ans\_body} \) is a list of atoms in a body to be proved

```
proofs = []
for atom in ans_body:
    proof_at = prove_atom(kb, atom, indent+"")
    if proof_at == "fail":
        return "fail" # fail if any proof fails
else:
    proofs.append(proof_at)
return proofs
```

The following provides a simple **unit test** that is hard wired for \( \text{triv\_KB} \):

```
from logicProblem import triv_KB

def test():
    a1 = prove_atom(triv_KB,'i_am')
    assert a1, f"triv_KB proving i_am gave {a1}"
    a2 = prove_atom(triv_KB,'i_smell')
    assert a2="fail", "triv_KB proving i_smell gave {a2}"
    print("Passed unit tests")

if __name__ == "__main__":
    test()

# try
from logicProblem import elect, elect_bug
# elect.max_display_level=3 # give detailed trace
# prove_atom(elect, 'live_w6')
# prove_atom(elect, 'lit_l1')
```

The interact(kb) provides an interactive interface to explore proofs for knowledge base kb. The user can ask to prove atoms and can ask how an atom was proved.

To ask how, there must be a current atom for which there is a proof. This starts as the atom asked. When the user asks "how n" the current atom becomes the n-th element of the body of the clause used to prove the (previous) current atom. The command "up" makes the current atom the atom in the head of the rule containing the (previous) current atom. Thus "how n" moves down the proof tree and "up" moves up the proof tree, allowing the user to explore the full proof.

```
helptext = """Commands are:
  ask atom  ask is there is a proof for atom (atom should not be in quotes)
  how       show the clause that was used to prove atom
  how n     show the clause used to prove the nth element of the body
  up         go back up proof tree to explore other parts of the proof tree
  kb         print the knowledge base
```

[https://aipython.org](https://aipython.org) Version 0.9.12 December 22, 2023
def interact(kb):
    going = True
    ups = []  # stack for going up
    proof = "fail"  # there is no proof to start
    while going:
        inp = input("logicExplain: ")
        inps = inp.split(" ")
        try:
            command = inps[0]
            if command == "quit":
                going = False
            elif command == "ask":
                proof = prove_atom(kb, inps[1])
                if proof == "fail":
                    print("fail")
                else:
                    print("yes")
            elif command == "how":
                if proof == "fail":
                    print("there is no proof")
                elif len(inps) == 1:
                    print_rule(proof)
                else:
                    try:
                        ups.append(proof)
                        proof = proof[1][int(inps[1])]  # nth argument of rule
                        print_rule(proof)
                    except:
                        print("In "how n", n must be a number between 0
                        and", len(proof[1])-1,"inclusive.")
            elif command == "up":
                if ups:
                    proof = ups.pop()
                else:
                    print("No rule to go up to.")
                    print_rule(proof)
            elif command == "kb":
                print(kb)
            elif command == "help":
                print(helptext)
            else:
                print("unknown command: ", inp)
                print("use help for help")
        except:
            print("unknown command: ", inp)
            print("use help for help")
```python
def print_rule(proof):
    (head, body) = proof
    if body == "answered":
        print(head, "was answered yes")
    elif body == []:
        print(head, "is a fact")
    else:
        print(head, "<-")
        for i, a in enumerate(body):
            print(i, ":", a[0])

# try
# interact(elect)
# Which clause is wrong in elect_bug? Try:
# interact(elect_bug)
# logicExplain: ask lit_l1

The following shows an interaction for the knowledge base elect:

```
```
5.5. Assumables

Exercise 5.6 The above code only ever explores one proof – the first proof found. Change the code to enumerate the proof trees (by returning a list of all proof trees, or, preferably, using `yield`). Add the command “retry” to the user interface to try another proof.

5.5 Assumables

Atom $a$ can be made assumable by including `Assumable(a)` in the knowledge base. A knowledge base that can include assumables is declared with `KBA`.

```python
from logicProblem import Clause, Askable, KB, yes

class Assumable(object):
    """An askable atom""
    
    def __init__(self, atom):
        """clause with atom head and lost of atoms body""
        self.atom = atom
    
    def __str__(self):
        """returns the string representation of a clause."
        return "assumable " + self.atom + "."

class KBA(KB):
    """A knowledge base that can include assumables""
    
    def __init__(self, statements):
        self.assumables = [c.atom for c in statements if isinstance(c, Assumable)]
        KB.__init__(self, statements)

The top-down Horn clause interpreter, `prove_all_ass` returns a list of the sets of assumables that imply `ans_body`. This list will contain all of the minimal sets of assumables, but can also find non-minimal sets, and repeated sets, if they can be generated with separate proofs. The set `assumed` is the set of assumables already assumed.

```
selected = ans_body[0] # select first atom from ans_body
if selected in self.askables:
    if yes(input("Is \"+selected+\" true? ")):
        return self.prove_all_ass(ans_body[1:], assumed)
    else:
        return [] # no answers
elif selected in self.assumables:
    return self.prove_all_ass(ans_body[1:], assumed|\{selected\})
else:
    return [ass
    for cl in self.clauses_for_atom(selected)
    for ass in self.prove_all_ass(cl.body+ans_body[1:], assumed)
    ] # union of answers for each clause with
    head=selected
else:
    return [assumed] # one answer

def conflicts(self):
    """returns a list of minimal conflicts""
    return minsets(self.prove_all_ass([\'false\']))

Given a list of sets, minsets returns a list of the minimal sets in the list. For example, minsets([\{2, 3, 4\}, \{2, 3\}, \{6, 2, 3\}, \{2, 3\}, \{2, 4, 5\}]) returns [[2, 3], [2, 4, 5]].

```
def minsets(ls):
    """ls is a list of sets
    returns a list of minimal sets in ls
    ""
    ans = [] # elements known to be minimal
    for c in ls:
        if not any(cl<=c for cl in ls) and not any(cl <= c for c in ans):
            ans.append(c)
    return ans

# minsets([\{2, 3, 4\}, \{2, 3\}, \{6, 2, 3\}, \{2, 3\}, \{2, 4, 5\}])
```

Warning: minsets works for a list of sets or for a set of (frozen) sets, but it does not work for a generator of sets (because ls is referenced in the loop). For example, try to predict and then test:

minsets(e for e in [[2, 3, 4], [2, 3], [6, 2, 3], [2, 3], [2, 4, 5]])

The diagnoses can be constructed from the (minimal) conflicts as follows. This also works if there are non-minimal conflicts, but is not as efficient.

```
def diagnoses(cons):
    """cons is a list of (minimal) conflicts.
    returns a list of diagnoses."
    if cons == []:
```
5.5. Assumables

```python
return [set()]
else:
    return minsets([[e]|d])  # | is set union
    for e in cons[0]
    for d in diagnoses(cons[1:])
```

Test cases:

```python
electa = KBA(
    Clause('light_l1'),
    Clause('light_l2'),
    Assumable('ok_l1'),
    Assumable('ok_l2'),
    Assumable('ok_s1'),
    Assumable('ok_s2'),
    Assumable('ok_s3'),
    Assumable('ok_cb1'),
    Assumable('ok_cb2'),
    Assumable('live_outside'),
    Clause('live_l1', ['live_w0']),
    Clause('live_l0', ['up_s2', 'ok_s2', 'live_w1']),
    Clause('live_l0', ['down_s2', 'ok_s2', 'live_w2']),
    Clause('live_w1', ['up_s1', 'ok_s1', 'live_w3']),
    Clause('live_w2', ['down_s1', 'ok_s1', 'live_w3']),
    Clause('live_w1', ['live_w4']),
    Clause('live_w4', ['up_s3', 'ok_s3', 'live_w3']),
    Clause('live_w3', ['live_w5', 'ok_cb1']),
    Clause('live_w3', ['live_w5', 'ok_cb2']),
    Clause('live_w5', ['live_outside']),
    Clause('lit_l1', ['light_l1', 'live_l1', 'ok_l1']),
    Clause('lit_l2', ['light_l2', 'live_l2', 'ok_l2']),
    Askable('up_s1'),
    Askable('down_s1'),
    Askable('up_s2'),
    Askable('down_s2'),
    Askable('up_s3'),
    Askable('down_s2'),
    Askable('dark_l1'),
    Askable('dark_l2'),
    Clause('false', ['dark_l1', 'lit_l1']),
    Clause('false', ['dark_l2', 'lit_l2'])
)
# electa.prove_all_ass(['false'])
# cs=electa.conflicts()
# print(cs)
# diagnoses(cs)  # diagnoses from conflicts
```

Exercise 5.7 To implement a version of conflicts that never generates non-minimal
122

5. Propositions and Inference

conflicts, modify prove all ass to implement iterative deepening on the number of
assumables used in a proof, and prune any set of assumables that is a superset of
a conflict.

Exercise 5.8 Implement explanations(self , body), where body is a list of atoms,
that returns a list of the minimal explanations of the body. This does not require
modification of prove all ass.
Exercise 5.9 Implement explanations, as in the previous question, so that it never
generates non-minimal explanations. Hint: modify prove all ass to implement iterative deepening on the number of assumptions, generating conflicts and explanations together, and pruning as early as possible.

5.6

Negation-as-failure

The negation af an atom a is written as Not(a) in a body.
logicNegation.py — Propositional negation-as-failure
11

from logicProblem import KB, Clause, Askable, yes

12
13
14
15

class Not(object):
def __init__(self, atom):
self.theatom = atom

16
17
18

def atom(self):
return self.theatom

19
20
21

def __repr__(self):
return f"Not({self.theatom})"

Prove with negation-as-failure (prove_naf) is like prove, but with the extra case
to cover Not:
logicNegation.py — (continued)
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37

def prove_naf(kb, ans_body, indent=""):
""" prove with negation-as-failure and askables
returns True if kb |- ans_body
ans_body is a list of atoms to be proved
"""
kb.display(2,indent,'yes <-',' & '.join(str(e) for e in ans_body))
if ans_body:
selected = ans_body[0] # select first atom from ans_body
if isinstance(selected, Not):
kb.display(2,indent,f"proving {selected.atom()}")
if prove_naf(kb, [selected.atom()], indent):
kb.display(2,indent,f"{selected.atom()} succeeded so
Not({selected.atom()}) fails")
return False
else:
kb.display(2,indent,f"{selected.atom()} fails so
Not({selected.atom()}) succeeds")

https://aipython.org

Version 0.9.12

December 22, 2023


5.6. Negation-as-failure

```
return prove_naf(kb, ans_body[1:], indent + "")
if selected in kb.askables:
    return (yes(input("Is " + selected + " true? "))
             and prove_naf(kb, ans_body[1:], indent + ""))
else:
    return any(prove_naf(kb, cl.body + ans_body[1:], indent + "")
                for cl in kb.clauses_for_atom(selected))
else:
    return True  # empty body is true
```

Test cases:

```
triv_KB_naf = KB([  
    Clause('i_am', ['i_think']),  
    Clause('i_think'),  
    Clause('i_smell', ['i_am', Not('dead')]),  
    Clause('i_bad', ['i_am', Not('i_think')])  
])
triv_KB_naf.max_display_level = 4
def test():  
    a1 = prove_naf(triv_KB_naf,['i_smell'])  
    assert a1, f"triv_KB_naf proving i_smell gave {a1}"  
    a2 = prove_naf(triv_KB_naf,['i_bad'])  
    assert not a2, f"triv_KB_naf proving i_bad gave {a2}"  
    print("Passed unit tests")
    if __name__ == "__main__":
        test()
```

Default reasoning about beaches at resorts (Example 5.28 of §):

```
beach_KB = KB([  
    Clause('away_from_beach', [Not('on_beach')]),  
    Clause('beach_access', ['on_beach', Not('ab_beach_access')])),  
    Clause('swim_at_beach', ['beach_access', Not('ab_swim_at_beach')]),  
    Clause('ab_swim_at_beach', ['enclosed_bay', 'big_city',  
                               Not('ab_no_swimming_near_city')]),  
    Clause('ab_no_swimming_near_city', ['in_BC', Not('ab_Beaches')]  
])
# prove_naf(beach_KB, ['away_from_beach'])  
# prove_naf(beach_KB, ['beach_access'])  
# beach_KB.add_clause(Clause('on_beach',[]))  
# prove_naf(beach_KB, ['away_from_beach'])  
# prove_naf(beach_KB, ['swim_at_beach'])  
# beach_KB.add_clause(Clause('enclosed_bay',[]))  
# prove_naf(beach_KB, ['swim_at_beach'])  
# beach_KB.add_clause(Clause('big_city',[]))  
# prove_naf(beach_KB, ['swim_at_beach'])  
# beach_KB.add_clause(Clause('in_BC',[]))
```

https://aipython.org  Version 0.9.12  December 22, 2023
83 | # prove_naf(beach_KB, ['swim_at_beach'])
Deterministic Planning

6.1 Representing Actions and Planning Problems

The STRIPS representation of an action consists of:

- the name of the action
- preconditions: a dictionary of feature:value pairs that specifies that the feature must have this value for the action to be possible
- effects: a dictionary of feature:value pairs that are made true by this action. In particular, a feature in the dictionary has the corresponding value (and not its previous value) after the action, and a feature not in the dictionary keeps its old value.

```python
class Strips(object):
    def __init__(self, name, preconds, effects, cost=1):
        """
        defines the STRIPS representation for an action:
        * name is the name of the action
        * preconds, the preconditions, is feature:value dictionary that
          must hold for the action to be carried out
        * effects is a feature:value map that this action makes
          true. The action changes the value of any feature specified
          here, and leaves other features unchanged.
        * cost is the cost of the action
        """
```

125
6. Deterministic Planning

A STRIPS domain consists of:

- A dictionary that maps each feature into a set of possible values for the feature.
- A set of actions, each represented using the Strips class.

```python
class STRIPS_domain(object):
    def __init__(self, feature_domain_dict, actions):
        feature_domain_dict is a feature:domain dictionary, mapping each feature to its domain
        actions
        self.feature_domain_dict = feature_domain_dict
        self.actions = actions
```

A planning problem consists of a planning domain, an initial state, and a goal. The goal does not need to fully specify the final state.

```python
class Planning_problem(object):
    def __init__(self, prob_domain, initial_state, goal):
        a planning problem consists of
        * a planning domain
        * the initial state
        * a goal
        self.prob_domain = prob_domain
        self.initial_state = initial_state
        self.goal = goal
```

6.1.1 Robot Delivery Domain

The following specifies the robot delivery domain of Section 6.1, shown in Figure 6.1.

```python
boolean = {False, True}
delivery_domain = STRIPS_domain(
```

https://aipython.org  Version 0.9.12  December 22, 2023
6.1. Representing Actions and Planning Problems

### Features to describe states

- **RLoc** – Rob’s location
- **RHC** – Rob has coffee
- **SWC** – Sam wants coffee
- **MW** – Mail is waiting
- **RHM** – Rob has mail

### Actions

- **mc** – move clockwise
- **mcc** – move counterclockwise
- **puc** – pickup coffee
- **dc** – deliver coffee
- **pum** – pickup mail
- **dm** – deliver mail

#### Figure 6.1: Robot Delivery Domain

```python
stripsProblem.py — (continued)

problem0 = Planning_problem(delivery_domain,
                            {'RLoc':'lab', 'MW':True, 'SWC':True, 'RHC':False,
                             'RHM':False},
                            https://aipython.org
                            Version 0.9.12 December 22, 2023
```
6.1.2 Blocks World

The blocks world consist of blocks and a table. Each block can be on the table or on another block. A block can only have one other block on top of it. Figure 6.2 shows 3 states with some of the actions between them.

A state is defined by the two features:

- **on** where on(x) = y when block x is on block or table y
- **clear** where clear(x) = True when block x has nothing on it.

There is one parameterized action

- **move(x, y, z)** move block x from y to z, where y and z could be a block or the table.

To handle parameterized actions (which depend on the blocks involved), the actions and the features are all strings, created for all the combinations of the blocks. Note that we treat moving to a block separately from moving to the table.

```python
problem1 = Planning_problem(delivery_domain,
{'RLoc': 'lab', 'MW': True, 'SWC': True, 'RHC': False,
'RHM': False},
{'SWC': False})
problem2 = Planning_problem(delivery_domain,
{'RLoc': 'lab', 'MW': True, 'SWC': True, 'RHC': False,
'RHM': False},
{'SWC': False, 'MW': False, 'RHM': False})
```
table, because the blocks needs to be clear, but the table always has room for another block.

```python
## blocks world

def move(x,y,z):
    """string for the 'move' action""
    return 'move_+'+x+'_from_+'+y+'_to_+'+z
def on(x):
    """string for the 'on' feature""
    return x+'_is_on'
def clear(x):
    """string for the 'clear' feature""
    return 'clear_'+x
def create_blocks_world(blocks = {'a','b','c','d'}):
    blocks_and_table = blocks | {'table'}
    stmap = {Strips(move(x,y,z),{on(x):y, clear(x):True, clear(z):True},
                (on(x):z, clear(y):True, clear(z):False))
        for x in blocks
        for y in blocks_and_table
        for z in blocks
        if x!=y and y!=z and z!=x)
    stmap.update({Strips(move(x,y,'table'), {on(x):y, clear(x):True},
                   (on(x):'table', clear(y):True))
        for x in blocks
        for y in blocks
        if x!=y})
    feature_domain_dict = {on(x):blocks_and_table-{x} for x in blocks}
    feature_domain_dict.update({clear(x):boolean for x in blocks_and_table})
    return STRIPS_domain(feature_domain_dict, stmap)
```

The problem \texttt{blocks1} is a classic example, with 3 blocks, and the goal consists of two conditions. See Figure \ref{fig:blocks1}. This example is challenging because you can't achieve one of the goals and then the other; whichever one you achieve first has to be undone to achieve the second.

```python
blocks1dom = create_blocks_world({'a','b','c'})
bloc1 = Planning_problem(blocks1dom,
    {on('a'): 'table', clear('a'):True,
     on('b'): 'c', clear('b'):True,
     on('c'): 'table', clear('c'):False}, # initial state
    {on('a'): 'b', on('c'): 'a'}) # goal
```

The problem \texttt{blocks2} is one to invert a tower of size 4.

```python
blocks2dom = create_blocks_world({'a','b','c','d'})
tower4 = {clear('a'):True, on('a'): 'b',
          clear('b'):False, on('b'): 'c',
          clear('c'):False, on('c'): 'd',
```

\url{https://aipython.org}
6. Deterministic Planning

The problem blocks2 is to move the bottom block to the top of a tower of size 4.

**Exercise 6.1** Represent the problem of given a tower of 4 blocks (a on b on c on d on table), the goal is to have a tower with the previous top block on the bottom (b on c on d on a). Do not include the table in your goal (the goal does not care whether a is on the table). [Before you run the program, estimate how many steps it will take to solve this.] How many steps does an optimal planner take?

**Exercise 6.2** Represent the domain so that on(x,y) is a Boolean feature that is True when x is on y. Does the representation of the state need to include negative on facts? Why or why not? (Note that this may depend on the planner; write your answer with respect to particular planners.)

**Exercise 6.3** It is possible to write the representation of the problem without using clear, where clear(x) means nothing is on x. Change the definition of the blocks world so that it does not use clear but uses on being false instead. Does this work better for any of the planners?

### 6.2 Forward Planning

To run the demo, in folder "aipython", load "stripsForwardPlanner.py", and copy and paste the commented-out example queries at the bottom of that file.

In a forward planner, a node is a state. A state consists of an assignment, which is a variable:value dictionary. In order to be able to do multiple-path pruning, we need to define a hash function, and equality between states.
6.2. Forward Planning

stripsForwardPlanner.py — Forward Planner with STRIPS actions

from searchProblem import Arc, Search_problem
from stripsProblem import Strips, STRIPS_domain

class State(object):
    def __init__(self, assignment):
        self.assignment = assignment
        self.hash_value = None
    def __hash__(self):
        if self.hash_value is None:
            self.hash_value = hash(frozenset(self.assignment.items()))
        return self.hash_value
    def __eq__(self, st):
        return self.assignment == st.assignment
    def __str__(self):
        return str(self.assignment)

In order to define a search problem (page 41), we need to define the goal condition, the start nodes, the neighbours, and (optionally) a heuristic function. Here zero is the default heuristic function.

def zero(*args,**nargs):
    """always returns 0""
    return 0

class Forward_STRIPS(Search_problem):
    """A search problem from a planning problem where:
    * a node is a state object.
    * the dynamics are specified by the STRIPS representation of actions
    """
    def __init__(self, planning_problem, heur=zero):
        """creates a forward search space from a planning problem.
        heur(state,goal) is a heuristic function,
        an underestimate of the cost from state to goal, where
        both state and goals are feature:value dictionaries.
        """
        self.prob_domain = planning_problem.prob_domain
        self.initial_state = State(planning_problem.initial_state)
        self.goal = planning_problem.goal
        self.heur = heur

    def is_goal(self, state):
        """is True if node is a goal."
        return all(state.assignment[prop]==self.goal[prop]
                    for prop in self.goal)

    def start_node(self):
        """returns start node"""
return self.initial_state

def neighbors(self, state):
    """returns neighbors of state in this problem""
    return [Arc(state, self.effect(act, state.assignment), act.cost,
              act)
              for act in self.prob_domain.actions
              if self.possible(act, state.assignment)]

def possible(self, act, state_asst):
    """True if act is possible in state.
    act is possible if all of its preconditions have the same value in
    the state""
    return all(state_asst[pre] == act.preconds[pre]
              for pre in act.preconds)

def effect(self, act, state_asst):
    """returns the state that is the effect of doing act given
    state_asst
    Python 3.9: return state_asst | act.effects"
    new_state_asst = state_asst.copy()
    new_state_asst.update(act.effects)
    return State(new_state_asst)

def heuristic(self, state):
    """in the forward planner a node is a state.
    the heuristic is an (under)estimate of the cost
    of going from the state to the top-level goal.
    ""
    return self.heur(state.assignment, self.goal)

Here are some test cases to try.

from searchBranchAndBound import DF_branch_and_bound
from searchMPP import SearcherMPP
import stripsProblem

# SearcherMPP(Forward_STRIPS(stripsProblem.problem1)).search() #A* with MPP
# DF_branch_and_bound(Forward_STRIPS(stripsProblem.problem1), 10).search() #B&B
# To find more than one plan:
# s1 = SearcherMPP(Forward_STRIPS(stripsProblem.problem1)) #A*
# s1.search() #find another plan

6.2.1 Defining Heuristics for a Planner

Each planning domain requires its own heuristics. If you change the actions,
you will need to reconsider the heuristic function, as there might then be a
lower-cost path, which might make the heuristic non-admissible.
6.2. Forward Planning

Here is an example of defining heuristics for the coffee delivery planning domain.

First we define the distance between two locations, which is used for the heuristics.

```python
def dist(loc1, loc2):
    """returns the distance from location loc1 to loc2
    ""
    if loc1==loc2:
        return 0
    if {loc1,loc2} in [{'cs','lab'},{'mr','off'}]:
        return 2
    else:
        return 1
```

Note that the current state is a complete description; there is a value for every feature. However the goal need not be complete; it does not need to define a value for every feature. Before checking the value for a feature in the goal, a heuristic needs to define whether the feature is defined in the goal.

```python
def h1(state,goal):
    """ the distance to the goal location, if there is one"
    if 'RLoc' in goal:
        return dist(state['RLoc'], goal['RLoc'])
    else:
        return 0

def h2(state,goal):
    """ the distance to the coffee shop plus getting coffee and delivering it
    if the robot needs to get coffee
    ""
    if ('SWC' in goal and goal['SWC']==False
        and state['SWC']==True
        and state['RHC']==False):
        return dist(state['RLoc'],'cs')+3
    else:
        return 0
```

The maximum of the values of a set of admissible heuristics is also an admissible heuristic. The function `maxh` takes a number of heuristic functions as arguments, and returns a new heuristic function that takes the maximum of the values of the heuristics. For example, `h1` and `h2` are heuristic functions and so `maxh(h1, h2)` is also. `maxh` can take an arbitrary number of arguments.

```python
def maxh(*heuristics):
    """Returns a new heuristic function that is the maximum of the functions in heuristics.
```
heuristics is the list of arguments which must be heuristic functions.

```python
# return lambda state,goal: max(h(state,goal) for h in heuristics)
def newh(state,goal):
    return max(h(state,goal) for h in heuristics)
return newh
```

The following runs the example with and without the heuristic.

```python
from searchMPP import SearcherMPP
from stripsForwardPlanner import Forward_STRIPS
import stripsProblem

def test_forward_heuristic(thisproblem=stripsProblem.problem1):
    print("\n***** FORWARD NO HEURISTIC")
    print(SearcherMPP(Forward_STRIPS(thisproblem)).search())
    print("\n***** FORWARD WITH HEURISTIC h1")
    print(SearcherMPP(Forward_STRIPS(thisproblem,h1)).search())
    print("\n***** FORWARD WITH HEURISTIC h2")
    print(SearcherMPP(Forward_STRIPS(thisproblem,h2)).search())
    print("\n***** FORWARD WITH HEURISTICs h1 and h2")
    print(SearcherMPP(Forward_STRIPS(thisproblem,maxh(h1,h2))).search())

if __name__ == "__main__":
    test_forward_heuristic()
```

**Exercise 6.4** For more than one start-state/goal combination, test the forward planner with a heuristic function of just $h_1$, with just $h_2$ and with both. Explain why each one prunes or doesn’t prune the search space.

**Exercise 6.5** Create a better heuristic than $\max(h_1, h_2)$. Try it for a number of different problems. In particular, try and include the following costs:

i) $h_3$ is like $h_2$ but also takes into account the case when $Rloc$ is in goal.

ii) $h_4$ uses the distance to the mail room plus getting mail and delivering it if the robot needs to get need to deliver mail.

iii) $h_5$ is for getting mail when goal is for the robot to have mail, and then getting to the goal destination (if there is one).

**Exercise 6.6** Create an admissible heuristic for the blocks world.
6.3 Regression Planning

To run the demo, in folder "aipython", load "stripsRegressionPlanner.py", and copy and paste the commented-out example queries at the bottom of that file.

In a regression planner a node is a subgoal that need to be achieved.

A Subgoal object consists of an assignment, which is a variable:value dictionary. We make it hashable so that multiple path pruning can work. The hash is only computed when necessary (and only once).

```python
from searchProblem import Arc, Search_problem

class Subgoal(object):
    def __init__(self, assignment):
        self.assignment = assignment
        self.hash_value = None
    def __hash__(self):
        if self.hash_value is None:
            self.hash_value = hash(frozenset(self.assignment.items()))
        return self.hash_value
    def __eq__(self, st):
        return self.assignment == st.assignment
    def __str__(self):
        return str(self.assignment)

A regression search has subgoals as nodes. The initial node is the top-level goal of the planner. The goal for the search (when the search can stop) is a subgoal that holds in the initial state.

```python
from stripsForwardPlanner import zero

class Regression_STRIPS(Search_problem):
    """A search problem where:
    * a node is a goal to be achieved, represented by a set of propositions.
    * the dynamics are specified by the STRIPS representation of actions
    """
    def __init__(self, planning_problem, heur=zero):
        """creates a regression search space from a planning problem.
        heur(state,goal) is a heuristic function;
        an underestimate of the cost from state to goal, where
        both state and goals are feature:value dictionaries
        """
        self.prob_domain = planning_problem.prob_domain
        self.top_goal = Subgoal(planning_problem.goal)
        self.initial_state = planning_problem.initial_state
        self.heur = heur
```

https://aipython.org  Version 0.9.12  December 22, 2023
def is_goal(self, subgoal):
    """if subgoal is true in the initial state, a path has been found""
    goal_asst = subgoal.assignment
    return all(self.initial_state[g]==goal_asst[g]
        for g in goal_asst)

def start_node(self):
    """the start node is the top-level goal""
    return self.top_goal

def neighbors(self, subgoal):
    """returns a list of the arcs for the neighbors of subgoal in this
    problem""
    goal_asst = subgoal.assignment
    return [ Arc(subgoal, self.weakest_precond(act,goal_asst),
        act.cost, act)
        for act in self.prob_domain.actions
        if self.possible(act,goal_asst)]

def possible(self, act, goal_asst):
    """True if act is possible to achieve goal_asst.
    the action achieves an element of the effects and
    the action doesn't delete something that needs to be achieved and
    the preconditions are consistent with other subgoals that need to
    be achieved
    """
    return (any(goal_asst[prop] == act.effects[prop]
        for prop in act.effects if prop in goal_asst)
        and all(goal_asst[prop] == act.effects[prop]
            for prop in act.effects if prop in goal_asst)
        and all(goal_asst[prop]== act.preconds[prop]
            for prop in act.preconds if prop not in act.effects
                and prop in goal_asst)
            )

def weakest_precond(self, act, goal_asst):
    """returns the subgoal that must be true so goal_asst holds after
    act
    should be: act.preconds | (goal_asst - act.effects)
    """
    new_asst = act.preconds.copy()
    for g in goal_asst:
        if g not in act.effects:
            new_asst[g] = goal_asst[g]
    return Subgoal(new_asst)

def heuristic(self, subgoal):
    """in the regression planner a node is a subgoal.
the heuristic is an (under)estimate of the cost of going from the
initial state to subgoal.

```python
return self.heur(self.initial_state, subgoal.assignment)
```

Exercise 6.7 Multiple path pruning could be used to prune more than the current
node. In particular, if the current node contains more conditions than a previously
visited node, it can be pruned. For example, if \{a: True, b: False\} has been visited,
then any node that is a superset, e.g., \{a: True, b: False, d: True\}, need not be
expanded. If the simpler subgoal does not lead to a solution, the more complicated
one will not either. Implement this more severe pruning. (Hint: This may require
modifications to the searcher.)

Exercise 6.8 It is possible that, as knowledge of the domain, that some as-
signment of values to variables can never be achieved. For example, the robot
cannot be holding mail when there is mail waiting (assuming it isn’t holding
mail initially). An assignment of values to (some of the) variables is incompat-
able if no possible (reachable) state can include that assignment. For example,
\{'MW': True, 'RHM': True\} is an incompatible assignment. This information may
be useful information for a planner; there is no point in trying to achieve these
together. Define a subclass of STRIPS_domain that can accept a list of incompatible
assignments. Modify the regression planner code to use such a list of incompatible
assignments. Give an example where the search space is smaller.

Exercise 6.9 After completing the previous exercise, design incompatible assign-
ments for the blocks world. (This should result in dramatic search improvements.)

6.3.1 Defining Heuristics for a Regression Planner

The regression planner can use the same heuristic function as the forward plan-
ner. However, just because a heuristic is useful for a forward planner does not
mean it is useful for a regression planner, and vice versa. You should exper-
iment with whether the same heuristic works well for both a regression planner
and a forward planner.

The following runs the same example as the forward planner with and
without the heuristic defined for the forward planner:
6.4 Planning as a CSP

Exercise 6.10 Try the regression planner with a heuristic function of just \( h_1 \) and with just \( h_2 \) (defined in Section 6.2.1). Explain how each one prunes or doesn’t prune the search space.

Exercise 6.11 Create a better heuristic than \textit{heuristic fun} defined in Section 6.2.1.

### 6.4 Planning as a CSP

To run the demo, in folder “aipython”, load ”stripsCSPPlanner.py”, and copy and paste the commented-out example queries at the bottom of that file. This assumes Python 3.

Here we implement the CSP planner assuming there is a single action at each step. This creates a CSP that can use any of the CSP algorithms to solve (e.g., stochastic local search or arc consistency with domain splitting).

This assumes the same action representation as before; we do not consider factored actions (action features), nor do we implement state constraints.
The following methods return methods which can be applied to the particular environment.

For example, $\text{is}(3)$ returns a function that when applied to 3, returns True

```python
feat_time_var = {feat: [Variable(f"\{feat\}_{t}\", dom)
    for t in range(number_stages+1)]
    for (feat, dom) in
    prob_domain.feature_domain_dict.items()}

# initial state constraints:
constraints = [Constraint((feat_time_var[feat][0],), is_(val))
    for (feat, val) in initial_state.items()]

# goal constraints on the final state:
constraints += [Constraint((feat_time_var[feat][number_stages],),
    is_(val))
    for (feat, val) in goal.items()]

# precondition constraints:
constraints += [Constraint((feat_time_var[feat][t],
    self.action_vars[t]),
    if_(val, act))  # feat@t==val if action@t==act
    for act in prob_domain.actions
    for (feat, val) in act.preconds.items()
    for t in range(number_stages)]

# effect constraints:
constraints += [Constraint((feat_time_var[feat][t+1],
    self.action_vars[t]),
    if_(val, act))  # feat@t+1==val if action@t==act
    for act in prob_domain.actions
    for feat, val in act.effects.items()
    for t in range(number_stages)]

# frame constraints:
constraints += [Constraint((feat_time_var[feat][t],
    self.action_vars[t], feat_time_var[feat][t+1]),
    eq_if_not_in_((act for act in
    prob_domain.actions
    if feat in act.effects))
    for feat in prob_domain.feature_domain_dict
    for t in range(number_stages) ]

variables = set(self.action_vars) | {feat_time_var[feat][t]
    for feat in
    prob_domain.feature_domain_dict
    for t in range(number_stages+1)}

CSP.__init__(self, "CSP_from_Strips", variables, constraints)

def extract_plan(self, soln):
    return [soln[a] for a in self.action_vars]
```
and when applied to any other value returns False. So is_(3)(3) returns True and is_(3)(7) returns False.

Note that the underscore (‘.’) is part of the name; here we use it as the convention that it is a function that returns a function. This uses two different styles to define is_ and if_; returning a function defined by lambda is equivalent to returning the embedded function, except that the embedded function has a name. The embedded function can also be given a docstring.

```python
# stripsCSPPlanner.py — (continued)

def is_(val):
    """returns a function that is true when it is applied to val."
    
    #return lambda x: x == val
    def is_fun(x):
        return x == val
    is_fun.__name__ = f'value_is_{val}"
    return is_fun

    def if_(v1,v2):
        """if the second argument is v2, the first argument must be v1"
        
        #return lambda x1,x2: x1==v1 if x2==v2 else True
        def if_fun(x1,x2):
            return x1==v1 if x2==v2 else True
        if_fun.__name__ = f"if x2 is {v2} then x1 is {v1}"
        return if_fun

    def eq_if_not_in_(actset):
        """first and third arguments are equal if action is not in actset"
        
        # return lambda x1, a, x2: x1==x2 if a not in actset else True
        def eq_if_not_fun(x1, a, x2):
            return x1==x2 if a not in actset else True
        eq_if_not_fun.__name__ = f"first and third arguments are equal if action is not in {actset}"
        return eq_if_not_fun

Putting it together, this returns a list of actions that solves the problem prob for a given horizon. If you want to do more than just return the list of actions, you might want to get it to return the solution. Or even enumerate the solutions (by using Search_with_AC_from_CSP).

```python
# stripsCSPPlanner.py — (continued)

def con_plan(prob,horizon):
    """finds a plan for problem prob given horizon."
    
    csp = CSP_from_STRIPSPS(prob, horizon)
    sol = Con_solver(csp).solve_one()
    return csp.extract_plan(sol) if sol else sol

The following are some example queries.
```
from searchGeneric import Searcher
from cspConsistency import Search_with_AC_from_CSP, Con_solver
from stripsProblem import Planning_problem
import stripsProblem

# Problem 0
# con_plan(stripsProblem.problem0, 1) # should it succeed?
# con_plan(stripsProblem.problem0, 2) # should it succeed?
# con_plan(stripsProblem.problem0, 3) # should it succeed?
# To use search to enumerate solutions
#searcher0a = Searcher(Search_with_AC_from_CSP(CSP_from_STRIPS(stripsProblem.problem0, 1)))
#print(searcher0a.search()) # returns path to solution

## Problem 1
# con_plan(stripsProblem.problem1, 5) # should it succeed?
# con_plan(stripsProblem.problem1, 4) # should it succeed?
## To use search to enumerate solutions:
#searcher15a = Searcher(Search_with_AC_from_CSP(CSP_from_STRIPS(stripsProblem.problem1, 5)))
#print(searcher15a.search()) # returns path to solution

## Problem 2
# con_plan(stripsProblem.problem2, 6) # should fail??
# con_plan(stripsProblem.problem2, 7) # should succeed???

## Example 6.13
problem3 = Planning_problem(stripsProblem.delivery_domain,
  {'SWC':True, 'RHC':False}, {'SWC':False})
# con_plan(problem3, 2) # Horizon of 2
# con_plan(problem3, 3) # Horizon of 3

problem4 = Planning_problem(stripsProblem.delivery_domain, {'SWC':True},
  {'SWC':False, 'MW':False, 'RHM':False})

# For the stochastic local search:
# from cspSLS import SLSearcher, Runtime_distribution
# cspplanning15 = CSP_from_STRIPS(stripsProblem.problem1, 5) # should succeed
# se0 = SLSearcher(cspplanning15); print(se0.search(100000,0.5))
# p = Runtime_distribution(cspplanning15)
# p.plot_runs(1000,1000,0.7) # warning will take a few minutes
6.5 Partial-Order Planning

To run the demo, in folder "aipython", load "stripsPOP.py", and copy and paste the commented-out example queries at the bottom of that file.

A partial order planner maintains a partial order of action instances. An action instance consists of a name and an index. We need action instances because the same action could be carried out at different times.

```python
from searchProblem import Arc, Search_problem
import random

class Action_instance(object):
    next_index = 0
    def __init__(self, action, index=None):
        if index is None:
            index = Action_instance.next_index
            Action_instance.next_index += 1
        self.action = action
        self.index = index
        
    def __str__(self):
        return f"{self.action}#{self.index}"

    __repr__ = __str__  # __repr__ function is the same as the __str__ function
```

A node (as in the abstraction of search space) in a partial-order planner consists of:

- **actions**: a set of action instances.

- **constraints**: a set of \((a_1, a_2)\) pairs, where \(a_1\) and \(a_2\) are action instances, which represents that \(a_1\) must come before \(a_2\) in the partial order. There are a number of ways that this could be represented. Here we represent the set of pairs that are in transitive closure of the before relation. This lets us quickly determine whether some before relation is consistent with the current constraints.

- **agenda**: a list of \((s, a)\) pairs, where \(s\) is a \((\text{var}, \text{val})\) pair and \(a\) is an action instance. This means that variable \text{var} must have value \text{val} before \(a\) can occur.

- **causal links**: a set of \((a_0, g, a_1)\) triples, where \(a_1\) and \(a_2\) are action instances and \(g\) is a \((\text{var}, \text{val})\) pair. This holds when action \(a_0\) makes \(g\) true for action \(a_1\).
6.5. Partial-Order Planning

```python
class POP_node(object):
    """a (partial) partial-order plan. This is a node in the search space."""
    def __init__(self, actions, constraints, agenda, causal_links):
        """* actions is a set of action instances
* constraints a set of (a0,a1) pairs, representing a0<a1, closed under transitivity
* agenda list of (subgoal,action) pairs to be achieved, where subgoal is a (variable,value) pair
* causal_links is a set of (a0,g,a1) triples, where ai are action instances, and g is a (variable,value) pair
"""
        self.actions = actions # a set of action instances
        self.constraints = constraints # a set of (a0,a1) pairs
        self.agenda = agenda # list of (subgoal,action) pairs to be achieved
        self.causal_links = causal_links # set of (a0,g,a1) triples

    def __str__(self):
        return ('actions: ' + str({str(a) for a in self.actions}) +
        'constraints: ' +
        str({(str(a1),str(a2)) for (a1,a2) in self.constraints}) +
        'agenda: ' +
        str([((str(s),str(a)) for (s,a) in self.agenda]) +
        'causal_links: ' +
        str({(str(a0),str(g),str(a2)) for (a0,g,a2) in self.causal_links} )

extract_plan constructs a total order of action instances that is consistent with the partial order.

```
The `neighbors` method is a coroutine that enumerates the neighbors of a given node.

```python
from display import Displayable

class POP_search_from_STRIPS(Search_problem, Displayable):
    def __init__(self, planning_problem):
        Search_problem.__init__(self)
        self.planning_problem = planning_problem
        self.start = Action_instance("start")
        self.finish = Action_instance("finish")

    def is_goal(self, node):
        return node.agenda == []

    def start_node(self):
        constraints = {(self.start, self.finish)}
        agenda = [(g, self.finish) for g in self.planning_problem.goal.items()]
        return POP_node([self.start, self.finish], constraints, agenda, [])

    def neighbors(self, node):
        """enumerates the neighbors of node""
        self.display(3, "finding neighbors of\n", node)
        if node.agenda:
            subgoal, act1 = node.agenda[0]
            self.display(2, "selecting", subgoal, "for", act1)
            new_agenda = node.agenda[1:]
            for act0 in node.actions:
                if (self.achieves(act0, subgoal) and
                    self.possible((act0, act1), node.constraints)):
                    self.display(2, "reusing", act0)
                    consts1 =
                        self.add_constraint((act0, act1), node.constraints)
                    new_clink = (act0, subgoal, act1)
                    new_cls = node.causal_links + [new_clink]
                    for consts2 in
                        self.protect_cl_for_actions(node.actions, consts1, new_clink):
                            yield Arc(node,
                                POP_node(node.actions, consts2, new_agenda, new_cls),
                                cost=0)
            for a0 in self.planning_problem.prob_domain.actions:  # a0 is an action
                if self.achieves(a0, subgoal):
                    # a0 achieves subgoal
                    new_a = Action_instance(a0)
                    self.display(2, "using new action", new_a)
                    new_actions = node.actions + [new_a]
```
Given a causal link \((a_0, \text{subgoal}, a_1)\), the following method protects the causal link from each action in \(\text{actions}\). Whenever an action deletes \(\text{subgoal}\), the action needs to be before \(a_0\) or after \(a_1\). This method enumerates all constraints that result from protecting the causal link from all actions.

```python
def protect_cl_for_actions(self, actions, constrs, clink):
    """yields constraints that extend constrs and protect causal link \((a_0, \text{subgoal}, a_1)\)
    for each action in actions
    """
    if actions:
        a = actions[0]
        rem_actions = actions[1:]
        a0, subgoal, a1 = clink
        if a != a0 and a != a1 and self.deletes(a, subgoal):
            if self.possible((a,a0),constrs):
                new_const = self.add_constraint((a,a0),constrs)
                for e in self.protect_cl_for_actions(rem_actions,new_const,clink):
                    yield e # could be "yield from"
            if self.possible((a1,a),constrs):
                new_const = self.add_constraint((a1,a),constrs)
                for e in self.protect_cl_for_actions(rem_actions,new_const,clink):
                    yield e
        else:
            for e in self.protect_cl_for_actions(rem_actions,constrs,clink):
                yield e
    else:
        yield constrs
```

Given an action \(act\), the following method protects all the causal links in \(\text{clinks}\) from \(act\). Whenever \(act\) deletes \(\text{subgoal}\) from some causal link \((a_0, \text{subgoal}, a_1)\),
the action \( act \) needs to be before \( a_0 \) or after \( a_1 \). This method enumerates all constraints that result from protecting the causal links from \( act \).

```python
def protect_all_cls(self, clinks, act, constrs):
    """yields constraints that protect all causal links from act""
    if clinks:
        (a0,cond,a1) = clinks[0] # select a causal link
        rem_clinks = clinks[1:] # remaining causal links
        if act != a0 and act != a1 and self.deletes(act,cond):
            if self.possible((act,a0),constrs):
                new_const = self.add_constraint((act,a0),constrs)
                for e in self.protect_all_cls(rem_clinks,act,new_const):
                    yield e
            if self.possible((a1,act),constrs):
                new_const = self.add_constraint((a1,act),constrs)
                for e in self.protect_all_cls(rem_clinks,act,new_const):
                    yield e
        else:
            for e in self.protect_all_cls(rem_clinks,act,constrs): yield e
    else:
        yield constrs
```

The following methods check whether an action (or action instance) achieves or deletes some subgoal.

```python
def achieves(self,action,subgoal):
    var,val = subgoal
    return var in self.effects(action) and self.effects(action)[var] == val

def deletes(self,action,subgoal):
    var,val = subgoal
    return var in self.effects(action) and self.effects(action)[var] != val

def effects(self,action):
    """returns the variable:value dictionary of the effects of action.
    works for both actions and action instances""
    if isinstance(action, Action_instance):
        action = action.action
    if action == "start":
        return self.planning_problem.initial_state
    elif action == "finish":
        return {}
    else:
        return action.effects
```

The constraints are represented as a set of pairs closed under transitivity. Thus if \((a,b)\) and \((b,c)\) are the list, then \((a,c)\) must also be in the list. This means
that adding a new constraint means adding the implied pairs, but querying whether some order is consistent is quick.

```python
def add_constraint(self, pair, const):
    if pair in const:
        return const
    todo = [pair]
    newconst = const.copy()
    while todo:
        x0, x1 = todo.pop()
        newconst.add((x0, x1))
        for x, y in newconst:
            if x == x1 and (x0, y) not in newconst:
                todo.append((x0, y))
            if y == x0 and (x, x1) not in newconst:
                todo.append((x, x1))
    return newconst

def possible(self, pair, constraint):
    (x, y) = pair
    return (y, x) not in constraint
```

Some code for testing:

```python
from searchBranchAndBound import DF_branch_and_bound
from searchMPP import SearcherMPP
import stripsProblem

rplanning0 = POP_search_from_STRIPS(stripsProblem.problem0)
rplanning1 = POP_search_from_STRIPS(stripsProblem.problem1)
rplanning2 = POP_search_from_STRIPS(stripsProblem.problem2)
searcher0 = DF_branch_and_bound(rplanning0, 5)
searcher0a = SearcherMPP(rplanning0)
searcher1 = DF_branch_and_bound(rplanning1, 10)
searcher1a = SearcherMPP(rplanning1)
searcher2 = DF_branch_and_bound(rplanning2, 10)
searcher2a = SearcherMPP(rplanning2)

# Try one of the following searchers
# a = searcher0.search()
# a = searcher0a.search()
# a.end().extract_plan() # print a plan found
# a.end().constraints # print the constraints
# SearcherMPP.max_display_level = 0 # less detailed display
# DF_branch_and_bound.max_display_level = 0 # less detailed display
# a = searcher1.search()
# a = searcher1a.search()
# a = searcher2.search()
# a = searcher2a.search()
```

https://aipython.org

Version 0.9.12
December 22, 2023
Supervised Machine Learning

This chapter is the first on machine learning. It covers the following topics:

- Data: how to load it, training and test sets
- Features: many of the features come directly from the data. Sometimes it is useful to construct features, e.g. \( \text{height} > 1.9m \) might be a Boolean feature constructed from the real-values feature \( \text{height} \). The next chapter is about neural networks and how to learn features; in this chapter we construct them explicitly in what is often known as \textit{feature engineering}.
- Learning with no input features: this is the base case of many methods. What should we predict if we have no input features? This provides the base cases for many algorithms (e.g., decision tree algorithm) and baselines that more sophisticated algorithms need to beat. It also provides ways to test various predictors.
- Decision tree learning: one of the classic and simplest learning algorithms, which is the basis of many other algorithms.
- Cross validation and parameter tuning: methods to prevent overfitting.
- Linear regression and classification: other classic and simple techniques that often work well (particularly combined with feature learning or engineering).
- Boosting: combining simpler learning methods to make even better learners.

A good source of classic datasets is the UCI Machine Learning Repository [?][?]. The SPECT, IRIS, and car datasets (carbool is a Boolean version of the car dataset) are from this repository.
7. Supervised Machine Learning

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Examples</th>
<th>#Columns</th>
<th>Input Types</th>
<th>Target Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECT</td>
<td>267</td>
<td>23</td>
<td>Boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>IRIS</td>
<td>150</td>
<td>5</td>
<td>numeric</td>
<td>categorical</td>
</tr>
<tr>
<td>carbool</td>
<td>1728</td>
<td>7</td>
<td>categorical/numeric</td>
<td>numeric</td>
</tr>
<tr>
<td>holiday</td>
<td>32</td>
<td>6</td>
<td>Boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>mail_reading</td>
<td>28</td>
<td>5</td>
<td>Boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>tv_likes</td>
<td>12</td>
<td>5</td>
<td>Boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>simp_regr</td>
<td>7</td>
<td>2</td>
<td>numeric</td>
<td>numeric</td>
</tr>
</tbody>
</table>

Figure 7.1: Some of the datasets used here.

7.1 Representations of Data and Predictions

The code uses the following definitions and conventions:

- A **dataset** is an enumeration of examples.

- An **example** is a list (or tuple) of values. The values can be numbers or strings.

- A **feature** is a function from examples into the range of the feature. Each feature \( f \) also has the following attributes:
  
  \( f.\text{ftype} \), the type of \( f \), one of: "boolean", "categorical", "numeric"

  \( f.\text{frange} \), the set of values of \( f \) seen in the dataset, represented as a list.

  The \text{ftype} is inferred from the \text{frange} if not given explicitly.

  \( f.\text{__doc__} \), the docstring, a string description of \( f \) (for printing).

Thus for example, a **Boolean feature** is a function from the examples into \{False, True\}. So, if \( f \) is a Boolean feature, \( f.\text{frange} == [\text{False, True}] \), and if \( e \) is an example, \( f(e) \) is either True or False.

```python
import math, random, statistics
import csv
from display import Displayable
from utilities import argmax

boolean = [False, True]
```

When creating a dataset, we partition the data into a training set (\textit{train}) and a test set (\textit{test}). The target feature is the feature that we are making a prediction of. A dataset \( ds \) has the following attributes:

\( ds.\text{train} \) a list of the training examples

\( ds.\text{test} \) a list of the test examples

https://aipython.org
ds.target_index the index of the target

ds.target the feature corresponding to the target (a function as described above)

ds.input_features a list of the input features

```python
class Data_set(Displayable):
    """ A dataset consists of a list of training data and a list of test data. """

    def __init__(self, train, test=None, prob_test=0.20, target_index=0, header=None, target_type=None, seed=None):
        """A dataset for learning.
        train is a list of tuples representing the training examples
        test is the list of tuples representing the test examples
        if test is None, a test set is created by selecting each
        example with probability prob_test
        target_index is the index of the target.
        If negative, it counts from right.
        If target_index is larger than the number of properties,
        there is no target (for unsupervised learning)
        header is a list of names for the features
        target_type is either None for automatic detection of target type
        or one of "numeric", "boolean", "categorical"
        seed is for random number; None gives a different test set each time
        """
        if seed: # given seed makes partition consistent from run-to-run
            random.seed(seed)
        if test is None:
            train,test = partition_data(train, prob_test)
        self.train = train
        self.test = test
        self.prob_test = prob_test
        self.num_properties = len(self.train[0])
        if target_index < 0: #allows for -1, -2, etc.
            self.target_index = self.num_properties + target_index
        else:
            self.target_index = target_index
        self.header = header
        self.domains = [set() for i in range(self.num_properties)]
        for example in self.train:
            for ind,val in enumerate(example):
```

https://aipython.org
A feature is a function that takes an example and returns a value in the range of the feature. Each feature has a \textit{frange}, which gives the range of the feature, and an \textit{ftype} that gives the type, one of "boolean", "numeric" or "categorical".

We try to infer the type of each feature. Sometimes this can be wrong, (e.g., when the numbers are really categorical) and may need to be set explicitly.

```
def infer_type(self,domain):
    """Infers the type of a feature with domain
    ""
    if all(v in (True,False) for v in domain):
        return "boolean"
```

https://aipython.org  Version 0.9.12  December 22, 2023
if all(isinstance(v,(float,int)) for v in domain):
    return "numeric"
else:
    return "categorical"

7.1.1 Creating Boolean Conditions from Features

Some of the algorithms require Boolean input features or features with range \{0,1\}. In order to be able to use these algorithms on datasets that allow for arbitrary domains of input variables, we construct Boolean conditions from the attributes.

There are 3 cases:

- When the range only has two values, we designate one to be the “true” value.

- When the values are all numeric, we assume they are ordered (as opposed to just being some classes that happen to be labelled with numbers) and construct Boolean features for splits of the data. That is, the feature is $e[ind] < cut$ for some value cut. We choose a number of cut values, up to a maximum number of cuts, given by \( \text{max_num_cuts} \).

- When the values are not all numeric, we create an indicator function for each value. An indicator function for a value returns true when that value is given and false otherwise. Note that we can’t create an indicator function for values that appear in the test set but not in the training set because we haven’t seen the test set. For the examples in the test set with a value that doesn’t appear in the training set for that feature, the indicator functions all return false.

There is also an option \text{categorical\_only} to create only Boolean features for categorical input features, and not to make cuts for numerical values.

def conditions(self, max_num_cuts=8, categorical_only = False):
    """returns a set of boolean conditions from the input features
    max_num_cuts is the maximum number of cute for numeric features
    categorical\_only is true if only categorical features are made binary
    """
    if (max_num_cuts, categorical_only) in self.conditions_cache:  
        return self.conditions_cache[(max_num_cuts, categorical_only)]
	conds = []
	for ind,frange in enumerate(self.domains):
            if ind != self.target_index and len(frange)>1: 
                if len(frange) == 2:
                    # two values, the feature is equality to one of them.
                    true_val = list(frange)[1] # choose one as true
```python
def feat(e, i=ind, tv=true_val):
    return e[i]==tv
if self.header:
    feat.__doc__ = f"{self.header[ind]}=={true_val}"
else:
    feat.__doc__ = f"e[{ind}]=={true_val}"
feat.frange = boolean
feat.ftype = "boolean"
conds.append(feat)
elif all(isinstance(val,(int,float)) for val in frange):
    if categorical_only: # numeric, don't make cuts
        def feat(e, i=ind):
            return e[i]
        feat.__doc__ = f"e[{ind}]"
        conds.append(feat)
    else:
        # all numeric, create cuts of the data
        sorted_frange = sorted(frange)
        num_cuts = min(max_num_cuts,len(frange))
        cut_positions = [len(frange)*i//num_cuts for i in range(1,num_cuts)]
        for cut in cut_positions:
            cutat = sorted_frange[cut]
            def feat(e, ind_=ind, cutat=cutat):
                return e[ind_] < cutat
            if self.header:
                feat.__doc__ = self.header[ind]+"<"+str(cutat)
            else:
                feat.__doc__ = "e["+str(ind)+"]<"+str(cutat)
            feat.frange = boolean
            feat.ftype = "boolean"
            conds.append(feat)
else:
    # create an indicator function for every value
    for val in frange:
        def feat(e, ind_=ind, val_=val):
            return e[ind_] == val_
        if self.header:
            feat.__doc__ = self.header[ind]+"=="+str(val)
        else:
            feat.__doc__ = "e["+str(ind)+"]=="+str(val)
        feat.frange = boolean
        feat.ftype = "boolean"
        conds.append(feat)
self.conditions_cache[(max_num_cuts, categorical_only)] = conds
return conds
```

Exercise 7.1 Change the code so that it splits using $e[\text{ind}] \leq \text{cut}$ instead of $e[\text{ind}] < \text{cut}$. Check boundary cases, such as 3 elements with 2 cuts. As a test case, make sure that when the range is the 30 integers from 100 to 129, and you want 2 cuts,

https://aipython.org Version 0.9.12 December 22, 2023
the resulting Boolean features should be \( e[\text{ind}] \leq 109 \) and \( e[\text{ind}] \leq 119 \) to make sure that each of the resulting domains is of equal size.

**Exercise 7.2** This splits on whether the feature is less than one of the values in the training set. Sam suggested it might be better to split between the values in the training set, and suggested using

\[
\text{cutat} = (\text{sorted.frange}[\text{cut}] + \text{sorted.frange}[\text{cut} - 1])/2
\]

Why might Sam have suggested this? Does this work better? (Try it on a few datasets).

### 7.1.2 Evaluating Predictions

A **predictor** is a function that takes an example and makes a prediction on the values of the target features.

A **loss** takes a prediction and the actual value and returns a non-negative real number; lower is better. The **error** for a dataset is either the mean loss, or sometimes the sum of the losses. When reporting results the mean is usually used. When it is the sum, this will be made explicit.

The function `evaluate_dataset` returns the average error for each example, where the error for each example depends on the evaluation criteria. Here we consider three evaluation criteria, the squared error (average of the square of the difference between the actual and predicted values), absolute errors (average of the absolute difference between the actual and predicted values) and the log loss (the average negative log-likelihood, which can be interpreted as the number of bits to describe an example using a code based on the prediction treated as a probability).

```python
def evaluate_dataset(self, data, predictor, error_measure):
    
    """Evaluates predictor on data according to the error_measure
    predictor is a function that takes an example and returns a
    prediction for the target features.
    error_measure(prediction,actual) -> non-negative real
    ""

    try:
        value = statistics.mean(error_measure(predictor(e),
                                            self.target(e))
                               for e in data)
    except ValueError: # if error_measure gives an error
        return float("inf") # infinity
    return value
else:
    return math.nan # not a number
```

The following evaluation criteria are defined. This is defined using a class, Evaluate but no instances will be created. Just use Evaluate.squared_loss etc.
The prediction is either a real value or a \code{\{value : probability\}} dictionary or a list. The actual is either a real number or a key of the prediction.

```python
class Evaluate(object):
    """A container for the evaluation measures"""

    def squared_loss(prediction, actual):
        """squared loss "
        if isinstance(prediction, (list, dict)):
            return (1-prediction[actual])**2 # the correct value is 1
        else:
            return (prediction-actual)**2

    def absolute_loss(prediction, actual):
        """absolute loss "
        if isinstance(prediction, (list, dict)):
            return abs(1-prediction[actual]) # the correct value is 1
        else:
            return abs(prediction-actual)

    def log_loss(prediction, actual):
        """log loss (bits)"
        try:
            if isinstance(prediction, (list, dict)):
                return -math.log2(prediction[actual])
            else:
                return -math.log2(prediction) if actual==1 else -math.log2(1-prediction)
        except ValueError:
            return float("inf") # infinity

    def accuracy(prediction, actual):
        """accuracy "
        if isinstance(prediction, dict):
            prev_val = prediction[actual]
            return 1 if all(prev_val >= v for v in prediction.values()) else 0
        if isinstance(prediction, list):
            prev_val = prediction[actual]
            return 1 if all(prev_val >= v for v in prediction) else 0
        else:
            return 1 if abs(actual-prediction) <= 0.5 else 0

    all_criteria = [accuracy, absolute_loss, squared_loss, log_loss]
```
7.1.3 Creating Test and Training Sets

The following method partitions the data into a training set and a test set. Note that this does not guarantee that the test set will contain exactly a proportion of the data equal to \( \text{prob\_test} \).

[An alternative is to use \texttt{random.sample()} which can guarantee that the test set will contain exactly a particular proportion of the data. However this would require knowing how many elements are in the dataset, which we may not know, as \textit{data} may just be a generator of the data (e.g., when reading the data from a file).]

```python
def partition_data(data, prob_test=0.30):
    # partitions the data into a training set and a test set, where
    # prob\_test is the probability of each example being in the test set.
    train = []
    test = []
    for example in data:
        if random.random() < prob_test:
            test.append(example)
        else:
            train.append(example)
    return train, test
```

7.1.4 Importing Data From File

A dataset is typically loaded from a file. The default here is that it loaded from a CSV (comma separated values) file, although the separator can be changed. This assumes that all lines that contain the separator are valid data (so we only include those data items that contain more than one element). This allows for blank lines and comment lines that do not contain the separator. However, it means that this method is not suitable for cases where there is only one feature.

Note that \textit{data\_all} and \textit{data\_tuples} are generators. \textit{data\_all} is a generator of a list of list of strings. This version assumes that CSV files are simple. The standard \textit{csv} package, that allows quoted arguments, can be used by uncommenting the line for \textit{data\_all} and commenting out the following line. \textit{data\_tuples} contains only those lines that contain the delimiter (others lines are assumed to be empty or comments), and tries to convert the elements to numbers whenever possible.

This allows for some of the columns to be included; specified by \textit{include\_only}. Note that if \textit{include\_only} is specified, the target index is the index for the included columns, not the original columns.
def read_data(file_name, separator=' ', num_train=None, prob_test=None, has_header=False, target_index=0, boolean_features=True, categorical=[], target_type=None, include_only=None, seed=None):  # seed=12345):
    """create a dataset from a file
separator is the character that separates the attributes
num_train is a number specifying the first num_train tuples are
    training, or None
prob_test is the probability an example should in the test set (if
    num_train is None)
has_header is True if the first line of file is a header
target_index specifies which feature is the target
boolean_features specifies whether we want to create Boolean
    features
    (if False, it uses the original features).
categorical is a set (or list) of features that should be treated
    as categorical
target_type is either None for automatic detection of target type
        or one of "numeric", "boolean", "categorical"
include_only is a list or set of indexes of columns to include
    ""
    self.boolean_features = boolean_features
    with open(file_name,'r',newline='') as csvfile:
        self.display(1,"Loading",file_name)
        # data_all = csv.reader(csvfile,delimiter=separator) # for more
            complicated CSV files
        data_all = (line.strip().split(separator) for line in csvfile)
        if include_only is not None:
            data_all = ([v for (i,v) in enumerate(line) if i in
                include_only]  
            for line in data_all)
        if has_header:
            header = next(data_all)
        else:
            header = None
        data_tuples = (interpret_elements(d) for d in data_all if
            len(d)>1)
        if num_train is not None:
            # training set is divided into training then test examples
            # the file is only read once, and the data is placed in
                appropriate list
            train = []
            for i in range(num_train):  # will give an error if
                insufficient examples
                train.append(next(data_tuples))
            test = list(data_tuples)
            Data_set.__init__(self,train, test=test,  
                target_index=target_index,header=header)
        else:  # randomly assign training and test examples
            Data_set.__init__(self,data_tuples, test=None,  
                prob_test=prob_test,
The following class is used for datasets where the training and test are in different files:

```python
def __init__(self, train_file_name, test_file_name, separator='\',
    has_header=False, target_index=0, boolean_features=True,
    categorical=[], target_type= None, include_only=None):
    """create a dataset from separate training and file
    separator is the character that separates the attributes
    num_train is a number specifying the first num_train tuples are
    training, or None
    prob_test is the probability an example should in the test set (if
    num_train is None)
    has_header is True if the first line of file is a header
    target_index specifies which feature is the target
    boolean_features specifies whether we want to create Boolean
    features
    (if False, it uses the original features).
    categorical is a set (or list) of features that should be treated
    as categorical
    target_type is either None for automatic detection of target type
    or one of "numeric", "boolean", "categorical"
    include_only is a list or set of indexes of columns to include
    """
    self.boolean_features = boolean_features
    with open(train_file_name, 'r', newline='') as train_file:
        with open(test_file_name, 'r', newline='') as test_file:
            # data_all = csv.reader(csvfile,delimiter=separator) # for more
            # complicated CSV files
            train_data = (line.strip().split(separator) for line in
                          train_file)
            test_data = (line.strip().split(separator) for line in
                         test_file)
            if include_only is not None:
                train_data = ([v for (i,v) in enumerate(line) if i in
                                include_only]
                              for line in train_data)
                test_data = ([v for (i,v) in enumerate(line) if i in
                               include_only]
                             for line in test_data)
            if has_header: # this assumes the training file has a header
                header = next(train_data)
            else:
                header = None
            train_tuples = [interpret_elements(d) for d in train_data if
                             len(d)>1]
```

https://aipython.org
7. Supervised Machine Learning

```python
    test_tuples = [interpret_elements(d) for d in test_data if len(d)>1]
    Data_set.__init__(self, train_tuples, test_tuples,
                      target_index=target_index, header=header)
```

When reading from a file all of the values are strings. This next method tries to convert each value into a number (an int or a float) or Boolean, if it is possible.

```python
    def interpret_elements(str_list):
        """make the elements of string list str_list numeric if possible.
        Otherwise remove initial and trailing spaces.
        """
        res = []
        for e in str_list:
            try:
                res.append(int(e))
            except ValueError:
                try:
                    res.append(float(e))
                except ValueError:
                    se = e.strip()
                    if se in ["True","true","TRUE"]: 
                        res.append(True)
                    elif se in ["False","false","FALSE"]: 
                        res.append(False)
                    else:
                        res.append(e.strip())
        return res
```

7.1.5 Augmented Features

Sometimes we want to augment the features with new features computed from the old features (e.g., the product of features). Here we allow the creation of a new dataset from an old dataset but with new features. Note that special cases of these are kernels; mapping the original feature space into a new space, which allow a neat way to do learning in the augmented space for many mappings (the “kernel trick”). This is beyond the scope of AIPython; those interested should read about support vector machines.

A feature is a function of examples. A unary feature constructor takes a feature and returns a new feature. A binary feature combiner takes two features and returns a new feature.
7.1. Representations of Data and Predictions

unary_function is a list of unary feature constructors
binary_functions is a list of binary feature combiners.
include_orig specifies whether the original features should be included

```python
self.orig_dataset = dataset
self.unary_functions = unary_functions
self.binary_functions = binary_functions
self.include_orig = include_orig
self.target = dataset.target
Data_set.__init__(self,dataset.train, test=dataset.test,
    target_index = dataset.target_index)
```

```python
def create_features(self):
    if self.include_orig:
        self.input_features = self.orig_dataset.input_features.copy()
    else:
        self.input_features = []
    for u in self.unary_functions:
        for f in self.orig_dataset.input_features:
            self.input_features.append(u(f))
    for b in self.binary_functions:
        for f1 in self.orig_dataset.input_features:
            for f2 in self.orig_dataset.input_features:
                if f1 != f2:
                    self.input_features.append(b(f1,f2))
```

The following are useful unary feature constructors and binary feature combiner.

```python
def square(f):
    """a unary feature constructor to construct the square of a feature
    ""
    def sq(e):
        return f(e)**2
    sq.__doc__ = f.__doc__+'**2'
    return sq

def power_feat(n):
    """given n returns a unary feature constructor to construct the nth
    power of a feature.
    e.g., power_feat(2) is the same as square, defined above
    ""
    def fn(f,n=n):
        def pow(e,n=n):
            return f(e)**n
        pow.__doc__ = f.__doc__+'**'+str(n)
        return pow
    return fn
```

[LearnProblem.py — (continued)](https://aipython.org)
def prod_feat(f1, f2):
    """a new feature that is the product of features f1 and f2
    """
    def feat(e):
        return f1(e)*f2(e)
    feat.__doc__ = f1.__doc__+'*'+f2.__doc__
    return feat

def eq_feat(f1, f2):
    """a new feature that is 1 if f1 and f2 give same value
    """
    def feat(e):
        return 1 if f1(e)==f2(e) else 0
    feat.__doc__ = f1.__doc__+'=='+f2.__doc__
    return feat

def neq_feat(f1, f2):
    """a new feature that is 1 if f1 and f2 give different values
    """
    def feat(e):
        return 1 if f1(e)!=f2(e) else 0
    feat.__doc__ = f1.__doc__+'!='+f2.__doc__
    return feat

Example:

# from learnProblem import Data_set_augmented, prod_feat
# data = Data_from_file('data/holiday.csv', has_header=True, num_train=19,
# target_index=-1)
# data = Data_from_file('data/iris.data', prob_test=1/3, target_index=-1)
## Data = Data_from_file('data/SPECT.csv', prob_test=0.5, target_index=0)
# dataplus = Data_set_augmented(data,[],[prod_feat])
# dataplus = Data_set_augmented(data,[],[prod_feat, neq_feat])

Exercise 7.3 For symmetric properties, such as product, we don’t need both 
f1 * f2 as well as f2 * f1 as extra properties. Allow the user to be able to declare 
feature constructors as symmetric (by associating a Boolean feature with them). 
Change construct features so that it does not create both versions for symmetric 
combiners.

7.2 Generic Learner Interface

A learner takes a dataset (and possibly other arguments specific to the method). 
To get it to learn, we call the learn() method. This implements Displayable so 
that we can display traces at multiple levels of detail (perhaps with a GUI).
7.3 Learning With No Input Features

If we make the same prediction for each example, what prediction should we make? This can be used as a naive baseline; if a more sophisticated method does not do better than this, it is not useful. This also provides the base case for some methods, such as decision-tree learning.

To run demo to compare different prediction methods on various evaluation criteria, in folder "aipython", load "learnNoInputs.py", using e.g., `ipython -i learnNoInputs.py`, and it prints some test results.

There are a few alternatives as to what could be allowed in a prediction:

- a point prediction, where we are only allowed to predict one of the values of the feature. For example, if the values of the feature are \{0, 1\} we are only allowed to predict 0 or 1 or of the values are ratings in \{1, 2, 3, 4, 5\}, we can only predict one of these integers.

- a point prediction, where we are allowed to predict any value. For example, if the values of the feature are \{0, 1\} we may be allowed to predict 0.3, 1, or even 1.7. For all of the criteria we can imagine, there is no point in predicting a value greater than 1 or less that zero (but that doesn’t mean we can’t), but it is often useful to predict a value between 0 and 1. If the values are ratings in \{1, 2, 3, 4, 5\}, we may want to predict 3.4.

- a probability distribution over the values of the feature. For each value \(v\), we predict a non-negative number \(p_v\), such that the sum over all predictions is 1.

For regression, we do the first of these. For classification, we do the second. The third can be implemented by having multiple indicator functions for the target.

Here are some prediction functions that take in an enumeration of values, a domain, and returns a value or dictionary of \{value : prediction\}. Note that \texttt{cmedian} returns one of the middle values when there are an even number of

```python
class Learner(Displayable):
    def __init__(self, dataset):
        raise NotImplementedError("Learner.__init__") # abstract method

def learn(self):
    """returns a predictor, a function from a tuple to a value for the target feature""
    raise NotImplementedError("learn") # abstract method
```

https://aipython.org
examples, whereas median gives the average of them (and so cmedian is applicable for ordinals that cannot be considered cardinal values). Similarly, cmode picks one of the values when more than one value has the maximum number of elements.

```python
from learnProblem import Evaluate
import math, random, collections, statistics
import utilities # argmax for (element,value) pairs

class Predict(object):
    """The class of prediction methods for a list of values.
    Please make the doc strings the same length, because they are used in
    tables.
    Note that we don't need self argument, as we are creating Predict
    objects,
    To use call Predict.laplace(data) etc.""

    ### The following return a distribution over values (for classification)
    def empirical(data, domain=[0,1], icount=0):
        "empirical dist"
        # returns a distribution over values
        counts = {v:icount for v in domain}
        for e in data:
            counts[e] += 1
        s = sum(counts.values())
        return {k:v/s for (k,v) in counts.items()}

    def bounded_empirical(data, domain=[0,1], bound=0.01):
        "bounded empirical"
        return {k:min(max(v,bound),1-bound) for (k,v) in 
            Predict.empirical(data, domain).items()}

    def laplace(data, domain=[0,1]):
        "Laplace " # for categorical data
        return Predict.empirical(data, domain, icount=1)

    def cmode(data, domain=[0,1]):
        "mode " # for categorical data
        md = statistics.mode(data)
        return {v: 1 if v==md else 0 for v in domain}

    def cmedian(data, domain=[0,1]):
        "median " # for categorical data
        md = statistics.median_low(data) # always return one of the values
        return {v: 1 if v==md else 0 for v in domain}

    ### The following return a single prediction (for regression). domain
    is ignored.
```
7.3. Learning With No Input Features

```python
def mean(data, domain=[0,1]):  # returns a real number
    return statistics.mean(data)

def rmean(data, domain=[0,1], mean0=0, pseudo_count=1):  
    "regularized mean"
    # returns a real number.
    # mean0 is the mean to be used for 0 data points
    # With mean0=0.5, pseudo_count=2, same as laplace for [0,1] data
    # this works for enumerations as well as lists
    sum = mean0 * pseudo_count
    count = pseudo_count
    for e in data:
        sum += e
        count += 1
    return sum/count

def mode(data, domain=[0,1]):
    "mode"
    return statistics.mode(data)

def median(data, domain=[0,1]):
    "median"
    return statistics.median(data)

all = [empirical, mean, rmean, bounded_empirical, laplace, cmode, mode, median, cmedian]

# The following suggests appropriate predictions as a function of the target type
select = {
    "boolean": [empirical, bounded_empirical, laplace, cmode, cmedian],
    "categorical": [empirical, bounded_empirical, laplace, cmode, cmedian],
    "numeric": [mean, rmean, mode, median]
}
```

7.3.1 Evaluation

To evaluate a point prediction, we first generate some data from a simple (Bernoulli) distribution, where there are two possible values, 0 and 1 for the target feature. Given $prob$, a number in the range $[0,1]$, this generate some training and test data where $prob$ is the probability of each example being 1. To generate a 1 with probability $prob$, we generate a random number in range $[0,1]$ and return 1 if that number is less than $prob$. A prediction is computed by applying the predictor to the training data, which is evaluated on the test set. This is repeated $num\_samples$ times.
Let’s evaluate the predictions of the possible selections according to the different evaluation criteria, for various training sizes.

```python
def test_no_inputs(error_measures = Evaluate.all_criteria, 
num_samples=10000, 
test_size=10, training_sizes= 
[1,2,3,4,5,10,20,100,1000]):
    for train_size in training_sizes:
        results = {predictor: {error_measure: 0 for error_measure in error_measures} 
            for predictor in Predict.all}
        for sample in range(num_samples):
            prob = random.random()
            training = [1 if random.random() < prob else 0 for i in range(train_size)]
            test = [1 if random.random() < prob else 0 for i in range(test_size)]
            for predictor in Predict.all:
                prediction = predictor(training)
                for error_measure in error_measures:
                    results[predictor][error_measure] += sum( 
                        error_measure(prediction, actual) for actual in test)/test_size
        print(f"For training size {train_size}:")
        print(" Predictor\t","\t".join(error_measure.__doc__ for 
            error_measure in error_measures),sep="\t")
        for predictor in Predict.all:
            print(f" {predictor.__doc__},"), 
            "\t".join("{:7f}".format(results[predictor][error_measure]/num_samples) 
                for error_measure in error_measures),sep="\t")
```

Exercise 7.4 Which predictor works best for low counts when the error is

(a) Squared error
(b) Absolute error
(c) Log loss

You may need to try this a few times to make sure your answer is supported by the evidence. Does the difference from the other methods get more or less as the number of examples grow?

Exercise 7.5 Suggest some other predictions that only take the training data. Does your method do better than the given methods? A simple way to get other predictors is to vary the threshold of bounded average, or to change the pseudocounts of the Laplace method (use other numbers instead of 1 and 2).
7.4 Decision Tree Learning

To run the decision tree learning demo, in folder "aipython", load "learnDT.py", using e.g., `ipython -i learnDT.py`, and it prints some test results. To try more examples, copy and paste the commented-out commands at the bottom of that file. This requires Python 3 with matplotlib.

The decision tree algorithm does binary splits, and assumes that all input features are binary functions of the examples. It stops splitting if there are no input features, the number of examples is less than a specified number of examples or all of the examples agree on the target feature.

```python
from learnProblem import Learner, Evaluate
from learnNoInputs import Predict
import math

class DT_learner(Learner):
    def __init__(self,
                 dataset,
                 split_to_optimize=Evaluate.log_loss, # to minimize for at each split
                 leaf_prediction=Predict.empirical, # what to use for value at leaves
                 train=None, # used for cross validation
                 max_num_cuts=8, # maximum number of conditions to split a numeric feature into
                 gamma=1e-7 , # minimum improvement needed to expand a node
                 min_child_weight=10):
        self.dataset = dataset
        self.target = dataset.target
        self.split_to_optimize = split_to_optimize
        self.leaf_prediction = leaf_prediction
        self.max_num_cuts = max_num_cuts
        self.gamma = gamma
        self.min_child_weight = min_child_weight
        if train is None:
            self.train = self.dataset.train
        else:
            self.train = train

    def learn(self, max_num_cuts=8):
        """learn a decision tree""
        return self.learn_tree(self.dataset.conditions(self.max_num_cuts),
                                self.train)
```

The main recursive algorithm, takes in a set of input features and a set of training data. It first decides whether to split. If it doesn’t split, it makes a point prediction, ignoring the input features.
It only splits if the best split increases the error by at least $\gamma$. This implies it does not split when:

- there are no more input features
- there are fewer examples than $\text{min\_number\_examples}$,
- all the examples agree on the value of the target, or
- the best split puts all examples in the same partition.

If it splits, it selects the best split according to the evaluation criterion (assuming that is the only split it gets to do), and returns the condition to split on (in the variable $split$) and the corresponding partition of the examples.

```python
def learn_tree(self, conditions, data_subset):
    """returns a decision tree
    conditions is a set of possible conditions
    data_subset is a subset of the data used to build this (sub)tree
    where a decision tree is a function that takes an example and
    makes a prediction on the target feature
    ""
    self.display(2,f"learn_tree with {len(conditions)} features and
    {len(data_subset)} examples")
    split, partn = self.select_split(conditions, data_subset)
    if split is None: # no split; return a point prediction
        prediction = self.leaf_value(data_subset, self.target.frange)
        self.display(2,f"leaf prediction for {len(data_subset)}
        examples is {prediction}")
    def leaf_fun(e):
        return prediction
    leaf_fun.__doc__ = str(prediction)
    leaf_fun.num_leaves = 1
    return leaf_fun
    else: # a split succeeded
        false_examples, true_examples = partn
        rem_features = [fe for fe in conditions if fe != split]
        self.display(2,"Splitting on",split.__doc__,"with examples
        split",
                len(true_examples),";",len(false_examples))
        true_tree = self.learn_tree(rem_features,true_examples)
        false_tree = self.learn_tree(rem_features,false_examples)
        def fun(e):
            if split(e):
                return true_tree(e)
            else:
                return false_tree(e)
        fun.__doc__ = (f"(if {split.__doc__} then {true_tree.__doc__}
        else {false_tree.__doc__})")
```
7.4. Decision Tree Learning

```python
f" else {false_tree.__doc__})")
fun.num_leaves = true_tree.num_leaves + false_tree.num_leaves
return fun
```

---

```python
def leaf_value(self, egs, domain):
    return self.leaf_prediction((self.target(e) for e in egs), domain)

def select_split(self, conditions, data_subset):
    """finds best feature to split on.

    conditions is a non-empty list of features.
    returns feature, partition
    where feature is an input feature with the smallest error as
    judged by split_to_optimize or
    feature==None if there are no splits that improve the error
    partition is a pair (false_examples, true_examples) if feature is
    not None
    ""
    best_feat = None # best feature
    # best_error = float("inf") # infinity - more than any error
    best_error = self.sum_losses(data_subset) - self.gamma
    self.display(3, " no split has
    error=",.best_error,"with",len(conditions),"conditions")
    best_partition = None
    for feat in conditions:
        false_examples, true_examples = partition(data_subset, feat)
        if min(len(false_examples),len(true_examples))>=self.min_child_weight:
            err = (self.sum_losses(false_examples)
            + self.sum_losses(true_examples))
            self.display(3, " split on",feat.__doc__,"has error=",.err,
            "splits
            into",len(true_examples),":",len(false_examples),"gamma=",.gamma)
        if err < best_error:
            best_feat = feat
            best_error=err
            best_partition = false_examples, true_examples
            self.display(2,"best split is on",best_feat.__doc__,
            "with err=",.best_error)
    return best_feat, best_partition

def sum_losses(self, data_subset):
    """returns sum of losses for dataset (with no more splits)
    There a single prediction for all leaves using leaf_prediction
    It is evaluated using split_to_optimize
    ""
    prediction = self.leaf_value(data_subset, self.target.frange)
    error = sum(self.split_to_optimize(prediction, self.target(e))
    for e in data_subset)
```
170
7. Supervised Machine Learning

```python
return error

def partition(data_subset, feature):
    """partitions the data_subset by the feature"""
    true_examples = []
    false_examples = []
    for example in data_subset:
        if feature(example):
            true_examples.append(example)
        else:
            false_examples.append(example)
    return false_examples, true_examples

Test cases:

from learnProblem import Data_set, Data_from_file

def testDT(data, print_tree=True, selections = None, **tree_args):
    """Prints errors and the trees for various evaluation criteria and ways to select leaves.
    """
    if selections == None: # use selections suitable for target type
        selections = Predict.select[data.target.ftype]
    evaluation_criteria = Evaluate.all_criteria
    print("Split Choice","Leaf Choice\t","#leaves",
          \t\t\t\".join(ecrit.__doc__
          for ecrit in evaluation_criteria),sep="\t")
    for crit in evaluation_criteria:
        for leaf in selections:
            tree = DT_learner(data, split_to_optimize=crit,
                leaf_prediction=leaf,
                **tree_args).learn()
            print(crit.__doc__, leaf.__doc__, tree.num_leaves,
                \t\t\t\".join("{:7f}\".format(data.evaluate_dataset(data.test,
                tree, ecrit))
            for ecrit in evaluation_criteria),sep="\t")
            if print_tree:
                print(tree.__doc__)

#DT_learner.max_display_level = 4
if __name__ == "__main__":
    # Choose one of the data files
    #data=Data_from_file('data/SPECT.csv', target_index=0);
    #print("SPECT.csv")
    #data=Data_from_file('data/iris.data', target_index=-1);
    #print("iris.data")
    data = Data_from_file('data/carbool.csv', target_index=-1, seed=123)
    #data = Data_from_file('data/mail_reading.csv', target_index=-1);
    #print("mail_reading.csv")
```

https://aipython.org  Version 0.9.12  December 22, 2023
7.5. Cross Validation and Parameter Tuning

```python
#data = Data_from_file('data/holiday.csv', has_header=True,
num_train=19, target_index=-1); print("holiday.csv")
testDT(data, print_tree=False)
```

Note that different runs may provide different values as they split the training and test sets differently. So if you have a hypothesis about what works better, make sure it is true for different runs.

**Exercise 7.6** The current algorithm does not have a very sophisticated stopping criterion. What is the current stopping criterion? (Hint: you need to look at both `learn_tree` and `select_split`.)

**Exercise 7.7** Extend the current algorithm to include in the stopping criterion

(a) A minimum child size; don’t use a split if one of the children has fewer elements that this.

(b) A depth-bound on the depth of the tree.

(c) An improvement bound such that a split is only carried out if error with the split is better than the error without the split by at least the improvement bound.

Which values for these parameters make the prediction errors on the test set the smallest? Try it on more than one dataset.

**Exercise 7.8** Without any input features, it is often better to include a pseudo-count that is added to the counts from the training data. Modify the code so that it includes a pseudo-count for the predictions. When evaluating a split, including pseudo counts can make the split worse than no split. Does pruning with an improvement bound and pseudo-counts make the algorithm work better than with an improvement bound by itself?

**Exercise 7.9** Some people have suggested using information gain (which is equivalent to greedy optimization of log loss) as the measure of improvement when building the tree, even if they want to have non-probabilistic predictions in the final tree. Does this work better than myopically choosing the split that is best for the evaluation criteria we will use to judge the final prediction?

---

7.5 Cross Validation and Parameter Tuning

To run the cross validation demo, in folder “aipython”, load “learnCrossValidation.py”, using e.g., `ipython -i learnCrossValidation.py`. Run the examples at the end to produce a graph like Figure 7.15. Note that different runs will produce different graphs, so your graph will not look like the one in the textbook. To try more examples, copy and paste the commented-out commands at the bottom of that file. This requires Python 3 with matplotlib.

[https://aipython.org](https://aipython.org)
The above decision tree overfits the data. One way to determine whether the prediction is overfitting is by cross validation. The code below implements $k$-fold cross validation, which can be used to choose the value of parameters to best fit the training data. If we want to use parameter tuning to improve predictions on a particular dataset, we can only use the training data (and not the test data) to tune the parameter.

In $k$-fold cross validation, we partition the training set into $k$ approximately equal-sized folds (each fold is an enumeration of examples). For each fold, we train on the other examples, and determine the error of the prediction on that fold. For example, if there are 10 folds, we train on 90% of the data, and then test on remaining 10% of the data. We do this 10 times, so that each example gets used as a test set once, and in the training set 9 times.

The code below creates one copy of the data, and multiple views of the data. For each fold, fold enumerates the examples in the fold, and fold_complement enumerates the examples not in the fold.

```python
from learnProblem import Data_set, Data_from_file, Evaluate
from learnNoInputs import Predict
from learnDT import DT_learner
import matplotlib.pyplot as plt
import random

class K_fold_dataset(object):
    def __init__(self, training_set, num_folds):
        self.data = training_set.train.copy()
        self.target = training_set.target
        self.input_features = training_set.input_features
        self.num_folds = num_folds
        self.conditions = training_set.conditions

        random.shuffle(self.data)
        self.fold_boundaries = [(len(self.data)*i)//num_folds
                     for i in range(0,num_folds+1)]

    def fold(self, fold_num):
        for i in range(self.fold_boundaries[fold_num],
                       self.fold_boundaries[fold_num+1]):
            yield self.data[i]

    def fold_complement(self, fold_num):
        for i in range(0,self.fold_boundaries[fold_num]):
            yield self.data[i]
        for i in range(self.fold_boundaries[fold_num+1],len(self.data)):
            yield self.data[i]
```

The validation error is the average error for each example, where we test on each fold, and learn on the other folds.
7.5. Cross Validation and Parameter Tuning

```python
def validation_error(self, learner, error_measure, **other_params):
    error = 0
    try:
        for i in range(self.num_folds):
            predictor = learner(self,
                train=list(self.fold_complement(i)),
                **other_params).learn()
            error += sum(error_measure(predictor(e), self.target(e))
                for e in self.fold(i))
        except ValueError:
            return float("inf")  # infinity
    return error/len(self.data)
```

The `plot_error` method plots the average error as a function of the minimum number of examples in decision-tree search, both for the validation set and for the test set. The error on the validation set can be used to tune the parameter — choose the value of the parameter that minimizes the error. The error on the test set cannot be used to tune the parameters; if it were to be used this way it could not be used to test how well the method works on unseen examples.

```python
def plot_error(data, criterion=Evaluate.squared_loss,
                  leaf_prediction=Predict.empirical,
                  num_folds=5, maxx=None, xscale='linear'):
    """Plots the error on the validation set and the test set with respect to settings of the minimum number of examples. xscale should be 'log' or 'linear' ""
    plt.ion()
    plt.xscale(xscale)  # change between log and linear scale
    plt.xlabel("min_child_weight")
    plt.ylabel("average " + criterion.__doc__")
    folded_data = K_fold_dataset(data, num_folds)
    if maxx == None:
        maxx = len(data.train)//2+1
    verrors = []  # validation errors
    terrors = []  # test set errors
    for mcw in range(1,maxx):
        verrors.append(folded_data.validation_error(DT_learner, criterion, leaf_prediction=leaf_prediction,
                                                    min_child_weight=mcw))
        tree = DT_learner(data, criterion, leaf_prediction=leaf_prediction,
                          min_child_weight=mcw).learn()
        terrors.append(data.evaluate_dataset(data.test, tree, criterion))
    plt.plot(range(1,maxx), verrors, ls='-', color='k',
             label="validation for " + criterion.__doc__")
    plt.plot(range(1,maxx), terrors, ls='--', color='k',
             label="test set for " + criterion.__doc__")
    plt.legend()
    plt.draw()
```
Figure 7.2 shows the average squared loss in the validation and test sets as a function of the \texttt{min\_child\_weight} in the decision-tree learning algorithm. (SPECT data with seed 12345 followed by \texttt{plot\_error(data)}). Different seeds will produce different graphs. The assumption behind cross validation is that the parameter that minimizes the loss on the validation set, will be a good parameter for the test set.

Note that different runs for the same data will have the same test error, but different validation error. If you rerun the \texttt{Data\_from\_file}, with a different seed, you will get the new test and training sets, and so the graph will change.

\textbf{Exercise 7.10} Change the error plot so that it can evaluate the stopping criteria of the exercise of Section 7.6 Which criteria makes the most difference?
7.6 Linear Regression and Classification

Here is a stochastic gradient descent searcher for linear regression and classification.

```python
from learnProblem import Learner
import random, math

class Linear_learner(Learner):
    def __init__(self, dataset, train=None,
                 learning_rate=0.1, max_init = 0.2,
                 squashed=True, batch_size=10):
        """Creates a gradient descent searcher for a linear classifier.
        The main learning is carried out by learn()
        ""
        self.dataset = dataset
        self.target = dataset.target
        if train==None:
            self.train = self.dataset.train
        else:
            self.train = train
        self.learning_rate = learning_rate
        self.squashed = squashed
        self.batch_size = batch_size
        self.input_features = [one]+dataset.input_features # one is defined below
        self.weights = {feat:random.uniform(-max_init,max_init)
                        for feat in self.input_features}

    predictor predicts the value of an example from the current parameter settings.
    predictor_string gives a string representation of the predictor.

    def predictor(self,e):
        """returns the prediction of the learner on example e""
        linpred = sum(w*f(e) for f,w in self.weights.items())
        if self.squashed:
            return sigmoid(linpred)
        else:
            return linpred

    def predictor_string(self, sig_dig=3):
```

https://aipython.org
"""returns the doc string for the current prediction function
sig_dig is the number of significant digits in the numbers"

doc = "+".join(str(round(val,sig_dig)) + "*" + feat.__doc__
    for feat,val in self.weights.items())

if self.squashed:
    return "sigmoid(" + doc + ")"
else:
    return doc

learn is the main algorithm of the learner. It does num_iter steps of stochastic
gradient descent. Only the number of iterations is specified; the other parame-
ters it gets from the class.

---

    def learn(self,num_iter=100):
        batch_size = min(self.batch_size, len(self.train))
        d = {feat:0 for feat in self.weights}
        for it in range(num_iter):
            self.display(2,"prediction=",self.predictor_string())
            for e in random.sample(self.train, batch_size):
                error = self.predictor(e) - self.target(e)
                update = self.learning_rate*error
                for feat in self.weights:
                    d[feat] += update*feat(e)
                    for feat in self.weights:
                        self.weights[feat] -= d[feat]
                        d[feat]=0
            return self.predictor

one is a function that always returns 1. This is used for one of the input prop-
erties.

---

    def one(e):
        "1"
    return 1

sigmoid(x) is the function

\[
\frac{1}{1+e^{-x}}
\]

The inverse of sigmoid is the logit function.

---

    def sigmoid(x):
        return 1/(1+math.exp(-x))
    def logit(x):
        return -math.log(1/x-1)
sigmoid([x0, v2, ...]) returns [v0, v2, ...] where

\[ v_i = \frac{\exp(x_i)}{\sum \exp(x_j)} \]

The inverse of sigmoid is the logit function

```python
def softmax(xs, domain=None):
    """xs is a list of values, and
domain is the domain (a list) or None if the list should be returned
returns a distribution over the domain (a dict)"
    m = max(xs) # use of m prevents overflow (and all values underflowing)
    exps = [math.exp(x-m) for x in xs]
    s = sum(exps)
    if domain:
        return {d:v/s for (d,v) in zip(domain,exps)}
    else:
        return [v/s for v in exps]
def indicator(v, domain):
    return [1 if v==dv else 0 for dv in domain]
```

The following tests the learner on a datasets. Uncomment the other datasets for different examples.

```python
from learnProblem import Data_set, Data_from_file, Evaluate
from learnProblem import Evaluate
import matplotlib.pyplot as plt
def test(**args):
    data = Data_from_file('data/SPECT.csv', target_index=0)
    # data = Data_from_file('data/mail_reading.csv', target_index=-1)
    # data = Data_from_file('data/carbool.csv', target_index=-1)
    learner = Linear_learner(data,**args)
    learner.learn()
    print("function learned is", learner.predictor_string())
    for ecrit in Evaluate.all_criteria:
        test_error = data.evaluate_dataset(data.test, learner.predictor, ecrit)
        print(" Average", ecrit.__doc__, "is", test_error)
```

The following plots the errors on the training and test sets as a function of the number of steps of gradient descent.

```python
def plot_steps(learner=None,
              data = None,
              criterion=Evaluate.squared_loss,
```
plots the training and test error for a learner.
data is the
learner_class is the class of the learning algorithm
criterion gives the evaluation criterion plotted on the y-axis
step specifies how many steps are run for each point on the plot
num_steps is the number of points to plot

if legend_label != ": legend_label+" "
plt.ion()
plt.xlabel("step")
plt.ylabel("Average " + criterion.__doc__)
if log_scale:
    plt.xscale('log') # plt.semilogx() # Makes a log scale
else:
    plt.xscale('linear')
if data is None:
    data = Data_from_file('data/holiday.csv', has_header=True,
    num_train=19, target_index=-1)
    # data = Data_from_file('data/SPECT.csv', target_index=0)
    # data = Data_from_file('data/mail_reading.csv', target_index=-1)
    # data = Data_from_file('data/carbool.csv', target_index=-1)
    # random.seed(None) # reset seed
if learner is None:
    learner = Linear_learner(data)
train_errors = []
test_errors = []
for i in range(1, num_steps+1, step):
    test_errors.append(data.evaluate_dataset(data.test,
        learner.predictor, criterion))
    train_errors.append(data.evaluate_dataset(data.train,
        learner.predictor, criterion))
    learner.display(2, "Train error:", train_errors[-1],
        "Test error:", test_errors[-1])
    learner.learn(num_iter=step)
    plt.plot(range(1, num_steps+1, step), train_errors, ls='-', label=legend_label+"training")
    plt.plot(range(1, num_steps+1, step), test_errors, ls='--', label=legend_label+"test")
    plt.legend()
    plt.draw()
    learner.display(1, "Train error:", train_errors[-1],
        "Test error:", test_errors[-1])
if __name__ == "__main__":
    test()
Figure 7.3 shows the result of `plot_steps(data=data, num_steps=1000)` in the code above. What would you expect to happen with the augmented data (with extra features)? Hint: think about underfitting and overfitting.

**Exercise 7.11** The squashed learner only makes predictions in the range \((0,1)\). If the output values are \(\{1,2,3,4\}\) there is no use predicting less than 1 or greater than 4. Change the squashed learner so that it can learn values in the range \((1,4)\). Test it on the file 'data/car.csv'.

The following plots the prediction as a function of the number of steps of gradient descent. We first define a version of `range` that allows for real numbers (integers and floats).
like the built-in range(start,stop,step) but allows for integers and floats.

Note that rounding errors are expected with real numbers. (or use numpy.arange)

```python
while start<stop:
    yield start
    start += step
```

def plot_prediction(data,
    learner = None,
    minx = 0,
    maxx = 5,
    step_size = 0.01, # for plotting
    label = "function"):
    plt.ion()
    plt.xlabel("x")
    plt.ylabel("y")
    if learner is None:
        learner = Linear_learner(data, squashed=False)
    learner.learning_rate=0.001
    learner.learn(100)
    learner.learning_rate=0.0001
    learner.learn(1000)
    learner.learning_rate=0.00001
    learner.learn(10000)
    learner.display(1,"function learned is", learner.predictor_string(),
        "error=",data.evaluate_dataset(data.train, learner.predictor,
        Evaluate.squared_loss))
    plt.plot([e[0] for e in data.train],[e[-1] for e in data.train],"bo",label="data")
    plt.plot(list(arange(minx,maxx,step_size)),[learner.predictor([x])
        for x in arange(minx,maxx,step_size)],
        label=label)
    plt.legend()
    plt.draw()
7.7 Boosting

The following code implements functional gradient boosting for regression.

A Boosted dataset is created from a base dataset by subtracting the prediction of the offset function from each example. This does not save the new dataset, but generates it as needed. The amount of space used is constant, independent on the size of the dataset.

---

```python
from learnProblem import Data_set, Learner, Evaluate
from learnNoInputs import Predict
from learnLinear import sigmoid
import statistics
import random

plt.xlabel("x")
plt.ylabel("y")
plt.plot([e[0] for e in data.train],[e[-1] for e in data.train],"ko",label="data")
x_values = list(arange(minx,maxx,step_size))
line_styles = ['-', '-.', '--', ':']
colors = ['0.5', 'k', 'k', 'k', 'k']
for degree in range(max_degree):
    data_aug = Data_set_augmented(data,[power_feat(n) for n in range(1,degree+1)],
    include_orig=False)
    learner = learner_class(data_aug,squashed=False)
    learner.learning_rate = learning_rate
    learner.learn(num_iter)
    learner.display(1,"For degree",degree,
    "function learned is", learner.predictor_string(),
    "error=",data.evaluate_dataset(data.train,
    learner.predictor, Evaluate.squared_loss))
    ls = line_styles[degree % len(line_styles)]
    col = colors[degree % len(colors)]
    plt.plot(x_values,[learner.predictor([x]) for x in x_values],
    linestyle=ls, color=col,
    label="degree=\"+str(degree)"
plt.legend(loc='upper left')
plt.draw()

# Try:
# data0 = Data_from_file('data/simp_regr.csv', prob_test=0,
#     boolean_features=False, target_index=-1)
# plot_prediction(data0)
# plot_polynomials(data0)
# What if the step size was bigger?
# datam = Data_from_file('data/mail_reading.csv', target_index=-1)
# plot_prediction(datam)
```

---

https://aipython.org
Version 0.9.12
December 22, 2023
A boosting learner takes in a dataset and a base learner, and returns a new predictor. The base learner, takes a dataset, and returns a Learner object.
7.7. Boosting

```python
def learn(self, num_ensembles=10):
    """adds num_ensemble learners to the ensemble.
    returns a new predictor.
    """
    for i in range(num_ensembles):
        train_subset = Boosted_dataset(self.dataset, self.predictor,
                                        subsample=self.subsample)
        learner = self.base_learner_class(train_subset)
        new_offset = learner.learn()
        def new_pred(e, old_pred=self.predictor, off=new_offset):
            return old_pred(e)+off(e)
        self.predictor = new_pred
        self.predictors.append(new_pred)
        self.offsets.append(new_offset)
        self.errors.append(data.evaluate_dataset(data.test,
                                                  self.predictor,
                                                  Evaluate.squared_loss))
        self.display(1,f"Iteration {len(self.offsets)-1}, treesize = {new_offset.num_leaves}. mean squared loss={self.errors[-1]}")
    return self.predictor
```

For testing, `sp_DT_learner` returns a learner that predicts the mean at the leaves and is evaluated using squared loss. It can also take arguments to change the default arguments for the trees.

```python
# Testing

from learnDT import DT_learner
from learnProblem import Data_set, Data_from_file

def sp_DT_learner(split_to_optimize=Evaluate.squared_loss,
                   leaf_prediction=Predict.mean,**nargs):
    """Creates a learner with different default arguments replaced by **nargs
    """
    def new_learner(dataset):
        return DT_learner(dataset,split_to_optimize=split_to_optimize,
                           leaf_prediction=leaf_prediction, **nargs)
    return new_learner

#data = Data_from_file('data/car.csv', target_index=-1) regression
#data = Data_from_file('data/student/student-mat-nq.csv', separator=';', has_header=True, target_index=-1, seed=13, include_only=list(range(30))+[32])
#2.0537973790924946
#data = Data_from_file('data/SPECT.csv', target_index=0, seed=62) #123)
#data = Data_from_file('data/mail_reading.csv', has_header=True, num_train=19,
#                      target_index=-1)
#data = Data_from_file('data/holiday.csv', has_header=True, num_train=19,
#                      target_index=-1)
#learner10 = Boosting_learner(data,
#                              sp_DT_learner(split_to_optimize=Evaluate.squared_loss,
```
7. Supervised Machine Learning

leaf_prediction=Predict.mean, min_child_weight=10))
#learner7 = Boosting_learner(data, sp_DT_learner(0.7))
#learner5 = Boosting_learner(data, sp_DT_learner(0.5))
#predictor9 = learner9.learn(10)
#For i in learner9.offsets: print(i.__doc__)
import matplotlib.pyplot as plt

def plot_boosting_trees(data, steps=10, mcws=[30,20,20,10], gammas=[100,200,300,500]):
    # to reduce clutter uncomment one of following two lines
    #mcws=[10]
    #gammas=[200]
    learners = [(mcw, gamma, Boosting_learner(data,
                                                sp_DT_learner(min_child_weight=mcw, gamma=gamma)))
                               for gamma in gammas for mcw in mcws
    plt.ion()
    plt.xscale('linear')  # change between log and linear scale
    plt.xlabel("number of trees")
    plt.ylabel("mean squared loss")
    markers = (m+c for c in ['k','g','r','m','c','y'] for m in ['-','--','-',':'])
    for (mcw,gamma,learner) in learners:
        data.display(1,f"min_child_weight={mcw}, gamma={gamma}")
        learner.learn(steps)
        plt.plot(range(steps+1), learner.errors, next(markers),
                  label=f"min_child_weight={mcw}, gamma={gamma}")
    plt.legend()
    plt.draw()

# plot_boosting_trees(data)

7.7.1 Gradient Tree Boosting

The following implements gradient Boosted trees for classification. If you want

to use this gradient tree boosting for a real problem, we recommend using
XGBoost[,] or LightGBM[,].

The method learn_tree is used unchanged. DT_learner assumes that the value at the leaf is the prediction of
the leaf, thus leaf_value needs to be overridden. It also assumes that all nodes
at a leaf have the same prediction, but in GBT the elements of a leaf can have
different values, depending on the previous trees. Thus sum_losses also needs
to be overridden.

```python
class GTB_learner(DT_learner):
    def __init__(self, dataset, number_trees, lambda_reg=1, gamma=0,
                 **dtargs):
```
DT_learner.__init__(self, dataset,
    split_to_optimize=Evaluate.log_loss, **dtargs)
self.number_trees = number_trees
self.lambda_reg = lambda_reg
self.gamma = gamma
self.trees = []

def learn(self):
    for i in range(self.number_trees):
        tree =
            self.learn_tree(self.dataset.conditions(self.max_num_cuts),
                self.train)
        self.trees.append(tree)
        self.display(1,f"""Iteration {i} treesize = {tree.num_leaves}
            train logloss={
                self.dataset.evaluate_dataset(self.dataset.train,
                    self.gtb_predictor, Evaluate.log_loss)
            } test logloss={
                self.dataset.evaluate_dataset(self.dataset.test,
                    self.gtb_predictor, Evaluate.log_loss)}""")
        return self.gtb_predictor

def gtb_predictor(self, example, extra=0):
    """prediction for example,
    extras is an extra contribution for this example being considered
    """
    return sigmoid(sum(t(example) for t in self.trees)+extra)

def leaf_value(self, egs, domain=[0,1]):
    """value at the leaves for examples egs
    domain argument is ignored""
    pred_acts = [(self.gtb_predictor(e),self.target(e)) for e in egs]
    return sum(a-p for (p,a) in pred_acts)/(sum(p*(1-p) for (p,a) in pred_acts)+self.lambda_reg)

def sum_losses(self, data_subset):
    """returns sum of losses for dataset (assuming a leaf is formed
    with no more splits)"
    leaf_val = self.leaf_value(data_subset)
    error = sum(Evaluate.log_loss(self.gtb_predictor(e,leaf_val),
        self.target(e))
        for e in data_subset) + self.gamma
    return error

Testing

# data = Data_from_file('data/carbool.csv', target_index=-1, seed=123)
# gtb_learner = GTB_learner(data, 10)
# gtb_learner.learn()
Warning: this is not meant to be an efficient implementation of deep learning. If you want to do serious machine learning on medium-sized or large data, we recommend Keras (https://keras.io) or PyTorch (https://pytorch.org), which are very efficient, particularly on GPUs. They are, however, black boxes. The AIPython neural network code should be seen like a car engine made of glass; you can see exactly how it works, even if it is not fast.

The parameters that are the same as in Keras have the same names.

8.1 Layers

A neural network is built from layers.

This provides a modular implementation of layers. Layers can easily be stacked in many configurations. A layer needs to implement a function to compute the output values from the inputs, a way to back-propagate the error, and perhaps update its parameters.
num outputs is the number of outputs for this layer.

```python
self.nn = nn
self.num_inputs = nn.num_outputs # output of nn is the input to
this layer
if num_outputs:
    self.num_outputs = num_outputs
else:
    self.num_outputs = nn.num_outputs # same as the inputs

def output_values(self, input_values, training=False):
    """Return the outputs for this layer for the given input values.
    input_values is a list of the inputs to this layer (of length
    num_inputs)
    returns a list of length self.num_outputs.
    It can act differently when training and when predicting.
    """
    raise NotImplementedError("output_values") # abstract method

def backprop(self, errors):
    """Backpropagate the errors on the outputs
    errors is a list of errors for the outputs (of length
    self.num_outputs).
    Returns the errors for the inputs to this layer (of length
    self.num_inputs).
    You can assume that this is only called after corresponding
    output_values,
    which can remember information information required for the
    back-propagation.
    """
    raise NotImplementedError("backprop") # abstract method

def update(self):
    """updates parameters after a batch.
    overridden by layers that have parameters
    """
    pass
```

A linear layer maintains an array of weights. `self.weights[o][i]` is the weight between input `i` and output `o`. A 1 is added to the end of the inputs. The default initialization is the Glorot uniform initializer [7], which is the default in Keras. An alternative is to provide a limit, in which case the values are selected uniformly in the range $[-\text{limit}, \text{limit}]$. Keras treats the bias separately, and defaults to zero.
nn is a neural network that the inputs come from
num_outputs is the number of outputs
the random initialization of parameters is in range \([-\text{limit}, \text{limit}]\)

```python
Layer.__init__(self, nn, num_outputs)
if limit is None:
    limit = math.sqrt(6/(self.num_inputs+self.num_outputs))
    self.weights[o][i] is the weight between input i and output o
self.weights = [[random.uniform(-limit, limit) if inf <
    self.num_inputs else 0
    for inf in range(self.num_inputs+1)]
    for outf in range(self.num_outputs)]
```

```python
def output_values(self,input_values, training=False):
    """Returns the outputs for the input values.
    It remembers the values for the backprop.

    Note in self.weights there is a weight list for every output,
    so wts in self.weights loops over the outputs.
    The bias is the *last* value of each list in self.weights.
    """
    self.inputs = input_values + [1]
    return [sum(w*val for (w,val) in zip(wts,self.inputs))
        for wts in self.weights]
```

```python
def backprop(self,errors):
    """Backpropagate the errors, updating the weights and returning the
    error in its inputs.
    """
    input_errors = [0]*(self.num_inputs+1)
    for out in range(self.num_outputs):
        for inp in range(self.num_inputs+1):
            input_errors[inp] += self.weights[out][inp] * errors[out]
            self.delta[out][inp] += self.inputs[inp] * errors[out]
    return input_errors[:-1] # remove the error for the "1"
```

```python
def update(self):
    """Updates parameters after a batch"
    batch_step_size = self.nn.learning_rate / self.nn.batch_size
    for out in range(self.num_outputs):
        for inp in range(self.num_inputs+1):
            self.weights[out][inp] -= batch_step_size *
            self.delta[out][inp]
            self.delta[out][inp] = 0
```

The standard activation function for hidden nodes is the ReLU.

```python
class ReLU_layer(Layer):
    ...
```

```
https://aipython.org Version 0.9.12 December 22, 2023
```
"""Rectified linear unit (ReLU) \( f(z) = \max(0, z) \).
The number of outputs is equal to the number of inputs.
"""

```python
def __init__(self, nn):
    Layer.__init__(self, nn)

def output_values(self, input_values, training=False):
    """Returns the outputs for the input values.
    It remembers the input values for the backprop.
    """
    self.input_values = input_values
    self.outputs = [max(0, inp) for inp in input_values]
    return self.outputs

def backprop(self, errors):
    """Returns the derivative of the errors"
    return [e if inp > 0 else 0 for e, inp in zip(errors, self.input_values)]
```

One of the old standards for the activation function for hidden layers is the sigmoid. It is included here to experiment with.

```python
class Sigmoid_layer(Layer):
    """sigmoids of the inputs.
    The number of outputs is equal to the number of inputs.
    Each output is the sigmoid of its corresponding input.
    """
    def __init__(self, nn):
        Layer.__init__(self, nn)

def output_values(self, input_values, training=False):
    """Returns the outputs for the input values.
    It remembers the output values for the backprop.
    """
    self.outputs = [sigmoid(inp) for inp in input_values]
    return self.outputs

def backprop(self, errors):
    """Returns the derivative of the errors"
    return [e*out*(1-out) for e, out in zip(errors, self.outputs)]
```

### 8.2 Feedforward Networks

```python
class NN(Learner):
    """Creates a neural network for a dataset,
```
8.2. Feedforward Networks

```python
layers is the list of layers

self.dataset = dataset
self.output_type = dataset.target.ftype
self.learning_rate = learning_rate
self.input_features = dataset.input_features
self.num_outputs = len(self.input_features)
validation_num = int(len(self.dataset.train)*validation_proportion)
if validation_num > 0:
    random.shuffle(self.dataset.train)
    self.validation_set = self.dataset.train[-validation_num:]
    self.training_set = self.dataset.train[:-validation_num]
else:
    self.validation_set = []
    self.training_set = self.dataset.train
self.layers = []
self.bn = 0 # number of batches run

def add_layer(self,layer):
    """add a layer to the network.
    Each layer gets number of inputs from the previous layers outputs.
    """
    self.layers.append(layer)
    self.num_outputs = layer.num_outputs

def predictor(self,ex):
    """Predicts the value of the first output for example ex.
    """
    values = [f(ex) for f in self.input_features]
    for layer in self.layers:
        values = layer.output_values(values)
    return sigmoid(values[0]) if self.output_type == "boolean" \
        else softmax(values, self.dataset.target.frange) if \
            self.output_type == "categorical" \
        else values[0]

def predictor_string(self):
    return "not implemented"

The `learn` method learns a network.

```
8. Neural Networks and Deep Learning

8.3 Improved Optimization

8.3.1 Momentum

```
class Linear_complete_layer_momentum(Linear_complete_layer):
    """a completely connected layer""
```

https://aipython.org Version 0.9.12 December 22, 2023
def __init__(self, nn, num_outputs, limit=None, alpha=0.9, epsilon = 1e-07, vel0=0):
    """A completely connected linear layer.
    nn is a neural network that the inputs come from
    num_outputs is the number of outputs
    max_init is the maximum value for random initialization of
    parameters
    vel0 is the initial velocity for each parameter
    """
    Linear_complete_layer.__init__(self, nn, num_outputs, limit=limit)
    # self.weights[o][i] is the weight between input i and output o
    self.velocity = [[vel0 for inf in range(self.num_inputs+1)]
                     for outf in range(self.num_outputs)]
    self.alpha = alpha
    self.epsilon = epsilon

def update(self):
    """updates parameters after a batch""
    batch_step_size = self.nn.learning_rate / self.nn.batch_size
    for out in range(self.num_outputs):
        for inp in range(self.num_inputs+1):
            self.velocity[out][inp] = self.alpha*self.velocity[out][inp]
            - batch_step_size * self.delta[out][inp]
            self.weights[out][inp] += self.velocity[out][inp]
            self.delta[out][inp] = 0

8.3.2 RMS-Prop

```python
class Linear_complete_layer_RMS_Prop(Linear_complete_layer):
    """a completely connected layer"
    def __init__(self, nn, num_outputs, limit=None, rho=0.9, epsilon =
                 1e-07):
        """A completely connected linear layer.
        nn is a neural network that the inputs come from
        num_outputs is the number of outputs
        max_init is the maximum value for random initialization of
        parameters
        """
        Linear_complete_layer.__init__(self, nn, num_outputs, limit=limit)
        # self.weights[o][i] is the weight between input i and output o
        self.ms = [[0 for inf in range(self.num_inputs+1)]
                   for outf in range(self.num_outputs)]
        self.rho = rho
        self.epsilon = epsilon

def update(self):
    """updates parameters after a batch"
    for out in range(self.num_outputs):
        for inp in range(self.num_inputs+1):
```

https://aipython.org  Version 0.9.12  December 22, 2023
gradient = self.delta[out][inp] / self.nn.batch_size
self.ms[out][inp] = self.rho*self.ms[out][inp]+ (1-self.rho) * gradient**2
self.weights[out][inp] -= self.nn.learning_rate/(self.ms[out][inp]+self.epsilon)**0.5 * gradient
self.delta[out][inp] = 0

8.4 Dropout

Dropout is implemented as a layer.

def __init__(self, nn, rate=0):
    """rate is fraction of the input units to drop. 0 <= rate < 1  """
    self.rate = rate
Layer.__init__(self, nn)

def output_values(self, input_values, training=False):
    """Returns the outputs for the input values. It remembers the input values for the backprop.  """
    if training:
        scaling = 1/(1-self.rate)
        self.mask = [0 if flip(self.rate) else 1 for _ in input_values]
        return [x*y*scaling for (x,y) in zip(input_values, self.mask)]
    else:
        return input_values

def backprop(self,errors):
    """Returns the derivative of the errors""
    return [x*y for (x,y) in zip(errors, self.mask)]
8.4. Dropout

```python
Layer.__init__(self, nn)

def output_values(self, input_values, training=False):
    """Returns the outputs for the input values.
It remembers the input values for the backprop.
""
    if training:
        scaling = 1/(1-self.rate)
        self.outputs= [0 if flip(self.rate) else inp*scaling # make 0
                       with probability rate
                       for inp in input_values]
        return self.outputs
    else:
        return input_values

def backprop(self,errors):
    """Returns the derivative of the errors""
    return errors
```

8.4.1 Examples

The following constructs a neural network with one hidden layer. The output is assumed to be Boolean or Real. If it is categorical, the final layer should have the same number of outputs as the number of categories (so it can use a softmax).

```python
#data = Data_from_file('data/mail_reading.csv', target_index=-1)
#data = Data_from_file('data/mail_reading_consis.csv', target_index=-1)
data = Data_from_file('data/SPECT.csv', prob_test=0.3, target_index=0,
                      seed=12345)
#data = Data_from_file('data/iris.data', prob_test=0.2, target_index=-1) # 150 examples approx 120 test + 30 test
#data = Data_from_file('data/iris.data', num_train=8, target_index=-1) # not linearly sep
#data = Data_from_file('data/holiday.csv', target_index=-1) #,
#random.seed(None)

# nn3 is has a single hidden layer of width 3
nn3 = NN(data, validation_proportion = 0)
nn3.add_layer(Linear_complete_layer(nn3,3))
#nn3.add_layer(Sigmoid_layer(nn3))
nn3.add_layer(ReLU_layer(nn3))
nn3.add_layer(Linear_complete_layer(nn3,1)) # when using
    output_type="boolean"
#nn3.learn(epochs = 100)
```

https://aipython.org  Version 0.9.12  December 22, 2023
# nn3do is like nn3 but with dropout on the hidden layer

nn3do = NN(data, validation_proportion = 0)
nn3do.add_layer(Linear_complete_layer(nn3do,3))
#nn3.add_layer(Sigmoid_layer(nn3)) # comment this or the next
nn3do.add_layer(ReLU_layer(nn3do))
nn3do.add_layer(Dropout_layer(nn3do, rate=0.5))
nn3do.add_layer(Linear_complete_layer(nn3do,1))
#nn3do.learn(epochs = 100)

# nn3_rmsp is like nn3 but uses RMS prop
nn3_rmsp = NN(data, validation_proportion = 0)
nn3_rmsp.add_layer(Linear_complete_layer_RMS_Prop(nn3_rmsp,3))
#nn3_rmsp.add_layer(Sigmoid_layer(nn3_rmsp)) # comment this or the next
nn3_rmsp.add_layer(ReLU_layer(nn3_rmsp))
nn3_rmsp.add_layer(Linear_complete_layer_RMS_Prop(nn3_rmsp,1))
#nn3_rmsp.learn(epochs = 100)

# nn3_m is like nn3 but uses momentum
mm1_m = NN(data, validation_proportion = 0)
mm1_m.add_layer(Linear_complete_layer_momentum(mm1_m,3))
#mm1_m.add_layer(Sigmoid_layer(mm1_m)) # comment this or the next
mm1_m.add_layer(ReLU_layer(mm1_m))
mm1_m.add_layer(Linear_complete_layer_momentum(mm1_m,1))
#mm1_m.learn(epochs = 100)

# nn2 has a single a hidden layer of width 2
nn2 = NN(data, validation_proportion = 0)
nn2.add_layer(Linear_complete_layer_RMS_Prop(nn2,2))
nn2.add_layer(ReLU_layer(nn2))
nn2.add_layer(Linear_complete_layer_RMS_Prop(nn2,1))

# nn5 is has a single hidden layer of width 5
nn5 = NN(data, validation_proportion = 0)
nn5.add_layer(Linear_complete_layer_RMS_Prop(nn5,5))
nn5.add_layer(ReLU_layer(nn5))
nn5.add_layer(Linear_complete_layer_RMS_Prop(nn5,1))

# nn0 has no hidden layers, and so is just logistic regression:
nn0 = NN(data, validation_proportion = 0) #learning_rate=0.05)
nn0.add_layer(Linear_complete_layer(nn0,1))
# Or try this for RMS-Prop:
#nn0.add_layer(Linear_complete_layer_RMS_Prop(nn0,1))

Plotting. Figure 8.1 shows the training and test performance on the SPECT dataset for the architectures above. Note the nn5 test has infinite log loss after about 45,000 steps. The noisyness of the predictions might indicate that the step size is too big. This was produced by the code below:

```

from learnLinear import plot_steps
from learnProblem import Evaluate
```

https://aipython.org  Version 0.9.12  December 22, 2023
8.4. Dropout

Figure 8.1: Plotting train and test log loss for various algorithms on SPECT dataset

```python
# To show plots first choose a criterion to use
# crit = Evaluate.log_loss
# crit = Evaluate.accuracy
# plot_steps(learner = nn0, data = data, criterion=crit, num_steps=10000,
# log_scale=False, legend_label="nn0")
# plot_steps(learner = nn2, data = data, criterion=crit, num_steps=10000,
# log_scale=False, legend_label="nn2")
# plot_steps(learner = nn3, data = data, criterion=crit, num_steps=10000,
# log_scale=False, legend_label="nn3")
# plot_steps(learner = nn5, data = data, criterion=crit, num_steps=10000,
# log_scale=False, legend_label="nn5")
# for (nn,nname) in [(nn0,"nn0"),(nn2,"nn2"),(nn3,"nn3"),(nn5,"nn5")]:
# plot_steps(learner = nn, data = data, criterion=crit,
# num_steps=100000, log_scale=False, legend_label=nname)
# Print some training examples
# for eg in random.sample(data.train,10): print(eg,nn3.predictor(eg))
# Print some test examples
# for eg in random.sample(data.test,10): print(eg,nn3.predictor(eg))
```
# To see the weights learned in linear layers
# nn3.layers[0].weights
# nn3.layers[2].weights

# Print test:
# for e in data.train: print(e,nn0.predictor(e))

def test(data, hidden_widths = [5], epochs=100,
          optimizers = [Linear_complete_layer,
                        Linear_complete_layer_momentum,
                        Linear_complete_layer_RMS_Prop]):
    data.display(0,"Batch\t","\t".join(criterion.__doc__ for criterion in Evaluate.all_criteria))
    for optimizer in optimizers:
        nn = NN(data)
        for width in hidden_widths:
            nn.add_layer(optimizer(nn,width))
            nn.add_layer(ReLU_layer(nn))
        if data.target.ftype == "boolean":
            nn.add_layer(optimizer(nn,1))
        else:
            error(F"Not implemented: {data.output_type}"
        nn.learn(epochs)

The following tests on MNIST. The original files are from http://yann.lecun.
com/exdb/mnist/. This code assumes you use the csv files from https://pjreddie.
com/projects/mnist-in-csv/ and put them in the directory ../MNIST/. Note
that this is very inefficient; you would be better to use Keras or Pytorch. There
are 28 * 28 = 784 input units and 512 hidden units, which makes 401,408 pa-
rameters for the lowest linear layer. So don’t be surprised when it takes many
hours in AIPython (even if it only takes a few seconds in Keras).

# Simplified version: (6000 training instances)
# data_mnist = Data_from_file('..../MNIST/mnist_train.csv', prob_test=0.9,
# target_index=0, boolean_features=False, target_type="categorical")

# Full version:
# data_mnist = Data_from_files('..../MNIST/mnist_train.csv',
# '..../MNIST/mnist_test.csv', target_index=0, boolean_features=False,
# target_type="categorical")

# nn_mnist = NN(data_mnist, validation_proportion = 0.02,
# learning_rate=0.001) #validation set = 1200
# nn_mnist.add_layer(Linear_complete_layer_RMS_Prop(nn_mnist,512));
# nn_mnist.add_layer(ReLU_layer(nn_mnist));
# nn_mnist.add_layer(Linear_complete_layer_RMS_Prop(nn_mnist,10))
# start_time = time.perf_counter();nn_mnist.learn(epochs=1,
# batch_size=128);end_time = time.perf_counter();print("Time:", end_time

https://aipython.org  Version 0.9.12  December 22, 2023
8.4. Dropout

- start_time,"seconds") #1 epoch

# determine test error:
# data_mnist.evaluate_dataset(data_mnist.test, nn_mnist.predictor,
  Evaluate.accuracy)

# Print some random predictions:
# for eg in random.sample(data_mnist.test,10):
  print(data_mnist.target(eg),nn_mnist.predictor(eg),nn_mnist.predictor(eg)[data_mnist.target(eg)])

Exercise 8.1  In the definition of nn3 above, for each of the following, first hypothesize what will happen, then test your hypothesis, then explain whether you testing confirms your hypothesis or not. Test it for more than one data set, and use more than one run for each data set.

(a) Which fits the data better, having a sigmoid layer or a ReLU layer after the first linear layer?

(b) Which is faster, having a sigmoid layer or a ReLU layer after the first linear layer?

(c) What happens if you have both the sigmoid layer and then a ReLU layer after the first linear layer and before the second linear layer?

(d) What happens if you have a ReLU layer then a sigmoid layer after the first linear layer and before the second linear layer?

(e) What happens if you have neither the sigmoid layer nor a ReLU layer after the first linear layer?

Exercise 8.2  Do some
Chapter 9

Reasoning with Uncertainty

9.1 Representing Probabilistic Models

A probabilistic model uses the same definition of a variable as a CSP (Section 4.1.1, page 69). A variable consists of a name, a domain and an optional (x,y) position (for displaying). The domain of a variable is a list or a tuple, as the ordering will matter in the representation of factors.

9.2 Representing Factors

A factor is, mathematically, a function from variables into a number; that is given a value for each of its variable, it gives a number. Factors are used for conditional probabilities, utilities in the next chapter, and are explicitly constructed by some algorithms (in particular variable elimination).

A variable assignment, or just assignment, is represented as a \{variable : value\} dictionary. A factor can be evaluated when all of its variables are assigned. The method get_value evaluates the factor for an assignment. The assignment can include extra variables not in the factor. This method needs to be defined for every subclass.

```
probFactors.py — Factors for graphical models

from display import Displayable
import math

class Factor(Displayable):
    nextid=0 # each factor has a unique identifier; for printing

    def __init__(self, variables, name=None):
        self.variables = variables # list of variables
```
```python
if name:
    self.name = name
else:
    self.name = f"f{Factor.nextid}"
    Factor.nextid += 1

def can_evaluate(self, assignment):
    """True when the factor can be evaluated in the assignment
    assignment is a {variable:value} dict
    """
    return all(v in assignment for v in self.variables)

def get_value(self, assignment):
    """Returns the value of the factor given the assignment of values
    to variables.
    Needs to be defined for each subclass.
    """
    assert self.can_evaluate(assignment)
    raise NotImplementedError("get_value") # abstract method

The method __str__ returns a brief definition (like "f7(X,Y,Z)"). The method
to_table returns string representations of a table showing all of the assignments
of values to variables, and the corresponding value.
```

```python
def __str__(self):
    """returns a string representing a summary of the factor"
    return f"{self.name}({', '.join(str(var) for var in self.variables)})"

def to_table(self, variables=None, given={}):
    """returns a string representation of the factor.
    Allows for an arbitrary variable ordering.
    variables is a list of the variables in the factor
    (can contain other variables)"
    if variables == None:
        variables = [v for v in self.variables if v not in given]
    else:
        variables = [v for v in variables if v in self.variables and v
                     not in given]
    head = "	".join(str(v) for v in variables)+"\t"+self.name
    return head+"\n"+self.ass_to_str(variables, given, variables)

def ass_to_str(self, vars, asst, allvars):
    #print(f"ass_to_str({vars}, {asst}, {allvars})")
    if vars:
        return "\n".join(self.ass_to_str(vars[1:], {**asst, vars[0]:val}, allvars)
                       for val in vars[0].domain)
    else:
        val = self.get_value(asst)
```
9.3 Conditional Probability Distributions

A conditional probability distribution (CPD) is a type of factor that represents a conditional probability. A CPD representing $P(X \mid Y_1 \ldots Y_k)$ is a type of factor, where given values for $X$ and each $Y_i$ returns a number.

A constant CPD has no parents, and has probability 1 when the variable has the value specified, and 0 when the variable has a different value.

9.3.1 Logistic Regression

A logistic regression CPD, for Boolean variable $X$ represents $P(X=\text{True} \mid Y_1 \ldots Y_k)$, using $k+1$ real-values weights so

$$P(X=\text{True} \mid Y_1 \ldots Y_k) = \text{sigmoid}(w_0 + \sum_i w_i Y_i)$$
where for Boolean \( Y_i \), True is represented as 1 and False as 0.

```python
from learnLinear import sigmoid, logit

class LogisticRegression(CPD):
    def __init__(self, child, parents, weights):
        """A logistic regression representation of a conditional probability.
        child is the Boolean (or 0/1) variable whose CPD is being defined
        parents is the list of parents
        weights is list of parameters, such that weights[i+1] is the weight
        for parents[i]
        ""
        assert len(weights) == 1 + len(parents)
        CPD.__init__(self, child, parents)
        self.weights = weights

    def get_value(self, assignment):
        assert self.can_evaluate(assignment)
        prob = sigmoid(self.weights[0])
        prob += sum(self.weights[i+1]*assignment[self.parents[i]]
                     for i in range(len(self.parents)))
        if assignment[self.child]: #child is true
            return prob
        else:
            return (1-prob)
```

### 9.3.2 Noisy-or

A **noisy-or**, for Boolean variable \( X \) with Boolean parents \( Y_1 \ldots Y_k \) is parametrized by \( k + 1 \) parameters \( p_0, p_1, \ldots, p_k \), where each \( 0 \leq p_i \leq 1 \). The semantics is defined as though there are \( k + 1 \) hidden variables \( Z_0, Z_1 \ldots Z_k \), where \( P(Z_0) = p_0 \) and \( P(Z_i | Y_i) = p_i \) for \( i \geq 1 \), and where \( X \) is true if and only if \( Z_0 \lor Z_1 \lor \cdots \lor Z_k \) (where \( \lor \) is “or”). Thus \( X \) is false if all of the \( Z_i \) are false. Intuitively, \( Z_0 \) is the probability of \( X \) when all \( Y_i \) are false and each \( Z_i \) is a noisy (probabilistic) measure that \( Y_i \) makes \( X \) true, and \( X \) only needs one to make it true.

```python
class NoisyOR(CPD):
    def __init__(self, child, parents, weights):
        """A noisy representation of a conditional probability.
        variable is the Boolean (or 0/1) child variable whose CPD is being defined
        parents is the list of Boolean (or 0/1) parents
        weights is list of parameters, such that weights[i+1] is the weight
        for parents[i]
        ""
        assert len(weights) == 1 + len(parents)
```

[https://aipython.org](https://aipython.org)  Version 0.9.12  December 22, 2023
9.3. Conditional Probability Distributions

```python
CPD.__init__(self, child, parents)
self.weights = weights

def get_value(self, assignment):
    assert self.can_evaluate(assignment)
    probfalse = (1-self.weights[0])*math.prod(1-self.weights[i+1]
        for i in range(len(self.parents))
        if assignment[self.parents[i]])

    if assignment[self.child]:
        return 1-probfalse
    else:
        return probfalse
```

9.3.3 Tabular Factors and Prob

A tabular factor is a factor that represents each assignment of values to variables separately. It is represented by a Python array (or python dict). If the variables are $V_1, V_2, \ldots, V_k$, the value of $f(V_1 = v_1, V_2 = v_1, \ldots, V_k = v_k)$ is stored in $f[v_1][v_2] \ldots [v_k]$.

If the domain of $V_i$ is $[0, \ldots, n_i - 1]$ this can be represented as an array. Otherwise we can use a dictionary. Python is nice in that it doesn’t care, whether an array or dict is used except when enumerating the values; enumerating a dict gives the keys (the variables) but enumerating an array gives the values. So we have to be careful not to do this.

```python
class TabFactor(Factor):
    def __init__(self, variables, values, name=None):
        Factor.__init__(self, variables, name=name)
        self.values = values

    def get_value(self, assignment):
        return self.get_val_rec(self.values, self.variables, assignment)

def get_val_rec(self, value, variables, assignment):
    if variables == []:
        return value
    else:
        return self.get_val_rec(value[assignment[variables[0]]],
            variables[1:], assignment)
```

Prob is a factor that represents a conditional probability by enumerating all of the values.

```python
class Prob(CPD, TabFactor):
```
9. Reasoning with Uncertainty

A factor defined by a conditional probability table

```python
def __init__(self, var, pars, cpt, name=None):
    creates a factor from a conditional probability table, cpt
    The cpt values are assumed to be for the ordering par+[var]
    TabFactor.__init__(self, pars+[var], cpt, name)
    self.child = var
    self.parents = pars
```

9.3.4 Decision Tree Representations of Factors

A decision tree representation of a conditional probability is either:

- `IfEq(var, val, true_cond, false_cond)` where `true_cond` and `false_cond` are decision trees. `true_cond` is used if variable `var` has value `val` in an assignment; `false_cond` is used if `var` has a different value.

- A distribution over the child variable

Note that not all parents need to be assigned to evaluate the decision tree; you only need the branch down the tree that gives the distribution.

```python
class ProbDT(CPD):
    def __init__(self, child, parents, dt):
        CPD.__init__(self, child, parents)
        self.dt = dt

    def get_value(self, assignment):
        return self.dt.get_value(assignment, self.child)

    def can_evaluate(self, assignment):
        return self.child in assignment and self.dt.can_evaluate(assignment)
```

Decision trees are made up of conditions; here equality of a value and a variable:

```python
class IfEq:
    def __init__(self, var, val, true_cond, false_cond):
        self.var = var
        self.val = val
        self.true_cond = true_cond
        self.false_cond = false_cond

    def get_value(self, assignment, child):
        if assignment[self.var] == self.val:
            return self.true_cond.get_value(assignment, child)
        else:
            return self.false_cond.get_value(assignment, child)
```
At the leaves are distributions over the child variable.

The following shows a decision representation of the Example 9.18 of AIFCA 3e. When the Action is to go out, the probability is a function of rain; otherwise it is a function of full.
9.4 Graphical Models

A graphical model consists of a set of variables and a set of factors. A belief network is a graphical model where all of the factors represent conditional probabilities. There are some operations (such as pruning variables) which are applicable to belief networks, but are not applicable to more general models. At the moment, we will treat them as the same.

```python
from display import Displayable
from probFactors import CPD
import matplotlib.pyplot as plt

class GraphicalModel(Displayable):
    """The class of graphical models.
    A graphical model consists of a title, a set of variables and a set of
    factors.
    """
    def __init__(self, title, variables=None, factors=None):
        self.title = title
        self.variables = variables
        self.factors = factors

A belief network (also known as a Bayesian network) is a graphical model where all of the factors are conditional probabilities, and every variable has a conditional probability of it given its parents. This only checks the first condition, and builds some useful data structures.

```
The following creates a topological sort of the nodes, where the parents of a node come before the node in the resulting order. This is based on Kahn’s algorithm from 1962.

```python
probGraphicalModels.py — (continued)

def topological_sort(self):
    """creates a topological ordering of variables such that the parents of a node are before the node. """
    if self.topological_sort_saved:
        return self.topological_sort_saved
    next_vars = {n for n in self.var2parents if not self.var2parents[n]}
    self.display(3,'topological_sort: next_vars',next_vars)
    top_order=[]
    while next_vars:
        var = next_vars.pop()
        self.display(3,'select variable',var)
        top_order.append(var)
        next_vars |= {ch for ch in self.children[var] if all(p in top_order for p in self.var2parents[ch])}
    self.display(3,'var_with_no_parents_left',next_vars)
    self.display(3,"top_order",top_order)
    assert set(top_order)==set(self.var2parents),(top_order,self.var2parents)
    self.topologicalsort_saved=top_order
    return top_order

9.4.1 Showing Belief Networks

The `show` method uses matplotlib to show the graphical structure of a belief network.

```python
probGraphicalModels.py — (continued)

def show(self, fontsize=10, facecolor='orange'):
    plt.ion() # interactive
    ax = plt.figure().gca()
    ax.set_axis_off()
    plt.title(self.title, fontsize=fontsize)
    bbox = dict(boxstyle="round4,pad=1.0,rounding_size=0.5",facecolor=facecolor)
    for var in self.variables: #reversed(self.topological_sort()):
        for par in self.var2parents[var]:
            ax.annotate(var.name, par.position, xytext=var.position,
            arrowprops={"arrowstyle": '<-',bbox=bbox,
            ha='center', va='center',
            fontsize=fontsize)
```

[https://aipython.org](https://aipython.org)  Version 0.9.12  December 22, 2023
4-chain

A Chain of 4 Variables

The first example belief network is a simple chain $A \rightarrow B \rightarrow C \rightarrow D$, shown in Figure 9.1.

Please do not change this, as it is the example used for testing.

```python
for var in self.variables:
    x, y = var.position
    plt.text(x, y, var.name, bbox=bbox, ha='center', va='center', fontsize=fontsize)
```

9.4.2 Example Belief Networks

A Chain of 4 Variables

The first example belief network is a simple chain $A \rightarrow B \rightarrow C \rightarrow D$, shown in Figure 9.1.

Please do not change this, as it is the example used for testing.

```python
from variable import Variable
from probFactors import CPD, Prob, LogisticRegression, NoisyOR, ConstantCPD
from probGraphicalModels import BeliefNetwork

#### Simple Example Used for Unit Tests ####
boolean = [False, True]
A = Variable("A", boolean, position=(0,0.8))
B = Variable("B", boolean, position=(0.333,0.7))
C = Variable("C", boolean, position=(0.666,0.6))
D = Variable("D", boolean, position=(1,0.5))
f_a = Prob(A, [], [0.4, 0.6])
f_b = Prob(B, [A], [[0.9, 0.1], [0.2, 0.8]])
f_c = Prob(C, [B], [[0.6, 0.4], [0.3, 0.7]])
f_d = Prob(D, [C], [[0.1, 0.9], [0.75, 0.25]])
bn_4ch = BeliefNetwork("4-chain", {A,B,C,D}, {f_a, f_b, f_c, f_d})
```
Report-of-Leaving Example

The second belief network, bn_report, is Example 9.13 of [http://artint.info](http://artint.info). The output of bn_report.show() is shown in Figure 9.2 of this document.

```python
# Belief network report-of-leaving example (Example 9.13 shown in Figure 9.3) of
# Poole and Mackworth, Artificial Intelligence, 2023 http://artint.info
boolean = [False, True]

Alarm = Variable("Alarm", boolean, position=(0.366,0.5))
Fire = Variable("Fire", boolean, position=(0.633,0.75))
Leaving = Variable("Leaving", boolean, position=(0.366,0.25))
Report = Variable("Report", boolean, position=(0.366,0.0))
Smoke = Variable("Smoke", boolean, position=(0.9,0.5))
Tamper = Variable("Tamper", boolean, position=(0.1,0.75))

f_ta = Prob(Tamper, [], [0.98, 0.02])
f_fi = Prob(Fire, [], [0.99, 0.01])
f_sm = Prob(Smoke, [Fire], [[0.99, 0.01], [0.1, 0.9]])
f_al = Prob(Alarm, [Fire, Tamper], [[0.9999, 0.0001], [0.15, 0.85], [[0.01, 0.99], [0.5, 0.5]]])
f_lv = Prob(Leaving, [Alarm], [[0.999, 0.001], [0.12, 0.88]])
```

[Figure 9.2: The report-of-leaving belief network]
Figure 9.3: Simple diagnosis example; simple_diagnosis.show()

Simple Diagnostic Example

This is the “simple diagnostic example” of Exercise 9.1 of ?, reproduced here as Figure 9.3.

```python
# Belief network simple-diagnostic example (Exercise 9.3 shown in Figure 9.39) of
# Poole and Mackworth, Artificial Intelligence, 2023 http://artint.info

Influenza = Variable("Influenza", boolean, position=(0.4,0.8))
Smokes = Variable("Smokes", boolean, position=(0.8,0.8))
SoreThroat = Variable("Sore Throat", boolean, position=(0.2,0.5))
HasFever = Variable("Fever", boolean, position=(0.4,0.5))
Bronchitis = Variable("Bronchitis", boolean, position=(0.6,0.5))
Coughing = Variable("Coughing", boolean, position=(0.4,0.2))
Wheezing = Variable("Wheezing", boolean, position=(0.8,0.2))

p_infl = Prob(Influenza,[],[0.95,0.05])
p_smokes = Prob(Smokes,[],[0.8,0.2])
p_sth = Prob(SoreThroat,[Influenza],[[0.999,0.001],[0.7,0.3]])
p_fever = Prob(HasFever,[Influenza],[[0.99,0.01],[0.9,0.1]])
p_bronc = Prob(Bronchitis,[Influenza,Smokes],[[0.9999, 0.0001], [0.3, 0.7]], [[0.1, 0.9], [0.01, 0.99]])
```

https://aipython.org Version 0.9.12 December 22, 2023
9.4. Graphical Models

Pearl's Sprinkler Example

Figure 9.4: The sprinkler belief network

The third belief network is the sprinkler example from Pearl. The output of `bn_sprinkler.show()` is shown in Figure 9.4 of this document.

```python
p_cough = Prob(Coughing,[Bronchitis],[[0.93,0.07],[0.2,0.8]])
p_wheeze = Prob(Wheezing,[Bronchitis],[[0.999,0.001],[0.4,0.6]])
simple_diagnosis = BeliefNetwork("Simple Diagnosis",
   {Influenza, Smokes, SoreThroat, HasFever, Bronchitis, Coughing, Wheezing},
   {p_infl, p_smokes, p_sth, p_fever, p_bronc, p_cough, p_wheeze})
```

Sprinkler Example

The third belief network is the sprinkler example from Pearl. The output of `bn_sprinkler.show()` is shown in Figure 9.4 of this document.
Bipartite Diagnostic Model with Noisy-or

The belief network `bn_no1` is a bipartite diagnostic model, with independent diseases, and the symptoms depend on the diseases, where the CPDs are defined using noisy-or. Bipartite means it is in two parts; the diseases are only connected to the symptoms and the symptoms are only connected to the diseases. The output of `bn_no1.show()` is shown in Figure 9.5 of this document.
#### Bipartite Diagnostic Network ####

Cough = Variable("Cough", boolean, (0.1,0.1))
Fever = Variable("Fever", boolean, (0.5,0.1))
Sneeze = Variable("Sneeze", boolean, (0.9,0.1))
Cold = Variable("Cold",boolean, (0.1,0.9))
Flu = Variable("Flu",boolean, (0.5,0.9))
Covid = Variable("Covid",boolean, (0.9,0.9))

p_cold_no = Prob(Cold,[],[0.9,0.1])
p_flu_no = Prob(Flu,[],[0.95,0.05])
p_covid_no = Prob(Covid,[],[0.99,0.01])

p_cough_no = NoisyOR(Cough, [Cold,Flu,Covid], [0.1, 0.3, 0.2, 0.7])
p_fever_no = NoisyOR(Fever, [ Flu,Covid], [0.01, 0.6, 0.7])
p_sneeze_no = NoisyOR(Sneeze, [Cold,Flu ], [0.05, 0.5, 0.2 ])

bn_no1 = BeliefNetwork("Bipartite Diagnostic Network (noisy-or)",
                     {Cough, Fever, Sneeze, Cold, Flu, Covid},
                     {p_cold_no, p_flu_no, p_covid_no, p_cough_no,
                      p_fever_no, p_sneeze_no})

# to see the conditional probability of Noisy-or do:
# print(p_cough_no.to_table())

# example from box "Noisy-or compared to logistic regression"
# X = Variable("X",boolean)
# w0 = 0.01
# print(NoisyOR(X,[A,B,C,D],[w0, 1-(1-0.05)/(1-w0), 1-(1-0.1)/(1-w0),
# 1-(1-0.2)/(1-w0), 1-(1-0.2)/(1-w0), ]).to_table(given={X:True}))

Bipartite Diagnostic Model with Logistic Regression

The belief network bn_lr1 is a bipartite diagnostic model, with independent
diseases, and the symptoms depend on the diseases, where the CPDs are de-
defined using logistic regression. It has the same graphical structure as the pre-
vious example (see Figure 9.5). This has the (approximately) the same con-
ditional probabilities as the previous example when zero or one diseases are
present. Note that sigmoid(−2.2) ≈ 0.1
9. Reasoning with Uncertainty

```python
p_fever_lr = LogisticRegression(Fever, [Flu, Covid], [-4.6, 5.02, 5.46])
p_sneeze_lr = LogisticRegression(Sneeze, [Cold, Flu], [-2.94, 3.04, 1.79])
bn_lr1 = BeliefNetwork("Bipartite Diagnostic Network - logistic regression",
    {Cough, Fever, Sneeze, Cold, Flu, Covid},
    {p_cold_lr, p_flu_lr, p_covid_lr, p_cough_lr, p_fever_lr, p_sneeze_lr})

# to see the conditional probability of Noisy-or do:
# print(p_cough_lr.to_table())

# example from box "Noisy-or compared to logistic regression"
# from learnLinear import sigmoid, logit
# w0=logit(0.01)
# X = Variable("X",boolean)
# print(LogisticRegression(X,[A,B,C,D],[w0, logit(0.05)-w0, logit(0.1)-w0, logit(0.2)-w0]).to_table(given={X:True}))
# try to predict what would happen (and then test) if we had
# w0=logit(0.01)
```

### 9.5 Inference Methods

Each of the inference methods implements the `query` method that computes the posterior probability of a variable given a dictionary of `{variable : value}` observations. The methods are Displayable because they implement the `display` method which is currently text-based.

```python
from display import Displayable
from probExamples import bn_4ch, B, D

class InferenceMethod(Displayable):
    """The abstract class of graphical model inference methods""
    method_name = "unnamed" # each method should have a method name

    def __init__(self, gm=None):
        self.gm = gm

    def query(self, qvar, obs={}):
        """returns a {value:prob} dictionary for the query variable""
        raise NotImplementedError("InferenceMethod query") # abstract method
```

We use `bn_4ch` as the test case, in particular $P(B \mid D = true)$. This needs an error threshold, particularly for the approximate methods, where the default threshold is much too accurate.
9.5. Inference Methods

Report-of-leaving observed: {Report: True}

![Image of a belief network with posterior distributions]

Figure 9.6: The report-of-leaving belief network with posterior distributions

---

def testIM(self, threshold=0.0000000001):
    solver = self(bn_4ch)
    res = solver.query(B,{D:True})
    correct_answer = 0.429632380245
    assert correct_answer-threshold < res[True] <
    correct_answer+threshold, \
    f"value {res[True]} not in desired range for
    {self.method_name}"
    print(f"Unit test passed for {self.method_name}.")

9.5.1 Showing Posterior Distributions

The show_post method draws the posterior distribution of all variables. Figure 9.6 shows the result of bn_reportRC.show_post({Report:True}) when run after loading probRC.py (see below).
9. Reasoning with Uncertainty

facecolor gives the color of the nodes

plt.ion()  # interactive
ax = plt.figure().gca()
ax.set_axis_off()
plt.title(self.gm.title + " observed: "+str(obs), fontsize=fontsize)
bbox = dict(boxstyle="round4,pad=1.0,rounding_size=0.5",
            facecolor=facecolor)
vartext = {}  # variable:text dictionary
for var in self.gm.variables:  # reversed(self.gm.topological_sort()):
    if var in obs:
        text = var.name + "=" + str(obs[var])
    else:
        distn = self.query(var, obs=obs)
        text = var.name + "\n" + "\n".join(str(d)+"+
"+num_format.format(v) for (d,v) in distn.items())
vartext[var] = text
    # Draw arcs
    for par in self.gm.var2parents[var]:
        ax.annotate(text, par.position, xytext=var.position,
                    arrowprops={'arrowstyle': '<-',bbox=bbox, ha='center', va='center',
                                 fontsize=fontsize})
    for var in self.gm.variables:
        x,y = var.position
        plt.text(x,y,vartext[var], bbox=bbox, ha='center', va='center',
                  fontsize=fontsize)

9.6 Naive Search

An instance of a ProbSearch object takes in a graphical model. The query method uses naive search to compute the probability of a query variable given observations on other variables. See Figure 9.9 of ?.

import math
from probGraphicalModels import GraphicalModel, InferenceMethod
from probFactors import Factor

class ProbSearch(InferenceMethod):
    """The class that queries graphical models using recursive conditioning
    """
    gm is graphical model to query
    """
    method_name = "naive search"
    def __init__(self,gm=None):
        InferenceMethod.__init__(self, gm)

https://aipython.org  Version 0.9.12  December 22, 2023
### self.max_display_level = 3

def query(self, qvar, obs={}, split_order=None):
    
    # computes P(qvar | obs) where
    # qvar is the query variable
    # obs is a variable:value dictionary
    # split_order is a list of the non-observed non-query variables in gm
    
    if qvar in obs:
        return {val:(1 if val == obs[qvar] else 0)
                   for val in qvar.domain}
    else:
        if split_order == None:
            split_order = [v for v in self.gm.variables
                            if (v not in obs) and v != qvar]
        unnorm = [self.prob_search({qvar:val}|obs, self.gm.factors,
                                     split_order)
                   for val in qvar.domain]
        p_obs = sum(unnorm)
        return {val:pr/p_obs for val,pr in zip(qvar.domain, unnorm)}

The following is the naive search-based algorithm. It is exponential in the number of variables, so is not very useful. However, it is simple, and useful to understand before looking at the more complicated algorithm used in the subclass.

def prob_search(self, context, factors, split_order):
    
    # simple search algorithm
    # context: a variable:value dictionary
    # factors: a set of factors
    # split_order: list of variables not assigned in context
    # returns sum over variable assignments to variables in split order
    # of product of factors
    
    self.display(2,"calling prob_search",(context,factors,split_order))
    if not factors:
        return 1
    elif to_eval := {fac for fac in factors
                     if fac.can_evaluate(context)):
        # evaluate factors when all variables are assigned
        self.display(3,"prob_search evaluating factors",to_eval)
        val = math.prod(fac.get_value(context) for fac in to_eval)
        return val * self.prob_search(context, factors-to_eval,
                                       split_order)
    else:
        total = 0
        var = split_order[0]
        self.display(3, "prob_search branching on", var)
        for val in var.domain:
            total += self.prob_search({var:val}|context, factors,
                                       split_order[1:])
9.7 Recursive Conditioning

The recursive conditioning algorithm adds forgetting and caching and recognizing disconnected components to the naive search. We do this by adding a cache and redefining the recursive search algorithm. In inherits the query method. See Figure 9.12 of ?.

```python
class ProbRC(ProbSearch):
    method_name = "recursive conditioning"

def __init__(self, gm=None):
    self.cache = {((frozenset(), frozenset()):1}
    ProbSearch.__init__(self, gm)

def prob_search(self, context, factors, split_order):
    """ returns \sum_{split_order} \prod_{factors} given assignment in context
    context is a variable:value dictionary
    factors is a set of factors
    split_order: list of variables in factors that are not in context
    ""
    self.display(3,"calling rc",(context,factors))
    ce = (frozenset(context.items()), frozenset(factors)) # key for the cache entry
    if ce in self.cache:
        self.display(3,"rc cache lookup",(context,factors))
        return self.cache[ce]
    elif to_eval := {fac for fac in factors if fac.can_evaluate(context)}:
        # evaluate factors when all variables are assigned
        val = math.prod(fac.get_value(context) for fac in to_eval)
        if val == 0:
            return 0
        else:
            self.cache[ce] = val
            return val
    elif vars_not_in_factors := {var for var in context if not any(var in fac.variables for fac in factors)}:
        # forget variables not in any factor
        self.display(3,"rc forgetting variables", vars_not_in_factors)
        return self.prob_search({key:val for (key,val) in context.items() if key not in vars_not_in_factors}, factors, split_order)
    elif to_eval := {fac for fac in factors if fac.can_evaluate(context)}:
        # evaluate factors when all variables are assigned
        val = math.prod(fac.get_value(context) for fac in to_eval)
        if val == 0:
            return 0
        else:
            self.cache[ce] = val
            return val
```

https://aipython.org  Version 0.9.12  December 22, 2023
9.7. Recursive Conditioning

```python
    return 0
else:
    return val * self.prob_search(context,
        {fac for fac in factors
            if fac not in to_eval},
        split_order)
elif len(comp := connected_components(context, factors, split_order)) > 1:
    # there are disconnected components
    self.display(3, "splitting into connected components", comp, "in context", context)
    return(math.prod(self.prob_search(context,f,eo) for (f, eo) in comp))
else:
    assert split_order, "split_order should not be empty to get here"
    total = 0
    var = split_order[0]
    self.display(3, "rc branching on", var)
    for val in var.domain:
        total += self.prob_search({var:val}|context, factors, split_order[1:])
    self.cache[ce] = total
    self.display(2, "rc branching on", var, "returning", total)
    return total
```

connected_components returns a list of connected components, where a connected component is a set of factors and a set of variables, where the graph that connects variables and factors that involve them is connected. The connected components are built one at a time; with a current connected component. At all times factors is partitioned into 3 disjoint sets:

- component_factors containing factors in the current connected component where all factors that share a variable are already in the component

- factors_to_check containing factors in the current connected component where potentially some factors that share a variable are not in the component; these need to be checked

- other_factors the other factors that are not (yet) in the connected component

```python
def connected_components(context, factors, split_order):
    """returns a list of (f, e) where f is a subset of factors and e is a subset of split_order such that each element shares the same variables that are disjoint from other elements."
    other_factors = set(factors)  # copies factors
```
factors_to_check = {other_factors.pop()} # factors in connected component still to be checked
component_factors = set() # factors in first connected component already checked
component_variables = set() # variables in first connected component
while factors_to_check:
    next_fac = factors_to_check.pop()
    component_factors.add(next_fac)
    new_vars = set(next_fac.variables) - component_variables - context.keys()
    component_variables |= new_vars
    for var in new_vars:
        factors_to_check |= {f for f in other_factors if var in f.variables}
    other_factors -= factors_to_check # set difference
if other_factors:
    return [(component_factors, [e for e in split_order if e in component_variables])]
else:
    return [(component_factors, split_order)]

Testing:

from probExamples import bn_4ch, A,B,C,D,f_a,f_b,f_c,f_d
bn_4chv = ProbRC(bn_4ch)
## bn_4chv.query(A,{})
## bn_4chv.query(D,{})
## InferenceMethod.max_display_level = 3 # show more detail in displaying
## InferenceMethod.max_display_level = 1 # show less detail in displaying
## bn_4chv.query(A,{D:True},{C,B})
## bn_4chv.query(B,{A:True,D:False})

from probExamples import bn_report,Alarm,Fire,Leaving,Report,Smoke,Tamper
bn_reportRC = ProbRC(bn_report) # answers queries using recursive conditioning
## bn_reportRC.query(Tamper,{})
## InferenceMethod.max_display_level = 0 # show no detail in displaying
## bn_reportRC.query(Leaving,{})
## bn_reportRC.query(Tamper,{},
##    split_order=[Smoke,Fire,Alarm,Leaving,Report])
## bn_reportRC.query(Tamper,{Report:True})
## bn_reportRC.query(Tamper,{Report:True,Smoke:False})

## To display resulting posteriors try:
# bn_reportRC.show_post({})
# bn_reportRC.show_post({Smoke:False})
# bn_reportRC.show_post({Report:True})
from probExamples import bn_sprinkler, Season, Sprinkler, Rained, Grass_wet, Grass_shiny, Shoes_wet
bn_sprinklerv = ProbRC(bn_sprinkler)
bn_sprinklerv.query(Shoes_wet, {Rained:True})
bn_sprinklerv.query(Shoes_wet, {Grass_shiny:True})
bn_sprinklerv.query(Shoes_wet, {Grass_shiny:False, Rained:True})

from probExamples import bn_no1, bn_lr1, Cough, Fever, Sneeze, Cold, Flu, Covid
bn_no1v = ProbRC(bn_no1)
bn_lr1v = ProbRC(bn_lr1)
bn_no1v.query(Flu, {Fever:1, Sneeze:1})
bn_no1v.query(Flu, {Fever:1, Sneeze:1})
bn_no1v.query(Cough, {})
bn_no1v.query(Cold, {Cough:1, Sneeze:0, Fever:1})
bn_no1v.query(Flu, {Cough:0, Sneeze:1, Fever:1})
bn_no1v.query(Covid, {Cough:1, Sneeze:0, Fever:1})
bn_no1v.query(Flu, {Cough:0, Sneeze:1, Fever:1, Flu:0})
bn_no1v.query(Covid, {Cough:1, Sneeze:0, Fever:1, Flu:1})

if __name__ == '__main__':
    InferenceMethod.testIM(ProbSearch)
    InferenceMethod.testIM(ProbRC)

The following example uses the decision tree representation of Section 9.3.4 (page 207). Does recursive conditioning split on variable full for the query commented out below? What can be done to guarantee that it does?

from probFactors import Prob, action, rain, full, wet, p_wet
from probGraphicalModels import BeliefNetwork
p_action = Prob(action,[],{'go_out':0.3, 'get_coffee':0.7})
p_rain = Prob(rain,[],[0.4,0.6])
p_full = Prob(full,[],[0.1,0.9])
wetBN = BeliefNetwork("Wet (decision tree CPD)", {action, rain, full, wet},
    {p_action, p_rain, p_full, p_wet})
wetRC = ProbRC(wetBN)
# wetRC.query(wet, {action:'go_out', rain:True})
# wetRC.show_post({action:'go_out', rain:True})
# wetRC.show_post({action:'go_out', wet:True})
9.8 Variable Elimination

An instance of a VE object takes in a graphical model. The query method uses variable elimination to compute the probability of a variable given observations on some other variables.

```python
from probFactors import Factor, FactorObserved, FactorSum, factor_times
from probGraphicalModels import GraphicalModel, InferenceMethod

class VE(InferenceMethod):
    """The class that queries Graphical Models using variable elimination."
    gm is graphical model to query
    ""
    method_name = "variable elimination"

    def __init__(self,gm=None):
        InferenceMethod.__init__(self, gm)

    def query(self,var,obs={},elim_order=None):
        """computes P(var|obs) where
        var is a variable
        obs is a {variable:value} dictionary"

        if var in obs:
            return {var:1 if val == obs[var] else 0 for val in var.domain}
        else:
            if elim_order == None:
                elim_order = self.gm.variables
            projFactors = [self.project_observations(fact,obs)
                            for fact in self.gm.factors]
            for v in elim_order:
                if v != var and v not in obs:
                    projFactors = self.eliminate_var(projFactors,v)
                    unnorm = factor_times(var,projFactors)
                    p_obs = sum(unnorm)
                    self.display(1,"Unnormalized probs:",unnorm,"Prob obs:",p_obs)
            return {val:pr/p_obs for val,pr in zip(var.domain, unnorm)}
```

A FactorObserved is a factor that is the result of some observations on another factor. We don’t store the values in a list; we just look them up as needed. The observations can include variables that are not in the list, but should have some intersection with the variables in the factor.

```python
class FactorObserved(Factor):
    def __init__(self,factor,obs):
        Factor.__init__(self, [v for v in factor.variables if v not in obs])
        self.observed = obs
        self.orig_factor = factor
```

https://aipython.org
A \textit{FactorSum} is a factor that is the result of summing out a variable from the product of other factors. I.e., it constructs a representation of:

\[
\sum \prod_{\text{var} \in \text{factors}} f. 
\]

We store the values in a list in a lazy manner; if they are already computed, we used the stored values. If they are not already computed we can compute and store them.

The method \textit{factor\_times} multiples a set of factors that are all factors on the same variable (or on no variables). This is the last step in variable elimination before normalizing. It returns an array giving the product for each value of \textit{variable}.

\[
\text{def factor\_times(variable, factors):
    
    """when factors are factors just on variable (or on no variables)""
    
    prods = []
}
To project observations onto a factor, for each variable that is observed in
the factor, we construct a new factor that is the factor projected onto that vari-
able. FactorObserved creates a new factor that is the result is assigning a value
to a single variable.

```python
def project_observations(self, factor, obs):
    """Returns the resulting factor after observing obs
    obs is a dictionary of {variable:value} pairs.
    """
    if any((var in obs) for var in factor.variables):
        # a variable in factor is observed
        return FactorObserved(factor, obs)
    else:
        return factor

def eliminate_var(self, factors, var):
    """Eliminate a variable var from a list of factors.
    Returns a new set of factors that has var summed out.
    """
    contains_var = []
    not_contains_var = []
    for fac in factors:
        if var in fac.variables:
            contains_var.append(fac)
        else:
            not_contains_var.append(fac)
    if contains_var == []:
        return factors
    else:
        newFactor = FactorSum(var, contains_var)
        self.display(2, "Multiplying:", [str(f) for f in contains_var])
        self.display(2, "Creating factor:", newFactor)
        self.display(3, newFactor.to_table()) # factor in detail
        not_contains_var.append(newFactor)
        return not_contains_var
```

```python
from probExamples import bn_4ch, A,B,C,D
bn_4chv = VE(bn_4ch)
## bn_4chv.query(A,{})
## bn_4chv.query(D,{})
## InferenceMethod.max_display_level = 3 # show more detail in displaying
## InferenceMethod.max_display_level = 1 # show less detail in displaying
```
9.9. Stochastic Simulation

9.9.1 Sampling from a discrete distribution

The method `sample_one` generates a single sample from a (possible unnormalized) distribution. `dist` is a `{value : weight}` dictionary, where `weight ≥ 0`. This returns a value with probability in proportion to its weight.

```python
import random

from probGraphicalModels import InferenceMethod

def sample_one(dist):
    """returns the index of a single sample from normalized distribution dist."""
    rand = random.random() * sum(dist.values())
    cum = 0  # cumulative weights
    for value, weight in dist.items():
        cum += weight
        if cum >= rand:
            return value
```

```python
if __name__ == '__main__':
    InferenceMethod.testIM(VE)
```
For \( v \) in \( \text{dist} \):
    \[ \text{cum} += \text{dist}[v] \]
    \[ \text{if} \ \text{cum} > \text{rand} : \]
    \[ \text{return} \ v \]

If we want to generate multiple samples, repeatedly calling \texttt{sample_one} may not be efficient. If we want to generate \( n \) samples, and the distribution is over \( m \) values, \texttt{sample_one} takes time \( O(mn) \). If \( m \) and \( n \) are of the same order of magnitude, we can do better.

The method \texttt{sample_multiple} generates multiple samples from a distribution defined by \( \text{dist} \), where \( \text{dist} \) is a \{\text{value : weight}\} dictionary, where \( \text{weight} \geq 0 \) and the weights cannot all be zero. This returns a list of values, of length \( \text{num_samples} \), where each sample is selected with a probability proportional to its weight.

The method generates all of the random numbers, sorts them, and then goes through the distribution once, saving the selected samples.

```python
def sample_multiple(dist, num_samples):
    """returns a list of num_samples values selected using distribution dist.
    dist is a \{\text{value:weight}\} dictionary that does not need to be normalized
    ""
    total = sum(dist.values())
    rands = sorted(random.random()*total for i in range(num_samples))
    result = []
    dist_items = list(dist.items())
    cum = dist_items[0][1] # cumulative sum
    index = 0
    for r in rands:
        while r > cum:
            index += 1
            cum += dist_items[index][1]
        result.append(dist_items[index][0])
    return result
```

Exercise 9.1
What is the time and space complexity the following 4 methods to generate \( n \) samples, where \( m \) is the length of \( \text{dist} \):
(a) \( n \) calls to \texttt{sample_one}
(b) \texttt{sample_multiple}
(c) Create the cumulative distribution (choose how this is represented) and, for each random number, do a binary search to determine the sample associated with the random number.
(d) Choose a random number in the range \( [i/n, (i+1)/n) \) for each \( i \in \text{range}(n) \), where \( n \) is the number of samples. Use these as the random numbers to select the particles. (Does this give random samples?)
9.9. Stochastic Simulation

For each method suggest when it might be the best method.

The `test_sampling` method can be used to generate the statistics from a number of samples. It is useful to see the variability as a function of the number of samples. Try it for few samples and also for many samples.

```python
from probStochSim import *

def test_sampling(dist, num_samples):
    """Given a distribution, dist, draw num_samples samples
    and return the resulting counts
    """
    result = {v:0 for v in dist}
    for v in sample_multiple(dist, num_samples):
        result[v] += 1
    return result
```

# try the following queries a number of times each:
# test_sampling({1:1,2:2,3:3,4:4}, 100)
# test_sampling({1:1,2:2,3:3,4:4}, 100000)

9.9.2 Sampling Methods for Belief Network Inference

A `SamplingInferenceMethod` is an `InferenceMethod`, but the query method also takes arguments for the number of samples and the sample-order (which is an ordering of factors). The first methods assume a belief network (and not an undirected graphical model).

```python
class SamplingInferenceMethod(InferenceMethod):
    """The abstract class of sampling-based belief network inference methods"
    
    def __init__(self,gm=None):
        InferenceMethod.__init__(self, gm)

    def query(self,qvar,obs={},number_samples=1000,sample_order=None):
        raise NotImplementedError("SamplingInferenceMethod query") # abstract
```

9.9.3 Rejection Sampling

```python
class RejectionSampling(SamplingInferenceMethod):
    """The class that queries Graphical Models using Rejection Sampling.
    gm is a belief network to query
    """
    method_name = "rejection sampling"
```
def __init__(self, gm=None):
    SamplingInferenceMethod.__init__(self, gm)

def query(self, qvar, obs={}, number_samples=1000, sample_order=None):
    """computes P(qvar | obs) where
    qvar is a variable.
    obs is a {variable:value} dictionary.
    sample_order is a list of variables where the parents
    come before the variable.
    """
    if sample_order is None:
        sample_order = self.gm.topological_sort()
    self.display(2,*sample_order,sep="\t")
    counts = {val:0 for val in qvar.domain}
    for i in range(number_samples):
        rejected = False
        sample = {}
        for nvar in sample_order:
            fac = self.gm.var2cpt[nvar] #factor with nvar as child
            val = sample_one({v:fac.get_value(**sample, nvar:v) for v in nvar.domain})
            self.display(2,val,end="\t")
            if nvar in obs and obs[nvar] != val:
                rejected = True
                self.display(2,"Rejected")
                break
            sample[nvar] = val
        if not rejected:
            counts[sample[qvar]] += 1
            self.display(2,"Accepted")
    tot = sum(counts.values())
    # As well as the distribution we also include raw counts
    dist = {c:v/tot if tot>0 else 1/len(qvar.domain) for (c,v) in counts.items()}
    dist["raw_counts"] = counts
    return dist

9.9.4 Likelihood Weighting

Likelihood weighting includes a weight for each sample. Instead of rejecting
samples based on observations, likelihood weighting changes the weights of
the sample in proportion with the probability of the observation. The weight
then becomes the probability that the variable would have been rejected.
method_name = "likelihood weighting"

def __init__(self, gm=None):
    SamplingInferenceMethod.__init__(self, gm)

def query(self, qvar, obs={}, number_samples=1000, sample_order=None):
    """computes P(qvar | obs) where
    qvar is a variable.
    obs is a {variable:value} dictionary.
    sample_order is a list of factors where factors defining the parents
    come before the factors for the child.
    """
    if sample_order is None:
        sample_order = self.gm.topological_sort()
    self.display(2,*[v for v in sample_order
         if v not in obs],sep="\t")
    counts = {val:0 for val in qvar.domain}
    for i in range(number_samples):
        sample = {}
        weight = 1.0
        for nvar in sample_order:
            fac = self.gm.var2cpt[nvar]
            if nvar in obs:
                sample[nvar] = obs[nvar]
                weight *= fac.get_value(sample)
            else:
                val = sample_one({v:fac.get_value(**sample,nvar=v) for
                    v in nvar.domain})
                self.display(2,val,end="\t")
                sample[nvar] = val
                counts[sample[qvar]] += weight
                self.display(2,weight)
        tot = sum(counts.values())
        # as well as the distribution we also include the raw counts
        dist = {c:v/tot for (c,v) in counts.items()}
        dist["raw_counts"] = counts
        return dist

Exercise 9.2 Change this algorithm so that it does importance sampling using a
proposal distribution. It needs sample_one using a different distribution and then
update the weight of the current sample. For testing, use a proposal distribution
that only specifies probabilities for some of the variables (and the algorithm uses
the probabilities for the network in other cases).

9.9.5 Particle Filtering

In this implementation, a particle is a {variable : value} dictionary. Because
adding a new value to dictionary involves a side effect, the dictionaries need
to be copied during resampling.

https://aipython.org
class ParticleFiltering(SamplingInferenceMethod):
    """The class that queries Graphical Models using Particle Filtering.
    gm is a belief network to query
    """
    method_name = "particle filtering"
    
    def __init__(self, gm=None):
        SamplingInferenceMethod.__init__(self, gm)
        
    def query(self, qvar, obs={}, number_samples=1000, sample_order=None):
        """computes P(qvar | obs) where
        qvar is a variable.
        obs is a {variable:value} dictionary.
        sample_order is a list of factors where factors defining the parents
        come before the factors for the child.
        """
        if sample_order is None:
            sample_order = self.gm.topological_sort()
        self.display(2,*[v for v in sample_order
                if v not in obs],sep="\t")
        particles = [{}, for i in range(number_samples)]
        for nvar in sample_order:
            fac = self.gm.var2cpt[nvar]
            if nvar in obs:
                weights = [fac.get_value(**part, nvar:obs[nvar])
                    for part in particles]
                particles = [{**p, nvar:obs[nvar]}
                    for p in resample(particles, weights,
                    number_samples)]
            else:
                for part in particles:
                    part[nvar] = sample_one({v:fac.get_value(**part,
                        nvar:v})
                        for v in nvar.domain})
                self.display(2,part[nvar],end="\t")
        counts = {val:0 for val in qvar.domain}
        for part in particles:
            counts[part[qvar]] += 1
        tot = sum(counts.values())
        # as well as the distribution we also include the raw counts
        dist = {c:v/tot for (c,v) in counts.items()}
        dist["raw_counts"] = counts
        return dist

Resampling

Resample is based on sample_multiple but works with an array of particles. (Aside: Python doesn’t let us use sample_multiple directly as it uses a dictionary,

https://aipython.org
and particles, represented as dictionaries can’t be the key of dictionaries).

```python
def resample(particles, weights, num_samples):
    """returns num_samples copies of particles resampled according to weights.
    particles is a list of particles
    weights is a list of positive numbers, of same length as particles
    num_samples is n integer
    """
    total = sum(weights)
    rands = sorted(random.random()*total for i in range(num_samples))
    result = []
    cum = weights[0]  # cumulative sum
    index = 0
    for r in rands:
        while r>cum:
            index += 1
            cum += weights[index]
        result.append(particles[index])
    return result
```

### 9.9.6 Examples

```python
from probExamples import bn_4ch, A,B,C,D
bn_4chr = RejectionSampling(bn_4ch)
bn_4chL = LikelihoodWeighting(bn_4ch)
# InferenceMethod.max_display_level = 2  # detailed tracing for all inference methods
# bn_4chr.query(A,{}
# bn_4chr.query(C,{})
# bn_4chr.query(A,{C:True})
# bn_4chr.query(B,{A:True,C:False})

from probExamples import bn_report, Alarm, Fire, Leaving, Report, Smoke, Tamper
bn_reportr = RejectionSampling(bn_report)  # answers queries using rejection sampling
bn_reportL = LikelihoodWeighting(bn_report)  # answers queries using likelihood weighting
bn_reportp = ParticleFiltering(bn_report)  # answers queries using particle filtering
# bn_reportr.query(Tamper,{})
# bn_reportr.query(Tamper,{})
# bn_reportr.query(Tamper,{}Report:True})
# InferenceMethod.max_display_level = 0  # no detailed tracing for all inference methods
# bn_reportr.query(Tamper,{}Report:True},number_samples=100000)
# bn_reportr.query(Tamper,{}Report:True,Smoke:False})
```
9. Reasoning with Uncertainty

```python
# bn_reportr.query(Tamper, {Report: True, Smoke: False}, number_samples=100)
# bn_reportr.query(Tamper, {Report: True, Smoke: False}, number_samples=100)
# bn_reportr.query(Tamper, {Report: True, Smoke: False}, number_samples=100)
from probExamples import bn_sprinkler, Season, Sprinkler
from probExamples import Rained, Grass_wet, Grass_shiny, Shoes_wet
bn_sprinkler = RejectionSampling(bn_sprinkler)  # answers queries using rejection sampling
bn_sprinklerL = LikelihoodWeighting(bn_sprinkler)  # answers queries using rejection sampling
bn_sprinklerp = ParticleFiltering(bn_sprinkler)  # answers queries using particle filtering
#bn_sprinklerL.query(Shoes_wet, {Grass_shiny: True, Rained: True})
#bn_sprinklerL.query(Shoes_wet, {Grass_shiny: True, Rained: True})
#bn_sprinklerp.query(Shoes_wet, {Grass_shiny: True, Rained: True})

if __name__ == "__main__":
    InferenceMethod.testIM(RejectionSampling, threshold=0.1)
    InferenceMethod.testIM(LikelihoodWeighting, threshold=0.1)
    InferenceMethod.testIM(ParticleFiltering, threshold=0.1)
```

Exercise 9.3  This code keeps regenerating the distribution of a variable given its parents. Implement one or both of the following, and compare them to the original. Make `cond_dist` return a slice that corresponds to the distribution, and then use the slice instead of the dictionary (a list slice does not generate new data structures). Make `cond_dist` remember values it has already computed, and only return these.

9.9.7 Gibbs Sampling

The following implements Gibbs sampling, a form of Markov Chain Monte Carlo MCMC.

```python
# probStochSim.py — (continued)

from random
from probGraphicalModels import InferenceMethod
from probStochSim import sample_one, SamplingInferenceMethod

class GibbsSampling(SamplingInferenceMethod):
    """The class that queries Graphical Models using Gibbs Sampling."
    bn is a graphical model (e.g., a belief network) to query """
    method_name = "Gibbs sampling"

    def __init__(self, gm=None):
        SamplingInferenceMethod.__init__(self, gm)
        self.gm = gm

https://aipython.org
```
def query(self, qvar, obs={}, number_samples=1000, burn_in=100, sample_order=None):
    """computes P(qvar | obs) where
    qvar is a variable.
    obs is a {variable:value} dictionary.
    sample_order is a list of non-observed variables in order, or
    if sample_order None, an arbitrary ordering is used
    """
    counts = {val: 0 for val in qvar.domain}
    if sample_order is not None:
        variables = sample_order
    else:
        variables = [v for v in self.gm.variables if v not in obs]
        random.shuffle(variables)
    var_to_factors = {v: set() for v in self.gm.variables}
    for fac in self.gm.factors:
        for var in fac.variables:
            var_to_factors[var].add(fac)
    sample = {var: random.choice(var.domain) for var in variables}
    self.display(3, "Sample:", sample)
    sample.update(obs)
    for i in range(burn_in + number_samples):
        for var in variables:
            # get unnormalized probability distribution of var given its
            # neighbors
            vardist = {val: 1 for val in var.domain}
            for val in var.domain:
                sample[var] = val
                for fac in var_to_factors[var]:  # Markov blanket
                    vardist[val] *= fac.get_value(sample)
                sample[var] = sample_one(vardist)
            if i >= burn_in:
                counts[sample[qvar]] += 1
                self.display(3, " ", sample)
        tot = sum(counts.values())
    # as well as the computed distribution, we also include raw counts
    dist = {c:v/tot for (c,v) in counts.items()}
    dist["raw_counts"] = counts
    self.display(2, f"Gibbs sampling P({qvar}|{obs}) = {dist}")
    return dist
from probExamples import bn_report, Alarm, Fire, Leaving, Report, Smoke, Tamper
bn_reportg = GibbsSampling(bn_report)
# bn_reportg.query(Tamper, {Report: True}, number_samples=1000)
if __name__ == "__main__":
    InferenceMethod.testIM(GibbsSampling, threshold=0.1)

Exercise 9.4 Change the code so that it can have multiple query variables. Make the list of query variable be an input to the algorithm, so that the default value is the list of all non-observed variables.

Exercise 9.5 In this algorithm, explain where it computes the probability of a variable given its Markov blanket. Instead of returning the average of the samples for the query variable, it is possible to return the average estimate of the probability of the query variable given its Markov blanket. Does this converge to the same answer as the given code? Does it converge faster, slower, or the same?

9.9.8 Plotting Behavior of Stochastic Simulators

The stochastic simulation runs can give different answers each time they are run. For the algorithms that give the same answer in the limit as the number of samples approaches infinity (as do all of these algorithms), the algorithms can be compared by comparing the accuracy for multiple runs. Summary statistics like the variance may provide some information, but the assumptions behind the variance being appropriate (namely that the distribution is approximately Gaussian) may not hold for cases where the predictions are bounded and often skewed.

It is more appropriate to plot the distribution of predictions over multiple runs. The plot_stats method plots the prediction of a particular variable (or for the partition function) for a number of runs of the same algorithm. On the x-axis, is the prediction of the algorithm. On the y-axis is the number of runs with prediction less than or equal to the x value. Thus this is like a cumulative distribution over the predictions, but with counts on the y-axis.

Note that for runs where there are no samples that are consistent with the observations (as can happen with rejection sampling), the prediction of probability is 1.0 (as a convention for 0/0).

That variable "what contains the query variable, or what is "prob.eo", the probability of evidence.

import matplotlib.pyplot as plt
def plot_stats(method, qvar, qval, obs, number_runs=1000, **queryargs):
    """Plots a cumulative distribution of the prediction of the model.
    method is a InferenceMethod (that implements appropriate query(.))
    plots P(qvar=qval | obs)
    qvar is the query variable, qval is corresponding value
obs is the \{variable:value\} dictionary representing the observations
number_iterations is the number of runs that are plotted
**queryargs is the arguments to query (often number_samples for
sampling methods)

```python
plt.ion()
plt.xlabel("value")
plt.ylabel("Cumulative Number")
method.max_display_level, prev_mdl = 0, method.max_display_level #no
display
answers = [method.query(qvar,obs,**queryargs)
    for i in range(number_runs)]
values = [ans[qval] for ans in answers]
label = f"""{method.method_name}\
    P({qvar}={qval}|\'''.join(f'\{var}={val}'
    for (var,val) in
    obs.items()))""
values.sort()
plt.plot(values,range(number_runs),label=label)
plt.legend() #loc="upper left")
plt.draw()
method.max_display_level = prev_mdl # restore display level

# Try:
# plot_stats(bn_reportr,Tamper,True,{Report:True,Smoke:True},
# number_samples=1000, number_runs=1000)
# plot_stats(bn_reportl,Tamper,True,{Report:True,Smoke:True},
# number_samples=1000, number_runs=1000)
# plot_stats(bn_reportp,Tamper,True,{Report:True,Smoke:True},
# number_samples=1000, number_runs=1000)
# plot_stats(bn_reportr,Tamper,True,{Report:True,Smoke:True},
# number_samples=100, number_runs=1000)
# plot_stats(bn_reportl,Tamper,True,{Report:True,Smoke:True},
# number_samples=100, number_runs=1000)
# plot_stats(bn_reportg,Tamper,True,{Report:True,Smoke:True},
# number_samples=1000, number_runs=1000)

def plot_mult(methods, example, qvar, qval, obs, number_samples=1000,
number_runs=1000):
    for method in methods:
        solver = method(example)
        if isinstance(method,SamplingInferenceMethod):
            plot_stats(solver, qvar, qval, obs,
            number_samples=number_samples, number_runs=number_runs)
        else:
            plot_stats(solver, qvar, qval, obs, number_runs=number_runs)
```

```
from probRC import ProbRC
# Try following (but it takes a while..)
methods =
```
9. Reasoning with Uncertainty

[ProbRC, RejectionSampling, LikelihoodWeighting, ParticleFiltering, GibbsSampling]

```
#plot_mult(methods, bn_report, Tamper, True, {Report: True, Smoke: False}, number_samples=100,
number_runs=1000)
```

```
# Sprinkler Example:
```

```
# plot_stats(bn_sprinklerR, Shoes_wet, True, {Grass_shiny: True, Rained: True}, number_samples=1000)
```

```
# plot_stats(bn_sprinklerL, Shoes_wet, True, {Grass_shiny: True, Rained: True}, number_samples=1000)
```

9.10 Hidden Markov Models

This code for hidden Markov models is independent of the graphical models code, to keep it simple. Section 9.11 gives code that models hidden Markov models, and more generally, dynamic belief networks, using the graphical models code.

This HMM code assumes there are multiple Boolean observation variables that depend on the current state and are independent of each other given the state.

```python
import random
from probStochSim import sample_one, sample_multiple

class HMM(object):
    def __init__(self, states, obsvars, pobs, trans, indist):
        """A hidden Markov model.
        states - set of states
        obsvars - set of observation variables
        pobs - probability of observations, pobs[i][s] is P(Obs_i=True | State=s)
        trans - transition probability - trans[i][j] gives P(State=j | State=i)
        indist - initial distribution - indist[s] is P(State_0 = s)
        ""
        self.states = states
        self.obsvars = obsvars
        self.pobs = pobs
        self.trans = trans
        self.indist = indist
```

Consider the following example. Suppose you want to unobtrusively keep track of an animal in a triangular enclosure using sound. Suppose you have 3 microphones that provide unreliable (noisy) binary information at each time
step. The animal is either close to one of the 3 points of the triangle or in the middle of the triangle.

The observation model is as follows. If the animal is in a corner, it will be detected by the microphone at that corner with probability 0.6, and will be independently detected by each of the other microphones with a probability of 0.1. If the animal is in the middle, it will be detected by each microphone with a probability of 0.4.

The transition model is as follows: If the animal is in a corner it stays in the same corner with probability 0.80, goes to the middle with probability 0.1 or goes to one of the other corners with probability 0.05 each. If it is in the middle, it stays in the middle with probability 0.7, otherwise it moves to one of the corners, each with probability 0.1.

Initially the animal is in one of the four states, with equal probability.
52 indist1 = {st: 1.0 / len(states1) for st in states1}
53 hmm1 = HMM(states1, obs1, pobs1, trans1, indist1)

9.10.1 Exact Filtering for HMMs

A HMM filter has a current state distribution which can be updated by observing or by advancing to the next time.

```python
from display import Displayable

class HMMVEfilter(Displayable):
    def __init__(self, hmm):
        self.hmm = hmm
        self.state_dist = hmm.indist

    def filter(self, obsseq):
        """updates and returns the state distribution following the sequence of observations in obsseq using variable elimination."

        Note that it first advances time.
        This is what is required if it is called sequentially.
        If that is not what is wanted initially, do an observe first.

        """
        for obs in obsseq:
            self.advance()  # advance time
            self.observe(obs)  # observe
        return self.state_dist

    def observe(self, obs):
        """updates state conditioned on observations.
        obs is a list of values for each observation variable""
        for i in self.hmm.obsvars:
            self.state_dist = {st: self.state_dist[st] * (self.hmm.pobs[i][st]
                if obs[i] else (1-self.hmm.pobs[i][st]))
                for st in self.hmm.states}
        norm = sum(self.state_dist.values())  # normalizing constant
        self.state_dist = {st: self.state_dist[st] / norm for st in self.hmm.states}
        self.display(2, "After observing", obs, "state distribution:", self.state_dist)

    def advance(self):
        """advance to the next time""
        nextstate = {st: 0.0 for st in self.hmm.states}  # distribution over next states
        for j in self.hmm.states:  # j ranges over next states
```

https://aipython.org

Version 0.9.12  December 22, 2023
for i in self.hmm.states: # i ranges over previous states
    nextstate[j] += self.hmm.trans[i][j]*self.state_dist[i]
self.state_dist = nextstate
self.display(2,"After advancing state
distribution:",self.state_dist)

The following are some queries for \textit{hmm1}.

Exercise 9.6 The representation assumes that there are a list of Boolean observations. Extend the representation so that the each observation variable can have multiple discrete values. You need to choose a representation for the model, and change the algorithm.

9.10.2 Localization

The localization example in the book is a controlled HMM, where there is a given action at each time and the transition depends on the action.
Instead of the transition probability, it has a function act2trans from action to transition probability.
Any algorithms need to select the transition probability according to the action.

```python
def __init__(self, states, obsvars, pobs, act2trans, indist):
    self.act2trans = act2trans
    HMM.__init__(self, states, obsvars, pobs, None, indist)

local_states = list(range(16))
door_positions = {2, 4, 7, 11}
def prob_door(loc):
    return 0.8 if loc in door_positions else 0.1
local_obs = {'door': [prob_door(i) for i in range(16)]}
act2trans = {'right': [[0.1 if next == current
    else 0.8 if next == (current+1)%16
    else 0.074 if next == (current+2)%16
    else 0.002 for next in range(16)]
    for current in range(16)],
    'left': [[0.1 if next == current
        else 0.8 if next == (current-1)%16
        else 0.074 if next == (current-2)%16
        else 0.002 for next in range(16)]
    for current in range(16)]

hmm_16pos = HMM_Controlled(local_states, {'door'}, local_obs,
act2trans, [1/16 for i in range(16)])

To change the VE localization code to allow for controlled HMMs, notice that the action selects which transition probability to us.

```

```
def __init__(self, hmm, fontsize=10):
    self.hmm = hmm
    self.fontsize = fontsize
    self.loc_filt = HMM_Local(hmm)
    fig, (self.ax) = plt.subplots()
    plt.subplots_adjust(bottom=0.2)
    ## Set up buttons:
    left_butt = Button(plt.axes([0.05, 0.02, 0.1, 0.05]), "left")
    left_butt.label.set_fontsize(self.fontsize)
    left_butt.on_clicked(self.left)
    right_butt = Button(plt.axes([0.25, 0.02, 0.1, 0.05]), "right")
    right_butt.label.set_fontsize(self.fontsize)
    right_butt.on_clicked(self.right)
    door_butt = Button(plt.axes([0.45, 0.02, 0.1, 0.05]), "door")
    door_butt.label.set_fontsize(self.fontsize)
    door_butt.on_clicked(self.door)
    nodoor_butt = Button(plt.axes([0.65, 0.02, 0.1, 0.05]), "no door")
    nodoor_butt.label.set_fontsize(self.fontsize)
    nodoor_butt.on_clicked(self.nodoor)
    reset_butt = Button(plt.axes([0.85, 0.02, 0.1, 0.05]), "reset")
    reset_butt.label.set_fontsize(self.fontsize)
    reset_butt.on_clicked(self.reset)
    ## draw the distribution
    plt.subplot(1, 1, 1)
    self.draw_dist()
    plt.show()

def draw_dist(self):
    self.ax.clear()
    plt.ylim(0, 1)
    plt.ylabel("Probability", fontsize=self.fontsize)
    plt.xlabel("Location", fontsize=self.fontsize)
    plt.title("Location Probability Distribution",
              fontsize=self.fontsize)
    plt.xticks(self.hmm.states, fontsize=self.fontsize)
    plt.yticks(fontsize=self.fontsize)
    vals = [self.loc_filt.state_dist[i] for i in self.hmm.states]
    self.bars = self.ax.bar(self.hmm.states, vals, color='black')
    self.ax.bar_label(self.bars,["{v:.2f}".format(v=v) for v in vals],
                       padding = 1, fontsize=self.fontsize)
    plt.draw()

def left(self, event):
    self.loc_filt.go("left")
    self.draw_dist()

def right(self, event):
    self.loc_filt.go("right")
    self.draw_dist()

def door(self, event):
    self.loc_filt.observe({'door':True})
9. Reasoning with Uncertainty

```python
self.draw_dist()
def noodoor(self, event):
    self.loc_filt.observe({'door':False})
    self.draw_dist()
def reset(self, event):
    self.loc_filt.state_dist = {i:1/16 for i in range(16)}
    self.draw_dist()

# sl = Show_Localization(hmm_16pos)
# sl = Show_Localization(hmm_16pos, fontsize=15) # for demos - enlarge window
```

9.10.3 Particle Filtering for HMMs

In this implementation a particle is just a state. If you want to do some form of smoothing, a particle should probably be a history of states. This maintains, 
*particles*, an array of states, *weights* an array of (non-negative) real numbers, such that *weights[i]* is the weight of *particles[i]*.

```python
from display import Displayable
from probStochSim import resample

class HMMparticleFilter(Displayable):
    def __init__(self, hmm, number_particles=1000):
        self.hmm = hmm
        self.particles = [sample_one(hmm.indist)
                         for i in range(number_particles)]
        self.weights = [1 for i in range(number_particles)]

    def filter(self, obsseq):
        """returns the state distribution following the sequence of
        observations in obsseq using particle filtering.

        Note that it first advances time.
        This is what is required if it is called after previous filtering.
        If that is not what is wanted initially, do an observe first.
        """
        for obs in obsseq:
            self.advance() # advance time
            self.observe(obs) # observe
            self.resample_particles()
            self.display(2, "After observing", str(obs),
                         "state distribution:",
                         self.histogram(self.particles))
            self.display(1, "Final state distribution:",
                         self.histogram(self.particles))
        return self.histogram(self.particles)
```

[https://aipython.org](https://aipython.org)  Version 0.9.12  December 22, 2023
def advance(self):
    """advance to the next time.
    This assumes that all of the weights are 1.""
    self.particles = [sample_one(self.hmm.trans[st])
                     for st in self.particles]

def observe(self, obs):
    """reweighs the particles to incorporate observations obs""
    for i in range(len(self.particles)):
        for obv in obs:
            if obs[obv]:
                self.weights[i] *= self.hmm.pobs[obv][self.particles[i]]
            else:
                self.weights[i] *= 1-self.hmm.pobs[obv][self.particles[i]]

def histogram(self, particles):
    """returns list of the probability of each state as represented by
    the particles""
    tot=0
    hist = {st: 0.0 for st in self.hmm.states}
    for (st,wt) in zip(particles,self.weights):
        hist[st]+=wt
    tot += wt
    return {st:hist[st]/tot for st in hist}

def resample_particles(self):
    """resamples to give a new set of particles.""
    self.particles = resample(self.particles, self.weights, len(self.particles))
    self.weights = [1] * len(self.particles)

The following are some queries for hmm1.

probHMM.py — (continued)

# probHMM.py — (continued)

hmm1pf1 = HMMparticleFilter(hmm1)
# HMMparticleFilter.max_display_level = 2 # show each step
# hmm1pf1.filter([{'m1':0, 'm2':1, 'm3':1}, {'m1':1, 'm2':0, 'm3':1}])
# hmm1pf2 = HMMparticleFilter(hmm1)
# hmm1pf2.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':1, 'm3':0},
#                 {'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0},
#                 {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0},
#                 {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':1},
#                 {'m1':0, 'm2':0, 'm3':1}])
# hmm1pf3 = HMMparticleFilter(hmm1)
# hmm1pf3.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0},
#                 {'m1':1, 'm2':0, 'm3':0}, {'m1':1, 'm2':0, 'm3':1}])

Exercise 9.7 A form of importance sampling can be obtained by not resampling.

https://aipython.org
Is it better or worse than particle filtering? Hint: you need to think about how they can be compared. Is the comparison different if there are more states than particles?

**Exercise 9.8** Extend the particle filtering code to continuous variables and observations. In particular, suppose the state transition is a linear function with Gaussian noise of the previous state, and the observations are linear functions with Gaussian noise of the state. You may need to research how to sample from a Gaussian distribution.

### 9.10.4 Generating Examples

The following code is useful for generating examples.

```python
def simulate(hmm, horizon):
    """returns a pair of (state sequence, observation sequence) of length horizon."
    for each time t, the agent is in state_sequence[t] and observes observation_sequence[t]
    """
    state = sample_one(hmm.indist)
    obsseq=[]
    stateseq=[]
    for time in range(horizon):
        stateseq.append(state)
        newobs =
        {obs:sample_one({0:1-hmm.pobs[obs][state],1:hmm.pobs[obs][state]})
            for obs in hmm.obsvars}
        obsseq.append(newobs)
        state = sample_one(hmm.trans[state])
    return stateseq,obsseq

def simobs(hmm, stateseq):
    """returns observation sequence for the state sequence""
    obsseq=[]
    for state in stateseq:
        newobs =
        {obs:sample_one({0:1-hmm.pobs[obs][state],1:hmm.pobs[obs][state]})
            for obs in hmm.obsvars}
        obsseq.append(newobs)
    return obsseq

def create_eg(hmm,n):
    """Create an annotated example for horizon n""
    seq,obs = simulate(hmm,n)
    print("True state sequence:",seq)
    print("Sequence of observations:\n",obs)
    hmmfilter = HMMVEfilter(hmm)
    dist = hmmfilter.filter(obs)
```

[https://aipython.org](https://aipython.org)
9.11 Dynamic Belief Networks

A dynamic belief network (DBN) is a belief network that extends in time.
There are a number of ways that reasoning can be carried out in a DBN, including:

- Rolling out the DBN for some time period, and using standard belief network inference. The latest time that needs to be in the rolled out network is the time of the latest observation or the time of a query (whichever is later). This allows us to observe any variables at any time and query any variables at any time. This is covered in Section 9.11.2.

- An unrolled belief network may be very large, and we might only be interested in asking about “now”. In this case we can just representing the variables “now”. In this approach we can observe and query the current variables. We can them move to the next time. This does not allow for arbitrary historical queries (about the past or the future), but can be much simpler. This is covered in Section 9.11.3.

9.11.1 Representing Dynamic Belief Networks

To specify a DBN, think about the distribution now. Now will be represented as time 1. Each variable will have a corresponding previous variable; these will be created together.

A dynamic belief network consists of:

- A set of features. A variable is a feature-time pair.

- An initial distribution over the features “now” (time 1). This is a belief network with all variables being time 1 variables.

- A specification of the dynamics. We define the how the variables now (time 1) depend on variables now and the previous time (time 0), in such a way that the graph is acyclic.
A variable can have both a name and an index. The index defaults to 1.

```python
def __init__(self,name,domain=[False,True],index=1):
    Variable.__init__(self,f"{name}_{index}",domain)
    self.basename = name
    self.domain = domain
    self.index = index
    self.previous = None

def __lt__(self,other):
    if self.name != other.name:
        return self.name<other.name
    else:
        return self.index<other.index

def __gt__(self,other):
    return other<self

def variable_pair(name,domain=[False,True]):
    """returns a variable and its predecessor. This is used to define 2-stage DBNs"
    var_now = DBNvariable(name,domain,index='now')
    var_prev = DBNvariable(name,domain,index='prev')
    var_now.previous = var_prev
    return var_prev, var_now
```

A FactorRename is a factor that is the result renaming the variables in the factor. It takes a factor, fac, and a \{new : old\} dictionary, where new is the name of a variable in the resulting factor and old is the corresponding name in fac. This assumes that the all variables are renamed.

```python
class FactorRename(Factor):
    def __init__(self,fac,renaming):
        """A renamed factor.
        fac is a factor
        renaming is a dictionary of the form {new:old} where old and new var variables,
        where the variables in fac appear exactly once in the renaming
        """
        Factor.__init__(self,[n for (n,o) in renaming.items() if o in fac.variables])
        self.orig_fac = fac
        self.renaming = renaming

    def get_value(self,assignment):
        return self.orig_fac.get_value({self.renaming[var]:val for (var,val) in assignment.items()}
```
The following class renames the variables of a conditional probability distribution. It is used for template models (e.g., dynamic decision networks or relational models).

```python
class CPDrename(FactorRename, CPD):
    def __init__(self, cpd, renaming):
        renaming_inverse = {old:new for (new,old) in renaming.items()}
        CPD.__init__(self,renaming_inverse[cpd.child],[renaming_inverse[p] for p in cpd.parents])
        self.orig_fac = cpd
        self.renaming = renaming
```

```python
class DBN(Displayable):
    """The class of stationary Dynamic Belief networks.
    * name is the DBN name
    * vars_now is a list of current variables (each must have previous variable).
    * transition_factors is a list of factors for P(X|parents) where X is a current variable and parents is a list of current or previous variables.
    * init_factors is a list of factors for P(X|parents) where X is a current variable and parents can only include current variables
    The graph of transition factors + init factors must be acyclic.
    ""
    def __init__(self, title, vars_now, transition_factors=None, init_factors=None):
        self.title = title
        self.vars_now = vars_now
        self.vars_prev = [v.previous for v in vars_now]
        self.transition_factors = transition_factors
        self.init_factors = init_factors
        self.var_index = {} # var_index[v] is the index of variable v
        for i,v in enumerate(vars_now):
            self.var_index[v]=i
```

Here is a 3 variable DBN:

```python
A0,A1 = variable_pair("A", domain=[False,True])
B0,B1 = variable_pair("B", domain=[False,True])
C0,C1 = variable_pair("C", domain=[False,True])
# dynamics
pc = Prob(C1,[B1,C0],([[0.03,0.97],[0.38,0.62]], [[0.23,0.77],[0.78,0.22]]))
pb = Prob(B1,[A0,A1],([[0.5,0.5],[0.77,0.23]], [[0.4,0.6],[0.83,0.17]]))
pa = Prob(A1,[A0,B0],([[0.1,0.9],[0.65,0.35]], [[0.3,0.7],[0.8,0.2]]))
```

[https://aipython.org](https://aipython.org)
# initial distribution
pa0 = Prob(A1,[],[0.9,0.1])
pb0 = Prob(B1,[A1],[[0.3,0.7],[0.8,0.2]])
pc0 = Prob(C1,[],[0.2,0.8])
dbn1 = DBN("Simple DBN",[A1,B1,C1],[pa,pb,pc],[pa0,pb0,pc0])

Here is the animal example

```python
from probHMM import closeMic, farMic, midMic, sm, mmc, sc, mcm, mcc

Pos_0,Pos_1 = variable_pair("Position",domain=[0,1,2,3])
Mic1_0,Mic1_1 = variable_pair("Mic1")
Mic2_0,Mic2_1 = variable_pair("Mic2")
Mic3_0,Mic3_1 = variable_pair("Mic3")

# conditional probabilities - see hmm for the values of sm,mmc, etc
ppos = Prob(Pos_1, [Pos_0],
            [[sm, mmc, mmc, mmc], #was in middle
             [mcm, sc, mcc, mcc], #was in corner 1
             [mcm, mcc, sc, mcc], #was in corner 2
             [mcm, mcc, mcc, sc]]) #was in corner 3
pm1 = Prob(Mic1_1, [Pos_1], [[1-midMic, midMic],
                              [1-closeMic, closeMic],
                              [1-farMic, farMic],
                              [1-farMic, farMic]])
pm2 = Prob(Mic2_1, [Pos_1], [[1-midMic, midMic],
                              [1-closeMic, closeMic],
                              [1-farMic, farMic],
                              [1-farMic, farMic]])
pm3 = Prob(Mic3_1, [Pos_1], [[1-midMic, midMic],
                              [1-farMic, farMic],
                              [1-farMic, farMic],
                              [1-closeMic, closeMic]])
ipos = Prob(Pos_1,[], [0.25, 0.25, 0.25, 0.25])
dbn_an =DBN("Animal DBN",[Pos_1,Mic1_1,Mic2_1,Mic3_1],
            [ppos, pm1, pm2, pm3],
            [ipos, pm1, pm2, pm3])
```

9.11.2 Unrolling DBNs

```python
class BNfromDBN(BeliefNetwork):
    """Belief Network unrolled from a dynamic belief network
    """

def __init__(self,dbn,horizon):
    """dbn is the dynamic belief network being unrolled
    horizon>0 is the number of steps (so there will be horizon+1
    variables for each DBN variable.
    """
    self.name2var = {var.basename:
                      [DBNvariable(var.basename,var.domain,index) for index in
                       range(horizon+1)]}
```

https://aipython.org
for var in dbn.vars_now
    self.display(1,f"name2var={self.name2var}"
variables = {v for vs in self.name2var.values() for v in vs
self.display(1,f"variables={variables}"
bnfactors = {CPDrename(fac,{self.name2var[var.basename][0]:var
    for var in fac.variables})
    for fac in dbn.init_factors}
bnfactors |= {CPDrename(fac,{self.name2var[var.basename][i]:var
    for var in fac.variables if
        var.index=='prev'}]
        {self.name2var[var.basename][i+1]:var
     for var in fac.variables if
        var.index=='now'})
    for fac in dbn.transition_factors
        for i in range(horizon)
            self.display(1,f"bnfactors={bnfactors}"
BeliefNetwork.__init__(self, dbn.title, variables, bnfactors)

Here are two examples. Note that we need to use bn.name2var['B'][2] to
get the variable B2 (B at time 2).

# Try
from probRC import ProbRC
#bn = BNfromDBN(dbn1,2) # construct belief network
#drc = ProbRC(bn) # initialize recursive conditioning
#B2 = bn.name2var['B'][2]
#drc.query(B2) #P(B2)
#drc.query(bn.name2var['B'][1],{bn.name2var['B'][0]:True,bn.name2var['C'][1]:False})
    #P(B1|B0,C1)

9.11.3 DBN Filtering

If we only wanted to ask questions about the current state, we can save space
by forgetting the history variables.

class DBNVEfilter(VE):
    def __init__(self,dbn):
        self.dbn = dbn
        self.current_factors = dbn.init_factors
        self.current_obs = {}

    def observe(self, obs):
        """updates the current observations with obs.
        obs is a variable:value dictionary where variable is a current
        variable."""
        assert all(self.current_obs[var]==obs[var] for var in obs
            if var in self.current_obs),"inconsistent current
            observations"
self.current_obs.update(obs) # note 'update' is a dict method

def query(self, var):
    """returns the posterior probability of current variable var"""
    return
    VE(GraphicalModel(self.dbn.title, self.dbn.vars_now, self.current_factors)).query(var, self.current_obs)

def advance(self):
    """advance to the next time"""
    prev_factors = [self.make_previous(fac) for fac in self.current_factors]
    prev_obs = {var.previous: val for var, val in self.current_obs.items()}
    two_stage_factors = prev_factors + self.dbn.transition_factors
    self.current_factors = self.elim_vars(two_stage_factors, self.dbn.vars_prev, prev_obs)
    self.current_obs = {}

def make_previous(self, fac):
    """Creates new factor from fac where the current variables in fac
    are renamed to previous variables.
    """
    return FactorRename(fac, {var.previous: var for var in fac.variables})

def elim_vars(self, factors, vars, obs):
    for var in vars:
        if var in obs:
            factors = [self.project_observations(fac, obs) for fac in factors]
        else:
            factors = self.eliminate_var(factors, var)
    return factors

Example queries:

# df = DBNVEfilter(dbn1)
# df.observe({B1:True}); df.advance(); df.observe({C1:False})
# df.query(B1) #P(B1|B0,C1)
# df.advance(); df.query(B1)
# dfa = DBNVEfilter(dbn_an)
# dfa.observe({Mic1_1:0, Mic2_1:1, Mic3_1:1})
# dfa.advance()
# dfa.observe({Mic1_1:1, Mic2_1:0, Mic3_1:1})
# dfa.query(Pos_1)
Chapter 10

Learning with Uncertainty

10.1 Bayesian Learning

The section contains two implementations of the (discretized) beta distribution. The first represents Bayesian learning as a belief network. The second is an interactive tool to understand the beta distribution.

The following uses a belief network representation from the previous chapter to learn (discretized) probabilities. Figure [10.1] shows the output after observing heads, heads, tails. Notice the prediction of future tosses.

```python
from variable import Variable
from probFactors import Prob
from probGraphicalModels import BeliefNetwork
from probRC import ProbRC

#### Coin Toss ###
# multiple coin tosses:
toss = ['tails','heads']
tosses = [Variable(f'Toss#{i}', toss, (0.8, 0.9-i/10) if i<10 else (0.4,0.2))
    for i in range(11)]

def coinTossBN(num_bins = 10):
    prob_bins = [x/num_bins for x in range(num_bins+1)]
    PH = Variable("P_heads", prob_bins, (0.1,0.9))
    p_PH = Prob(PH,[],{x:0.5/num_bins if x in [0,1] else 1/num_bins for x in prob_bins})
    p_tosses = [ Prob(tosses[ii],[PH], {x:{'tails':1-x,'heads':x} for x in prob_bins})
        for i in range(11)]
```
Coin Tosses observed: {Toss#0: 'heads', Toss#1: 'heads', Toss#2: 'tails'}

Figure 10.1: coinTossBN after observing heads, heads, tails
10.1. Bayesian Learning

Figure 10.2 shows a plot of the Beta distribution (the P(head variable in the previous belief network) given some sets of observations.

This is a plot that is produced by the following interactive tool.

```python
return BeliefNetwork("Coin Tosses",
[PH]+tosses,
[p_PH]+p_tosses)

# coinRC = ProbRC(coinTossBN(20))
# coinRC.query(tosses[10],{tosses[0]:'heads'})
# coinRC.show_post({})
# coinRC.show_post({tosses[0]:'heads'})
# coinRC.show_post({tosses[0]:'heads',tosses[1]:'heads'})
# coinRC.show_post({tosses[0]:'heads',tosses[1]:'heads',tosses[2]:'tails'})
```

Figure 10.2 shows a plot of the Beta distribution (the $P_{\text{head}}$ variable in the previous belief network) given some sets of observations.

This is a plot that is produced by the following interactive tool.
def __init__(self, num=100, fontsize=10):
    self.num = num
    self.dist = [1 for i in range(num)]
    self.vals = [i/num for i in range(num)]
    self.fontsize = fontsize
    self.saves = []
    self.num_heads = 0
    self.num_tails = 0
    plt.ioff()
    fig, (self.ax) = plt.subplots()
    plt.subplots_adjust(bottom=0.2)
    ## Set up buttons:
    heads_butt = Button(plt.axes([0.05, 0.02, 0.1, 0.05]), "heads")
    heads_butt.label.set_fontsize(self.fontsize)
    heads_butt.on_clicked(self.heads)
    tails_butt = Button(plt.axes([0.25, 0.02, 0.1, 0.05]), "tails")
    tails_butt.label.set_fontsize(self.fontsize)
    tails_butt.on_clicked(self.tails)
    save_butt = Button(plt.axes([0.45, 0.02, 0.1, 0.05]), "save")
    save_butt.label.set_fontsize(self.fontsize)
    save_butt.on_clicked(self.save)
    reset_butt = Button(plt.axes([0.85, 0.02, 0.1, 0.05]), "reset")
    reset_butt.label.set_fontsize(self.fontsize)
    reset_butt.on_clicked(self.reset)
    ## draw the distribution
    plt.subplot(1, 1, 1)
    self.draw_dist()
    plt.show()

    def draw_dist(self):
        sv = self.num / sum(self.dist)
        self.dist = [v*sv for v in self.dist]
        self.ax.clear()
        plt.ylabel("Probability", fontsize=self.fontsize)
        plt.xlabel("P(Heads)", fontsize=self.fontsize)
        plt.title("Beta Distribution", fontsize=self.fontsize)
        plt.xticks(fontsize=self.fontsize)
        plt.yticks(fontsize=self.fontsize)
        self.ax.plot(self.vals, self.dist, color='black', label = f"{self.num_heads} heads; {self.num_tails} tails")
        for (nh, nt, d) in self.saves:
            self.ax.plot(self.vals, d, label = f"{nh} heads; {nt} tails")
        self.ax.legend()
        plt.draw()

    def heads(self, event):
        self.num_heads += 1
        self.dist = [self.dist[i] * self.vals[i] for i in range(self.num)]
        self.draw_dist()
10.2 K-means

The k-means learner maintains two lists that suffice as sufficient statistics to classify examples, and to learn the classification:

- \textit{class\_counts} is a list such that \textit{class\_counts}[c] is the number of examples in the training set with \textit{class} = c.

- \textit{feature\_sum} is a list such that \textit{feature\_sum}[i][c] is sum of the values for the \textit{i}’th feature \textit{i} for members of class \textit{c}. The average value of the \textit{i}th feature in class \textit{c} is

\[
\frac{\textit{feature\_sum}[i][c]}{\textit{class\_counts}[c]}
\]

The class is initialized by randomly assigning examples to classes, and updating the statistics for \textit{class\_counts} and \textit{feature\_sum}.

```python
from learnProblem import Data_set, Learner, Data_from_file
import random
import matplotlib.pyplot as plt

class K_means_learner(Learner):
    def __init__(self, dataset, num_classes):
        self.dataset = dataset
        self.num_classes = num_classes
        self.random_initialize()

    def random_initialize(self):
        # class\_counts[c] is the number of examples with class=c
        self.class_counts = [0]*self.num_classes

```

[https://aipython.org](https://aipython.org)
The distance from (the mean of) a class to an example is the sum, over all features, of the sum-of-squares differences of the class mean and the example value.

One step of k-means updates the class counts and feature sum. It uses the old values to determine the classes, and so the new values for class counts and feature sum. At the end it determines whether the values of these have changed, and then replaces the old ones with the new ones. It returns an indicator of whether the values are stable (have not changed).
10.2. K-means

# feature_sum[i][c] is the sum of the values of feature i for class c
new_feature_sum = [[0]*self.num_classes
    for feat in self.dataset.input_features]
for eg in self.dataset.train:
    cl = self.class_of_eg(eg)
    new_class_counts[cl] += 1
    for (ind, feat) in enumerate(self.dataset.input_features):
        new_feature_sum[ind][cl] += feat(eg)
stable = (new_class_counts == self.class_counts)
    (self.feature_sum == new_feature_sum)
self.class_counts = new_class_counts
self.feature_sum = new_feature_sum
self.num_iterations += 1
return stable

def learn(self, n=100):
    """do n steps of k-means, or until convergence""
    i=0
    stable = False
    while i<n and not stable:
        stable = self.k_means_step()
        i += 1
        self.display(1, "Iteration", self.num_iterations,
            "class counts: ", self.class_counts,
            "Stable": stable)
    return stable

def show_classes(self):
    """sorts the data by the class and prints in order.
    For visualizing small data sets
    ""
    class_examples = [[] for i in range(self.num_classes)]
    for eg in self.dataset.train:
        class_examples[self.class_of_eg(eg)].append(eg)
        print("Class", "Example", sep='\t')
    for cl in range(self.num_classes):
        for eg in class_examples[cl]:
            print(cl, *eg, sep='\t')

def plot_error(self, maxstep=20):
    """Plots the sum-of-squares error as a function of the number of
    steps""
    plt.ion()
    plt.xlabel("step")
    plt.ylabel("Ave sum-of-squares error")
    train_errors = []
    if self.dataset.test:
        test_errors = []
for i in range(maxstep):
    self.learn(1)
    train_errors.append(sum(self.distance(self.class_of_eg(eg), eg)
                           for eg in self.dataset.train) / len(self.dataset.train))

    if self.dataset.test:
        test_errors.append(sum(self.distance(self.class_of_eg(eg), eg)
                                for eg in self.dataset.test) / len(self.dataset.test))
    plt.plot(range(1, maxstep + 1), train_errors,
             label='{} classes. Training set'.format(self.num_classes))
    if self.dataset.test:
        plt.plot(range(1, maxstep + 1), test_errors,
                 label='{} classes. Test set'.format(self.num_classes))
    plt.legend()
plt.draw()

#data = Data_from_file('data/emdata1.csv', num_train=10,
#                        target_index=2000) # trivial example
#data = Data_from_file('data/emdata2.csv', num_train=10, target_index=2000)
#data = Data_from_file('data/emdata0.csv', num_train=14,
#                      target_index=2000) # example from textbook
kml = K_means_learner(data, 2)
num_iter=4
print("Class assignment after" , num_iter,"iterations:")
kml.learn(num_iter); kml.show_classes()

# Plot the error
# km2 = K_means_learner(data, 2); km2.plot_error(20) # 2 classes
# km3 = K_means_learner(data, 3); km3.plot_error(20) # 3 classes
# km13 = K_means_learner(data, 13); km13.plot_error(20) # 13 classes

# data = Data_from_file('data/carbool.csv',
#                        target_index=2000, boolean_features=True)
# kml = K_means_learner(data, 3)
# kml.learn(20); kml.show_classes()
# km3 = K_means_learner(data, 3); km3.plot_error(20) # 3 classes
# km3 = K_means_learner(data, 30); km3.plot_error(20) # 30 classes

Exercise 10.1 Change boolean_features = True flag to allow for numerical features. K-means assumes the features are numerical, so we want to make non-numerical features into numerical features (using characteristic functions) but we probably don’t want to change numerical features into Boolean.

Exercise 10.2 If there are many classes, some of the classes can become empty (e.g., try 100 classes with carbool.csv). Implement a way to put some examples into a class, if possible. Two ideas are:

(a) Initialize the classes with actual examples, so that the classes will not start empty. (Do the classes become empty?)
(b) In class prediction, we test whether the code is empty, and make a prediction of 0 for an empty class. It is possible to make a different prediction to “steal” an example (but you should make sure that a class has a consistent value for each feature in a loop).

Make your own suggestions, and compare it with the original, and whichever of these you think may work better.

10.3 EM

In the following definition, a class, $c$, is an integer in range $[0, num\_classes)$. $i$ is an index of a feature, so $feat[i]$ is the $i$th feature, and a feature is a function from tuples to values. $val$ is a value of a feature.

A model consists of 2 lists, which form the sufficient statistics:

- **class_counts** is a list such that $class\_counts[c]$ is the number of tuples with $class = c$, where each tuple is weighted by its probability, i.e.,

$$class\_counts[c] = \sum_{t: class(t) = c} P(t)$$

- **feature_counts** is a list such that $feature\_counts[i][val][c]$ is the weighted count of the number of tuples $t$ with $feat[i](t) = val$ and $class(t) = c$, each tuple is weighted by its probability, i.e.,

$$feature\_counts[i][val][c] = \sum_{t: feat[i](t) = val \text{ and } class(t) = c} P(t)$$

The function $em\_step$ goes through the training examples, and updates these counts. The first time it is run, when there is no model, it uses random distributions.

```python
from learnProblem import Data_set, Learner, Data_from_file
import random
import math
import matplotlib.pyplot as plt

class EM_learner(Learner):
    def __init__(self, dataset, num_classes):
        self.dataset = dataset
        self.num_classes = num_classes
        self.class_counts = None
        self.feature_counts = None

    def em_step(self, orig_class_counts, orig_feature_counts):
```
class_counts = [0]*self.num_classes
feature_counts = [{val:[0]*self.num_classes
                      for val in feat.frange}
                      for feat in self.dataset.input_features]
for tple in self.dataset.train:
    if orig_class_counts: # a model exists
        tpl_class_dist = self.prob(tple, orig_class_counts,
                                    orig_feature_counts)
    else: # initially, with no model, return a random
distribution
        tpl_class_dist = random_dist(self.num_classes)
        for cl in range(self.num_classes):
            class_counts[cl] += tpl_class_dist[cl]
            for (ind,feat) in enumerate(self.dataset.input_features):
                feature_counts[ind][feat(tple)][cl] += tpl_class_dist[cl]
        return class_counts, feature_counts

prob computes the probability of a class c for a tuple tple, given the current statistics.

\[
P(c \mid tple) \propto P(c) \prod_i P(X_i=tple(i) \mid c)
\]

\[
= \frac{\text{class_counts}[c]}{\text{len(self.dataset)}} \prod_i \frac{\text{feature_counts}[i][\text{feat}(tple)][c]}{\text{class_counts}[c]^{\text{len}(\text{feats})-1}}
\]

The last step is because \( \text{len(self.dataset)} \) is a constant (independent of \( c \)). \( \text{class_counts}[c] \) can be taken out of the product, but needs to be raised to the power of the number of features, and one of them cancels.

```python
def prob(self, tple, class_counts, feature_counts):
    """returns a distribution over the classes for tuple tple in the
    model defined by the counts
    """
    feats = self.dataset.input_features
    unnorm = [prod(feature_counts[i][feat(tple)][c]
                   for (i,feat) in enumerate(feats))
               /class_counts[c]**(len(feats)-1))
               for c in range(self.num_classes)]
    thesum = sum(unnorm)
    return [un/thesum for un in unnorm]
```

learn does n steps of EM:

```python
def learn(self,n):
    """do n steps of em""
```
for i in range(n):
    self.class_counts, self.feature_counts =
    self.em_step(self.class_counts, self.feature_counts)

The following is for visualizing the classes. It prints the dataset ordered by the probability of class c.

```python
def show_class(self, c):
    """sorts the data by the class and prints in order.
    For visualizing small data sets
    """
    sorted_data =
    sorted((self.prob(tpl, self.class_counts, self.feature_counts)[c],
        ind,  # preserve ordering for equal probabilities
tpl)
    for (ind, tpl) in enumerate(self.dataset.train))

    for cc, r, tpl in sorted_data:
        print(cc, *tpl, sep='\t')
```

The following are for evaluating the classes.

The probability of a tuple can be evaluated by marginalizing over the classes:

\[
P(tple) = \sum_c P(c) \prod_i P(X_i=tple(i) | c)
= \sum_c \frac{cc[c]}{\text{len}(self.dataset)} \prod_i \frac{fc[i][\text{feat}(tple)]}[c]
\]

where \( cc \) is the class count and \( fc \) is feature count. \( \text{len}(self.dataset) \) can be distributed out of the sum, and \( cc[c] \) can be taken out of the product:

\[
= \frac{1}{\text{len}(self.dataset)} \sum_c \frac{1}{cc[c]^{\text{# feats} - 1}} \prod_i fc[i][\text{feat}(tple)][c]
\]

Given the probability of each tuple, we can evaluate the logloss, as the negative of the log probability:

```python
def logloss(self, tple):
    """returns the logloss of the prediction on tple, which is
    -log(P(tple))
    based on the current class counts and feature counts
    """
    feats = self.dataset.input_features
    res = 0
    cc = self.class_counts
    fc = self.feature_counts
    for c in range(self.num_classes):
        res += prod(fc[i][feat(tple)][c])
```

https://aipython.org
for (i, feat) in enumerate(feats)/(cc[c]**(len(feats)-1))
    if res>0:
        return -math.log2(res/len(self.dataset.train))
    else:
        return float("inf") #infinity

def plot_error(self, maxstep=20):
    """Plots the logloss error as a function of the number of steps""
    self.learn()
    plt.xlabel("step")
    plt.ylabel("Ave Logloss (bits)"")
    train_errors = []
    if self.dataset.test:
        test_errors = []
    for i in range(maxstep):
        train_errors.append( sum(self.logloss(tple) for tple in
            self.dataset.train) )
            /len(self.dataset.train))
        if self.dataset.test:
            test_errors.append( sum(self.logloss(tple) for tple in
                self.dataset.test) )
                /len(self.dataset.test))
        plt.plot(range(1,maxstep+1),train_errors, label=str(self.num_classes)+" classes. Training set")
        if self.dataset.test:
            plt.plot(range(1,maxstep+1),test_errors, label=str(self.num_classes)+" classes. Test set")
        plt.legend()
    plt.draw()

def prod(L):
    """returns the product of the elements of L""
    res = 1
    for e in L:
        res *= e
    return res

def random_dist(k):
    """generate k random numbers that sum to 1""
    res = [random.random() for i in range(k)]
    s = sum(res)
    return [v/s for v in res]
data = Data_from_file('data/emdata2.csv', num_train=10, target_index=2000)
eml = EM_learner(data,2)
um_iter=2
print("Class assignment after",num_iter,"iterations:")
eml.learn(num_iter); eml.show_class(0)
# Plot the error

```python
# em2=EM_learner(data,2); em2.plot_error(40) # 2 classes
# em3=EM_learner(data,3); em3.plot_error(40) # 3 classes
# em13=EM_learner(data,13); em13.plot_error(40) # 13 classes

# data = Data_from_file('data/carbool.csv',
#           target_index=2000,boolean_features=False)
# [f.frange for f in data.input_features]
# eml = EM_learner(data,3)
# eml.learn(20); eml.show_class(0)
# em3=EM_learner(data,3); em3.plot_error(60) # 3 classes
# em3=EM_learner(data,30); em3.plot_error(60) # 30 classes
```

**Exercise 10.3** For the EM data, where there are naturally 2 classes, 3 classes does better on the training set after a while than 2 classes, but worse on the test set. Explain why. Hint: look what the 3 classes are. Use “em3.show_class(i)” for each of the classes $i \in [0, 3)$.

**Exercise 10.4** Write code to plot the logloss as a function of the number of classes (from 1 to say 15) for a fixed number of iterations. (From the experience with the existing code, think about how many iterations is appropriate.)
Chapter 11

Causality

11.1 Do Questions

A causal model can answer “do” questions.

The intervene function takes a belief network and a variable:value dictionary specifying what to “do”, and returns a belief network resulting from intervening to set each variable in the dictionary to its value specified. It replaces each CPD with a constant CPD.

The following adds the queryDo method to the InferenceMethod class, so it can be used with any inference method. It replaces the graphical model with the modified one, runs the inference algorithm, and restores the initial belief network.
11. Causality

```python
oldBN, self.gm = self.gm, intervene(self.gm, do)
result = self.query(qvar, obs)
self.gm = oldBN # restore original
return result

# make queryDo available for all inference methods
InferenceMethod.queryDo = queryDo
```

Test cases:

### Showing posterior distributions:
```python
from probRC import ProbRC
from probExamples import bn_sprinkler, Season, Sprinkler, Rained,
                     Grass_wet, Grass_shiny, Shoes_wet
bn_sprinkler = ProbRC(bn_sprinkler)
## bn_sprinkler.queryDo(Shoes_wet)
## bn_sprinkler.queryDo(Shoes_wet,obs={Sprinkler:"on"})
## bn_sprinkler.queryDo(Shoes_wet,do={Sprinkler:"on"})
## bn_sprinkler.queryDo(Season, obs={Sprinkler:"on"})
## bn_sprinkler.queryDo(Season, do={Sprinkler:"on"})

### Showing posterior distributions:
# bn_sprinkler.show_post({})
# bn_sprinkler.show_post({Sprinkler:"on"})
# spon = intervene(bn_sprinkler, do={Sprinkler:"on"})
# ProbRC(spon).show_post({})
```

The following is a representation of a possible model where marijuana is a
gateway drug to harder drugs (or not). Try the queries at the end.

```python
from variable import Variable
from probFactors import Prob
from probGraphicalModels import BeliefNetwork
boolean = [False, True]

Drug_Prone = Variable("Drug_Prone", boolean, position=(0.1,0.5)) #
             (0.5,0.9))
Side_Effects = Variable("Side_Effects", boolean, position=(0.1,0.5)) #
                  (0.5,0.1))
Takes_Marijuana = Variable("\nTakes_Marijuana\n", boolean,
                  position=(0.1,0.5))
Takes_Hard_Drugs = Variable("Takes_Hard_Drugs", boolean,
                  position=(0.9,0.5))

p_dp = Prob(Drug_Prone, [], [0.8, 0.2])
p_be = Prob(Side_Effects, [Takes_Marijuana], [[1, 0], [0.4, 0.6]])
p_tm = Prob(Takes_Marijuana, [Drug_Prone], [[0.98, 0.02], [0.2, 0.8]])
p_thd = Prob(Takes_Hard_Drugs, [Side_Effects, Drug_Prone],
              # Drug_Prone=False Drug_Prone=True
              [[[0.999, 0.001], [0.6, 0.4]], # Side_Effects=False
```

https://aipython.org
11.2 Counterfactual Example

This is for a chain $A \to B \to C$ where the query is $A=true, C=true$ is observed; what is the probability of $C$ is $A$ were false. See Figure [11.1]

```python
from probVariables import Variable
from probFactors import Prob, ProbDT, Ifeq, Dist
from probGraphicalModels import BeliefNetwork
from probRC import ProbRC
from probDo import queryDo

boolean = [False, True]

# without a deterministic system
Ap = Variable("Ap", boolean, position=(0.2,0.8))
Bp = Variable("Bp", boolean, position=(0.2,0.4))
 Cp = Variable("Cp", boolean, position=(0.2,0.0))
p_Ap = Prob(Ap, [], [0.5,0.5])
p_Bp = Prob(Bp, [Ap], [[0.6,0.4], [0.6,0.4]]) # does not depend on A!
p_Cp = Prob(Cp, [Bp], [[0.2,0.8], [0.9,0.1]])
abcSimple = BeliefNetwork("ABC Simple",
    [Ap,Bp,Cp],
    [p_Ap, p_Bp, p_Cp])
ABCsimpq = ProbRC(abcSimple)
# ABCsimpq.show_post(obs = {Ap:True, Cp:True})
```
ABC Counterfactual Example

Figure 11.1: \(A \rightarrow B \rightarrow C\) belief network for “what if A”

```python
# as a deterministic system with independent noise
A = Variable("A", boolean, position=(0.2,0.8))
B = Variable("B", boolean, position=(0.2,0.4))
C = Variable("C", boolean, position=(0.2,0.0))
Aprime = Variable("A'", boolean, position=(0.8,0.8))
Bprime = Variable("B'", boolean, position=(0.8,0.4))
Cprime = Variable("C'", boolean, position=(0.8,0.0))
BifA = Variable("B if a", boolean, position=(0.4,0.8))
BifnA = Variable("B if not a", boolean, position=(0.6,0.8))
CifB = Variable("C if b", boolean, position=(0.4,0.4))
CifnB = Variable("C if not b", boolean, position=(0.6,0.4))
p_A = Prob(A, [], [0.5,0.5])
p_B = Prob(B, [A, BifA, BifnA], [[[1,0],[0,1]],[[1,0],[0,1]]], # A=0
    [[[0,1],[1,0]],[[0,1],[0,1]]]) # A=1
p_C = Prob(C, [B, CifB, CifnB], [[[1,0],[0,1]],[[1,0],[0,1]]], # B=0
    [[[0,1],[1,0]],[[0,1],[0,1]]]) # B=1
p_Aprime = Prob(Aprime, [], [0.6,0.4])
p_Bprime = Prob(Bprime, [Aprime, BifA, BifnA],
    [[[1,0],[0,1]],[[1,0],[0,1]]], # A=0
    [[[1,0],[1,0]],[[0,1],[0,1]]]) # A=1
p_Cprime = Prob(Cprime, [Bprime, CifB, CifnB],
    [[[1,0],[0,1]],[[1,0],[0,1]]], # B=0
    [[[1,0],[1,0]],[[0,1],[0,1]]]) # B=1
```

https://aipython.org
11.2. Counterfactual Example

\[
\begin{bmatrix}
[1,0], [1,0], [0,1], [0,1]
\end{bmatrix}
\]  # B=1

\[\text{p\_bifa} = \text{Prob}(\text{BifA}, [], [0.6, 0.4]) \]  # Does not actually depend on A!!

\[\text{p\_bifna} = \text{Prob}(\text{BifnA}, [], [0.6, 0.4])\]

\[\text{p\_cifb} = \text{Prob}(\text{CifB}, [], [0.9, 0.1])\]

\[\text{p\_cifnb} = \text{Prob}(\text{CifnB}, [], [0.2, 0.8])\]

\(\text{abcCounter} = \text{BeliefNetwork}(\text{"ABC Counterfactual Example"},\)

\[\begin{bmatrix}
[A, B, C, A\prime, B\prime, C\prime, B\text{ifA}, B\text{ifnA}, C\text{ifB}, C\text{ifnB}],

[p\_A, p\_B, p\_C, p\_A\prime, p\_B\prime, p\_C\prime, p\_b\text{ifA}, p\_b\text{ifnA}, p\_c\text{ifB}, p\_c\text{ifnB}]
\end{bmatrix}\]

\(\text{abcq} = \text{ProbRC}(\text{abcCounter})\)

# abcq.queryDo(\text{Cprime}, obs = \{A\prime:False, A:True\})
# abcq.queryDo(\text{Cprime}, obs = \{C:True, A\prime:False\})
# abcq.queryDo(\text{Cprime}, obs = \{A:True, C:True, A\prime:False\})
# abcq.queryDo(\text{CifB}, obs = \{C:True, A\prime:False\})
# abcq.queryDo(\text{CifnB}, obs = \{C:True, A\prime:False\})
# abcq.show_post(obs = \{\})
# abcq.show_post(obs = \{A\prime:False, A:True\})
# abcq.show_post(obs = \{A:True, C:True, A\prime:False\})
# abcq.show_post(obs = \{A:True, C:True, A\prime:True\})

The following is the firing squad example of Pearl. See Figure 11.2

-order = \text{Variable}(\"Order\", \text{boolean}, \text{position}=(0.4,0.8))

-S1 = \text{Variable}(\"S1\", \text{boolean}, \text{position}=(0.3,0.4))

-S1o = \text{Variable}(\"S1o\", \text{boolean}, \text{position}=(0.1,0.8))

-S1n = \text{Variable}(\"S1n\", \text{boolean}, \text{position}=(0.0,0.6))

-S2 = \text{Variable}(\"S2\", \text{boolean}, \text{position}=(0.5,0.4))

-S2o = \text{Variable}(\"S2o\", \text{boolean}, \text{position}=(0.7,0.8))

-S2n = \text{Variable}(\"S2n\", \text{boolean}, \text{position}=(0.8,0.6))

-\text{Dead} = \text{Variable}(\"Dead\", \text{boolean}, \text{position}=(0.4,0.0))

-def eqto(var):
\quad return \text{Ifeq(var, True, \text{Dist}([0,1]), \text{Dist}([1,0])))

-p\_S1 = \text{ProbDT}(\text{S1}, [\text{Order}, \text{S1o}, \text{S1n}],
\quad \text{Ifeq(\text{Order}, True, eqto(\text{S1o}), eqto(\text{S1n}))})

-p\_S2 = \text{ProbDT}(\text{S2}, [\text{Order}, \text{S2o}, \text{S2n}],
\quad \text{Ifeq(\text{Order}, True, eqto(\text{S2o}), eqto(\text{S2n}))})

-#p\_S1 = \text{Prob}(\text{S1}, [\text{Order}, \text{S1o}, \text{S1n}], \text{[[[1,0],[0,1]],[[1,0],[0,1]]]}, \#
\quad \text{Order}=0

-# \text{[[[1,0],[1,0]],[[0,1],[0,1]]]}, \# \text{Order}=1

-#p\_S2 = \text{Prob}(\text{S2}, [\text{Order}, \text{S2o}, \text{S2n}], \text{[[[1,0],[0,1]],[[1,0],[0,1]]]}, \#
\quad \text{Order}=0

-# \text{[[[1,0],[1,0]],[[0,1],[0,1]]]}, \# \text{Order}=1

https://aipython.org  Version 0.9.12  December 22, 2023
11. Causality

Firing squad observed: {}

![Firing squad belief network]

Figure 11.2: Firing squad belief network

```python
p_dead = Prob(Dead, [S1, S2], [[[1, 0], [0, 1]], [[0, 1], [0, 1]]])
p_order = Prob(Order, [], [0.9, 0.1])
p_s1o = Prob(S1o, [], [0.01, 0.99])
p_s1n = Prob(S1n, [], [0.99, 0.01])
p_s2o = Prob(S2o, [], [0.01, 0.99])
p_s2n = Prob(S2n, [], [0.99, 0.01])

firing_squad = BeliefNetwork("Firing squad",
                            [Order, S1, S1o, S1n, S2, S2o, S2n, Dead],
                            [p_order, p_dead, p_s1o, p_s1n, p_s2o, p_s2n])

fsq = ProbRC(firing_squad)
# fsq.queryDo(Dead)
# fsq.queryDo(Order, obs={Dead:True})
# fsq.queryDo(Dead, obs={Order:True})
# fsq.show_post({})
# fsq.show_post({Dead:True})
# fsq.show_post({Order:True})
```
Chapter 12

Planning with Uncertainty

12.1 Decision Networks

The decision network code builds on the representation for belief networks of Chapter 9.

We first allow for factors that define the utility. Here the utility is a function of the variables in \(\text{vars}\). In a utility table the utility is defined in terms of a tabular factor – a list that enumerates the values – as in Section 9.3.3.

A decision variable is a like a random variable with a string name, and a domain, which is a list of possible values. The decision variable also includes the parents, a list of the variables whose value will be known when the decision is made. It also includes a potion, which is only used for plotting.

```python
from probGraphicalModels import GraphicalModel, BeliefNetwork
from probFactors import Factor, CPD, TabFactor, factor_times, Prob
from variable import Variable
import matplotlib.pyplot as plt

class Utility(Factor):
    """A factor defining a utility""
    pass

class UtilityTable(TabFactor, Utility):
    """A factor defining a utility using a table""
    def __init__(self, vars, table, position=None):
        """Creates a factor on vars from the table."
        """The table is ordered according to vars.
        """
        TabFactor.__init__(self, vars, table, name="Utility")
        self.position = position
```
A decision network is a graphical model where the variables can be random variables or decision variables. Among the factors we assume there is one utility factor.

The split order ensures that the parents of a decision node are split before the decision node, and no other variables (if that is possible).
12.1.1 Example Decision Networks

Umbrella Decision Network

Here is a simple "umbrella" decision network. The output of `umbrella_dn.show()` is shown in Figure 12.1.
12. Planning with Uncertainty

The following is a variant with the umbrella decision having 2 parents; nothing else has changed. This is interesting because one of the parents is not needed; if the agent knows the weather, it can ignore the forecast.

```python
umbrella_dn2p = DecisionNetwork("Umbrella Decision Network (extra arc)",
    {Weather, Forecast, Umbrella2p},
    {p_weather, p_forecast, umb_utility2p})
# umbrella_dn2p.show()
```

The diagram for the umbrella decision network is as follows:

```
Umbrella Decision Network

Weather

Forecast

Umbrella

Figure 12.1: The umbrella decision network
```
Fire Decision Network

The fire decision network of Figure 12.2 (showing the result of `fire_dn.show()`) is represented as:

```python
defnNetworks.py — (continued)

boolean = [False, True]
Alarm = Variable("Alarm", boolean, position=(0.25,0.633))
Fire = Variable("Fire", boolean, position=(0.5,0.9))
Leaving = Variable("Leaving", boolean, position=(0.25,0.366))
Report = Variable("Report", boolean, position=(0.25,0.1))
Smoke = Variable("Smoke", boolean, position=(0.75,0.633))
Tamper = Variable("Tamper", boolean, position=(0,0.9))
See_Sm = Variable("See_Sm", boolean, position=(0.75,0.366) )
Chk_Sm = DecisionVariable("Chk_Sm", boolean, {Report}, position=(0.5,0.366))
Call = DecisionVariable("Call", boolean,{See_Sm,Chk_Sm,Report},
    position=(0.75,0.1))
f_ta = Prob(Tamper,[],[0.98,0.02])
```

Figure 12.2: Fire Decision Network
Planning with Uncertainty

```python
f_fi = Prob(Fire,[],[0.99,0.01])
f_sm = Prob(Smoke,[Fire],[[0.99,0.01],[0.1,0.9]])
f_al = Prob(Alarm,[Fire,Tamper],[[[0.9999, 0.0001], [0.15, 0.85]], [[0.01, 0.99], [0.5, 0.5]])
f_lv = Prob(Leaving,[Alarm],[[0.99, 0.001], [0.12, 0.88]])
f_re = Prob(Report,[Leaving],[[0.99, 0.01], [0.25, 0.75]])
f_ss = Prob(See_Sm,[Chk_Sm,Smoke],[[[1,[0],[1],[0,1]])

ut = UtilityTable([Chk_Sm,Fire,Call],
    [[[0,-200],[-5000,-200]],[[-20,-220],[-5020,-220]]],
    position=(1,0.633))

fire_dn = DecisionNetwork("Fire Decision Network",
    {Tamper,Fire,Alarm,Leaving,Smoke,Call,See_Sm,Chk_Sm,Report},
    {f_ta,f_fi,f_sm,f_al,f_lv,f_re,f_ss,ut})

# print(ut.to_table())
# fire_dn.show()
# fire_dn.show(fontsize=15)
```

Cheating Decision Network

The following is the representation of the cheating decision of Figure 12.3. Note that we keep the names of the variables short (less than 8 characters) so that the tables look good when printed.

```python
grades = ['A','B','C','F']

Watched = Variable("Watched", boolean, position=(0,0.9))
Caught1 = Variable("Caught1", boolean, position=(0.2,0.7))
Caught2 = Variable("Caught2", boolean, position=(0.6,0.7))
Punish = Variable("Punish", ["None","Suspension","Recorded"],
    position=(0.8,0.9))
Grade_1 = Variable("Grade_1", grades, position=(0.2,0.3))
Grade_2 = Variable("Grade_2", grades, position=(0.6,0.3))
Fin_Grd = Variable("Fin_Grd", grades, position=(0.8,0.1))
Cheat_1 = DecisionVariable("Cheat_1", boolean, set()
    #no parents
    Cheat_2 = DecisionVariable("Cheat_2", boolean, {Cheat_1,Caught1},
    position=(0.4,0.5))

p_wa = Prob(Watched,[],[0.7, 0.3])
p_c1 = Prob(Caught1,[Watched,Cheat_1],[[1.0, 0.0], [0.9, 0.1]]),
    [[1.0, 0.0], [0.5, 0.5]])
    p_r1 = Prob(Report,[Leaving],[[0.99, 0.01], [0.25, 0.75]])

# print(ut.to_table())
# fire_dn.show()
# fire_dn.show(fontsize=15)
```
12.1. Decision Networks

```
utc = UtilityTable([Punish, Fin_Grd],
                  {Punish: 0.5, Fin_Grd: 0.5})

p_gr1 = Prob(Grade_1, [Cheat_1], [{A': 0.2, B': 0.3, C': 0.3, F': 0.2},
                                 {A': 0.5, B': 0.3, C': 0.2, F': 0.0}])

p_gr2 = Prob(Grade_2, [Cheat_2], [{A': 0.2, B': 0.3, C': 0.3, F': 0.2},
                                 {A': 0.5, B': 0.3, C': 0.2, F': 0.0}])

p_fg = Prob(Fin_Grd, [Grade_1, Grade_2],
            {'A': {'A': 0.2, 'B': 0.2, 'C': 0.0, 'F': 0.0},
             'B': {'A': 0.0, 'B': 1.0, 'C': 0.0, 'F': 0.0},
             'C': {'A': 0.5, 'B': 0.5, 'C': 0.0, 'F': 0.0},
             'F': {'A': 0.0, 'B': 0.5, 'C': 0.0, 'F': 0.0}},
            {'A': {'A': 0.5, 'B': 0.2, 'C': 0.0, 'F': 0.0},
             'B': {'A': 0.0, 'B': 1.0, 'C': 0.0, 'F': 0.0},
             'C': {'A': 0.2, 'B': 0.3, 'C': 0.0, 'F': 0.0},
             'F': {'A': 0.0, 'B': 0.5, 'C': 0.0, 'F': 0.0}}))

utc = UtilityTable([Punish, Fin_Grd],
                    {Punish: 0.5, Fin_Grd: 0.5})
```

---

**Figure 12.3: Cheating Decision Network**

```python
import networkx as nx
import matplotlib.pyplot as plt

g = nx.DiGraph()
g.add_node('Watched')
g.add_node('Cheat_1')
g.add_node('Cheat_2')
g.add_node('Grade_1')
g.add_node('Grade_2')
g.add_node('Fin_Grd')
g.add_node('Punish')
g.add_node('Caught1')
g.add_node('Caught2')

g.add_edge('Watched', 'Cheat_1')
g.add_edge('Watched', 'Cheat_2')
g.add_edge('Cheat_1', 'Caught1')
g.add_edge('Cheat_2', 'Caught2')
g.add_edge('Caught1', 'Caught2')
g.add_edge('Caught1', 'Punish')
g.add_edge('Caught2', 'Punish')
g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught2', 'Fin_Grd')
g.add_edge('Caught1', 'Grade_1')
g.add_edge('Caught1', 'Grade_2')
g.add_edge('Caught1', 'Caught2')
g.add_edge('Caught2', 'Grade_1')
g.add_edge('Caught2', 'Grade_2')
g.add_edge('Grade_1', 'Caught1')
g.add_edge('Grade_2', 'Caught2')

g.add_edge('Grade_1', 'Grade_2')
g.add_edge('Grade_1', 'Caught1')
g.add_edge('Grade_2', 'Caught2')

g.add_edge('Caught1', 'Caught2')
g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
g.add_edge('Caught1', 'Caught2')

g.add_edge('Caught2', 'Caught1')
```
3-chain

Figure 12.4: A decision network that is a chain of 3 decisions

The following example is a finite-stage fully-observable Markov decision process with a single reward (utility) at the end. It is interesting because the parents do not include all predecessors. The methods we use will work without change on this, even though the agent does not condition on all of its previous observations and actions. The output of `ch3.show()` is shown in Figure 12.4.

```
{'None':{'A':100, 'B':90, 'C':70, 'F':50},
'Suspension':{'A':40, 'B':20, 'C':10, 'F':0},
'Recorded':{'A':70, 'B':60, 'C':40, 'F':20},
position=(1,0.5))

cheating_dn = DecisionNetwork("Cheating Decision Network",
    {Punish,Caught2,Watched,Fin_Grd,Grade_2,Grade_1,Cheat_2,Caught1,Cheat_1},
    {p_wa, p_cc1, p_cc2, p_pun, p_gr1, p_gr2,p_fg,utc})

# cheating_dn.show()
# cheating_dn.show(fontsize=15)
```

Chain of 3 decisions

The following example is a finite-stage fully-observable Markov decision process with a single reward (utility) at the end. It is interesting because the parents do not include all predecessors. The methods we use will work without change on this, even though the agent does not condition on all of its previous observations and actions. The output of `ch3.show()` is shown in Figure 12.4.
12.1. Decision Networks

D1 = DecisionVariable('D1', boolean, {S1}, position=(3/7, 0.1))
S2 = Variable('S2', boolean, position=(4/7, 0.5))
D2 = DecisionVariable('D2', boolean, {S2}, position=(5/7, 0.1))
S3 = Variable('S3', boolean, position=(6/7, 0.5))

p_s0 = Prob(S0, [], [0.5, 0.5])
tr = [[[0.1, 0.9], [0.9, 0.1]], [[0.2, 0.8], [0.8, 0.2]]] # 0 is flip, 1 is keep value
p_s1 = Prob(S1, [D0, S0], tr)
p_s2 = Prob(S2, [D1, S1], tr)
p_s3 = Prob(S3, [D2, S2], tr)

ch3U = UtilityTable([S3], [0, 1], position=(7/7, 0.9))

ch3 = DecisionNetwork("3-chain",
    {S0,D0,S1,D1,S2,D2,S3},{p_s0,p_s1,p_s2,p_s3,ch3U})

# ch3.show()
# ch3.show(fontsize=15)

12.1.2 Decision Functions

The output of an optimization function is an optimal policy, a list of decision functions, and the expected value of the optimal policy. A decision function is the action for each decision variable as a function of its parents.

class DictFactor(Factor):
    """A factor the represents the values using a dictionary"""
    def __init__(self, *pargs, **kwargs):
        self.values = {}
        Factor.__init__(self, *pargs, **kwargs)
    
    def assign(self, assignment, value):
        self.values[frozenset(assignment.items())] = value

    def get_value(self, assignment):
        ass = frozenset(assignment.items())
        assert ass in self.values, f"assignment {assignment} cannot be evaluated"
        return self.values[ass]

class DecisionFunction(DictFactor):
    def __init__(self, decision, parents):
        """A decision function"
        decision is a decision variable
        parents is a set of variables
        """
        self.decision = decision
        self.parent = parents

https://aipython.org  Version 0.9.12  December 22, 2023
12. Planning with Uncertainty

12.1.3 Recursive Conditioning for decision networks

An instance of a RC_DN object takes in a decision network. The query method uses recursive conditioning to compute the expected utility of the optimal policy. self.opt_policy becomes the optimal policy.

```python
dictFactor.__init__(self, parents, name=decision.name)
```

The following is the simplest search-based algorithm. It is exponential in the number of variables, so is not very useful. However, it is simple, and useful to understand before looking at the more complicated algorithm. Note that the above code does not call rc0; you will need to change the self.rc to self.rc0 in above code to use it.

```python
import math
from probGraphicalModels import GraphicalModel, InferenceMethod
from probFactors import Factor
from probRC import connected_components

class RC_DN(InferenceMethod):
    """The class that queries graphical models using recursive conditioning
    gm is graphical model to query
    """

    def __init__(self, gm=None):
        self.gm = gm
        self.cache = {((frozenset(), frozenset()),):1}
        # self.max_display_level = 3

    def optimize(self, split_order=None, algorithm=None):
        """computes expected utility, and creates optimal decision
        functions, where
        elim_order is a list of the non-observed non-query variables in gm
        algorithm is the (search algorithm to use). Default is self.rc
        """
        if algorithm is None:
            algorithm = self.rc
        if split_order == None:
            split_order = self.gm.split_order()
        self.opt_policy = {v:DecisionFunction(v, v.parents)
                           for v in self.gm.variables
                           if isinstance(v, DecisionVariable)}
        return algorithm({}, self.gm.factors, split_order)

    def show_policy(self):
        print(\"n\'.join(df.to_table() for df in self.opt_policy.values())
```

https://aipython.org
def rc0(self, context, factors, split_order):
    """simplest search algorithm
    context is a variable:value dictionary
    factors is a set of factors
    split_order is a list of variables in factors that are not in
    context
    """
    self.display(3,"calling rc0",(context,factors),"with
    SO",split_order)
    if not factors:
        return 1
    elif to_eval := {fac for fac in factors if
        fac.can_evaluate(context)}:
        self.display(3,"rc0 evaluating factors",to_eval)
        val = math.prod(fac.get_value(context) for fac in to_eval)
        return val * self.rc0(context, factors-to_eval, split_order)
    else:
        var = split_order[0]
        self.display(3, "rc0 branching on", var)
        if isinstance(var,DecisionVariable):
            assert set(context) <= set(var.parents), f"cannot optimize
            {var} in context {context}"
            maxres = -math.inf
            for val in var.domain:
                self.display(3,"In rc0, branching on",var,"=",[val),
                newres = self.rc0({var:val}|context, factors,
                split_order[1:])
                if newres > maxres:
                    maxres = newres
                    theval = val
            self.opt_policy[var].assign(context,theval)
            return maxres
        else:
            total = 0
            for val in var.domain:
                total += self.rc0({var:val}|context, factors,
                split_order[1:])
            self.display(3, "rc0 branching on", var,"returning", total)
            return total

We can combine the optimization for decision networks above, with the
improvements of recursive conditioning used for graphical models (Section

```python

```
split_order is a list of variables in factors that are not in context

self.display(3,"calling rc",(context,factors))
ce = (frozenset(context.items()), frozenset(factors)) # key for the cache entry
if ce in self.cache:
    self.display(2,"rc cache lookup",(context,factors))
    return self.cache[ce]
# if not factors: # no factors; needed if you don't have forgetting and caching
    return 1
elif vars_not_in_factors := {var for var in context if not any(var in fac.variables for fac in factors)}:
    # forget variables not in any factor
    self.display(3,"rc forgetting variables", vars_not_in_factors)
    return self.rc({key:val for (key,val) in context.items() if key not in vars_not_in_factors}, factors, split_order)
elif to_eval := {fac for fac in factors if fac.can_evaluate(context)}:
    # evaluate factors when all variables are assigned
    self.display(3,"rc evaluating factors",to_eval)
    val = math.prod(fac.get_value(context) for fac in to_eval)
    if val == 0:
        return 0
    else:
        return val * self.rc(context, {fac for fac in factors if fac not in to_eval}, split_order)
elif len(comp := connected_components(context, factors, split_order)) > 1:
    # there are disconnected components
    self.display(2,"splitting into connected components",comp)
    return(math.prod(self.rc(context,f,eo) for (f,eo) in comp))
else:
    assert split_order, f"split_order empty rc({context},{factors})"
    var = split_order[0]
    self.display(3, "rc branching on", var)
    if isinstance(var,DecisionVariable):
        assert set(var.parents) <= set(var.parents), f"cannot optimize {var} in context {context}"
        maxres = -math.inf
        for val in var.domain:
            self.display(3,"In rc, branching on",var,"=",val)
            newres = self.rc({var:val}|context, factors, split_order[1:])
            if newres > maxres:
                maxres = newres
                theval = val
Here is how to run the optimize the example decision networks:

```python
# Umbrella decision network
#urc = RC_DN(umbrella_dn)
#urc.optimize(algorithm=urc.rc0)  #RC0
#urc.optimize()  #RC
#urc.show_policy()

#rc_fire = RC_DN(fire_dn)
#rc_fire.optimize()
#rc_fire.show_policy()

#rc_cheat = RC_DN(cheating_dn)
#rc_cheat.optimize()
#rc_cheat.show_policy()

#rc_ch3 = RC_DN(ch3)
#rc_ch3.optimize()
#rc_ch3.show_policy()
#rc_ch3.optimize(algorithm=rc_ch3.rc0)  # why does that happen?
```

### 12.1.4 Variable elimination for decision networks

VE_DN is variable elimination for decision networks. The method `optimize` is used to optimize all the decisions. Note that `optimize` requires a legal elimination ordering of the random and decision variables, otherwise it will give an exception. (A decision node can only be maximized if the variables that are not its parents have already been eliminated.)
```python
self.dn = dn

def optimize(self, elim_order=None, obs={}):
    if elim_order == None:
        elim_order = reversed(self.gm.split_order())
    self.opt_policy = {}
    proj_factors = [self.project_observations(fact, obs)
                    for fact in self.dn.factors]
    for v in elim_order:
        if isinstance(v, DecisionVariable):
            to_max = [fac for fac in proj_factors
                        if v in fac.variables and set(fac.variables) <= v.all_vars]
            assert len(to_max) == 1, "illegal variable order "+str(elim_order)+" at "+str(v)
            newFac = FactorMax(v, to_max[0])
            self.opt_policy[v] = newFac.decision_fun
            proj_factors = [fac for fac in proj_factors
                             if fac is not to_max[0]] + [newFac]
            self.display(2, "maximizing", v)
            self.display(3, newFac)
        else:
            proj_factors = self.eliminate_var(proj_factors, v)
            assert len(proj_factors) == 1, "Should there be only one element of proj_factors?"
            return proj_factors[0].get_value({})

    def show_policy(self):
        print("\n"join(df.to_table() for df in self.opt_policy.values())")

class FactorMax(TabFactor):
    """A factor obtained by maximizing a variable in a factor.
    Also builds a decision_function. This is based on FactorSum."
    
    def __init__(self, dvar, factor):
        """dvar is a decision variable. factor is a factor that contains dvar and only parents of dvar"
        self.dvar = dvar
        self.factor = factor
        vars = [v for v in factor.variables if v is not dvar]
        Factor.__init__(self, vars)
        self.values = {}
        self.decision_fun = DecisionFunction(dvar, dvar.parents)

    def get_value(self, assignment):
        """"""
new_asst = {x:v for (x,v) in assignment.items() if x in self.variables}
asst = frozenset(new_asst.items())
if asst in self.values:
    return self.values[asst]
else:
    max_val = float("-inf") # -infinity
    for elt in self.dvar.domain:
        fac_val = self.factor.get_value(assignment|{self.dvar:elt})
        if fac_val>max_val:
            max_val = fac_val
            best_elt = elt
    self.values[asst] = max_val
    self.decision_fun.assign(assignment, best_elt)
    return max_val

Here are some example queries:

decnNetworks.py — (continued)

def test(dn):
    rc0dn = RC_DN(dn)
    rc0v = rc0dn.optimize(algorithm=rc0dn.rc0)
    rcdn = RC_DN(dn)
    rcv = rcdn.optimize()
    assert abs(rc0v-rcv)<1e-10, f"rc0 produces {rc0v}; rc produces {rcv}"
    vcdn = VE_DN(dn)
    vev = vcdn.optimize()
    assert abs(vev-rcv)<1e-10, f"VE_DN produces {vev}; RC produces {rcv}"
    print(f"passed unit test. rc0, rc and VE gave same result for {dn}")
    if __name__ == "__main__":
        test(fire_dn)

12.2 Markov Decision Processes

The following represent a Markov decision process (MDP) directly, rather than using the recursive conditioning or variable elimination code, as was done for decision networks.
import random
from display import Displayable
from utilities import argmaxd

class MDP(Displayable):
    """A Markov Decision Process. Must define:
    title a string that gives the title of the MDP
    states the set (or list) of states
    actions the set (or list) of actions
    discount a real-valued discount
    ""

    def __init__(self, title, states, actions, discount, init=0):
        self.title = title
        self.states = states
        self.actions = actions
        self.discount = discount
        self.initv = self.V = {s: init for s in self.states}
        self.initq = self.Q = {s: {a: init for a in self.actions} for s in self.states}

        def P(self, s, a):
            """Transition probability function
            returns a dictionary of {s1:p1} such that P(s1 | s,a)=p1. Other
            probabilities are zero.
            ""
            raise NotImplementedError("P") # abstract method

        def R(self, s, a):
            """Reward function R(s,a)
            returns the expected reward for doing a in state s.
            ""
            raise NotImplementedError("R") # abstract method

Two state partying example (Example 12.29 in?):

import matplotlib.pyplot as plt

class partyMDP(MDP):
    """Simple 2-state, 2-Action Partying MDP Example"

    def __init__(self, discount=0.9):
        states = {'healthy', 'sick'}
        actions = {'relax', 'party'}
        MDP.__init__(self, "party MDP", states, actions, discount)

    def R(self, s, a):
        """R(s,a)""
12.2. Markov Decision Processes

```python
return {'healthy': {'relax': 7, 'party': 10},
       'sick': {'relax': 0, 'party': 2}}[s][a]

def P(self, s, a):
    "returns a dictionary of {s1:p1} such that P(s1 | s,a)=p1. Other
    probabilities are zero."
    phealthy = {} # P('healthy' | s, a)
    phealthy['healthy'] = {'relax': 0.95, 'party': 0.7},
    'sick': {'relax': 0.5, 'party': 0.1 }[s][a]
    return {'healthy': phealthy, 'sick': 1-phealthy}
```

The distribution class is used to represent distributions as they are being created. Probability distributions are represented as item:value dictionaries. When being constructed, adding an item:value to the dictionary has to act differently when the item is already in the dictionary and when it isn’t. The add_prob method works whether the item is in the dictionary or not.

12.2.1 Problem Domains

An MDP does not contain enough information to simulate a domain, because

(a) the rewards and resulting state can be correlated (e.g., in the grid domains below, crashing into a wall results in both a negative reward and the agent not moving), and

(b) it represents the expected reward (e.g., a reward of 1 is has the same expected value as a reward of 100 with probability 1/100 and 0 otherwise, but these are different in a simulation).

A problem domain represents a problem as a function result from states and actions into a distribution of (state, reward) pairs. This can be a subclass of MDP because it implements R and P. A problem domain also specifies an initial state and coordinate information used by the graphical user interfaces.
class ProblemDomain(MDP):
    """A ProblemDomain implements
    self.result(state, action) -> {(reward, state):probability}.
    Other pairs have probability are zero.
    The probabilities must sum to 1.
    """
    def __init__(self, title, states, actions, discount,
                 initial_state=None, x_dim=0, y_dim=0,
                 vinit=0, offsets={}):
        """A problem domain
        * title is list of titles
        * states is the list of states
        * actions is the list of actions
        * discount is the discount factor
        * initial_state is the state the agent starts at (for simulation)
          if known
        * x_dim and y_dim are the dimensions used by the GUI to show the
          states in 2-dimensions
        * vinit is the initial value
        * offsets is a {action:(x,y)} map which specifies how actions are
          displayed in GUI
        """
        MDP.__init__(self, title, states, actions, discount)
        if initial_state is not None:
            self.state = initial_state
        else:
            self.state = random.choice(states)
        self.vinit = vinit # value to reset v,q to
        # The following are for the GUI:
        self.x_dim = x_dim
        self.y_dim = y_dim
        self.offsets = offsets

    def state2pos(self, state):
        """When displaying as a grid, this specifies how the state is
        mapped to (x,y) position.
        The default is for domains where the (x,y) position is the state
        ""
        return state

    def state2goal(self, state):
        """When displaying as a grid, this specifies how the state is
        mapped to goal position.
        The default is for domains where there is no goal
        ""
        return None

    def pos2state(self, pos):
        """When displaying as a grid, this specifies how the state is
12.2. Markov Decision Processes

mapped to (x,y) position.
The default is for domains where the (x,y) position is the state

```
return pos
```

def P(self, state, action):
    """Transition probability function
    returns a dictionary of {s1:p1} such that P(s1 | state,action)=p1.
    Other probabilities are zero.
    ""
    res = self.result(state, action)
    acc = 1e-6 # accuracy for test of equality
    assert 1-acc<sum(res.values())<1+acc, f"result({state},{action})
    not a distribution, sum={sum(res.values())}"
    dist = distribution()  
    for ((r,s),p) in res.items():
        dist.add_prob(s,p)
    return dist

def R(self, state, action):
    """Reward function R(s,a)
    returns the expected reward for doing a in state s.
    ""
    return sum(r*p for ((r,s),p) in self.result(state, action).items())

Tiny Game

The next example is the tiny game from Example 13.1 and Figure 13.1 of ?
The state is represented as (x,y) where x counts from zero from the left, and y
counts from zero upwards, so the state (0,0) is on the bottom-left state. The
actions are upC for up-careful, upR for up-risky, left, and left. (Note that
GridDomain means that it can be shown with the MDP GUI in Section [12.2.3].

```python

```
Grid World

Here is the domain of Example 12.30 of ?, shown here in Figure 12.5. A state is represented as \((x, y)\) where \(x\) counts from zero from the left, and \(y\) counts from zero upwards, so the state \((0, 0)\) is on the bottom-left.

https://aipython.org

Version 0.9.12  December 22, 2023
12.2. Markov Decision Processes

```python
class grid(ProblemDomain, GridDomain):
    """ x_dim * y_dim grid with rewarding states"
    def __init__(self, discount=0.9, x_dim=10, y_dim=10):
        ProblemDomain.__init__(self,
            "Grid World",
            [(x,y) for x in range(y_dim) for y in range(y_dim)], #states
            ['up', 'down', 'right', 'left'], #actions
discount,
            x_dim = x_dim, y_dim = y_dim,
            offsets = { 'right':(0.25,0), 'up':(0,0.25), 'left':(-0.25,0),
                       'down':(0,-0.25) }
        self.rewarding_states = {(3,2):-10, (3,5):-5, (8,2):10, (7,7):3 }
        self.fling_states = {(8,2), (7,7)} # assumed a subset of
        rewarding_states

    def intended_next(self,s,a):
        """returns the (reward, state) in the direction a.
        This is where the agent will end up if to goes in its
        intended_direction
        (which it does with probability 0.7).
        """
        (x,y) = s
        if a=='up':
            return (0, (x,y+1)) if y+1 < self.y_dim else (-1, (x,y))
        if a=='down':
            return (0, (x,y-1)) if y > 0 else (-1, (x,y))
        if a=='right':
            return (0, (x+1,y)) if x+1 < self.x_dim else (-1, (x,y))
        if a=='left':
            return (0, (x-1,y)) if x > 0 else (-1, (x,y))

    def result(self,s,a):
        """return a dictionary of {(r,s):p} where p is the probability of
        reward r, state s.
        a state is an (x,y) pair
        """
        r0 = self.rewarding_states[s] if s in self.rewarding_states else 0
        if s in self.fling_states:
            return {(r0,(0,0)): 0.25, (r0,(self.x_dim-1,0)):0.25,
                     (r0,(0,self.y_dim-1)):0.25, (r0,(self.x_dim-1,self.y_dim-1)):0.25}
        dist = distribution({})
        for a1 in self.actions:
            (r1,s1) = self.intended_next(s,a1)
            rs = (r1+r0, s1)
            p = 0.7 if a1==a else 0.1
            dist.add_prob(rs,p)
        return dist
```

https://aipython.org  Version 0.9.12  December 22, 2023
Figure 12.6: Monster game

Monster Game

This is for the game depicted in Figure 13.1 (Example 13.2 of ?).

```python
class Monster_game(ProblemDomain, GridDomain):
    
    vwalls = [(0,3), (0,4), (1,4)] # vertical walls right of these locations
    crash_reward = -1
    
    prize_locs = [(0,0), (0,4), (4,0), (4,4)]
    prize_appears_prob = 0.3
    prize_reward = 10
    
    monster_locs = [(0,1), (1,1), (2,3), (3,1), (4,2)]
    monster_appears_prob = 0.4
    monster_reward_when_damaged = -10
    repair_stations = [(1,4)]

    def __init__(self, discount=0.9):
        x_dim = 5
        y_dim = 5
        # which damaged and prize to show
        ProblemDomain.__init__(self,
            "Monster Game",
            [(x,y,damaged,prize)
            for x in range(x_dim)
            for y in range(y_dim)
            for damaged in [False,True]
            for prize in [None]+self.prize_locs], #states
            ['up', 'down', 'right', 'left'], #actions
```

https://aipython.org | Version 0.9.12 | December 22, 2023
12.2. Markov Decision Processes

```python
discount,
x_dim = x_dim, y_dim = y_dim,
offsets = {'right':(0.25,0), 'up':(0,0.25), 'left':(-0.25,0),
          'down':(0,-0.25))
self.state = (2,2,False,None)

def intended_next(self,xy,a):
    """returns the (reward, (x,y)) in the direction a.
    This is where the agent will end up if to goes in its
    intended_direction
    (which it does with probability 0.7).
    """
    (x,y) = xy # original x-y position
    if a=='up':
        return (0, (x,y+1)) if y+1 < self.y_dim else 
          (self.crash_reward, (x,y))
    if a=='down':
        return (0, (x,y-1)) if y > 0 else (self.crash_reward, (x,y))
    if a=='right':
        if (x,y) in self.vwalls or x+1==self.x_dim: # hit wall
            return (self.crash_reward, (x,y))
        else:
            return (0, (x+1,y))
    if a=='left':
        if (x-1,y) in self.vwalls or x==0:  # hit wall
            return (self.crash_reward, (x,y))
        else:
            return (0, (x-1,y))

def result(self,s,a):
    """return a dictionary of {(r,s):p} where p is the probability of
    reward r, state s.
    a state is an (x,y) pair
    """
    (x,y,damaged,prize) = s
    dist = distribution({})
    for a1 in self.actions: # possible results
        mp = 0.7 if a1==a else 0.1
        mr,(xn,yn) = self.intended_next((x,y),a1)
        if (xn,yn) in self.monster_locs:
            if damaged:
                dist.add_prob((mr+self.monster_reward_when_damaged,(xn,yn,True,prize)),
                              mp*self.monster_appears_prob)
                dist.add_prob((mr,(xn,yn,True,prize)),
                              mp*(1-self.monster_appears_prob))
            else:
                dist.add_prob((mr,(xn,yn,True,prize)),
                              mp+self.monster_appears_prob)
                dist.add_prob((mr,(xn,yn,False,prize)),
                              mp*(1-self.monster_appears_prob))
```

https://aipython.org
elif (xn, yn) == prize:
    dist.add_prob((mr+self.prize_reward, (xn, yn, damaged, None)), mp)
elif (xn, yn) in self.repair_stations:
    dist.add_prob((mr, (xn, yn, False, prize)), mp)
else:
    dist.add_prob((mr, (xn, yn, damaged, prize)), mp)

if prize is None:
    res = distribution({})
    for (r, (x2, y2, d, p2)), p in dist.items():
        res.add_prob((r, (x2, y2, d, None)),
                      p*(1-self.prize_apears_prob))
    for pz in self.prize_locs:
        res.add_prob((r, (x2, y2, d, pz)),
                      p*self.prize_apears_prob/len(self.prize_locs))
    return res
else:
    return dist

def state2pos(self, state):
    """When displaying as a grid, this specifies how the state is
    mapped to (x,y) position.
    The default is for domains where the (x,y) position is the state
    """
    (x, y, d, p) = state
    return (x, y)

def pos2state(self, pos):
    """When displaying as a grid, this specifies how the state is
    mapped to (x,y) position.
    """
    (x, y) = pos
    (xs, ys, damaged, prize) = self.state
    return (x, y, damaged, prize)

def state2goal(self, state):
    """the (x,y) position for the goal
    """
    (x, y, damaged, prize) = state
    return prize

# To see value iterations:
# mg = Monster_game()
# mg.viGUI() # then run vi a few times
# to see other states, exit the GUI
# mg.state = (2,2,True,(4,4)) # or other damaged/prize states
# mg.viGUI()
12.2.2 Value Iteration

The following implements value iteration for Markov decision processes.

A Q function is represented as a dictionary so $Q[s][a]$ is the value for doing action $a$ in state $s$. The value function is represented as a dictionary so $V[s]$ is the value of state $s$. Policy $\pi$ is represented as a dictionary where $\pi[s]$, where $s$ is a state, returns the the action.

Note that the following defines $vi$ to be a method in MDP.

```python
def vi(self, n):
    # carries out n iterations of value iteration, updating value
    # function self.V
    Returns a Q-function, value function, policy
    
    self.display(3, f"calling vi({n})")
    for i in range(n):
        self.Q = {s: {a: self.R(s, a) + self.discount * 
                       sum(p1*self.V[s1]
                        for (s1,p1) in self.P(s,a).items())
                       for a in self.actions}
        self.V = {s: max(self.Q[s][a] for a in self.actions)
                   for s in self.states}
        self.pi = {s: argmaxd(self.Q[s])
                   for s in self.states}
    return self.Q, self.V, self.pi

MDP.vi = vi
```

The following shows how this can be used.

```python
## Testing value iteration
# Try the following:
# pt = partyMDP(discount=0.9)
# pt.vi(1)
# pt.vi(100)
# partyMDP(discount=0.99).vi(100)
# partyMDP(discount=0.4).vi(100)
# gr = grid(discount=0.9)
# gr.viGUI()
# q,v,pi = gr.vi(100)
# q[(7,2)]
```
12. Planning with Uncertainty

12.2.3 Value Iteration GUI for Grid Domains

A GridDomain is a domain where the states can be mapped into \((x, y)\) positions, and the actions can be mapped into up-down-left-right. They are special because the \texttt{viGUI()} method to interact with them. It requires the following values/methods be defined:

- \texttt{self.x_dim} and \texttt{self.y_dim} define the dimensions of the grid (so the states are \((x, y)\), where \(0 \leq x < \texttt{self.x_dim}\) and \(0 \leq y < \texttt{self.y_dim}\).

- \texttt{self.state2pos(state)} gives the \((x, y)\) position of state. The default is that that states are already \((x, y)\) positions.

- \texttt{self.state2goal(state)} gives the \((x, y)\) position of the goal in state. The default is \texttt{None}.

- \texttt{self.pos2state(pos)} where \texttt{pos} is an \((x, y)\) pair, gives the state that is shown at position \((x, y)\). When the state contain more information than the \((x, y)\) pair, the extra information is taked from \texttt{self.state}.

- \texttt{self.offsets[a]} defines where to display action \(a\), as \((x, y)\) offset for action \(a\) when displaying Q-values.

```python
import matplotlib.pyplot as plt
from matplotlib.widgets import Button, CheckButtons, TextBox
from mdpProblem import MDP

class GridDomain(object):
    def viGUI(self):
        #plt.ion() # interactive
        fig, self.ax = plt.subplots()
        plt.subplots_adjust(bottom=0.2)
        stepB = Button(self.ax([0.8,0.05,0.1,0.075]), "step")
        stepB.on_clicked(self.on_step)
        resetB = Button(self.ax([0.65,0.05,0.1,0.075]), "reset")
        resetB.on_clicked(self.on_reset)
        self.qcheck = CheckButtons(self.ax([0.2,0.05,0.35,0.075]),
                                   ["show Q-values","show policy"])
        self.qcheck.on_clicked(self.show_vals)
        self.font_box = TextBox(self.ax([0.1,0.05,0.05,0.075]),"Font: ",
                                 textalignment="center")
        self.font_box.on_submit(self.set_font_size)
        self.show_vals(None)
        plt.show()

    def set_font_size(self, s):
        plt.rcParams.update({'font.size': eval(s)})
```

https://aipython.org
```python
plt.draw()

def show_vals(self, event):
    self.ax.cla()  # clear the axes

    array = [[self.V[self.pos2state((x, y))] for x in range(self.x_dim)]
             for y in range(self.y_dim)]
    self.ax.pcolormesh([x-0.5 for x in range(self.x_dim+1)],
                        [y-0.5 for y in range(self.y_dim+1)],
                        array, edgecolors='black', cmap='summer')

    # for cmap see https://matplotlib.org/stable/tutorials/colors/colormaps.html
    if self.qcheck.get_status()[1]:  # "show policy"
        for x in range(self.x_dim):
            for y in range(self.y_dim):
                state = self.pos2state((x, y))
                maxv = max(self.Q[state][a] for a in self.actions)
                for a in self.actions:
                    if self.Q[state][a] == maxv:
                        # draw arrow in appropriate direction
                        xoff, yoff = self.offsets[a]
                        self.ax.arrow(x, y, xoff*2, yoff*2,
                                       color='red', width=0.05, head_width=0.2,
                                       length_includes_head=True)
    if self.qcheck.get_status()[0]:  # "show q-values"
        self.show_q(event)
    else:
        self.show_v(event)

    self.ax.set_xticks(range(self.x_dim))
    self.ax.set_xticklabels(range(self.x_dim))
    self.ax.set_yticks(range(self.y_dim))
    self.ax.set_yticklabels(range(self.y_dim))

    plt.draw()

def on_step(self, event):
    self.step()
    self.show_vals(event)

def step(self):
    """The default step is one step of value iteration""
    self.vi(1)

def show_v(self, event):
    """show values""
    for x in range(self.x_dim):
        for y in range(self.y_dim):
            state = self.pos2state((x, y))
            self.ax.text(x, y, "{val:.2f}".format(val=self.V[state]), ha='center')

    def show_q(self, event):
```

---

https://aipython.org  Version 0.9.12  December 22, 2023
12. Planning with Uncertainty

Figure 12.7 shows the user interface for the tiny domain, which can be obtained using

\texttt{MDPtiny(discount=0.9).viGUI()}

resizing it, checking “show q-values” and “show policy”, and clicking “step” a few times.

Figure 12.8 shows the user interface for the grid domain, which can be obtained using

\texttt{grid(discount=0.9).viGUI()}

resizing it, checking “show q-values” and “show policy”, and clicking “step” a few times.

Exercise 12.1 Computing \( q \) before \( v \) may seem like a waste of space because we don’t need to store \( q \) in order to compute value function or the policy. Change the algorithm so that it loops through the states and actions once per iteration, and only stores the value function and the policy. Note that to get the same results as before, you would need to make sure that you use the previous value of \( v \) in the computation not the current value of \( v \). Does using the current value of \( v \) hurt the algorithm or make it better (in approaching the actual value function)?

12.2.4 Asynchronous Value Iteration

This implements asynchronous value iteration, storing \( Q \).
12.2. Markov Decision Processes

Figure 12.7: Interface for tiny example, after a number of steps. Each rectangle represents a state. In each rectangle are the 4 Q-values for the state. The leftmost number is the for the left action; the rightmost number is for the right action; the upper most is for the upR (up-risky) action and the lowest number is for the upC action. The arrow points to the action(s) with the maximum Q-value. Use MDPtiny().viGUI() after loading mdpExamples.py
Figure 12.8: Interface for grid example, after a number of steps. Each rectangle represents a state. In each rectangle are the 4 Q-values for the state. The leftmost number is the for the left action; the rightmost number is for the right action; the upper most is for the up action and the lowest number is for the down action. The arrow points to the action(s) with the maximum Q-value. From grid(discount=0.9).viGUI()
A Q function is represented so $q[s][a]$ is the value for doing action with index $a$ state with index $s$.

Note that the following defines avi to be a method of MDP.

```python
def avi(self,n):
    states = list(self.states)
    actions = list(self.actions)
    for i in range(n):
        s = random.choice(states)
        a = random.choice(actions)
        self.Q[s][a] = (self.R(s,a) + self.discount *
                        sum(p1 * max(self.Q[s1][a1]
                                    for a1 in self.actions)
                                for (s1,p1) in self.P(s,a).items()))
    return self.Q
```

The following shows how avi can be used.

```python
# Testing asynchronous value iteration
# Try the following:
# pt = partyMDP(discount=0.9)
# pt.avi(10)
# pt.vi(1000)
#
# gr = grid(discount=0.9)
# q = gr.avi(100000)
# q[(7,2)]

def test_MDP(mdp, discount=0.9, eps=0.01):
    """tests vi and avi give the same answer for a MDP class mdp """
    mdp1 = mdp(discount=discount)
    q1,v1,pi1 = mdp1.vi(100)
    mdp2 = mdp(discount=discount)
    q2 = mdp2.avi(1000)
    same = all(abs(q1[s][a]-q2[s][a]) < eps
               for s in mdp1.states
               for a in mdp1.actions)
    assert same, "vi and avi are different:\n{q1}\n{q2}"
    print(f"passed unit test. vi and avi gave same result for {mdp1.title}")
```

**Exercise 12.2** Implement value iteration that stores the $V$-values rather than the $Q$-values. Does it work better than storing $Q$? (What might “better” mean?)
Exercise 12.3  In asynchronous value iteration, try a number of different ways to choose the states and actions to update (e.g., sweeping through the state-action pairs, choosing them at random). Note that the best way may be to determine which states have had their Q-values change the most, and then update the previous ones, but that is not so straightforward to implement, because you need to find those previous states.
13.1 Representing Agents and Environments

The reinforcement learning agents and environments are instances of the general agent architecture of Section 2.1, where the percepts are reward–state pairs. The state is the world state; this is the fully observable assumption. In particular:

- An agent implements the method `select_action` that takes the reward (and environment state) and returns the next action (and updates the state of the agent).
- An environment implements the method `do` that takes the action and returns a pair of the reward and the resulting environment state.

These are chained together to simulate the system.

This follows the architecture of Section 2.1, here the percept is the state. The simulation starts by calling the agent method `initial_action(state)`, which is generally to remember the state and return a random action.

### 13.1.1 Environments

The environments have names for the roles of agents participating. In this chapter, where we assume a single agent, this is used as the name of the environment.

```python
import random
import math
from display import Displayable
```
from agents import Agent, Environment
from utilities import select_from_dist, argmaxe, argmaxd, flip

class RL_env(Environment):
    def __init__(self, name, actions, state):
        """creates an environment given name, list of actions, and initial state""
        self.name = name  # the role for an agent
        self.actions = actions  # list of all actions
        self.state = state  # initial state
        self.reward = None  # last reward

    # must implement do(action)->(reward,state)

13.1.2 Agents

class RL_agent(Agent):
    """An RL_Agent
    has percepts (s, r) for some state s and real reward r
    ""
    def __init__(self, actions):
        self.actions = actions

    def initial_action(self, env_state):
        """return the initial action, and remember the state and action
        Act randomly initially
        Could be overridden to initialize data structures (as the agent now
        knows about one state)
        ""
        self.state = env_state
        self.action = random.choice(self.actions)
        return self.action

    def select_action(self, reward, state):
        """Select the action given the reward and next state
        Remember the action in self.action
        This implements "Act randomly" and should be overridden!
        ""
        self.reward = reward
        self.action = random.choice(self.actions)
        return self.action

    def v(self, state):
        """v needed for GUI; an agent must also implement q()"
        return max(self.q(state,a) for a in self.actions)
13.1. Representing Agents and Environments

13.1.3 Simulating an Environment-Agent Interaction

The interaction between the agents and the environment is mediated by a simulator that calls each in turn. Simulate below is similar to Simulate of Section 2.1 except it is initialized by agent.initial_action(state).

```python
import matplotlib.pyplot as plt

class Simulate(Displayable):
    """simulate the interaction between the agent and the environment for n time steps.
    Returns a pair of the agent state and the environment state."
    
    def __init__(self, agent, environment):
        self.agent = agent
        self.env = environment
        self.reward_history = [] # for plotting
        self.step = 0
        self.sum_rewards = 0

    def start(self):
        self.action = self.agent.initial_action(self.env.state)
        return self

    def go(self, n):
        for i in range(n):
            self.step += 1
            (reward, state) = self.env.do(self.action)
            self.display(2, f"step={self.step} reward={reward}, state={state}"
            self.sum_rewards += reward
            self.reward_history.append(reward)
            self.action = self.agent.select_action(reward, state)
            self.display(2, f" action={self.action}"
        return self

    def plot(self, label=None, step_size=None, xscale='linear'):
        """plots the rewards history in the simulation
        label is the label for the plot
        step_size is the number of steps between each point plotted
        xscale is 'log' or 'linear'
        returns sum of rewards"
```

The following plots the sum of rewards as a function of the step in a simulation.
if step_size is None: # for long simulations (> 999), only plot some points
    step_size = max(1,len(self.reward_history)//500)
if label is None:
    label = self.agent.method
plt.ion()
plt.xscale(xscale)
plt.xlabel("step")
plt.ylabel("Sum of rewards")
sum_history, sum_rewards = acc_rews(self.reward_history, step_size)
plt.plot(range(0,len(self.reward_history),step_size), sum_history,
         label=label)
plt.legend()
plt.draw()
return sum_rewards

def acc_rews(rews,step_size):
    """returns the rolling sum of the values, sampled each step_size, and
    the sum
    """
    acc = []
    sumr = 0; i=0
    for e in rews:
        sumr += e
        i += 1
        if (i%step_size == 0): acc.append(sumr)
    return acc, sumr

13.1.4 Party Environment

Here is the definition of the simple 2-state, 2-action decision about whether to
party or relax (Example 12.29 in ?). (Compare to the MDP representation of
page 288)

from rlProblem import RL_env
class Party_env(RL_env):
    def __init__(self):
        RL_env.__init__(self, "Party Decision", ["party", "relax"],
                        "healthy")

    def do(self, action):
        """updates the state based on the agent doing action.
        returns reward,state
        """
        if self.state=="healthy":
            if action=="party":
                self.state = "healthy" if flip(0.7) else "sick"
                self.reward = 10
            else: # action=="relax"
13.1. Representing Agents and Environments

```python
self.state = "healthy" if flip(0.95) else "sick"
self.reward = 7
else: # self.state=="sick"
    if action=="party":  
        self.state = "healthy" if flip(0.1) else "sick"
        self.reward = 2
    else:
        self.state = "healthy" if flip(0.5) else "sick"
        self.reward = 0
return self.reward, self.state
```

13.1.5 Environment from a Problem Domain

Env_fom_ProblemDomain takes a ProblemDomain (page 289) and constructs an environment that can be used for reinforcement learners.

As explained in Section 12.2.1, the representation of an MDP does not contain enough information to simulate a system, because it loses any dependency between the rewards and the resulting state (e.g., hitting the wall and having a negative reward may be correlated), and only represents the expected value of rewards, not how they are distributed. The ProblemDomain class defines the result method to map states and actions into distributions over (reward, state) pairs.

```python
class Env_from_ProblemDomain(RL_env):
    def __init__(self, prob_dom):
        RL_env.__init__(self, prob_dom.title, prob_dom.actions, prob_dom.state)
        self.problem_domain = prob_dom
        self.state = prob_dom.state
        self.x_dim = prob_dom.x_dim
        self.y_dim = prob_dom.y_dim
        self.offsets = prob_dom.offsets
        self.state2pos = self.problem_domain.state2pos
        self.state2goal = self.problem_domain.state2goal
        self.pos2state = self.problem_domain.pos2state

    def do(self, action):
        """updates the state based on the agent doing action.
        returns state,reward
        """
        (self.reward, self.state) = select_from_dist(self.problem_domain.result(self.state, action))
        self.problem_domain.state = self.state
        self.display(2,f"do({action} -> ({self.reward}, {self.state})")
        return (self.reward, self.state)
```

https://aipython.org
13.1.6 Monster Game Environment

This is for the game depicted in Figure 13.1 (Example 13.2 of ?). This is an alternative representation to that of Section 12.2.1, which defined the distribution over reward-state pairs. This directly builds a simulator, which might be easier to understand or adapt to new environments.

```python
import random
from utilities import flip
from rlProblem import RL_env

class Monster_game_env(RL_env):
    x_dim = 5
    y_dim = 5

    vwalls = [(0,3), (0,4), (1,4)] # vertical walls right of these locations
    hwalls = [] # not implemented
    crashed_reward = -1

    prize_locs = [(0,0), (0,4), (4,0), (4,4)]
    prize_apears_prob = 0.3
    prize_reward = 10

    monster_locs = [(0,1), (1,1), (2,3), (3,1), (4,2)]
    monster_appears_prob = 0.4
    monster_reward_when_damaged = -10
    repair_stations = [(1,4)]

    actions = ['up','down','left','right']
```

Figure 13.1: Monster game
13.1. Representing Agents and Environments

```python
def __init__(self):
    # State:
    self.x = 2
    self.y = 2
    self.damaged = False
    self.prize = None
    # Statistics
    self.number_steps = 0
    self.accumulated_rewards = 0  # sum of rewards received
    self.min_accumulated_rewards = 0
    self.min_step = 0
    self.zero_crossing = 0
    RL_env.__init__(self, "Monster Game", self.actions, (self.x,
                self.y, self.damaged, self.prize))
    self.display(2,"","Step","Tot Rew","Ave Rew",sep="\t")

def do(self, action):
    """updates the state based on the agent doing action.
    returns reward,state
    """
    assert action in self.actions, f"Monster game, unknown action: {action}"
    self.reward = 0.0
    # A prize can appear:
    if self.prize is None and flip(self.prize_apears_prob):
        self.prize = random.choice(self.prize_locs)
    # Actions can be noisy
    if flip(0.4):
        actual_direction = random.choice(self.actions)
    else:
        actual_direction = action
    # Modeling the actions given the actual direction
    if actual_direction == "right":
        if self.x==self.x_dim-1 or (self.x, self.y) in self.vwalls:
            self.reward += self.crashed_reward
        else:
            self.x += 1
    elif actual_direction == "left":
        if self.x==0 or (self.x-1, self.y) in self.vwalls:
            self.reward += self.crashed_reward
        else:
            self.x += -1
    elif actual_direction == "up":
        if self.y==self.y_dim-1:
            self.reward += self.crashed_reward
        else:
            self.y += 1
    elif actual_direction == "down":
        if self.y==0:
            self.reward += self.crashed_reward
```
else:
    self.y += -1
else:
    raise RuntimeError(f"unknown_direction: {actual_direction}"

# Monsters
if (self.x, self.y) in self.monster_locs and flip(self.monster_appears_prob):
    if self.damaged:
        self.reward += self.monster_reward_when_damaged
    else:
        self.damaged = True
    if (self.x, self.y) in self.repair_stations:
        self.damaged = False

# Prizes
if (self.x, self.y) == self.prize:
    self.reward += self.prize_reward
    self.prize = None

# Statistics
self.number_steps += 1
self.accumulated_rewards += self.reward
if self.accumulated_rewards < self.min_accumulated_rewards:
    self.min_accumulated_rewards = self.accumulated_rewards
    self.min_step = self.number_steps
if self.accumulated_rewards > 0 and self.reward > self.accumulated_rewards:
    self.zero_crossing = self.number_steps
    self.display(2, ",", self.number_steps, self.accumulated_rewards,
    self.accumulated_rewards/self.number_steps, sep="\t")

return self.reward, (self.x, self.y, self.damaged, self.prize)

The following methods are used by the GUI (Section 13.7, page 332) so that the
states can be shown.

---

```python
### For GUI
def state2pos(self, state):
    """the (x,y) position for the state
    """
    (x, y, damaged, prize) = state
    return (x,y)

def state2goal(self, state):
    """the (x,y) position for the goal
    """
    (x, y, damaged, prize) = state
    return prize
```
13.2 Q Learning

def pos2state(self, pos):
    """the state corresponding to the (x,y) position. The damages and prize are not shown in the GUI"
    (x, y) = pos
    return (x, y, self.damaged, self.prize)

13.2 Q Learning

To run the Q-learning demo, in folder “aipython”, load “rlQLearner.py”, and copy and paste the example queries at the bottom of that file.

__init__(self, role, actions, discount, exploration_strategy=epsilon_greedy, es_kwargs={},
          alpha_fun=lambda _,:0.2, Qinit=0, method="Q_learner"): """
RL_agent.__init__(self, actions)
self.role = role
self.discount = discount
self.exploration_strategy = exploration_strategy
self.es_kwargs = es_kwargs
self.alpha_fun = alpha_fun
self.Qinit = Qinit
self.method = method
self.acc_rewards = 0
self.Q = {}
self.visits = {}

The initial action is a random action. It remembers the state, and initializes the data structures.

```python
def initial_action(self, state):
    """ Returns the initial action; selected at random
    Initialize Data Structures
    ""
    self.state = state
    self.visits[state] = {act:0 for act in self.actions}
    self.action = self.exploration_strategy(state, self.Q[state],
                                           self.visits[state],**self.es_kwargs)
    self.display(2, f"Initial State: {state} Action {self.action}"
                 f"s\ta\tr\ts\tQ")
    return self.action

def select_action(self, reward, next_state):
    """give reward and next state, select next action to be carried out"
    if next_state not in self.visits: # next state not seen before
        self.visits[next_state] = {act:0 for act in self.actions}
        self.visits[self.state][self.action] += 1
        alpha = self.alpha_fun(self.visits[self.state][self.action])
        self.Q[self.state][self.action] += alpha*(
            reward + self.discount * max(self.Q[next_state].values())
            - self.Q[self.state][self.action])
        self.display(2,self.state, self.action, reward, next_state,
                     self.Q[self.state][self.action], sep='\t')
        self.action = self.exploration_strategy(next_state,
                                                self.Q[next_state],
                                                self.visits[next_state],**self.es_kwargs)
        self.state = next_state
    self.display(3,f"Agent {self.role} doing {self.action} in state {self.state}"
                 f"s\ta\tr\tv")
    return self.action
```

The GUI assumes $q(s,a)$ and $v(s)$ functions:

https://aipython.org
13.2. Q Learning

```python
def q(self, s, a):
    if s in self.Q and a in self.Q[s]:
        return self.Q[s][a]
    else:
        return self.Qinit
def v(self, s):
    if s in self.Q:
        return max(self.Q[s].values())
    else:
        return self.Qinit
```

**SARSA** is the same as Q-learning except in the action selection. SARSA changes 3 lines:

```python
class SARSA(Q_learner):
    def __init__(self, *args, **nargs):
        Q_learner.__init__(self, *args, **nargs)
        self.method = "SARSA"
        def select_action(self, reward, next_state):
            """give reward and next state, select next action to be carried out"""
            if next_state not in self.visits: # next state not seen before
                self.visits[next_state] = {act: 0 for act in self.actions}
                self.visits[next_state][self.action] += 1
                alpha = self.alpha_fun(self.visits[next_state][self.action])
                next_action = self.exploration_strategy(next_state, self.Q[next_state],
                                                        self.visits[next_state], **self.es_kwargs)
                self.display(2, self.state, self.action, reward, next_state, self.Q[next_state][self.action], sep='\t')
                self.state = next_state
                self.action = next_action
                self.display(3, f"Agent {self.role} doing {self.action} in state {self.state}
                                                        {self.Q[next_state][self.action]}")
            return self.action
```

### 13.2.1 Exploration Strategies

Two explorations strategies are defined: epsilon-greedy and UCB.

In general an exploration strategy takes two arguments, and some optional arguments depending on the strategy.

[https://aipython.org](https://aipython.org)
• **State** is the state that action is chosen for
• \(Q_s\) is a \{\text{action:} q\text{-value}\} dictionary for the state
• \(V_s\) is a \{\text{action:} visits\} dictionary for the current state; where \text{visits} is the number of times that the action has been carried out in the current state.

```python
def epsilon_greedy(state, Qs, Vs={}, epsilon=0.2):
    """select action given epsilon greedy
    Qs is the \{\text{action:}\ q\text{-value}\} dictionary for current state
    Vs is ignored
    epsilon is the probability of acting randomly
    ""
    if flip(epsilon):
        return random.choice(list(Qs.keys())) # act randomly
    else:
        return argmaxd(Qs) # pick an action with max Q

def ucb(state, Qs, Vs, c=1.4):
    """select action given upper-confidence bound
    Qs is the \{\text{action:}\ q\text{-value}\} dictionary for current state
    Vs is the \{\text{action:}\ visits\} dictionary for current state
    0.01 is to prevent divide-by-zero when Vs[a]==0
    ""
    Ns = sum(Vs.values())
    ucb1 = {a:Qs[a]+c*math.sqrt(Ns/(0.01+Vs[a]))
            for a in Qs.keys()}
    action = argmaxd(ucb1)
    return action
```

**Exercise 13.1** Implement a soft-max action selection. Choose a temperature that works well for the domain. Explain how you picked this temperature. Compare the epsilon-greedy, soft-max and optimism in the face of uncertainty.

### 13.2.2 Testing Q-learning

The first tests are for the 2-action 2-state decision about whether to relax or party (Example 12.29 of ?).
13.2. Q Learning

"""tests whether RL on env has the same (within eps) Q-values as vi on mdp"

mdp1 = mdp(discount=discount)
q1,v1,p1 = mdp1.vi(1000)
ag = learnerClass(env.name, env.actions, discount, **kwargs)
sim = Simulate(ag,env).start()
sim.go(100000)
same = all(abs(ag.q(s,a)-q1[s][a]) < eps for s in mdp1.states for a in mdp1.actions)
assert same, (f"""Unit test failed for {env.name}, in {ag.method} Q=""""+
str({(s,a):ag.q(s,a) for s in mdp1.states for a in mdp1.actions})+
""""in vi Q={q1}"")
print(f"""Unit test passed. For {env.name}, {ag.method} has same Q-value as value iteration"
if __name__ == '__main__':
test_RL(Q_learner, alpha_fun=lambda k:10/(9+k))
# test_RL(SARSA) # should this pass? Why?

#env = Party_env()
env = Env_from_ProblemDomain(MDPtiny())
# Some RL agents with different parameters:
ag = Q_learner(env.name, env.actions, 0.7, method="eps (0.1) greedy" )
ag_ucb = Q_learner(env.name, env.actions, 0.7, exploration_strategy = ucb,
es_kwargs={'c':0.1}, method="ucb")
ag_opt = Q_learner(env.name, env.actions, 0.7, Qinit=100,
es_kwargs={'epsilon':0}, method="optimistic" )
ag_exp_m = Q_learner(env.name, env.actions, 0.7,
es_kwargs={'epsilon':0.5}, method="more explore")
ag_greedy = Q_learner(env.name, env.actions, 0.1, Qinit=100, method="disc 0.1")
sa = SARSA(env.name, env.actions, 0.9, method="SARSA")
sucb = SARSA(env.name, env.actions, 0.9, exploration_strategy = ucb,
es_kwargs={'c':1}, method="SARSA ucb")
sim_ag = Simulate(ag,env).start()

# sim_ag.go(1000)
# ag.Q # get the learned Q-values
# sim_ag.plot()
# sim_ucb = Simulate(ag_ucb,env).start(); sim_ucb.go(1000); sim_ucb.plot()
# Simulate(ag_opt,env).start().go(1000).plot()
# Simulate(ag_exp_m,env).start().go(1000).plot()
# Simulate(sa,env).start().go(1000).plot()
from mdpExamples import MDPtiny
envt = Env_from_ProblemDomain(MDPtiny())
### 13.3 Q-learning with Experience Replay

A bounded buffer remembers values up to size buffer_size. Once it is full, all old experiences have the same chance of being in the buffer.

```python
from rlQLearner import Q_learner
from utilities import flip
import random

class BoundedBuffer(object):
    def __init__(self, buffer_size=1000):
        self.buffer_size = buffer_size
        self.buffer = [0]*buffer_size
        self.number_added = 0

    def add(self, experience):
        if self.number_added < self.buffer_size:
            self.buffer[self.number_added] = experience
        else:
            if flip(self.buffer_size/self.number_added):
                position = random.randrange(self.buffer_size)
                self.buffer[position] = experience
```

---

https://aipython.org  Version 0.9.12  December 22, 2023
A QERLearner does Q-learning with experience replay. It only uses action replay after burn-in number of steps.

```python
class QERLearner(QLearner):
    def __init__(self, role, actions, discount,
                 max_buffer_size=10000,
                 num_updates_per_action=5, burn_in=1000,
                 method="QERLearner", **q_kwargs):
        
        """Q-learner with experience replay
        role is the role of the agent (e.g., in a game)
        actions is the set of actions the agent can do
        discount is the discount factor
        max_buffer_size is the maximum number of past experiences that is remembered
        burn_in is the number of steps before using old experiences
        num_updates_per_action is the number of q-updates for past experiences per action
        q_kwargs are any extra parameters for QLearner
        ""
        QLearner.__init__(self, role, actions, discount, method=method,
                          **q_kwargs)
        self.experience_buffer = BoundedBuffer(max_buffer_size)
        self.num_updates_per_action = num_updates_per_action
        self.burn_in = burn_in

    def select_action(self, reward, next_state):
        """give reward and new state, select next action to be carried out"
        self.experience_buffer.add(((self.state, self.action, reward, next_state))
        # remember experience
        if next_state not in self.Q: # Q and visits are defined on the same states
            self.visits[next_state] = {act: 0 for act in self.actions}
            self.visits[self.state][self.action] += 1
            alpha = self.alpha_fun(self.visits[self.state][self.action])
            self.Q[self.state][self.action] += alpha*(
                reward + self.discount * max(self.Q[next_state].values())
                - self.Q[self.state][self.action])
        self.display(2, self.state, self.action, reward, next_state,
                     self.Q[self.state][self.action], sep='\t')
        self.state = next_state
        # do some updates from experience buffer
```

https://aipython.org  Version 0.9.12  December 22, 2023
13. Reinforcement Learning

```python
if self.experience_buffer.number_added > self.burn_in:
    for i in range(self.num_updates_per_action):
        (s,a,r,ns) = self.experience_buffer.get()
        self.visits[s][a] +=1 # is this correct?
        alpha = self.alpha_fun(self.visits[s][a])
        self.Q[s][a] += alpha * (r +
                            self.discount* max(self.Q[ns][na]
                            for na in self.actions)
                            -self.Q[s][a] )
    ### CHOOSE NEXT ACTION ##
    self.action = self.exploration_strategy(next_state,
                                            self.Q[next_state],
                                            self.visits[next_state],**self.es_kwargs)
    self.display(3,f"Agent {self.role} doing {self.action} in state
                                      {self.state}"
    return self.action
```

---

```python
from rlProblem import Simulate
from rlExamples import Monster_game_env
from rlQLearner import mag1, mag2, mag3
mon_env = Monster_game_env()
mag1ar = Q_ER_learner(mon_env.name, mon_env.actions,0.9,method="Q_ER")
    # Simulate(mag1ar,mon_env).start().go(100000).plot()  
    mag3ar = Q_ER_learner(mon_env.name, mon_env.actions, 0.9, alpha_fun=lambda k:10/(9+k),method="Q_ER alpha=10/(9+k)"
    # Simulate(mag3ar,mon_env).start().go(100000).plot()
from rlQLearner import test_RL
if __name__ == "__main__":
    test_RL(Q_ER_learner)
```

### 13.4 Stochastic Policy Learning Agent

The following agent is like a Q-learning agent but maintains a stochastic policy. The policy is represented as unnormalized counts for each action in a state (like a Dirichlet distribution). This is the code described in Section 14.7.2 and Figure 14.10 of ?.

---

```python
from display import Displayable
import utilities # argmaxall for (element,value) pairs
import matplotlib.pyplot as plt
import random
from rlQLearner import Q_learner
```
13.4. Stochastic Policy Learning Agent

```python
class StochasticPIAgent(Q_learner):
    """This agent maintains the Q-function for each state.
    Chooses the best action using empirical distribution over actions
    """
    def __init__(self, role, actions, discount=0, pi_init=1,
                 method="Stochastic Q_learner", **nargs):
        """
        role is the role of the agent (e.g., in a game)
        actions is the set of actions the agent can do.
        discount is the discount factor (0 is appropriate if there is a
                     single state)
        pi_init gives the prior counts (Dirichlet prior) for the policy
                     (must be >0)
        """
        #self.max_display_level = 3
        Q_learner.__init__(self, role, actions, discount,
                           exploration_strategy=self.action_from_stochastic_policy,
                           method=method, **nargs)
        self.pi_init = pi_init
        self.pi = {}

    def initial_action(self, state):
        """ update policy pi then do initial action from Q_learner
        """
        self.pi[state] = {act:self.pi_init for act in self.actions}
        return Q_learner.initial_action(self, state)

    def action_from_stochastic_policy(self, next_state, qs, vs):
        a_best = utilities.argmaxd(self.Q[self.state])
        self.pi[self.state][a_best] +=1
        if next_state not in self.pi:
            self.pi[next_state] = {act:self.pi_init for act in self.actions}
            return select_from_dist(self.pi[next_state])
        return select_from_dist(self.pi[next_state])

    def normalize(dist):
        """dict is a {value:number} dictionary, where the numbers are all
        non-negative
        returns dict where the numbers sum to one
        """
        tot = sum(dist.values())
        return {var:val/tot for (var,val) in dist.items()}

    def select_from_dist(dist):
        rand = random.random()
        for (act,prob) in normalize(dist).items():
            rand -= prob
            if rand < 0:
                return act
```

The agent can be tested on the reinforcement learning benchmarks:

https://aipython.org
from rlProblem import Simulate
import rlExamples
mon_env = rlExamples.Monster_game_env()
magspi = StochasticPIAgent(mon_env.name, mon_env.actions, 0.9)
#Simulate(magspi, mon_env).start().go(100000).plot()
magspi10 = StochasticPIAgent(mon_env.name, mon_env.actions, 0.9,
    alpha_fun=lambda k:10/(9+k), method="stoch 10/(9+k)")
#Simulate(magspi10, mon_env).start().go(100000).plot()

from rlQLearner import test_RL
if __name__ == "__main__":
    test_RL(StochasticPIAgent, alpha_fun=lambda k:10/(9+k))

Exercis 13.2 Test some other ways to determine the probabilities for the stochastic policy in StochasticPIAgent. (It currently can be seen as using a Dirichlet where the probability represents the proportion of times each action is best plus pseudo-counts).

Replace self.pi[self.state][a_best] +=1 with something like
self.pi[self.state][a_best] *= c for some c > 1. E.g., c = 1.1 so it chooses that action 10% more, independently of the number of times tried. (Try to change the code as little as possible; make it so that either the original or different values of c can be run without changing your code. Warning: watch out for overflow.)

(a) Try for multiple c; which one works best for the Monster game?
(b) Suggest an alternative way to update the probabilities in the policy (e.g., adding δ to policy that is then normalized or some other methods). How well does it work?

13.5 Model-based Reinforcement Learner

To run the demo, in folder “aipython”, load “rlModelLearner.py”, and copy and paste the example queries at the bottom of that file. This assumes Python 3.

A model-based reinforcement learner builds a Markov decision process model of the domain, simultaneously learns the model and plans with that model.

The model-based reinforcement learner used the following data structures:

- **Q[s][a]** is dictionary that, given state s and action a returns the Q-value, the estimate of the future (discounted) value of being in state s and doing action a.

- **R[s][a]** is dictionary that, given a (s,a) state s and action a is the average reward received from doing a in state s.
13.5. Model-based Reinforcement Learner

- \( T[s][a][s'] \) is dictionary that, given states \( s \) and \( s' \) and action \( a \) returns the number of times \( a \) was done in state \( s \) and the result was state \( s' \). Note that \( s' \) is only a key if it has been the result of doing \( a \) in \( s \); there are no 0 counts recorded.

- \( \text{visits}[s][a] \) is dictionary that, given state \( s \) and action \( a \) returns the number of times action \( a \) was carried out in state \( s \). This is the C of Figure 13.6 of ?.

Note that \( \text{visits}[s][a] = \sum_{s'} T[s][a][s'] \) but is stored separately to keep the code more readable.

The main difference to Figure 13.6 of ? is the code below does a fixed number of asynchronous value iteration updates per step.

```
import random
from rlProblem import RL_agent, Simulate, epsilon_greedy, ucb
from display import Displayable
from utilities import argmaxe, flip

class Model_based_reinforcement_learner(RL_agent):
    """A Model-based reinforcement learner""

    def __init__(self, role, actions, discount,
        exploration_strategy=epsilon_greedy, es_kwargs={},
        Qinit=0,
        updates_per_step=10, method="MBR_learner"):
        """role is the role of the agent (e.g., in a game)
        actions is the list of actions the agent can do
        discount is the discount factor
        explore is the proportion of time the agent will explore
        Qinit is the initial value of the Q's
        updates_per_step is the number of AVI updates per action
        label is the label for plotting
        ""
        RL_agent.__init__(self, actions)
        self.role = role
        self.actions = actions
        self.discount = discount
        self.exploration_strategy = exploration_strategy
        self.es_kwargs = es_kwarges
        self.Qinit = Qinit
        self.updates_per_step = updates_per_step
        self.method = method

    def initial_action(self, state):
        """Returns the initial action; selected at random
```
Initialize Data Structures

```python
self.action = RL_agent.initial_action(self, state)
self.T = {self.state: {a: {} for a in self.actions}}
self.visits = {self.state: {a: 0 for a in self.actions}}
self.Q = {self.state: {a: self.Qinit for a in self.actions}}
self.R = {self.state: {a: 0 for a in self.actions}}
self.states_list = [self.state] # list of states encountered
```

```python
self.display(2, f"Initial State: {state} Action {self.action}")
```

```python
return self.action
```

---

```python
def select_action(self, reward, next_state):
    
    """do num_steps of interaction with the environment
    for each action, do updates_per_step iterations of asynchronous
    value iteration
    """
    if next_state not in self.visits: # has not been encountered before
        self.states_list.append(next_state)
        self.visits[next_state] = {a: 0 for a in self.actions}
        self.T[next_state] = {a: {} for a in self.actions}
        self.Q[next_state] = {a: self.Qinit for a in self.actions}
        self.R[next_state] = {a: 0 for a in self.actions}
    else:
        self.T[next_state][self.action] += 1
        self.visits[next_state][self.action] += 1
        self.R[next_state][self.action] +=
        (reward-self.R[next_state][self.action])/self.visits[next_state][self.action]
    st,act = self.state,self.action #initial state-action pair for AVI
    for update in range(self.updates_per_step):
            sum(self.T[st][act][nst]/self.visits[st][act]*self.v(nst)
              for nst in self.T[st][act].keys()))
        st = random.choice(self.states_list)
        act = random.choice(self.actions)
    self.state = next_state
    self.action = self.exploration_strategy(next_state,
        self.Q[next_state],
        self.visits[next_state],**self.es_kwargs)
    return self.action
```

```python
def q(self, state, action):
    if state in self.Q and action in self.Q[state]:
        return self.Q[state][action]
    else:
        return self.Qinit
```
Exercise 13.3 If there was only one update per step, the algorithm can be made simpler and use less space. Explain how. Does it make it more efficient? Is it worthwhile having more than one update per step for the games implemented here?

Exercise 13.4 It is possible to implement the model-based reinforcement learner by replacing \( Q, R, T, \) \( \text{visits}, \) \( \text{res} \) \( \text{states} \) with a single dictionary that, given a state and action returns a tuple corresponding to these data structures. Does this make the algorithm easier to understand? Does this make the algorithm more efficient?

Exercise 13.5 If the states and the actions were mapped into integers, the dictionaries could be implemented perhaps more efficiently as arrays. How does the code need to change? Implement this for the monster game. Is it more efficient?

Exercise 13.6 In \texttt{random\_choice} in the updates of \texttt{select\_action}, all state-action pairs have the same chance of being chosen. Does selecting state-action pairs proportionally to the number of times visited work better than what is implemented? Provide evidence for your answer.

13.6 Reinforcement Learning with Features

To run the demo, in folder “aipython”, load “rlFeatures.py”, and copy and paste the example queries at the bottom of that file. This assumes Python 3.

13.6.1 Representing Features

A feature is a function from state and action. To construct the features for a domain, we construct a function that takes a state and an action and returns the list of all feature values for that state and action. This feature set is redesigned for each problem.

https://aipython.org
party_features3 and party_features4 return lists of feature values for the party decision. party_features4 has one extra feature.

```python
from rlExamples import Monster_game_env
from rlProblem import RL_env

def party_features3(state, action):
    return [1, state=="sick", action=="party"]

def party_features4(state, action):
    return [1, state=="sick", action=="party", state=="sick" and
            action=="party"]
```

**Exercise 13.7** With party_features3 what policies can be discovered? Suppose one action is optimal for one state; what happens in other states.

The monster_features defines the vector of features values for the given state and action.

```python
def monster_features(state, action):
    """returns the list of feature values for the state-action pair
    ""
    assert action in Monster_game_env.actions, f"Monster game, unknown
    action: {action}"
    (x,y,d,p) = state
    # f1: would go to a monster
    f1 = monster_ahead(x,y,action)
    # f2: would crash into wall
    f2 = wall_ahead(x,y,action)
    # f3: action is towards a prize
    f3 = towards_prize(x,y,action,p)
    # f4: damaged and action is toward repair station
    f4 = towards_repair(x,y,action) if d else 0
    # f5: damaged and towards monster
    f5 = 1 if d and f1 else 0
    # f6: damaged
    f6 = 1 if d else 0
    # f7: not damaged
    f7 = 1-f6
    # f8: damaged and prize ahead
    f8 = 1 if d and f3 else 0
    # f9: not damaged and prize ahead
    f9 = 1 if not d and f3 else 0
    features = [1,f1,f2,f3,f4,f5,f6,f7,f8,f9]
    # the next 20 features are for 5 prize locations
    # and 4 distances from outside in all directions
    for pr in Monster_game_env.prize_locs+[None]:
        if p==pr:
            features += [x, 4-x, y, 4-y]
```

else:
    features += \[0, 0, 0, 0\]
# fp04 feature for y when prize is at 0,4
# this knows about the wall to the right of the prize
if p==(0,4):
    if x==0:
        fp04 = y
    elif y<3:
        fp04 = y
    else:
        fp04 = 4-y
else:
    fp04 = 0
features.append(fp04)
return features

def monster_ahead(x,y,action):
    """returns 1 if the location expected to get to by doing
    action from (x,y) can contain a monster.
    """
    if action == "right" and (x+1,y) in Monster_game_env.monster_locs:
        return 1
    elif action == "left" and (x-1,y) in Monster_game_env.monster_locs:
        return 1
    elif action == "up" and (x,y+1) in Monster_game_env.monster_locs:
        return 1
    elif action == "down" and (x,y-1) in Monster_game_env.monster_locs:
        return 1
    else:
        return 0

def wall_ahead(x,y,action):
    """returns 1 if there is a wall in the direction of action from (x,y).
    This is complicated by the internal walls.
    """
    if action == "right" and (x==Monster_game_env.x_dim-1 or (x,y) in
        Monster_game_env.vwalls):
        return 1
    elif action == "left" and (x==0 or (x-1,y) in Monster_game_env.vwalls):
        return 1
    elif action == "up" and y==Monster_game_env.y_dim-1:
        return 1
    elif action == "down" and y==0:
        return 1
    else:
        return 0

def towards_prize(x,y,action,p):
    """action goes in the direction of the prize from (x,y)"""
    if p is None:
The following uses a simpler set of features. In particular, it only considers whether the action will most likely result in a monster position or a wall, and whether the action moves towards the current prize.

```python
def simp_features(state, action):
    # returns a list of feature values for the state-action pair
    assert action in Monster_game_env.actions
    (x, y, d, p) = state
    # f1: would go to a monster
    f1 = monster_ahead(x, y, action)
    if action == "up" and (x>0 and y<4 or x==0 and y<2):
        return 1
    elif action == "left" and x>1:
        return 1
    elif action == "right" and x==0 and y<3:
        return 1
    elif action == "down" and x==0 and y>2:
        return 1
    else:
        return 0
```
13.6. Reinforcement Learning with Features

13.6.2 Feature-based RL learner

This learns a linear function approximation of the Q-values. It requires the function *get_features* that given a state and an action returns a list of values for all of the features. Each environment requires this function to be provided.

```python
import random
from rlProblem import RL_agent, epsilon_greedy, ucb
from display import Displayable
from utilities import argmaxe, flip
import rlGameFeature

class SARSA_LFA_learner(RL_agent):
    """A SARSA with linear function approximation (LFA) learning agent has
    """
    def __init__(self, role, actions, discount,
                 get_features=rlGameFeature.party_features4,
                 exploration_strategy=epsilon_greedy, es_kwargs={},
                 step_size=0.01, winit=0, method="SARSA_LFA"):
        """role is the role of the agent (e.g., in a game)
        actions is the set of actions the agent can do
        discount is the discount factor
        get_features is a function get_features(state,action) -> list of
        feature values
        exploration_strategy is the exploration function, default
        "epsilon_greedy"
        es_kwargs is extra keyword arguments of the exploration_strategy
        step_size is gradient descent step size
        winit is the initial value of the weights
        method gives the method used to implement the role (for plotting)
        """
        RL_agent.__init__(self, actions)
        self.role = role
        self.discount = discount
        self.exploration_strategy = exploration_strategy
        self.es_kwargs = es_kwargs
        self.get_features = get_features
        self.step_size = step_size
        self.winit = winit
        self.method = method
```

The initial action is a random action. It remembers the state, and initializes the data structures.
def initial_action(self, state):
    """ Returns the initial action; selected at random
    Initialize Data Structures
    """
    self.action = RL_agent.initial_action(self, state)
    self.features = self.get_features(state, self.action)
    self.weights = [self.winit for f in self.features]
    self.display(2, f"Initial State: {state} Action {self.action}")
    self.display(2,"s	a	 r	s"
    return self.action

do takes in the number of steps.

def q(self, state, action):
    """ returns Q-value of the state and action for current weights
    ""
    return dot_product(self.weights, self.get_features(state, action))

def v(self, state):
    return max(self.q(state, a) for a in self.actions)

def select_action(self, reward, next_state):
    """ do num_steps of interaction with the environment ""
    feature_values = self.get_features(self.state, self.action)
    oldQ = self.q(self.state, self.action)
    next_action = self.exploration_strategy(next_state,
        {a: self.q(next_state,a)
        for a in self.actions}, {})
    nextQ = self.q(next_state, next_action)
    delta = reward + self.discount * nextQ - oldQ
    for i in range(len(self.weights)):
        self.weights[i] += self.step_size * delta * feature_values[i]
    self.display(2, self.state, self.action, reward, next_state,
        self.q(self.state, self.action), delta, sep='\t')
    self.state = next_state
    self.action = next_action
    return self.action

def show_actions(self, state=None):
    """ prints the value for each action in a state.
    This may be useful for debugging.
    ""
    if state is None:
        state = self.state
    for next_act in self.actions:
        print(next_act, dot_product(self.weights, self.get_features(state, next_act)))
13.6. Reinforcement Learning with Features

```python
def dot_product(l1, l2):
    return sum(e1*e2 for (e1,e2) in zip(l1,l2))
```

Test code:

```python
from rlProblem import Simulate
from rlExamples import Party_env, Monster_game_env
import rlGameFeature
from rlGUI import rlGUI

party = Party_env()
pa3 = SARSA_LFA_learner(party.name, party.actions, 0.9,
    rlGameFeature.party_features3)
    # Simulate(pa3,party).start().go(300).plot()
pa4 = SARSA_LFA_learner(party.name, party.actions, 0.9,
    rlGameFeature.party_features4)
    # Simulate(pa4,party).start().go(300).plot()

mon_env = Monster_game_env()
fa1 = SARSA_LFA_learner(mon_env.name, mon_env.actions, 0.9,
    rlGameFeature.monster_features)
    # Simulate(fa1,mon_env).start().go(100000).plot()
fas1 = SARSA_LFA_learner(mon_env.name, mon_env.actions, 0.9,
    rlGameFeature.simp_features, method="LFA (simp features)")
    #Simulate(fas1,mon_env).start().go(100000).plot()
    # rlGUI(mon_env, SARSA_LFA_learner(mon_env.name, mon_env.actions, 0.9,
    # rlGameFeature.monster_features))

from rQLearner import test_RL
if __name__ == "__main__":
    test_RL(SARSA_LFA_learner, es_kwargs={'epsilon':1})  # random exploration
```

Exercise 13.8 How does the step-size affect performance? Try different step sizes (e.g., 0.1, 0.001, other sizes in between). Explain the behavior you observe. Which step size works best for this example. Explain what evidence you are basing your prediction on.


Exercise 13.10 For each of the following first predict, then plot, then explain the behavior you observed:

(a) SARSA_LFA, Model-based learning (with 1 update per step) and Q-learning for 10,000 steps 20% exploring followed by 10,000 steps 100% exploiting

(b) SARSA_LFA, model-based learning and Q-learning for
   i) 100,000 steps 20% exploring followed by 100,000 steps 100% exploit
   ii) 10,000 steps 20% exploring followed by 190,000 steps 100% exploit

https://aipython.org
(c) Suppose your goal was to have the best accumulated reward after 200,000 steps. You are allowed to change the exploration rate at a fixed number of steps. For each of the methods, which is the best position to start exploiting more? Which method is better? What if you wanted to have the best reward after 10,000 or 1,000 steps?

Based on this evidence, explain when it is preferable to use SARSA, LFA, Model-based learner, or Q-learning.

**Important:** you need to run each algorithm more than once. Your explanation should include the variability as well as the typical behavior.

**Exercise 13.11** In the call to self.exploration_strategy, what should the counts be? (The code above will fail for ucb, for example.) Think about the case where there are too many states. Suppose we are just learning for a neighborhood of a current state (e.g., a fixed number of steps away the from the current state); how could the algorithm be modifies to make sure it has at least explored the close neighborhood of the current state?

### 13.7 GUI for RL

This implements an interactive graphical user interface for reinforcement learners. It lets the uses choose the actions and visualize the value function and/or the q-function.

**Warning:** Exit is not working, because it is only interuppting one thread.

---

```python
import matplotlib.pyplot as plt
from matplotlib.widgets import Button, CheckButtons, TextBox
from rlProblem import Simulate

class rGUI(object):
    def __init__(self, env, agent):
        """
        ""
        self.env = env
        self.agent = agent
        self.state = self.env.state
        self.x_dim = env.x_dim
        self.y_dim = env.y_dim
        if 'offsets' in vars(env): # 'offsets' is defined in environment
            self.offsets = env.offsets
        else: # should be more general
            self.offsets = {'right':(0.25,0), 'up':(0,0.25),
                            'left':(-0.25,0), 'down':(0,-0.25))
        # replace the exploration strategy with GUI
        self.orig_exp_strategy = self.agent.exploration_strategy
        self.agent.exploration_strategy = self.actionFromGUI
        self.do_steps = 0
        self.quit = False
```

---

[https://aipython.org](https://aipython.org)
self.action = None

def go(self):
    self.q = self.agent.q
    self.v = self.agent.v
    try:
        self.fig, self.ax = plt.subplots()
        plt.subplots_adjust(bottom=0.2)
        self.actButtons =
        {self.fig.text(0.8+self.offsets[a][0]*0.4,0.1+self.offsets[a][1]*0.1,a,
            bbox={'boxstyle': 'square', 'color': 'yellow', 'ec': 'black'},
            picker=True): a #, fontsize=fontsize): a
        for a in self.env.actions}
        self.fig.canvas.mpl_connect('pick_event', self.sel_action)
        self.sim = Simulate(self.agent, self.env)
        self.show()
        self.sim.start()
        self.sim.go(1000000000000)  # go forever
    except ExitGUI:
        plt.close()

def show(self):
    #plt.ion() # interactive (why doesn't this work?)
    self.qcheck = CheckButtons(plt.axes([0.2, 0.05, 0.25, 0.075]),
        ["show q-values", "show policy", "show visits"])
    self.qcheck.on_clicked(self.show_vals)
    self.font_box = TextBox(plt.axes([0.125, 0.05, 0.05, 0.05]), "Font: ",
        textalignment="center")
    self.font_box.on_submit(self.set_font_size)
    self.font_box.set_val(str(plt.rcParams['font.size']))
    self.step_box = TextBox(plt.axes([0.5, 0.05, 0.1, 0.05]), ",",
        textalignment="center")
    self.step_box.set_val("100")
    self.stepsButton = Button(plt.axes([0.6, 0.05, 0.075, 0.05]), "steps",
        color='yellow')
    self.stepsButton.on_clicked(self.steps)
    self.exitButton = Button(plt.axes([0.0, 0.05, 0.05, 0.05]), "exit",
        color='yellow')
    self.exitButton.on_clicked(self.exit)
    self.show_vals(None)

    def set_font_size(self, s):
        plt.rcParams.update({'font.size': eval(s)})
        plt.draw()

    def exit(self, s):
        self.quit = True
raise ExitGUI

def show_vals(self, event):
    self.ax.cla()
    self.ax.set_title(f"{self.sim.step}: State: {self.state} Reward:
    {self.env.reward} Sum rewards: {self.sim.sum_rewards}")
    array = [[self.v(self.env.pos2state((x,y))) for x in
        range(self.x_dim)]
        for y in range(self.y_dim)]
    self.ax.pcolormesh([x-0.5 for x in range(self.x_dim+1)],
        [x-0.5 for x in range(self.y_dim+1)],
        array, edgecolors='black', cmap='summer')
    if self.qcheck.get_status()[1]: # "show policy"
        for x in range(self.x_dim):
            for y in range(self.y_dim):
                state = self.env.pos2state((x,y))
                maxv = max(self.agent.q(state, a) for a in
                    self.env.actions)
                for a in self.env.actions:
                    xoff, yoff = self.offsets[a]
                    if self.agent.q(state, a) == maxv:
                        # draw arrow in appropriate direction
                        self.ax.arrow(x, y, xoff*2, yoff*2,
                            color='red', width=0.05, head_width=0.2,
                            length_includes_head=True)
    if goal := self.env.state2goal(self.state):
        self.ax.add_patch(plt.Circle(goal, 0.1, color='lime'))
        self.ax.add_patch(plt.Circle(self.env.state2pos(self.state), 0.1,
            color='w'))
    if self.qcheck.get_status()[0]: # "show q-values"
        self.show_q(event)
    elif self.qcheck.get_status()[2] and 'visits' in vars(self.agent):
        # "show visits"
        self.show_visits(event)
    else:
        self.show_v(event)
        self.ax.set_xticks(range(self.x_dim))
        self.ax.set_xticklabels(range(self.x_dim))
        self.ax.set_yticks(range(self.y_dim))
        self.ax.set_yticklabels(range(self.y_dim))
        plt.draw()

def sel_action(self, event):
    self.action = self.actButtons[event.artist]

def show_v(self, event):
    """show values"""
for \texttt{x} in \texttt{range}(self.x\_dim):
    for \texttt{y} in \texttt{range}(self.y\_dim):
        \texttt{state} = self.env.pos2state((x,y))
        \texttt{self.ax.text(x,y,\"\{val:.2f\}\".format(val=self.agent.v(state)),ha='center')}

def show\_q(self,event):
    """show q-values"""
    for \texttt{x} in \texttt{range}(self.x\_dim):
        for \texttt{y} in \texttt{range}(self.y\_dim):
            \texttt{state} = self.env.pos2state((x,y))
            \texttt{for a in self.env.actions:}
                \texttt{xoff, yoff = self.offsets[a]}
                \texttt{self.ax.text(x+xoff,y+yoff,}
                \texttt{\"\{val:.2f\}\".format(val=self.agent.q(state,a)),ha='center')}

def show\_visits(self,event):
    """show q-values"""
    for \texttt{x} in \texttt{range}(self.x\_dim):
        for \texttt{y} in \texttt{range}(self.y\_dim):
            \texttt{state} = self.env.pos2state((x,y))
            \texttt{for a in self.env.actions:}
                \texttt{xoff, yoff = self.offsets[a]}
                \texttt{if state in self.agent.visits and a in}
                \texttt{self.agent.visits[state]:}
                    \texttt{num\_visits = self.agent.visits[state][a]}
                \texttt{else:}
                    \texttt{num\_visits = 0}
                \texttt{self.ax.text(x+xoff,y+yoff,}
                \texttt{str(num\_visits),ha='center')}

def steps(self,event):
    """do the steps given in step box"""
    \texttt{num\_steps = int(self.step\_box.text)}
    \texttt{if num\_steps > 0:}
        \texttt{self.do\_steps = num\_steps-1}
        \texttt{self.action = self.action\_from\_orig\_exp\_strategy()}

def action\_from\_orig\_exp\_strategy(self):
    """returns the action from the original explorations strategy""
    \texttt{visits = self.agent.visits[self.state] if 'visits' in}
    \texttt{vars(self.agent) else {}}
    \texttt{return}
    \texttt{self.orig\_exp\_strategy(self.state,{a:self.agent.q(self.state,a)}
    \texttt{for a in self.agent.actions},
    \texttt{visits,**self.agent.es\_kwargs})}

def action\_from\_GUI(self, state, *args, **kwargs):
    """called as the exploration strategy by the RL agent.
    returns an action, either from the GUI or the original exploration strategy""

https://aipython.org  Version 0.9.12  December 22, 2023
```python

""
self.state = state
if self.do_steps > 0:  # use the original
    self.do_steps -= 1
    return self.action_from_orig_exp_strategy()
else:  # get action from the user
    self.show_vals(None)
    while self.action == None and not self.quit:  # wait for user
        action
        plt.pause(0.05)  # controls reaction time of GUI
        act = self.action
        self.action = None
    return act

class ExitGUI(Exception):
    pass

from rlExamples import Monster_game_env
from mdpExamples import MDPtiny, Monster_game
from rlQlearner import Q_learner, SARSA
from rlStochasticPolicy import StochasticPIAgent
from rlProblem import Env_from_ProblemDomain, epsilon_greedy, ucb
env = Env_from_ProblemDomain(MDPtiny())
# env = Env_from_ProblemDomain(Monster_game())
# gui = rlGUI(env, Q_learner("Q", env.actions, 0.9)); gui.go()
# gui = rlGUI(env, SARSA("Q", env.actions, 0.9)); gui.go()
# gui = rlGUI(env, SARSA("Q", env.actions, 0.9, alpha_fun=lambda k:10/(9+k))); gui.go()
# gui = rlGUI(env, SARSA-UCB, env.actions, 0.9,
# exploration_strategy = ucb, es_kwargs={'c':0.1}); gui.go()
# gui = rlGUI(env, StochasticPIAgent("Q", env.actions, 0.9,
# alpha_fun=lambda k:10/(9+k))); gui.go()
```

Chapter 14

Multiagent Systems

14.1 Minimax

Here we consider two-player zero-sum games. Here a player only wins when another player loses. This can be modeled as where there is a single utility which one agent (the maximizing agent) is trying to minimize and the other agent (the minimizing agent) is trying to minimize.

14.1.1 Creating a two-player game

```python
from display import Displayable

class Node(Displayable):
    """A node in a search tree. It has a name a string
    isMax is True if it is a maximizing node, otherwise it is minimizing
    node
    children is the list of children
    value is what it evaluates to if it is a leaf.
    ""
    def __init__(self, name, isMax, value, children):
        self.name = name
        self.isMax = isMax
        self.value = value
        self.allchildren = children

    def isLeaf(self):
        """returns true of this is a leaf node"
        return self.allchildren is None

masProblem.py — A Multiagent Problem
```

337
The following gives the tree from Figure 11.5 of the book. Note how 888 is used as a value here, but never appears in the trace.

```python
fig10_5 = Node("a", True, None, [
    Node("b", False, None, [
        Node("d", True, None, [
            Node("h", False, None, [
                Node("h1", True, 7, None),
                Node("h2", True, 9, None)],
            Node("i", False, None, [
                Node("i1", True, 6, None),
                Node("i2", True, 888, None)]),
        Node("e", True, None, [
            Node("j", False, None, [
                Node("j1", True, 11, None),
                Node("j2", True, 12, None)],
            Node("k", False, None, [
                Node("k1", True, 888, None),
                Node("k2", True, 888, None)]),
        Node("c", False, None, [
            Node("f", True, None, [
                Node("l", False, None, [
                    Node("l1", True, 5, None),
                    Node("l2", True, 888, None)],
                Node("m", False, None, [
                    Node("m1", True, 4, None),
                    Node("m2", True, 888, None)]),
            Node("g", True, None, [
                Node("n", False, None, [
                    Node("n1", True, 888, None),
                    Node("n2", True, 888, None)],
                Node("o", False, None, [
                    Node("o1", True, 888, None),
                    Node("o2", True, 888, None)]))])])])])])
```

The following is a representation of a magic-sum game, where players take turns picking a number in the range $[1, 9]$, and the first player to have 3 numbers that sum to 15 wins. Note that this is a syntactic variant of tic-tac-toe or naughts and crosses. To see this, consider the numbers on a magic square (Figure 14.1); 3 numbers that add to 15 correspond exactly to the winning positions

https://aipython.org

Version 0.9.12 December 22, 2023
of tic-tac-toe played on the magic square.

Note that we do not remove symmetries. (What are the symmetries? How do the symmetries of tic-tac-toe translate here?)

```python
class Magic_sum(Node):
    def __init__(self, xmove=True, last_move=None, available=[1, 2, 3, 4, 5, 6, 7, 8, 9], x=[], o=[]):
        """This is a node in the search for the magic-sum game.
        xmove is True if the next move belongs to X.
        last_move is the number selected in the last move
        available is the list of numbers that are available to be chosen
        x is the list of numbers already chosen by x
        o is the list of numbers already chosen by o
        """
        self.isMax = self.xmove = xmove
        self.last_move = last_move
        self.available = available
        self.x = x
        self.o = o
        self.allchildren = None #computed on demand
        lm = str(last_move)
        self.name = "start" if not last_move else "o="+lm if xmove else "x="+lm

    def children(self):
        if self.allchildren is None:
            if self.xmove:
                self.allchildren = [
                    Magic_sum(xmove = not self.xmove,
                                last_move = sel,
                                available = [e for e in self.available if e is not sel],
                                x = self.x+[sel],
                                o = self.o)
                        for sel in self.available]
            else:
                self.allchildren = [
                    Magic_sum(xmove = not self.xmove,
                                last_move = sel,
                                available = [e for e in self.available if e is not sel],
                                x = self.x+[sel],
                                o = self.o
]```
x = self.x,
o = self.o+[sel])
for sel in self.available
    return self.allchildren

def isLeaf(self):
    """A leaf has no numbers available or is a win for one of the
players.
We only need to check for a win for o if it is currently x's turn,
and only check for a win for x if it is o's turn (otherwise it would
have been a win earlier).
""
    return (self.available == [] or
        (sum_to_15(self.last_move,self.o)
            if self.xmove
            else sum_to_15(self.last_move,self.x)))

def evaluate(self):
    if self.xmove and sum_to_15(self.last_move,self.o):
        return -1
    elif not self.xmove and sum_to_15(self.last_move,self.x):
        return 1
    else:
        return 0

def sum_to_15(last,selected):
    """is true if last, together with two other elements of selected sum to
15.
""
    return any(last+a+b == 15
        for a in selected
        if a != last
        for b in selected
        if b != last and b != a)

14.1.2 Minimax and α-β Pruning

This is a naive depth-first minimax algorithm:

def minimax(node,depth):
    """returns the value of node, and a best path for the agents
    ""
    if node.isLeaf():
        return node.evaluate(),None
    elif node.isMax:
        max_score = float("-inf")
        max_path = None
        for C in node.children():
            score,path = minimax(C,depth+1)
            if score > max_score:
                max_score = score
                max_path = C.name,path

https://aipython.org
return max_score, max_path

else:
    min_score = float("inf")
    min_path = None
    for C in node.children():
        score, path = minimax(C, depth + 1)
        if score < min_score:
            min_score = score
            min_path = C.name, path
    return min_score, min_path

The following is a depth-first minimax with $\alpha$-$\beta$ pruning. It returns the value for a node as well as a best path for the agents.

```python
def minimax_alpha_beta(node, alpha, beta, depth=0):
    """node is a Node, alpha and beta are cutoffs, depth is the depth
    returns value, path
    where path is a sequence of nodes that results in the value
    """
    node.display(2, "*depth," + minimax_alpha_beta(" + node.name, " + , alpha, ", beta, " + " + ")
    best=None # only used if it will be pruned
    if node.isLeaf():
        node.display(2, "*depth," + returning leaf value", node.evaluate())
        return node.evaluate(), None
    elif node.isMax:
        for C in node.children():
            score, path = minimax_alpha_beta(C, alpha, beta, depth + 1)
            if score >= beta: # beta pruning
                node.display(2, "*depth," + pruned due to
                beta=" + , beta, " + C=" + C.name)
                return score, None
            if score > alpha:
                alpha = score
                best = C.name, path
                node.display(2, "*depth," + returning max alpha", alpha, "best", best)
                return alpha, best
    else:
        for C in node.children():
            score, path = minimax_alpha_beta(C, alpha, beta, depth + 1)
            if score <= alpha: # alpha pruning
                node.display(2, "*depth," + pruned due to
                alpha=" + alpha, " + C=" + C.name)
                return score, None
            if score < beta:
                beta = score
                best = C.name, path
                node.display(2, "*depth," + returning min beta", beta, "best", best)
                return beta, best
```

Testing:

[https://aipython.org](https://aipython.org)  
Version 0.9.12  
December 22, 2023
14. Multiagent Systems

The next code is for multiple agents that learn when interacting with other agents. The main difference with the simulator from the last chapter is that the games take actions from all agents and provide a separate reward to each agent. Any of the reinforcement learning agents from the last chapter can be used.

14.2 Multiagent Learning

The simulation for a game passes the joint action from all agents to the environment, which returns a tuple of rewards – one for each agent – and the next state.

```python
from display import Displayable
import matplotlib.pyplot as plt
from rlProblem import RL_agent

class SimulateGame(Displayable):
    def __init__(self, game, agent_types):
        #self.max_display_level = 3
        self.game = game
        self.agents = [agent_types[i](game.players[i], game.actions[i], 0) for i in range(game.num_agents)] # list of agents
        self.action_dists = [{act:0 for act in game.actions[i]} for i in range(game.num_agents)]
        self.action_history = []
        self.state_history = []
        self.reward_history = []
```

https://aipython.org
self.dist = {}
self.dist_history = []
self.actions = tuple(ag.initial_action(game.initial_state) for ag in self.agents)
self.num_steps = 0

def go(self, steps):
    for i in range(steps):
        self.num_steps += 1
        (self.rewards, state) = self.game.play(self.actions)
        self.display(3, f"In go rewards={self.rewards}, state={state}")
        self.reward_history.append(self.rewards)
        self.state_history.append(state)
        self.actions = tuple(agent.select_action(reward, state)
                             for (agent, reward) in zip(self.agents, self.rewards))
        self.action_history.append(self.actions)
        for i in range(self.game.num_agents):
            self.action_dists[i][self.actions[i]] += 1
            self.dist_history.append(dict((a, self.dist_history[-1][i][a] / sum(self.dist_history[-1][i].values())) for a in self.game.actions[i])
                for i in range(self.game.num_agents))
#return self.reward_history, self.action_history

def action_dist(self, which_actions=[1,1]):
    """ which actions is [a0,a1]
    returns the empirical distribution of actions for agents,
    where ai specifies the index of the actions for agent i
    remove this???
    """
    return [sum(1 for a in sim.action_history
               if a[i]==gm.actions[i][which_actions[i]])/len(sim.action_history)
               for i in range(2)]

The plotting shows how the empirical distributions of the first two agents changes as the learning continues.
plt.xlabel(f"Probability \{self.game.players[0]
{self.agents[0].actions[x_action]}")
plt.ylabel(f"Probability \{self.game.players[1]
{self.agents[1].actions[y_action]}")
plt.plot([self.dist_history[i][0][x_act]/
sum(self.dist_history[i][0].values())
for i in range(len(self.dist_history))],
[self.dist_history[i][1][y_act]/
sum(self.dist_history[i][1].values())
for i in range(len(self.dist_history))])
#plt.legend()
#plt.savefig('soccerplot.pdf') # if you want to save plot
plt.show()
14.2. Multiagent Learning

```python
return ({'left', 'left'}: (0.6, 0.4),
       {'left', 'right'}: (0.3, 0.7),
       {'right', 'left'}: (0.2, 0.8),
       {'right', 'right'}: (0.9, 0.1))
```

```python
class GameShow(Displayable):
    def __init__(self):
        self.num_agents = 2
        self.states = ['s']
        self.initial_state = 's'
        self.actions = [['takes', 'gives']] * 2
        self.players = ['Agent 1', 'Agent 2']

    def play(self, actions):
        return ({'takes', 'takes'}: (1, 1),
                {'takes', 'gives'}: (11, 0),
                {'gives', 'takes'}: (0, 11),
                {'gives', 'gives'}: (10, 10))
```

```python
class UniqueNEGameExample(Displayable):
    def __init__(self):
        self.num_agents = 2
        self.states = ['s']
        self.initial_state = 's'
        self.actions = [['a1', 'b1', 'c1'], ['d2', 'e2', 'f2']]
        self.players = ['agent 1 does', 'agent 2 does']

    def play(self, actions):
        return ({'a1', 'd2'}: (3, 5),
                {'a1', 'e2'}: (5, 1),
                {'a1', 'f2'}: (1, 2),
                {'b1', 'd2'}: (1, 1),
                {'b1', 'e2'}: (2, 9),
                {'b1', 'f2'}: (6, 4),
                {'c1', 'd2'}: (2, 6),
                {'c1', 'e2'}: (4, 7),
                {'c1', 'f2'}: (0, 8))
```

14.2.3 Testing Games and Environments

---

```python
# Choose a game:
# gm = ShoppingGame()
# gm = SoccerGame()
# gm = GameShow()
```

[https://aipython.org](https://aipython.org)  
Version 0.9.12  
December 22, 2023
# gm = UniqueNEGameExample()

from rlQLearner import Q_learner
from rlProblem import RL_agent
from rlStochasticPolicy import StochasticPIAgent

# Choose one of the combinations of learners:
# sim=SimulateGame(gm,[StochasticPIAgent, StochasticPIAgent]); sim.go(10000)
# sim=SimulateGame(gm,[Q_learner, Q_learner]); sim.go(10000)
# sim=SimulateGame(gm,[Q_learner, StochasticPIAgent]); sim.go(10000)

# sim.plot_dynamics()

# empirical proportion that agents did their action at index 1:
# sim.action_dist([1,1])

# (unnormalized) empirical distribution for agent 0
# sim.agents[0].dist

Exercise 14.1 Consider the alternative ways to implement stochastic policy iteration of Exercise 13.2.

(a) What value(s) of $c$ converge for the soccer game? Explain your results.
(b) Suggest another method that works well for the soccer game, the other games and other RL environments.

Exercise 14.2 For the soccer game, how can a Q_learner be regularly beaten? Assume that random number generator is secret. (Hint: can you predict what it will do?) What happens when it is played against an adversary that knows how it learns? What happens if two of these agents are played against each other? Can a StochasticPIAgent be defeated in the same way?

Exercise 14.3 Try the game show game (prisoner’s dilemma) with two StochasticPIAgent agents and alpha_fun=lambda k:0.1. Try also 0.01. Why does this work qualitatively different? Is this better?
Chapter 15

Individuals and Relations

Here we implement top-down proofs for datalog and logic programming. This is much less efficient than Prolog, which is typically implemented by compiling to an abstract machine. If you want to do serious work, we suggest using Prolog; SWI Prolog [https://www.swi-prolog.org] is good.

15.1 Representing Datalog and Logic Programs

The following extends the knowledge bases of Chapter 5 to include logical variables. In that chapter, atoms did not have structure and were represented as strings. Here atoms can have arguments including variables (defined below) and constants (represented by strings).

Function symbols have the same representation as atoms. To make unification simpler and for when we treat clauses as data, Func is defined as an abbreviation for Atom.

```python
from display import Displayable
import logicProblem

class Var(Displayable):
    """A logical variable""
    def __init__(self, name):
        """name""
        self.name = name
    def __str__(self):
        return self.name
    __repr__ = __str__
```

347
def __eq__(self, other):
    return isinstance(other, Var) and self.name == other.name

def __hash__(self):
    return hash(self.name)

class Atom(object):
    """An atom""
    def __init__(self, name, args):
        self.name = name
        self.args = args

    def __str__(self):
        return f"{self.name}({', '.join(str(a) for a in self.args)})"

    __repr__ = __str__

Func = Atom  # same syntax is used for function symbols

The following extends Clause of Section 5.1 to also include a set of logical variables in the clause. It also allows for atoms that are strings (as in Chapter 5) and makes them into atoms.

```python
class Clause(logicProblem.Clause):
    next_index = 0
    def __init__(self, head, *args, **nargs):
        if not isinstance(head, Atom):
            head = Atom(head)
        logicProblem.Clause.__init__(self, head, *args, **nargs)
        self.logical_variables = log_vars([self.head, self.body], set())

    def rename(self):
        """create a unique copy of the clause""
        if self.logical_variables:
            sub = {v: Var(f"{v.name}_{Clause.next_index}") for v in self.logical_variables}
            Clause.next_index += 1
            return Clause(apply(self.head, sub), apply(self.body, sub))
        else:
            return self

def log_vars(exp, vs):
    """the union the logical variables in exp and the set vs""
    if isinstance(exp, Var):
        return {exp} | vs
    elif isinstance(exp, Atom):
        return log_vars(exp.name, log_vars(exp.args, vs))
    elif isinstance(exp, (list, tuple)):
        for e in exp:
            vs = log_vars(e, vs)
    return vs
```
15.2 Unification

```python
unifdisp = Var(None) # for display

def unify(t1,t2):
    e = [(t1,t2)]
    s = {} # empty dictionary
    while e:
        (a,b) = e.pop()
        unifdisp.display(2,f"unifying{(a,b)}, e={e},s={s})"
        if a != b:
            if isinstance(a,Var):
                e = apply(e,{a:b})
                s = apply(s,{a:b})
                s[a]=b
            elif isinstance(b,Var):
                e = apply(e,{b:a})
                s = apply(s,{b:a})
                s[b]=a
            elif isinstance(a,Atom) and isinstance(b,Atom) and
                a.name==b.name and len(a.args)==len(b.args):
                e += zip(a.args,b.args)
            elif isinstance(a,(list,tuple)) and isinstance(b,(list,tuple))
                and len(a)==len(b):
                e += zip(a,b)
        else:
            return False
    return s

def apply(e,sub):
    """e is an expression
    sub is a {var:val} dictionary
    returns e with all occurrence of var replaces with val"
    if isinstance(e,Var) and e in sub:
        return sub[e]
    if isinstance(e,Atom):
        return Atom(e.name, apply(e.args,sub))
    if isinstance(e,list):
        return [apply(a,sub) for a in e]
    if isinstance(e,tuple):
        return tuple(apply(a,sub) for a in e)
    if isinstance(e,dict):
        return {k:apply(v,sub) for (k,v) in e.items()}
    else:
        return e
```

Test cases:
15.3 Knowledge Bases

The following modifies KB of Section 5.1 so that clause indexing is only on the predicate symbol of the head of clauses.

```python
class KB(logicProblem.KB):
    """A first-order knowledge base.
    only the indexing is changed to index on name of the head.""
    def add_clause(self, c):
        """Add clause c to clause dictionary""
        if c.head.name in self.atom_to_clauses:
            self.atom_to_clauses[c.head.name].append(c)
        else:
            self.atom_to_clauses[c.head.name] = [c]
```

simp_KB is the simple knowledge base of Figure 15.1 of ?.

```python
from logicRelation import Var, Atom, Clause, KB
simp_KB = KB([  
    Clause(Atom('in', ['kim', 'r123'])),
    Clause(Atom('part_of', ['r123', 'cs_building'])),
    Clause(Atom('in', [Var('X'), Var('Y')]),
        [Atom('part_of', [Var('Z'), Var('Y')]),
        Atom('in', [Var('X'), Var('Z')])])  
])
```

elect_KB is the relational version of the knowledge base for the electrical system of a house, as described in Example 15.11 of ?.

```python
# define abbreviations to make the clauses more readable:
def lit(x): return Atom('lit', [x])
def light(x): return Atom('light', [x])
def ok(x): return Atom('ok', [x])
def live(x): return Atom('live', [x])
def connected_to(x,y): return Atom('connected_to', [x,y])
def up(x): return Atom('up', [x])
```
15.3. Knowledge Bases

```python
def down(x):
    return Atom('down', [x])
L = Var('L')
W = Var('W')
W1 = Var('W1')
elect_KB = KB([  
    # lit(L) is true if light L is lit.
    Clause(lit(L),  
        [light(L),  
         ok(L),  
         live(L)]),
    # live(W) is true if W is live (i.e., current will flow through it)
    Clause(live(W),  
        [connected_to(W,W1),  
         live(W1)]),
    Clause(live('outside')),  
    # light(L) is true if L is a light
    Clause(light('l1')),  
    Clause(light('l2')),  
    # connected_to(W0,W1) is true if W0 is connected to W1 such that
    # current will flow from W1 to W0.
    Clause(connected_to('l1','w0')),  
    Clause(connected_to('w0','w1'),  
        [up('s2'), ok('s2')]),
    Clause(connected_to('w0','w2'),  
        [down('s2'), ok('s2')]),
    Clause(connected_to('w1','w3'),  
        [up('s1'), ok('s1')]),
    Clause(connected_to('w2','w3'),  
        [down('s1'), ok('s1')]),
    Clause(connected_to('l2','w4')),  
    Clause(connected_to('w4','w3'),  
        [up('s3'), ok('s3')]),
    Clause(connected_to('p1','w3')),  
    Clause(connected_to('w3','w5'),  
        [ok('cb1')]),
    Clause(connected_to('p2','w6')),  
    Clause(connected_to('w6','w5'),  
        [ok('cb2')]),
    Clause(connected_to('w5','outside'),  
        [ok('outside_connection')]),
    # up(S) is true if switch S is up
    # down(S) is true if switch S is down
```

https://aipython.org  
Version 0.9.12  
December 22, 2023
15.4 Top-down Proof Procedure

The top-down proof procedure is the one defined in Section 15.5.4 of and shown in Figure 15.5. It is like prove defined in Section 5.3. It implements the iterator interface so that answers can be generated one at a time (or put in a list), and returns answers. To implement “choose” it loops over all alternatives and yields (returns one element at a time) the successful proofs.

```python
def ask(self, query):
    """self is the current KB
    query is a list of atoms to be proved
    generates (variable:value) dictionary"
    qvars = list(log_vars(query, set()))
    for ans in self.prove(qvars, query):
        yield {x:v for (x,v) in zip(qvars,ans)}

def ask_all(self, query):
    """returns a list of all answers to the query given kb"
    return list(self.ask(query))

def ask_one(self, query):
    """returns an answer to the query given kb or None of there are no answers"
    for ans in self.ask(query):
        return ans

def prove(self, ans, ans_body, indent=""):
    """enumerates the proofs for ans_body
    ans_body is a list of atoms to be proved
    ans is the list of values of the query variables"
    self.display(2,indent,f"(yes({ans}) <=", & ".join(str(a) for a in ans_body))
    if ans_body==[]:
        yield ans
    else:
        selected, remaining = self.select_atom(ans_body)
        if self.built_in(selected):
```

[https://aipython.org](https://aipython.org)
15.4. Top-down Proof Procedure

```python
yield from self.eval_built_in(ans, selected, remaining, indent)
else:
    for chosen_clause in self.atom_to_clauses[selected.name]:
        clause = chosen_clause.rename() # rename variables
        sub = unify(selected, clause.head)
        if sub is not False:
            self.display(3, indent, "KB.prove: selected=", selected, "clause=", clause, "sub=", sub)
            resans = apply(ans, sub)
            new_ans_body = apply(clause.body + remaining, sub)
            yield from self.prove(resans, new_ans_body, indent + "")

def select_atom(self, lst):
    """given list of atoms, return (selected atom, remaining atoms)"""
    return lst[0], lst[1:]

def built_in(self, atom):
    return atom.name in ['lt', 'triple']

def eval_built_in(self, ans, selected, remaining, indent):
    if selected.name == 'lt': # less than
        [a1, a2] = selected.args
        if a1 < a2:
            yield from self.prove(ans, remaining, indent + "")
    if selected.name == 'triple': # use triple store (AIFCA Ch 16)
        yield from self.prove_triple(ans, selected, remaining, indent)
```

---

Example Queries:

```python
# simp_KB.max_display_level = 2 # show trace
# ask_all(simp_KB, [Atom('in', [Var('A'), Var('B')])])

def test_ask_all(kb=simp_KB, query=[Atom('in', [Var('A'), Var('B')])])
    res=[[Var('A'): 'kim', Var('B'): 'r123'], [Var('A'): 'kim', Var('B'): 'cs_building']]:
    ans= kb.ask_all(query)
    assert ans == res, f"ask_all({query}) gave answer {ans}"
    print("ask_all: Passed unit test")

if __name__ == "__main__":
    test_ask_all()

# elect_KB.max_display_level = 2 # show trace
# elect_KB.ask_all([light('l1')])
# elect_KB.ask_all([light('l6')])
# elect_KB.ask_all([up(Var('X'))])
# elect_KB.ask_all([connected_to('w0', W)])
```
Exercise 15.1  Implement ask-ther-user similar to Section 5.3. Augment this by allowing the user to specify which instances satisfy a instances. For example, by asking the user “for what X is w1 connected to X?”; or perhaps in a more user friendly way.

15.5 Logic Program Example

The following is an append program and the query of Example 15.30 of ?.

append(nil,W,W).
append(c(A,X),Y,c(A,Z)) <-
    append(X,Y,Z).

The term c(A,X) is represented using Atom
    In Prolog syntax:
append(nil,W,W).
append([A|X],Y,[A|Z]) :-
    append(X,Y,Z).

The value if lst is [1,i,s,t]. The query is
?
append(F,[L],[1,i,s,t]).

We first define some constants and functions to make it more readable.

---

logicRelation.py — (continued)
---

```
A = Var('A')
W = Var('W')
X = Var('X')
Y = Var('Y')
Z = Var('Z')
def cons(h,t): return Atom('cons',[h,t])
def append(a,b,c): return Atom('append',[a,b,c])
app_KB = KB([
    Clause(append('nil',W,W)),
    Clause(append(cons(A,X),Y,cons(A,Z)),
       [append(X,Y,Z)])
])
F = Var('F')
```
15.5. Logic Program Example

\[ \text{lst} = \text{cons('l', cons('i', cons('s', cons('t', 'nil'))))} \]

# app_KB.max_display_level = 2 # show derivation

# Think about the expected answer before trying:

#ask_all(app_KB, [append(X, cons('s', Y), L)])

#ask_all(app_KB, [append(lst, lst, L), append(X, cons('s', Y), L)])
Chapter 16

Knowledge Graphs and Ontologies

16.1 Triple Store

A triple store provides efficient indexing for triples. For any combination of the subject-verb-object being provided or not, it can efficiently retrieve the corresponding triples. This should be as fast as commercial triple stores, but can probably handle fewer triples, as it is not optimized for space. It also have fewer bells and whistles (e.g., ways to visualize triples and traverse the graph).

A triple store implements an index that covers all cases of where the subject, verb, or object are provided or not. The unspecified parts are given using Q (with value '?'). Thus, for example, index[(Q, vrb, Q)] is the list of triples with verb vrb. index[(sub, Q, obj)] is the list of triples with subject sub and object obj.

```python
from display import Displayable

class TripleStore(Displayable):
    Q = '?'  # query position

    def __init__(self):
        self.index = {}

    def add(self, triple):
        (sb, vb, ob) = triple
        Q = self.Q  # make it easier to read
        add_to_index(self.index, (Q, Q, Q), triple)
        add_to_index(self.index, (Q, vb, ob), triple)
```

357
add_to_index(self.index, (Q,vb,Q), triple)
add_to_index(self.index, (Q,vb,ob), triple)
add_to_index(self.index, (sb,Q,Q), triple)
add_to_index(self.index, (sb,Q,ob), triple)
add_to_index(self.index, (sb,vb,Q), triple)
add_to_index(self.index, triple, triple)

def __len__(self):
    """number of triples in the triple store""
    return len(self.index[(Q,Q,Q)])

The lookup method returns a list of triples that match a pattern. The pattern is a triple of the form \((i,j,k)\) where each of \(i, j,\) and \(k\) is either "Q" or a given value; specifying whether the subject, verb, and object are provided in the query or not. lookup\((Q,Q,Q)\) returns all triples. lookup\((s,v,o)\) can be used to check whether the triple \((s,v,o)\) is in the triple store; it returns [] if the triple is not in the knowledge graph, and \([(s,v,o)]\) if it is.

```python
def lookup(self, query):
    """pattern is a triple of the form (i,j,k) where
    each i, j, k is either Q or a value for the
    subject, verb and object respectively.
    returns all triples with the specified non-Q vars in corresponding
    position
    """
    if query in self.index:
        return self.index[query]
    else:
        return []

def add_to_index(dict, key, value):
    if key in dict:
        dict[key].append(value)
    else:
        dict[key] = [value]
```

Here is a simple test triple store. In Wikidata Q262802 denotes the football (soccer) player Christine Sinclair, P27 is the country of citizenship, and Q16 is Canada.

```python
# test cases:
sts = TripleStore() # simple triple store
Q = TripleStore.Q # makes it easier to read
sts.add(('/entity/Q262802', 'http://schema.org/name','Christine Sinclair'))
sts.add(('/entity/Q262802', '/prop/direct/P27', '/entity/Q16'))
sts.add(('/entity/Q16', 'http://schema.org/name', 'Canada'))

# sts.lookup('/entity/Q262802,Q,Q))
# sts.lookup(Q,'http://schema.org/name',Q))
```
16.1. Triple Store

```python
# sts.lookup((Q,'http://schema.org/name',"Canada"))
# sts.lookup('/entity/Q16', 'http://schema.org/name', "Canada")
# sts.lookup('/entity/Q262802', 'http://schema.org/name', "Canada")
# sts.lookup((Q,Q,Q))

def test_kg(kg=sts, q=('/entity/Q262802',Q,Q),
    res=[('/entity/Q262802','http://schema.org/name','Christine Sinclair'), ('/entity/Q262802', '/prop/direct/P27', '/entity/Q16')]):
    """Knowledge graph unit test""
    ans = kg.lookup(q)
    assert res==ans, f"test_kg answer {ans}"
    print("knowledge graph unit test passed")

if __name__ == '__main__':
    test_kg()
```


The default in load_file is to only include English names; multiple languages can be included in the list. If the language restriction is None, all tuples are included. Converting to strings, as done here, loses information, e.g., the language associated with the literals. If you don’t want to lose information, you can use rdflib objects.

```python
# before using do:
# pip install rdflib

def load_file(ts, filename, language_restriction=['en']):
    import rdflib
    g = rdflib.Graph()
    g.parse(filename)
    for (s,v,o) in g:
        if language_restriction and isinstance(o,rdflib.term.Literal) and o._language and o._language not in language_restriction:
            pass
        else:
            ts.add((str(s),str(v),str(o)))
    print(f"{len(g)} triples read. Triple store has {len(ts)} triples.")
```

 TripleStore.load_file = load_file

### Test cases ###

ts = TripleStore()
#ts.load_file('http://www.wikidata.org/wiki/Special:EntityData/Q262802.nt')
q262802 = 'http://www.wikidata.org/entity/Q262802'
#res=ts.lookup(q262802, 'http://www.wikidata.org/prop/P27',Q) # country of citizenship
# The attributes of the object in the first answer to the above query:
#ts.lookup([res[0][2],Q,Q])
```
16.2 Integrating Datalog and Triple Store

The following extends the definite clause reasoner in the previous chapter to include a built-in “triple” predicate (an atom with name “triple” and three arguments). The instances of this predicate are retrieved from the triple store. This is a simplified version of what can be done with the semweb library of SWI Prolog (https://www.swi-prolog.org/pldoc/doc?object=section(%27packages/semweb.html%27). For anything serious, we suggest you use that. Note that the semweb library uses “rdf” as the predicate name, and ? uses “prop” in Section 16.1.3 for the same predicate as “triple”.

```python
from logicRelation import Var, Atom, Clause, KB, unify, apply
from knowledgeGraph import TripleStore, sts
import random

class KBT(KB):
    def __init__(self, triplestore, statements=[]):
        self.triplestore = triplestore
        KB.__init__(self, statements)

    def eval_triple(self, ans, selected, remaining, indent):
        query = selected.args
        Q = self.triplestore.Q
        pattern = tuple(Q if isinstance(e,Var) else e for e in query)
        retrieved = self.triplestore.lookup(pattern)
        self.display(3,indent,"eval_triple:
        query="query,"pattern="pattern,"retrieved="retrieved")
        for tr in random.sample(retrieved,len(retrieved)):
            sub = unify(tr, query)
            self.display(3,indent,"KB.prove:
            selected="selected,"triple="tr,""sub="sub")
            if sub is not False:
                yield from self.prove(apply(ans,sub), apply(remaining,sub),
                indent+" ")

# simple test case:
kbt = KBT(sts) # sts is simple triplestore from knowledgeGraph.py
# kbt.ask_all([Atom('triple',('http://www.wikidata.org/entity/Q262802',
    Var('P'),Var('O')))])
```

The following are some larger examples from Wikidata. You must run load_file to load the triples related to Christine Sinclair (Q262802). Otherwise the queries won’t work.

https://aipython.org
The first query is how Christine Sinclair (Q262802) is related to Portland Thorns (Q1446672) with two hops in the knowledge graph. It is asking for a $P$, $O$ and $P1$ such that

$$(Q262802, P, O) \& (0, P1, Q1446672)$$

The second is asking for the name of a team that Christine Sinclair (Q262802) played for. It is asking for a $O$, $T$ and $N$, where $O$ is the reified object that gives the relationship, $T$ is the team and $N$ is the name of the team. Informally (with variables staring with uppercase and constants in lower case) this is

$$(q262802, p54, O) \& (O, p54, T) \& (T, name, N)$$

Notice how the reified relation ‘P54’ (member of sports team) is represented:

### KnowledgeReasoning.py — (continued)
```python
# What is the name of a team that Christine Sinclair played for:
# kbts.ask_one([triple(q262802, 'http://www.wikidata.org/prop/P54', O),
# triple(O, 'http://www.wikidata.org/prop/statement/P54', T),
# triple(T, 'http://schema.org/name', N)])
```

The third asks for the name of a team that Christine Sinclair (Q262802) played for at two different start times. It is asking for a $N$, $D1$ and $D2$, $N$ is the name of the team and $D1$ and $D2$ are the start dates. In Wikidata, P54 is “member of sports team” and P580 is “start time”.

### KnowledgeReasoning.py — (continued)
```python
# The name of a team that Christine Sinclair played for at two different times, and the dates
def playedtwice(s, n, d0, d1): return Atom('playedtwice', [s, n, d0, d1])
S = Var('S')
N = Var('N')
```
D0 = Var('D0')
D1 = Var('D1')
kbs.add_clause(Clause(playedtwice(S,N,D0,D1), [
    triple(S, 'http://www.wikidata.org/prop/P54', O),
    triple(O, 'http://www.wikidata.org/prop/statement/P54', T),
    triple(S, 'http://www.wikidata.org/prop/P54', O1),
    triple(O1, 'http://www.wikidata.org/prop/statement/P54', T),
    lt(O, O1), # ensure different and only generated once
    triple(T, 'http://schema.org/name', N),
    triple(O, 'http://www.wikidata.org/prop/qualifier/P580', D0),
    triple(O1, 'http://www.wikidata.org/prop/qualifier/P580', D1)
]))
# kbs.ask_all([playedtwice(q262802,N,D0,D1)])
Chapter 17

Relational Learning

17.1 Collaborative Filtering

The code here is based on the gradient descent algorithm for matrix factorization of \( I \).

A rating set consists of training and test data, each a list of \((user, item, rating)\) tuples.

```python
import random
import matplotlib.pyplot as plt
import urllib.request
from learnProblem import Learner
from display import Displayable

class Rating_set(Displayable):
    '''A rating contains:
    training_data: list of (user, item, rating) triples
    test_data: list of (user, item, rating) triples
    '''
    def __init__(self, training_data, test_data):
        self.training_data = training_data
        self.test_data = test_data

    grades_rs = Rating_set(  # 3='A', 2='B', 1='C'
        [('s1','c1',3),  # training data
         ('s2','c1',1),
         ('s1','c2',2),
```

The following is a representation of Examples 17.5-17.7 of \( I \). This is a much smaller dataset than one would expect to work well.
A CF_learner does stochastic gradient descent to make a predictor of ratings for user-item pairs.

```python
class CF_learner(Learner):
    def __init__(self,
        rating_set,  # a Rating_set
        step_size = 0.01,  # gradient descent step size
        regularization = 1.0,  # L2 regularization for full dataset
        num_properties = 10,  # number of hidden properties
        property_range = 0.02  # properties are initialized to be between
                                # -property_range and property_range
    ):  
        self.rating_set = rating_set  
        self.training_data = rating_set.training_data  
        self.test_data = self.rating_set.test_data  
        self.step_size = step_size  
        self.regularization = regularization  
        self.num_properties = num_properties  
        self.num_ratings = len(self.training_data)  
        self.ave_rating = (sum(r for (u,i,r) in self.training_data) 
                                /self.num_ratings)  
        self.users = {u for (u,i,r) in self.training_data}  
        self.items = {i for (u,i,r) in self.training_data}  
        self.user_bias = {u:0 for u in self.users}  
        self.item_bias = {i:0 for i in self.items}  
        self.user_prop = {u:[random.uniform(-property_range,property_range) 
                               for p in range(num_properties)]  
                         for u in self.users}  
        self.item_prop = {i:[random.uniform(-property_range,property_range) 
                               for p in range(num_properties)]  
                         for i in self.items}  
        # the _delta variables are the changes internal to a batch:  
        self.user_bias_delta = {u:0 for u in self.users}  
        self.item_bias_delta = {i:0 for i in self.items}  
        self.user_prop_delta = {u:[0 for p in range(num_properties)]  
                                for u in self.users}  
        self.item_prop_delta = {i:[0 for p in range(num_properties)]  
                                for i in self.items}  
        # zeros is used for users and items not in the training set  
        self.zeros = [0 for p in range(num_properties)]  
        self.epoch = 0  
        self.display(1, "Predict mean:" "(Ave Abs,AveSumSq)",
```

https://aipython.org
17.1. Collaborative Filtering

```
"training =",self.eval2string(self.training_data,
    useMean=True),
"test =",self.eval2string(self.test_data, useMean=True))
```

Prediction returns the current prediction of a user on an item.

```
def prediction(self,user,item):
    """Returns prediction for this user on this item.
    The use of .get() is to handle users or items in test set but not
    in the training set.
    """
    if user in self.user_bias: # user in training set
        if item in self.item_bias: # item in training set
            return (self.ave_rating
                + self.user_bias[user]
                + self.item_bias[item]
                + sum([self.user_prop[user][p]*self.item_prop[item][p]
                    for p in range(self.num_properties)]))
        else: # training set contains user but not item
            return (self.ave_rating + self.user_bias[user])
    elif item in self.item_bias: # training set contains item but not
        user
            return self.ave_rating + self.item_bias[item]
    else:
        return self.ave_rating
```

Learn carries out num_epochs epochs of stochastic gradient descent with
batch_size giving the number of training examples in a batch. The number
of epochs is approximately the average number of times each training data
point is used. It is approximate because it processes the integral number of the
batch size.

```
def learn(self, num_epochs = 50, batch_size=1000):
    """ do (approximately) num_epochs iterations through the dataset
    batch_size is the size of each batch of stochastic gradient
gradient descent.
    """
    batch_size = min(batch_size, len(self.training_data))
    batch_per_epoch = len(self.training_data) // batch_size #
    approximate
    num_iter = batch_per_epoch*num_epochs
    reglz =
        self.step_size*self.regularization*batch_size/len(self.training_data)
    #regularization per batch
    for i in range(num_iter):
        if i % batch_per_epoch == 0:
            self.epoch += 1
            self.display(1, "Epoch", self.epoch, "(Ave Abs,AveSumSq)",
```

https://aipython.org  Version 0.9.12  December 22, 2023
"training =",self.eval2string(self.training_data),
"test =",self.eval2string(self.test_data))

# determine errors for a batch
for (user,item,rating) in random.sample(self.training_data, batch_size):
    error = self.prediction(user,item) - rating
    self.user_bias_delta[user] += error
    self.item_bias_delta[item] += error
    for p in range(self.num_properties):
        self.user_prop_delta[user][p] += error*self.item_prop[item][p]
        self.item_prop_delta[item][p] += error*self.user_prop[user][p]

# Update all parameters
for user in self.users:
    self.user_bias[user] -=
        (self.step_size*self.user_bias_delta[user]
        + reglz*self.user_bias[user])
    self.user_bias_delta[user] = 0
    for p in range(self.num_properties):
        self.user_prop[user][p] -=
            (self.step_size*self.user_prop_delta[user][p]
            + reglz*self.user_prop[user][p])
        self.user_prop_delta[user][p] = 0

for item in self.items:
    self.item_bias[item] -=
        (self.step_size*self.item_bias_delta[item]
        + reglz*self.item_bias[item])
    self.item_bias_delta[item] = 0
    for p in range(self.num_properties):
        self.item_prop[item][p] -=
            (self.step_size*self.item_prop_delta[item][p]
            + reglz*self.item_prop[item][p])
        self.item_prop_delta[item][p] = 0

# The evaluate method evaluates current predictions on the rating set:

def evaluate(self, ratings, useMean=False):
    """returns (average_absolute_error, average_sum_squares_error) for ratings
    """
    abs_error = 0
    sumsq_error = 0
    if not ratings: return (0,0)
    for (user,item,rating) in ratings:
        prediction = self.ave_rating if useMean else self.prediction(user,item)
        error = prediction - rating
        abs_error += abs(error)
        sumsq_error += error * error

https://aipython.org
17.1. Collaborative Filtering

```python
return abs_error/len(ratings), sumsq_error/len(ratings)

def eval2string(self, *args, **nargs):
    """returns a string form of evaluate, with fewer digits
    """
    (abs, ssq) = self.evaluate(*args, **nargs)
    return f"""{(abs:.4f}, {ssq:.4f})"
```

Let's test the code on the grades rating set:

Let's test the code on the grades rating set:

```
relnCollFilt.py — (continued)

# lg = CF_learner(grades_rs, step_size = 0.1, regularization = 0.01,
# num_properties = 1)
# lg.learn(num_epochs = 500)
# lg.item_bias
# lg.user_bias
# lg.plot_property(0, plot_all=True) # can you explain why?
```

**Exercise 17.1** In using CF_learner with grades_rs, does it work better with 0 properties? Is it overfitting to the data? How can overfitting be adjusted?

**Exercise 17.2** Modify the code so that self.ave_rating is also learned. It should start as the average rating. Should it be regularized? Does it change from the initialized value? Does it work better or worse?

**Exercise 17.3** With the Movielens 100K dataset and the batch size being the whole training set, the error go to infinity? How can this be fixed?

**Exercise 17.4** Can the regularization avoid iterating through the parameters for all users and items after a batch? Consider items that are in many batches versus those in a few or even no batches. (Warning: This is challenging to get right.)

17.1.1 Plotting

The plot_predictions method plots the cumulative distributions for each ground truth. Figure 17.1 shows a plot for the Movielens 100K dataset. Consider the rating = 1 line. The value for x is the proportion of the predictions with predicted value ≤ x when the ground truth has a rating of 1. Similarly for the other lines.

```
def plot_predictions(self, examples="test"):
    """
    examples is either "test" or "training" or the actual examples
    """
    if examples == "test":
        examples = self.test_data
    elif examples == "training":
        examples = self.training_data
```

https://aipython.org  Version 0.9.12  December 22, 2023
The `plot_property` method plots a single latent property; see Figure 17.2. Each `(user, item, rating)` is plotted where the x-value is the value of the property for the user, the y-value is the value of the property for the item, and the rating is plotted at this `(x, y)` position. That is, `rating` is plotted at the `(x, y)` position `(p(user), p(item))`.

Because there are too many ratings to show, `plot_property` selects a random number of points. It is difficult to see what is going on; the `create_top_subset` method was created to show the most rated items and the users who rated the most of these. This should help visualize how the latent property helps.
17.1. Collaborative Filtering

Figure 17.2: learner1.plot_property(0) with 200 random ratings plotted. Rating \((u,i,r)\) has \(r\) plotted a position \((p(u),p(i))\) where \(p\) is the selected latent property.

```python
def plot_property(self,
    p,       # property
    plot_all=False, # true if all points should be plotted
    num_points=200 # number of random points plotted if not all
)
    
    """plot some of the user-movie ratings,
    if plot_all is true
    num_points is the number of points selected at random plotted.

    the plot has the users on the x-axis sorted by their value on
    property \(p\) and
    with the items on the y-axis sorted by their value on property \(p\) and
    the ratings plotted at the corresponding x-y position.
    """
    plt.ion()
    plt.xlabel("users")
    plt.ylabel("items")
    user_vals = [self.user_prop[u][p]
        for u in self.users]
    item_vals = [self.item_prop[i][p]
for i in self.items
plt.axis([min(user_vals)-0.02,
         max(user_vals)+0.05,
         min(item_vals)-0.02,
         max(item_vals)+0.05])
if plot_all:
    for (u,i,r) in self.training_data:
        plt.text(self.user_prop[u][p],
                 self.item_prop[i][p],
                 str(r))
else:
    for i in range(num_points):
        (u,i,r) = random.choice(self.training_data)
        plt.text(self.user_prop[u][p],
                 self.item_prop[i][p],
                 str(r))
plt.show()

17.1.2 Loading Rating Sets from Files and Websites

This assumes the form of the Movielens datasets ?, available from http://grouplens.org/datasets/movielens/.

The Movielens datasets consist of (user, movie, rating, timestamp) tuples. The aim here is to predict the future from the past. Tuples before data_split form the training set, and those after the test set.

A rating set can be read from the Internet or read from a local file. The default is to read the Movielens 100K dataset from the Internet. It would be more efficient to save the dataset as a local file, and then set local_file = True, as then it will not need to download the dataset every time the program is run.

class Rating_set_from_file(Rating_set):
    def __init__(self,
                 date_split=892000000,
                 local_file=False,
                 url="http://files.grouplens.org/datasets/movielens/ml-100k/u.data",
                 file_name="u.data"):
        self.display(1,"reading...")
        if local_file:
            lines = open(file_name, 'r')
        else:
            lines = (line.decode('utf-8') for line in urllib.request.urlopen(url))
        all_ratings = (tuple(int(e) for e in line.strip().split('	'))
                       for line in lines)
        self.training_data = []
        self.training_stats = {1:0, 2:0, 3:0, 4:0, 5:0}
        self.test_data = []
        self.test_stats = {1:0, 2:0, 3:0, 4:0, 5:0}

https://aipython.org Version 0.9.12 December 22, 2023
17.1. Collaborative Filtering

```python
for (user, item, rating, timestamp) in all_ratings:
    if timestamp < date_split:  # rate[3] is timestamp
        self.training_data.append((user, item, rating))
        self.training_stats[rating] += 1
    else:
        self.test_data.append((user, item, rating))
        self.test_stats[rating] += 1
self.display(1, "...read:", len(self.training_data), "training ratings and",
            len(self.test_data), "test ratings")
tr_users = {user for (user, item, rating) in self.training_data}
test_users = {user for (user, item, rating) in self.test_data}
self.display(1, "users:", len(tr_users), "training,",
            len(test_users), "test,",
            len(tr_users & test_users), "in common")
tr_items = {item for (user, item, rating) in self.training_data}
test_items = {item for (user, item, rating) in self.test_data}
self.display(1, "items:", len(tr_items), "training,",
            len(test_items), "test,",
            len(tr_items & test_items), "in common")
self.display(1, "Rating statistics for training set:", self.training_stats)
self.display(1, "Rating statistics for test set:", self.test_stats)
```

17.1.3 Ratings of top items and users

Sometimes it is useful to plot a property for all 
(user, item, rating) triples. There are too many such triples in the data set. The method create_top_subset creates a much smaller dataset where this makes sense. It picks the most rated items, then picks the users who have the most ratings on these items. It is designed for depicting the meaning of properties, and may not be useful for other purposes. The resulting plot is shown in Figure 17.3

```python
class Rating_set_top_subset(Rating_set):
    def __init__(self, rating_set, num_items = (20,40), num_users =
        (20,24)):
        """Returns a subset of the ratings by picking the most rated items,
        and then the users that have most ratings on these, and then all of
        the ratings that involve these users and items.
        num_items is (ni,si) which selects ni users at random from the top
        si users
        num_users is (nu,su) which selects nu items at random from the top
        su items
        """
        (ni, si) = num_items
        (nu, su) = num_users
        items = {item for (user,item,rating) in rating_set.training_data}
        item_counts = {i:0 for i in items}
```
for (user,item,rating) in rating_set.training_data:
    item_counts[item] += 1

items_sorted = sorted((item_counts[i],i) for i in items)
top_items = random.sample([item for (count, item) in items_sorted[-si:]], ni)
set_top_items = set(top_items)

users = {user for (user,item,rating) in rating_set.training_data}
user_counts = {u:0 for u in users}
for (user,item,rating) in rating_set.training_data:
    if item in set_top_items:
        user_counts[user] += 1

users_sorted = sorted((user_counts[u],u) for u in users)
top_users = random.sample([user for (count, user) in users_sorted[-su:]], nu)
set_top_users = set(top_users)

self.training_data = [(user,item,rating)
for (user,item,rating) in rating_set.training_data
    if user in set_top_users and item in set_top_items]
17.2 Relational Probabilistic Models

The following implements relational belief networks – belief networks with plates. Plates correspond to logical variables.

```python
self.test_data = []
movielens = Rating_set_from_file()
learner1 = CF_learner(movielens, num_properties = 1)
# learner10 = CF_learner(movielens, num_properties = 10)
# learner1.learn(50)
# learner1.plot_predictions(examples = "training")
# learner1.plot_predictions(examples = "test")
# learner1.plot_property(0)
# movielens_subset = Rating_set_top_subset(movielens,num_items = (20,40),
# num_users = (20,40))
# learner_s = CF_learner(movielens_subset, num_properties=1)
# learner_s.learn(1000)
# learner_s.plot_property(0,plot_all=True)
```

17.2 Relational Probabilistic Models

The following implements relational belief networks – belief networks with plates. Plates correspond to logical variables.

```
from display import Displayable
from probGraphicalModels import BeliefNetwork
from variable import Variable
from probRC import ProbRC
from probFactors import Prob
import random

boolean = [False, True]
```

A ParVar is a parametrized random variable, which consists of the name, a list of logical variables (plates), a domain, and a position. For each assignment of an entity to each logical variable, there is a random variable in a grounding.

```
class ParVar(object):
    """Parametrized random variable"""
    def __init__(self, name, log_vars, domain, position=None):
        self.name = name # string
        self.log_vars = log_vars
        self.domain = domain # list of values
        self.position = position if position else (random.random(),
            random.random())
        self.size = len(domain)
```

The class RBN is of relational belief networks. A relational belief networks consists of a title, a set of parvariables, and a set of parfactors.

```
class RBN(Displayable):
    https://aipython.org Version 0.9.12 December 22, 2023
```
def __init__(self, title, parvars, parfactors):
    self.title = title
    self.parvars = parvars
    self.parfactors = parfactors
    self.log_vars = {V for PV in parvars for V in PV.log_vars}

The grounding of a belief network with a population for each logical variable
is a belief network, for which any of the belief network inference algorithms
work.

def ground(self, populations, offsets=None):
    """Ground the belief network with the populations of the logical
    variables.
    populations is a dictionary that maps each logical variable to the
    list of individuals.
    Returns a belief network representation of the grounding.
    ""
    assert all(lv in populations for lv in self.log_vars), f"[{[lv for
    lv in self.log_vars if lv not in populations]} have no
    population"
    self.cps = [] # conditional probabilities in the grounding
    self.var_dict = {} # ground variables created
    for pp in self.parfactors:
        self.ground_parfactor(pp, list(self.log_vars), populations, {},
        offsets)
    return BeliefNetwork(self.title+'_grounded',
        self.var_dict.values(), self.cps)

def ground_parfactor(self, parfactor, lvs, populations, context,
    offsets):
    """
    parfactor is the parfactor to get instances of
    lvs is a list of the logical variables in parfactor not assigned in
    context
    populations is {logical_variable: population} dictionary
    context is a {logical_variable:value} dictionary for
    logical_variable in parfactor
    offsets a {loc_var:(x_offset,y_offset)} dictionary or None
    ""
    if lvs == []:
        if isinstance(parfactor, Prob):
            self.cps.append(Prob(self.ground_pvr(parfactor.child,context,offsets),
                [self.ground_pvr(p,context,offsets)
                for p in parfactor.parents],
                parfactor.values))
        else:
            print("Parfactor not implemented for",parfactor,"of
type",type(parfactor))
    else:
        for val in populations[lvs[0]]:
17.2. Relational Probabilistic Models

```python
self.ground_parfactor(parfactor, lvs[1:], populations,
{lvs[0]:val}|context, offsets)

def ground_pvr(self, prv, context, offsets):
    """grounds a parametrized random variable with respect to a context
    prv is a parametrized random variable
    context is a logical_variable:value dictionary that assigns all
    logical variables in prv
    offsets a {loc_var:(x_offset,y_offset)} dictionary or None
    """
    if isinstance(prv, ParVar):
        args = tuple(context[lv] for lv in prv.log_vars)
        if (prv, args) in self.var_dict:
            return self.var_dict[(prv, args)]
        else:
            new_gv = GrVar(prv, args, offsets)
            self.var_dict[(prv, args)] = new_gv
            return new_gv
    else: # allows for non-parametrized random variables
        return prv

A GrVar is a variable constructed by grounding a parametrized random variable with respect to a tuple of values for the logical variables.

class GrVar(Variable):
    """Grounded Variable""
    def __init__(self, parvar, args, offsets = None):
        """A grounded variable
        parvar is the parametrized variable
        args is a tuple of a value for each random variable
        offsets is a map between the value and the (x,y) offsets
        """
        if offsets:
            pos = sum_positions([parvar.position]+[offsets[a] for a in args])
        else:
            pos = sum_positions([parvar.position,
                (random.uniform(-0.2,0.2),random.uniform(-0.2,0.2))])
        Variable.__init__(self,parvar.name+"("+\ .join(args)+")",
            parvar.domain, pos)
        self.parvar= parvar
        self.args = tuple(args)
        self.hash_value = None

def __hash__(self):
    if self.hash_value is None: # only hash once
        self.hash_value = hash((self.parvar, self.args))
    return self.hash_value

def __eq__(self, other):
```

https://aipython.org Version 0.9.12 December 22, 2023
return isinstance(other, GrVar) and self.parvar == other.parvar and 
    self.args == other.args

def sum_positions(poslist):
    (x, y) = (0, 0)
    for (xo, yo) in poslist:
        x += xo
        y += yo
    return (x, y)

The following is a representation of Examples 17.5-17.7 of ?.

The plate model – represented here using grades – is shown in Figure 17.4. The observation in obs corresponds to the dataset of Figure 17.3. The grounding in grades_gr corresponds to Figure 17.5, but also includes the Grade variables no needed to answer the query (see exercise below).

Try the commented out queries to the Python shell:

grades = RBN("Grades RBN", {Int, Grade, Diff}, {pg, pi, pd})

students = ["s1", "s2", "s3", "s4"]
st_offsets = {st: (0, -0.2*i) for (i, st) in enumerate(students)}
courses = ["c1", "c2", "c3", "c4"]
co_offsets = {co: (0.2*i, 0) for (i, co) in enumerate(courses)}
grades_gr = grades.ground({"St": students, "Co": courses},
                          offsets=st_offsets | co_offsets)
ob = (GrVar(Grade, ["s1", "c1"])="A", GrVar(Grade, ["s2", "c1"])="C",
     GrVar(Grade, ["s1", "c2"])="B", 
     GrVar(Grade, ["s2", "c3"])="B", GrVar(Grade, ["s3", "c2"])="B",
     GrVar(Grade, ["s4", "c3"])="B")

# grades_rc = ProbRC(grades_gr)
# grades_rc.show_post({GrVar(Grade, ["s1", "c1"])="A"}, fontsize=10)
#
# grades_rc.show_post({GrVar(Grade, ["s1", "c1"])="A", GrVar(Grade, ["s2", "c1"])="C"})
#
# grades_rc.show_post({GrVar(Grade, ["s1", "c1"])="A", GrVar(Grade, ["s2", "c1"])="C", 
#                      GrVar(Grade, ["s1", "c2"])="B")})
# grades_rc.show_post(obs, fontsize=10)
Grades RBN grounded observed: {Grade(s1,c1): 'A', Grade(s2,c1): 'C', Grade(s1,c2): 'B'}

Figure 17.4: Grounded network with three observations

Exercise 17.5 What are advantages and disadvantages of using this formulation over using CF_learner with grades_rs? Think about overfitting, and where the parameters come from.

Exercise 17.6 The grounding above creates a random variable for each element for each possible combination of individuals in the populations. Change it so that it only creates as many random variables as needed to answer a query. For example, for the observations and queries above, only the variables in Figure 17.5 in ? need to be created.
Version History

- 2023-12-06 Version 0.9.12: Top-down proof for Datalog (ch 15) and triple store (ch 16)
- 2023-11-21 Version 0.9.11 updated and simplified relational learning, show relational belief networks
- 2023-11-07 Version 0.9.10 Improved GUIs and test cases for decision-theoretic planning (MDPs) and reinforcement learning.
- 2023-10-6 Version 0.9.8 GUIs for search, Bayesian learning, causality and many smaller changes.
- 2023-07-31 Version 0.9.7 includes relational probabilistic models and smaller changes
- 2023-06-06 Version 0.9.6 controllers are more consistent. Many smaller changes.
- 2022-08-13 Version 0.9.5 major revisions including extra code for causality and deep learning
- 2021-07-08 Version 0.9.1 updated the CSP code to have the same representation of variables as used by the probability code
- 2021-05-13 Version 0.9.0 Major revisions to chapters 8 and 9. Introduced recursive conditioning, simplified much code. New section on multiagent reinforcement learning.
- 2020-11-04 Version 0.8.6 simplified value iteration for MDPs.
- 2020-10-20 Version 0.8.4 planning simplified and fixed arc costs.
• 2020-07-21 Version 0.8.2 added positions and string to constraints

• 2019-09-17 Version 0.8.0 rerepresented blocks world (Section 6.1.2) due to bug found by Donato Meoli.


Index

α-β pruning, 341
A* search, 53
A* Search, 60
action, 125
agent, 25, 305
argmax, 19
assignment, 71, 201
assumable, 119
asynchronous value iteration, 300
augmented feature, 160
Bayesian network, 208
belief network, 208
blocks world, 128
Boolean feature, 150
botton-up proof, 112
branch-and-bound search, 64
class
Action_instance, 142
Agent, 26
Arc, 42
Askable, 109
Assumable, 119
BNfromDBN, 250
BeliefNetwork, 208
Boosted_dataset, 181
Boosting_Learner, 182
Branch_and_bound, 106
CF_Learner, 363
CPD, 203
CPDrename, 249
CSP, 74
CSP_from_CHARS, 138
Clause, 109, 348
Con_solver, 86
ConstantCPD, 203
Constraint, 70
DBN, 249
DBNVEfilter, 251
DBNvariable, 247
DF_Branch_and_bound, 65
DT_Learner, 167
Data_from_file, 157
Data_from_files, 159
Data_set, 151
Data_set_augmented, 160
DecisionFunction, 281
DecisionNetwork, 274
DecisionVariable, 273
Displayable, 18
Dist, 207
| Dropout_layer | 194 |
| EM_learner | 261 |
| Env_from_MDP | 309 |
| Environment | 26 |
| Evaluate | 156 |
| Factor | 201 |
| FactorMax | 286 |
| FactorObserved | 224 |
| FactorRename | 248 |
| FactorSum | 225 |
| Forward_STRIPS | 131 |
| FrontierPQ | 59 |
| GTB_learner | 184 |
| GibbsSampling | 234 |
| GrVar | 375 |
| GraphicalModel | 208 |
| GridDomain | 298 |
| HMM | 238 |
| HMMFilter | 240 |
| HMM_Controlled | 241 |
| HMM_Local | 242 |
| HMMParticleFilter | 244 |
| IEq | 206 |
| InferenceMethod | 216 | 267 |
| KB | 110 | 350 |
| KBA | 119 |
| KBT | 360 |
| K_fold_dataset | 172 |
| K_means_learner | 257 |
| Layer | 187 |
| Learner | 162 |
| LikelihoodWeighting | 230 |
| Linear_complete_layer | 188 |
| Linear_complete_layer_RMS_Prop | 193 |
| Linear_complete_layer_momentum | 192 |
| Linear_Learner | 175 |
| LogisticRegression | 204 |
| MDP | 287 |
| MDPtiny | 291 |
| Magic_sum | 339 |
| Model_based_reinforcement_learner | 323 |
| Monster_game_env | 294 | 310 |

| NN | 190 |
| Node | 337 |
| NoisyOR | 204 |
| POP_node | 142 |
| POP_search_from_STRIPS | 143 |
| ParVar | 373 |
| ParticleFiltering | 231 |
| Party_nv | 308 |
| Path | 45 |
| Planning_problem | 126 |
| Plot_env | 36 |
| Plot_prices | 29 |
| Predict | 164 |
| Prob | 205 |
| ProbDT | 206 |
| ProbRC | 220 |
| ProbSearch | 218 |
| Q_learner | 313 |
| RBN | 373 |
| RC_DN | 282 |
| RL_agent | 306 |
| RL_env | 305 |
| Rating_set | 370 |
| ReLU_layer | 189 |
| Regression_STRIPS | 135 |
| RejectionSampling | 229 |
| Rob_body | 31 |
| Rob_env | 31 |
| Rob_middle_layer | 34 |
| Rob_top_layer | 35 |
| Runtime_distribution | 102 |
| SARSAR | 315 |
| SARSAR_LFA_learner | 329 |
| SLSearcher | 95 |
| STRIPS_domain | 126 |
| SamplingInferenceMethod | 229 |
| Search_from_CSP | 83 | 85 |
| Search_problem | 41 |
| Search_problem_from_explicit_graph | 43 |
| Search_with_AC_from_CSP | 93 |
| Searcher | 53 |
| SearcherGUI | 56 |
| SearcherMPP | 62 |
| Show_Localization | 242 |
Index

Sigmoid Layer, 190
SoftConstraint, 104
State, 130
Strips, 125
Subgoal, 135
TP_agent, 28
TP_env, 27
TabFactor, 205
TripleStore, 337
Updatable_priority_queue, 101
Utility, 273
UtilityTable, 273
VE, 224
VE_DN, 285
Variable, 69
cross validation, 171
CSP, 69
consistency, 86
domain splitting, 89 93
search, 84
stochastic local search, 95
currying, 74
datalog, 347
dataset, 150
DBN
filtering, 251
unrolling, 250
DBN (dynamic belief network), 247
deep learning, 187
display, 19
Displayable, 18
domain splitting, 89 93
Dropout, 194
dynamic belief network (DBN), 247
representation, 247
EM, 261
environment, 25 26 305
error, 155
example, 150
explanation, 115
explicit graph, 43
factor, 201 205
factor_times, 225
feature, 150 152
feature engineering, 149
file
agentBuying.py, 27
agentEnv.py, 31
agentFollowTarget.py, 38
agentMiddle.py, 34
agentTop.py, 35
agents.py, 26
cspConsistency.py, 86
cspConsistencyGUI.py, 91
cspDFS.py, 83
cspExamples.py, 74
cspProblem.py, 70
cspSLS.py, 95
cspSearch.py, 85
cspSoft.py, 104
decnNetworks.py, 273
display.py, 18
knowledgeGraph.py, 357
knowledgeReasoning.py, 360
learnBayesian.py, 253
learnBoosting.py, 181
learnCrossValidation.py, 172
learnDT.py, 167
learnEM.py, 261
learnKMeans.py, 257
learnLinear.py, 175
https://aipython.org
learnNN.py, 187
learnNoInputs.py, 164
learnProblem.py, 150
logicAssumables.py, 119
logicBottomUp.py, 112
logicExplain.py, 115
logicNegation.py, 122
logicProblem.py, 109
logicRelation.py, 347
logicTopDown.py, 114
masLearn.py, 342
masMiniMax.py, 340
masProblem.py, 337
mdpExamples.py, 288
mdpGUI.py, 298
mdpProblem.py, 287
probCounterfactual.py, 269
probDBN.py, 247
probDo.py, 267
probExamples.py, 210
probFactors.py, 201
probGraphicalModels.py, 208
probHMM.py, 238
probLocalization.py, 241
probRC.py, 218
probStochSim.py, 227
probVE.py, 224
pythonDemo.py, 13
relnCollFilt.py, 363
relnExamples.py, 350
relnProbModels.py, 375
rlExamples.py, 308
rlFeatures.py, 329
rlGUI.py, 332
rlGameFeature.py, 326
rlModelLearner.py, 323
rlProblem.py, 305
rlQExperienceReplay.py, 318
rlQLearner.py, 313
rlStochasticPolicy.py, 320
searchBranchAndBound.py, 65
searchExample.py, 47
searchGUI.py, 56
searchGeneric.py, 53
searchGrid.py, 63
searchMPP.py, 62
searchProblem.py, 41
searchTest.py, 66
stripsCSPPlanner.py, 138
stripsForwardPlanner.py, 130
stripsHeuristic.py, 133
stripsPOP.py, 142
stripsProblem.py, 125
stripsRegressionPlanner.py, 135
utilities.py, 19
variable.py, 69
filtering, 240 244
DBN, 251
flip, 20
forward planning, 130
frange, 152
ftype, 152
game, 337
Gibbs sampling, 234
graphical model, 208
heuristic planning, 132 137
hidden Markov model, 238
hierarchical controller, 31
HMM
  exact filtering, 240
  particle filtering, 244
HMM (hidden Markov models), 238
importance sampling, 231
interact
  proofs, 116
ipython, 10
k-means, 257
kernel, 160
knowledge base, 110
knowledge graph, 357
learner, 162
learning, 149 199 253 265 305 336
  cross validation, 363 373
decision tree, 167
deep, 187 199
Index

deep learning, 187
EM, 261
k-means, 257
linear regression, 175
linear classification, 175
neural network, 187
no inputs, 163
reinforcement, 305–336
relational, 363
supervised, 149–186
with uncertainty, 253–265

LightGBM, 184
likelihood weighting, 230
linear regression, 175
linear classification, 175
localization, 241
logic program, 347
logistic regression, 203
logit, 176, 177
loss, 155

magic square, 338
magic-sum game, 338
Markov Chain Monte Carlo, 234
Markov decision process, 287
max_display_level, 19
MCMC, 234
MDP, 287
GUI, 298
method
consistent, 72
holds, 71
maxh, 133
zero, 131
minimax, 337
minimax algorithm, 340
minsets, 120
model-based reinforcement learner, 322
multiagent system, 337
multiple path pruning, 62

n-queens problem, 82
naive search probabilistic inference, 218

naughts and crosses, 338
neural network, 187
noisy-or, 204
NotImplementedError, 26

partial-order planner, 142
particle filtering, 231
HMMs, 244
planning, 125–147, 273–304
CSP, 138
decision network, 273
forward, 130
MDP, 287
partial order, 142
regression, 135
with certainty, 125–147
with learning, 322
with uncertainty, 273–304

plotting
agents in time, 29
reinforcement learning, 307
robot environment, 36
run-time distribution, 102
stochastic simulation, 236

predictor, 155
Prob, 205
probabilistic inference methods, 216
probability, 201
proof
bottom-up, 112
explanation, 115
top-down, 114
propagation, 109
Python, 9

Q learning, 313
query, 216
queryD0, 267

RC, 220
recursive conditioning, 220
recursive conditioning (RC), 220
recursive conditioning for decision
networks, 282
regression planning, 135
reinforcement learning, 305–336

https://aipython.org
environment, 305
feature-based, 325
model-based, 322
Q-learning, 313
rejection sampling, 229
relational learning, 363
relations, 347
ReLU, 189
resampling, 232
robot
  body, 31
  environment, 31
  middle layer, 34
  plotting, 36
  top layer, 35
robot delivery domain, 126
run time, 16
runtime distribution, 102
sampling, 227
  importance sampling, 231
  belief networks, 229
  likelihood weighting, 230
  particle filtering, 231
  rejection, 229
SARSA, 315
scope, 70
search, 41
  A*, 53
    branch-and-bound, 64
    multiple path pruning, 62
    search_with_any_conflict, 97
    search_with_var_pq, 98
  show, 72, 209
  sigmoid, 176
  softmax, 177
stochastic local search, 95
  any-conflict, 97
  two-stage choice, 98
  stochastic simulation, 227
  tabular factor, 205
test
  SLS, 104
tic-tac-toe, 338
top-down proof, 114
  triple store, 357, 360
uncertainty, 201
unification, 349, 350
unit test, 21, 61, 82, 113, 114, 116
unrolling
  DBN, 250
  updatable priority queue, 100
utility, 273
  utility table, 273
value iteration, 297
variable, 69
variable elimination (VE), 224
variable elimination for decision networks, 285
VE, 224
XGBoost, 184
yield, 14

https://aipython.org