Learning Objectives

At the end of the class you should be able to:

- explain how cycle checking and multiple-path pruning can improve efficiency of search algorithms
- explain the complexity of cycle checking and multiple-path pruning for different search algorithms
- justify why the monotone restriction is useful for A* search
- predict whether forward, backward, bidirectional or island-driven search is better for a particular problem
- demonstrate how dynamic programming works for a particular problem
# Summary of Search Strategies

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Frontier Selection</th>
<th>Complete</th>
<th>Halts</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth-first</td>
<td>Last node added</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Breadth-first</td>
<td>First node added</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best-first</td>
<td>Global min $h(p)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lowest-cost-first</td>
<td>Minimal $cost(p)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A^*$</td>
<td>Minimal $f(p)$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Complete** — if there is a path to a goal, it can find one, even on infinite graphs.

**Halts** — on finite graph (perhaps with cycles).

**Space** — as a function of the length of current path.
Summary of Search Strategies

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Frontier Selection</th>
<th>Complete</th>
<th>Halts</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth-first</td>
<td>Last node added</td>
<td>No</td>
<td>No</td>
<td>Linear</td>
</tr>
<tr>
<td>Breadth-first</td>
<td>First node added</td>
<td>Yes</td>
<td>No</td>
<td>Exp</td>
</tr>
<tr>
<td>Best-first</td>
<td>Global min $h(p)$</td>
<td>No</td>
<td>No</td>
<td>Exp</td>
</tr>
<tr>
<td>Lowest-cost-first</td>
<td>Minimal $cost(p)$</td>
<td>Yes</td>
<td>No</td>
<td>Exp</td>
</tr>
<tr>
<td>$A^*$</td>
<td>Minimal $f(p)$</td>
<td>Yes</td>
<td>No</td>
<td>Exp</td>
</tr>
</tbody>
</table>

**Complete** — if there a path to a goal, it can find one, even on infinite graphs.

**Halts** — on finite graph (perhaps with cycles).

**Space** — as a function of the length of current path.
A searcher can prune a path that ends in a node already on the path, without removing an optimal solution.
**Input:** a graph, a set of start nodes, Boolean procedure \( \text{goal}(n) \) that tests if \( n \) is a goal node.

\[
\text{frontier} := \{ \langle s \rangle : s \text{ is a start node} \}
\]

while \( \text{frontier} \) is not empty:

  select and remove path \( \langle n_0, \ldots, n_k \rangle \) from \( \text{frontier} \)

  if \( n_k \not\in \{ n_0, \ldots, n_{k-1} \} \):
    if \( \text{goal}(n_k) \):
      return \( \langle n_0, \ldots, n_k \rangle \)

    \( \text{Frontier} := \text{Frontier} \cup \{ \langle n_0, \ldots, n_k, n \rangle : \langle n_k, n \rangle \in A \} \)
In depth-first search, checking for cycles can be done in \textit{constant} time in path length.
In depth-first search, checking for cycles can be done in constant time in path length.
In depth-first search, checking for cycles can be done in **constant** time in path length.

For other methods, checking for cycles can be done in ____ time in path length.
In depth-first search, checking for cycles can be done in \textit{constant} time in path length.

For other methods, checking for cycles can be done in \textit{linear} time in path length.
In depth-first search, checking for cycles can be done in constant time in path length.

For other methods, checking for cycles can be done in linear time in path length.

With cycle pruning, which algorithms halt on finite graphs?
Multiple-path pruning: prune a path to node $n$ that the searcher has already found a path to.
Multiple-path pruning: prune a path to node $n$ that the searcher has already found a path to.

What needs to be stored?
Multiple path pruning: prune a path to node $n$ that the searcher has already found a path to.

What needs to be stored?

Lowest-cost-first search with multiple-path pruning is Dijkstra’s algorithm, and is the same as $A^*$ with multiple-path pruning and a heuristic function of 0.
Graph searching with multiple-path pruning

**Input:** a graph,
a set of start nodes,
Boolean procedure $\text{goal}(n)$ that tests if $n$ is a goal node.

$\text{frontier} := \{ \langle s \rangle : s \text{ is a start node} \}$

$\text{expanded} := \{ \}$

**while** $\text{frontier}$ is not empty:

**select** and **remove** path $\langle n_0, \ldots, n_k \rangle$ from $\text{frontier}$

**if** $n_k \not\in \text{expanded}$:

add $n_k$ to $\text{expanded}$

**if** $\text{goal}(n_k)$:

**return** $\langle n_0, \ldots, n_k \rangle$

$\text{Frontier} := \text{Frontier} \cup \{ \langle n_0, \ldots, n_k, n \rangle : \langle n_k, n \rangle \in A \}$
Multiple-Path Pruning

- How does multiple-path pruning compare to cycle pruning?
Multiple-Path Pruning

- How does multiple-path pruning compare to cycle pruning?
- Which search algorithms with multiple-path pruning always halt on finite graphs?
Multiple-Path Pruning

- How does multiple-path pruning compare to cycle pruning?
- Which search algorithms with multiple-path pruning always halt on finite graphs?
- What is the time overhead of multiple-path pruning?
Multiple-Path Pruning

- How does multiple-path pruning compare to cycle pruning?
- Which search algorithms with multiple-path pruning always halt on finite graphs?
- What is the time overhead of multiple-path pruning?
- What is the space overhead of multiple-path pruning?
Multiple-Path Pruning

- How does multiple-path pruning compare to cycle pruning?
- Which search algorithms with multiple-path pruning always halt on finite graphs?
- What is the time overhead of multiple-path pruning?
- What is the space overhead of multiple-path pruning?
- Is it better for depth-first or breadth-first searches?
Multiple-Path Pruning

- How does multiple-path pruning compare to cycle pruning?
- Which search algorithms with multiple-path pruning always halt on finite graphs?
- What is the time overhead of multiple-path pruning?
- What is the space overhead of multiple-path pruning?
- Is it better for depth-first or breadth-first searches?
- Can multiple-path pruning prevent an optimal solution being found?
Problem: what if a subsequent path to $n$ has a lower cost than the first path to $n$?
Problem: what if a subsequent path to $n$ has a lower cost than the first path to $n$?

- remove all paths from the frontier that use the longer path.
- change the initial segment of the paths on the frontier to use the lower-cost path.
- ensure this doesn’t happen. Make sure that the lower-cost path to a node is expanded first.
Multiple-Path Pruning & $A^*$

- Suppose path $p$ to $n$ was selected, but there is a lower-cost path to $n$. Suppose this lower-cost path is via path $p'$ on the frontier.
- Suppose path $p'$ ends at node $n'$.
Multiple-Path Pruning & $A^*$

- Suppose path $p$ to $n$ was selected, but there is a lower-cost path to $n$. Suppose this lower-cost path is via path $p'$ on the frontier.
- Suppose path $p'$ ends at node $n'$.
- $p$ was selected before $p'$, so:

\[
\text{cost}(p) + h(n) \leq \text{cost}(p') + h(n')
\]

Suppose $\text{cost}(n', n)$ is the actual cost of a path from $n'$ to $n$.

The path to $n$ via $p'$ has a lower cost than $p$ so:

\[
\text{cost}(p') + \text{cost}(n', n) < \text{cost}(p).
\]

$\text{cost}(n', n) < \text{cost}(p) - \text{cost}(p') \leq h(n') - h(n)$.

We can ensure this doesn't occur if $|h(n') - h(n)| \leq \text{cost}(n', n)$. 
Multiple-Path Pruning & $A^*$

- Suppose path $p$ to $n$ was selected, but there is a lower-cost path to $n$. Suppose this lower-cost path is via path $p'$ on the frontier.
- Suppose path $p'$ ends at node $n'$.
- $p$ was selected before $p'$, so:
  $$\text{cost}(p) + h(n) \leq \text{cost}(p') + h(n').$$
- Suppose $\text{cost}(n', n)$ is the actual cost of a path from $n'$ to $n$. The path to $n$ via $p'$ has a lower cost than $p$ so:
Multiple-Path Pruning & $A^*$

- Suppose path $p$ to $n$ was selected, but there is a lower-cost path to $n$. Suppose this lower-cost path is via path $p'$ on the frontier.
- Suppose path $p'$ ends at node $n'$.
- $p$ was selected before $p'$, so:
  \[ \text{cost}(p) + h(n) \leq \text{cost}(p') + h(n'). \]
- Suppose $\text{cost}(n', n)$ is the actual cost of a path from $n'$ to $n$. The path to $n$ via $p'$ has a lower cost than $p$ so:
  \[ \text{cost}(p') + \text{cost}(n', n) < \text{cost}(p). \]

\[ \text{cost}(n', n) < \]
Suppose path $p$ to $n$ was selected, but there is a lower-cost path to $n$. Suppose this lower-cost path is via path $p'$ on the frontier.

Suppose path $p'$ ends at node $n'$.

$p$ was selected before $p'$, so:
$$\text{cost}(p) + h(n) \leq \text{cost}(p') + h(n').$$

Suppose $\text{cost}(n', n)$ is the actual cost of a path from $n'$ to $n$. The path to $n$ via $p'$ has a lower cost than $p$ so:
$$\text{cost}(p') + \text{cost}(n', n) < \text{cost}(p).$$

$$\text{cost}(n', n) < \text{cost}(p) - \text{cost}(p') \leq ...$$
Suppose path $p$ to $n$ was selected, but there is a lower-cost path to $n$. Suppose this lower-cost path is via path $p'$ on the frontier.

Suppose path $p'$ ends at node $n'$.

$p$ was selected before $p'$, so:
$$\text{cost}(p) + h(n) \leq \text{cost}(p') + h(n').$$

Suppose $\text{cost}(n', n)$ is the actual cost of a path from $n'$ to $n$. The path to $n$ via $p'$ has a lower cost than $p$ so:
$$\text{cost}(p') + \text{cost}(n', n) < \text{cost}(p).$$

$$\text{cost}(n', n) < \text{cost}(p) - \text{cost}(p') \leq h(n') - h(n).$$

We can ensure this doesn't occur if
$$|h(n') - h(n)| \leq \text{cost}(n', n).$$
Heuristic function $h$ satisfies the monotone restriction if 
\[ |h(m) - h(n)| \leq \text{cost}(m, n) \] for every arc $\langle m, n \rangle$. 

This is a strengthening of the admissibility criterion.
Heuristics function $h$ satisfies the **monotone restriction** if $|h(m) - h(n)| \leq \text{cost}(m, n)$ for every arc $\langle m, n \rangle$.

If $h$ satisfies the monotone restriction, $A^*$ with multiple path pruning always finds a least-cost path to a goal.
Heuristic function $h$ satisfies the monotone restriction if $|h(m) - h(n)| \leq \text{cost}(m, n)$ for every arc $\langle m, n \rangle$.

If $h$ satisfies the monotone restriction, $A^*$ with multiple path pruning always finds a least-cost path to a goal.

This is a strengthening of the admissibility criterion.
The definition of searching is symmetric: find path from start nodes to goal node or from goal node to start nodes (with reversed arcs).
The definition of searching is symmetric: find path from start nodes to goal node or from goal node to start nodes (with reversed arcs).

- **Forward branching factor:** number of arcs out of a node.
- **Backward branching factor:** number of arcs into a node.
Direction of Search

- The definition of searching is symmetric: find path from start nodes to goal node or from goal node to start nodes (with reversed arcs).
- **Forward branching factor:** number of arcs out of a node.
- **Backward branching factor:** number of arcs into a node.
- Search complexity is $b^n$. Should use forward search if forward branching factor is less than backward branching factor, and vice versa.
Direction of Search

- The definition of searching is symmetric: find path from start nodes to goal node or from goal node to start nodes (with reversed arcs).

- **Forward branching factor:** number of arcs out of a node.

- **Backward branching factor:** number of arcs into a node.

- Search complexity is $b^n$. Should use forward search if forward branching factor is less than backward branching factor, and vice versa.

- Note: when graph is dynamically constructed, the backwards graph may not be available. One might be more difficult to compute than the other.
Bidirectional Search

- Idea: search backward from the goal and forward from the start simultaneously.
Bidirectional Search

- Idea: search backward from the goal and forward from the start simultaneously.
- This wins as $2b^{k/2} \ll b^k$.
  This can result in an exponential saving in time and space.
Bidirectional Search

- Idea: search backward from the goal and forward from the start simultaneously.
- This wins as $2b^{k/2} \ll b^k$.
  This can result in an exponential saving in time and space.
- The main problem is making sure the frontiers meet.
Bidirectional Search

- Idea: search backward from the goal and forward from the start simultaneously.
- This wins as $2b^{k/2} \ll b^k$.
  This can result in an exponential saving in time and space.
- The main problem is making sure the frontiers meet.
- This is often used with
  - a breadth-first method (e.g., least-cost-first search) that builds a set of states that can lead to the goal quickly.
  - in the other direction, another method (typically depth-first) can be used to find a path to these interesting states.
Bidirectional Search

- Idea: search backward from the goal and forward from the start simultaneously.
- This wins as $2b^{k/2} \ll b^k$.
  This can result in an exponential saving in time and space.
- The main problem is making sure the frontiers meet.
- This is often used with
  - a breadth-first method (e.g., least-cost-first search) that builds a set of states that can lead to the goal quickly.
  - in the other direction, another method (typically depth-first) can be used to find a path to these interesting states.
  - How much is stored in the breadth-first method, can be tuned depending on the space available.
Island Driven Search

- **Idea**: find a set of islands between $s$ and $g$.

  $s \rightarrow i_1 \rightarrow i_2 \rightarrow \ldots \rightarrow i_{m-1} \rightarrow g$

  There are $m$ smaller problems rather than 1 big problem.

- This can win as $mb^{k/m} \ll b^k$. 
Idea: find a set of islands between $s$ and $g$.

$$s \rightarrow i_1 \rightarrow i_2 \rightarrow \ldots \rightarrow i_{m-1} \rightarrow g$$

There are $m$ smaller problems rather than 1 big problem.

This can win as $mb^k/m \ll b^k$.

The problem is to identify the islands that the path must pass through. It is difficult to guarantee optimality.

Requires more knowledge than just the graph and a heuristic function.
Island Driven Search

- **Idea:** find a set of islands between \( s \) and \( g \).

\[
s \rightarrow i_1 \rightarrow i_2 \rightarrow \ldots \rightarrow i_{m-1} \rightarrow g
\]

There are \( m \) smaller problems rather than 1 big problem.

- This can win as \( mb^{k/m} \ll b^k \).

- The problem is to identify the islands that the path must pass through. It is difficult to guarantee optimality.

- Requires more knowledge than just the graph and a heuristic function.

- The subproblems can be solved using islands \( \Rightarrow \) hierarchy of abstractions.
Idea: for statically stored graphs, build a table of $dist(n)$ the actual distance of the shortest path from node $n$ to a goal. This can be built backwards from the goal:

$$dist(n) = \begin{cases} 
0 & \text{if } \text{is\_goal}(n), \\
\min_{\langle n, m \rangle \in A} (|\langle n, m \rangle| + dist(m)) & \text{otherwise}.
\end{cases}$$

using least-cost-first search in the reverse graph.
Idea: for statically stored graphs, build a table of $\text{dist}(n)$ the actual distance of the shortest path from node $n$ to a goal. This can be built backwards from the goal:

$$\text{dist}(n) = \begin{cases} 
0 & \text{if } \text{is\_goal}(n), \\
\min_{\langle n, m \rangle \in A}(|\langle n, m \rangle| + \text{dist}(m)) & \text{otherwise.}
\end{cases}$$

using least-cost-first search in the reverse graph.

This can be used locally to determine what to do from any state.
**Dynamic Programming**

**Idea:** for statically stored graphs, build a table of $\text{dist}(n)$ the actual distance of the shortest path from node $n$ to a goal. This can be built backwards from the goal:

$$\text{dist}(n) = \begin{cases} 
0 & \text{if } \text{is\_goal}(n), \\
\min_{\langle n, m \rangle \in A} (|\langle n, m \rangle| + \text{dist}(m)) & \text{otherwise.}
\end{cases}$$

using least-cost-first search in the reverse graph.

- This can be used locally to determine what to do from any state.
- Why not use $A^*$?
Dynamic Programming

Idea: for statically stored graphs, build a table of $dist(n)$ the actual distance of the shortest path from node $n$ to a goal. This can be built backwards from the goal:

$$dist(n) = \begin{cases} 
0 & \text{if } is\_goal(n), \\
\min_{\langle n, m \rangle \in A} (|\langle n, m \rangle| + dist(m)) & \text{otherwise.}
\end{cases}$$

using least-cost-first search in the reverse graph.

- This can be used locally to determine what to do from any state.
- Why not use $A^*$?
- There are two main problems:

©D.L. Poole and A.K. Mackworth 2010-2020

Artificial Intelligence, Lecture 3.4  15 / 15
Dynamic Programming

Idea: for statically stored graphs, build a table of $\text{dist}(n)$ the actual distance of the shortest path from node $n$ to a goal. This can be built backwards from the goal:

$$
\text{dist}(n) = \begin{cases} 
0 & \text{if is\_goal}(n), \\
\min_{\langle n, m \rangle \in A} (|\langle n, m \rangle| + \text{dist}(m)) & \text{otherwise}.
\end{cases}
$$

using least-cost-first search in the reverse graph.

- This can be used locally to determine what to do from any state.
- Why not use $A^*$?
- There are two main problems:
  - It requires enough space to store the graph.
  - The $\text{dist}$ function needs to be recomputed for each goal.