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

Strategy	Frontier Selection	Complete	Halts	Space
Depth-first	Last node added			
Breadth-first	First node added			
Best-first	Global min $h(p)$			
Lowest-cost-first	Minimal $cost(p)$			
A^*	Minimal $f(p)$			

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

Summary of Search Strategies

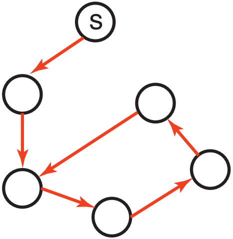
Strategy	Frontier Selection	Complete	Halts	Space
Depth-first	Last node added	No	No	Linear
Breadth-first	First node added	Yes	No	Exp
Best-first	Global min $h(p)$	No	No	Exp
Lowest-cost-first	Minimal $cost(p)$	Yes	No	Exp
A^*	Minimal $f(p)$	Yes	No	Exp

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

Cycle Pruning

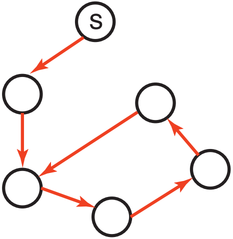


- A searcher can prune a path that ends in a node already on the path, without removing an optimal solution.

Graph searching with cycle pruning

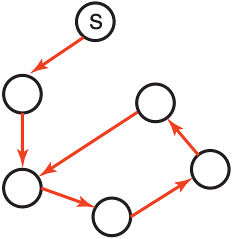
Input: a graph,
a set of start nodes,
Boolean procedure $goal(n)$ that tests if n is a goal node.
$frontier := \{\langle s \rangle : s \text{ is a start node}\}$
while *frontier* is not empty:
  select and **remove** path $\langle n_0, \ldots, n_k \rangle$ from *frontier*
  if $n_k \notin \{n_0, \ldots, n_{k-1}\}$:
    if $goal(n_k)$:
      return $\langle n_0, \ldots, n_k \rangle$
    $Frontier := Frontier \cup \{\langle n_0, \ldots, n_k, n \rangle : \langle n_k, n \rangle \in A\}$

Cycle Pruning



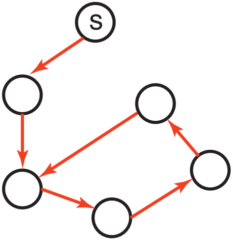
- In depth-first search, checking for cycles can be done in ________ time in path length.

Cycle Pruning



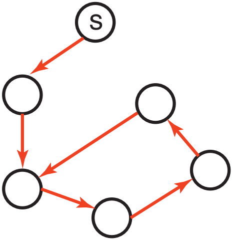
- In depth-first search, checking for cycles can be done in <u>constant</u> time in path length.

Cycle Pruning



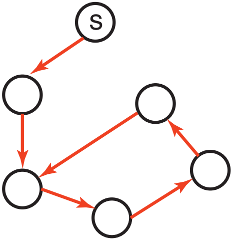
- In depth-first search, checking for cycles can be done in <u>constant</u> time in path length.
- For other methods, checking for cycles can be done in _____ time in path length.

Cycle Pruning



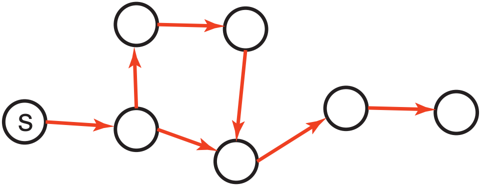
- In depth-first search, checking for cycles can be done in <u>constant</u> time in path length.
- For other methods, checking for cycles can be done in <u>linear</u> time in path length.

Cycle Pruning



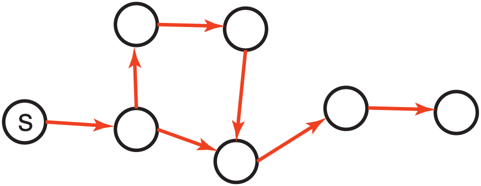
- In depth-first search, checking for cycles can be done in constant time in path length.
- For other methods, checking for cycles can be done in linear time in path length.
- With cycle pruning, which algorithms halt on finite graphs?

Multiple-Path Pruning



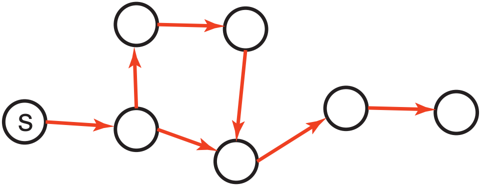
- Multiple path pruning: prune a path to node *n* that the searcher has already found a path to.

Multiple-Path Pruning



- 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



- 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 $goal(n)$ that tests if n is a goal node.
$frontier := \{\langle s \rangle : s \text{ is a start node}\}$
$expanded := \{\}$
while *frontier* is not empty:
select and **remove** path $\langle n_0, \ldots, n_k \rangle$ from *frontier*
if $n_k \notin expanded$:
add n_k to *expanded*
if $goal(n_k)$:
return $\langle n_0, \ldots, n_k \rangle$
$Frontier := Frontier \cup \{\langle n_0, \ldots, n_k, n \rangle : \langle n_k, n \rangle \in A\}$

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?

Multiple-Path Pruning & Optimal Solutions

Problem: what if a subsequent path to n has a lower cost than the first path to n?

Multiple-Path Pruning & Optimal Solutions

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:

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:
  $cost(p) + h(n) \leq cost(p') + h(n')$.
- Suppose $cost(n', n)$ is the actual cost of a path from n' to n. The path to n via p' has a lower cost that 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: $cost(p) + h(n) \leq cost(p') + h(n')$.
- Suppose $cost(n', n)$ is the actual cost of a path from n' to n. The path to n via p' has a lower cost that p so: $cost(p') + cost(n', n) < cost(p)$.

$$cost(n', n) <$$

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:
  $cost(p) + h(n) \leq cost(p') + h(n')$.
- Suppose $cost(n', n)$ is the actual cost of a path from n' to n. The path to n via p' has a lower cost that p so:
  $cost(p') + cost(n', n) < cost(p)$.

$$cost(n', n) < cost(p) - cost(p') \leq$$

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:
  $cost(p) + h(n) \leq cost(p') + h(n')$.
- Suppose $cost(n', n)$ is the actual cost of a path from n' to n. The path to n via p' has a lower cost that p so:
  $cost(p') + cost(n', n) < cost(p)$.

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

  We can ensure this doesn't occur if
  $h(n') - h(n) \leq cost(n', n)$.

Monotone Restriction

- Heuristic function h satisfies the monotone restriction if $h(m) - h(n) \leq cost(m, n)$ for every arc $\langle m, n \rangle$.

Monotone Restriction

- Heuristic function h satisfies the monotone restriction if $h(m) - h(n) \leq cost(m, n)$ 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.

Monotone Restriction

- Heuristic function h satisfies the monotone restriction if $h(m) - h(n) \leq cost(m, n)$ 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.

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).

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.

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.

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

  There are m smaller problems rather than 1 big problem.
- This can win as $mb^{k/m} \ll b^k$.

Island Driven Search

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

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

  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.

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

  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 $\Longrightarrow$ hierarchy of abstractions.

Dynamic Programming

Idea: Let $cost\_to\_goal(n)$ be the actual cost of a lowest-cost path from node n to a goal; $cost\_to\_goal(n)$ can be defined as

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

For a finite graph, we can precompute and store this using least-cost-first search with MPP, in the reverse graph.

Dynamic Programming

Idea: Let $cost\_to\_goal(n)$ be the actual cost of a lowest-cost path from node n to a goal; $cost\_to\_goal(n)$ can be defined as

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

For a finite graph, we can precompute and store this using least-cost-first search with MPP, in the reverse graph.

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

Dynamic Programming

Idea: Let $cost\_to\_goal(n)$ be the actual cost of a lowest-cost path from node n to a goal; $cost\_to\_goal(n)$ can be defined as

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

For a finite graph, we can precompute and store this using least-cost-first search with MPP, in the reverse graph.

- This can be used locally to determine what to do from *any* state.
- There are two main problems:

Dynamic Programming

Idea: Let $cost\_to\_goal(n)$ be the actual cost of a lowest-cost path from node n to a goal; $cost\_to\_goal(n)$ can be defined as

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

For a finite graph, we can precompute and store this using least-cost-first search with MPP, in the reverse graph.

- This can be used locally to determine what to do from *any* state.
- There are two main problems:
  - It requires enough space to store the graph.
  - The *cost_to_goal* function needs to be recomputed for each goal.

Dynamic Programming

Idea: Let $cost\_to\_goal(n)$ be the actual cost of a lowest-cost path from node n to a goal; $cost\_to\_goal(n)$ can be defined as

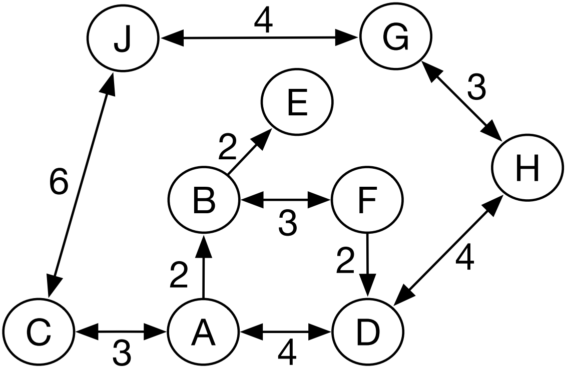
$$cost\_to\_goal(n) = \begin{cases} 0 & \text{if } goal(n), \\ \min_{\langle n,m\rangle \in A}(cost(\langle n,m\rangle) + cost\_to\_goal(m)) & \text{otherwise.} \end{cases}$$

For a finite graph, we can precompute and store this using least-cost-first search with MPP, in the reverse graph.

- This can be used locally to determine what to do from *any* state.
- There are two main problems:
  - It requires enough space to store the graph.
  - The *cost_to_goal* function needs to be recomputed for each goal.
- Implementation detail: in Python, make *expanded* in MPP a dictionary, so *expanded*[*s*] returns the cost from *s* to goal (cost found in search).

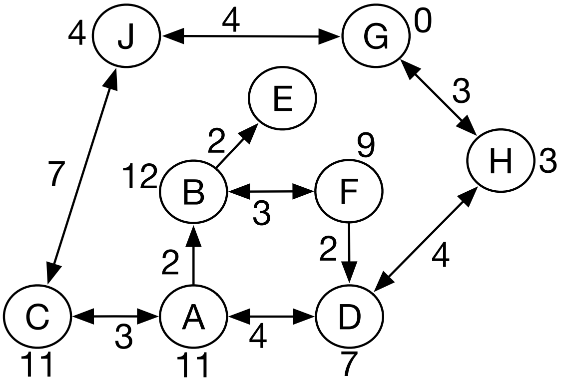
Example graph with heuristics

Goal: G.



Example graph cost-to-goal

Goal: G.



Value on nodes are *cost_to_goal* of arc.

(Partial) dynamic programming as a source of heuristics

Suppose

- there is not enough time or space to store the cost-to-goal for all nodes

(Partial) dynamic programming as a source of heuristics

Suppose

- there is not enough time or space to store the cost-to-goal for all nodes
- we stop the least-cost-first search early, and have expanded all paths with cost less than c. *expanded* is only defined for some states

(Partial) dynamic programming as a source of heuristics

Suppose

- there is not enough time or space to store the cost-to-goal for all nodes
- we stop the least-cost-first search early, and have expanded all paths with cost less than c. *expanded* is only defined for some states
- h is any admissible heuristic function that satisfies the montone restriction.

(Partial) dynamic programming as a source of heuristics

Suppose

- there is not enough time or space to store the cost-to-goal for all nodes
- we stop the least-cost-first search early, and have expanded all paths with cost less than c. *expanded* is only defined for some states
- h is any admissible heuristic function that satisfies the montone restriction.

The heuristic function

$$h'(n) = \begin{cases} expanded[n] & \text{if } expanded[n] \text{ is defined,} \\ \max(c, h(n)) & \text{otherwise.} \end{cases}$$

is an admissible heuristic function that that satisfies the montone restriction and (generally) improves h, as it is perfect for all values less than c.