Heuristic Search

- Idea: don't ignore the goal when selecting paths.
- Often there is extra knowledge that can be used to guide the search: heuristics.
- A heuristic function $h(n)$ is a nonnegative estimate of the cost of the least-cost path from node n to a goal node.
- $h(n)$ needs to be efficient to compute.
- h can be extended to paths: $h(\langle n_0, \ldots, n_k \rangle) = h(n_k)$.
- $h(n)$ is an underestimate if there is no path from n to a goal with cost less than $h(n)$.
- An admissible heuristic is a heuristic function that is an underestimate of the actual cost of a path to a goal.

Example Heuristic Functions

- If the nodes are points on a Euclidean plane and the cost is the distance, $h(n)$ can be the straight-line distance from n to the closest goal.
- If the nodes are locations and cost is time, we can use the distance to a goal divided by the maximum speed.
- If the goal is to collect all of the coins and not run out of fuel, the cost is an estimate of how many steps it will take to collect the rest of the coins, refuel when necessary, and return to goal position.
- A heuristic function can be found by solving a simpler (less constrained) version of the problem.

Heuristic depth-first Search

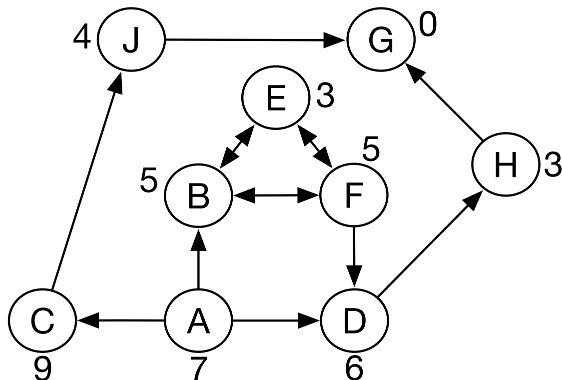
- Idea: in depth-first search select a neighbor that is closest to a goal according to the heuristic function.
- It inherits all of the advantages/disadvantages of depth-first search, but locally heads towards a goal.

Best-first Search

- Idea: select a path whose end is closest to a goal according to the heuristic function.
- Best-first search selects a path on the frontier with minimal h-value.
- It treats the frontier as a priority queue ordered by h.

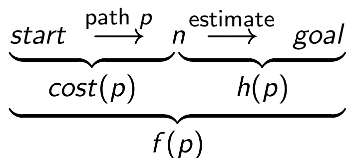
Illustrative Graph — Heuristic Search

From A get to G:



A^* Search

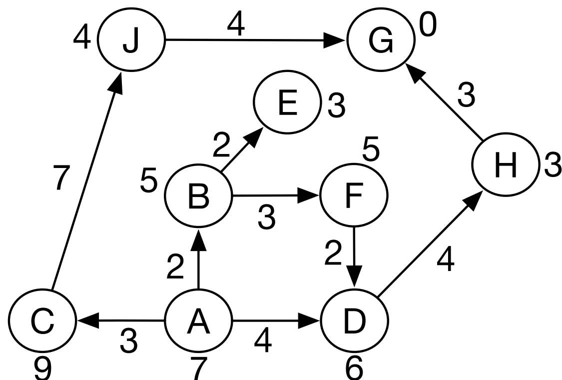
- A^* search uses both path cost and heuristic values
- $cost(p)$ is the cost of path p.
- $h(p)$ estimates the cost from the end of p to a goal.
- Let $f(p) = cost(p) + h(p)$.
  $f(p)$ estimates the total path cost of going from a start node to a goal via p.



- In A^* search, the frontier is a priority queue ordered by $f(p)$.
- It always selects the path on the frontier with the lowest estimated cost from the start to a goal node constrained to go via that path.

Example graph with heuristics (acyclic)

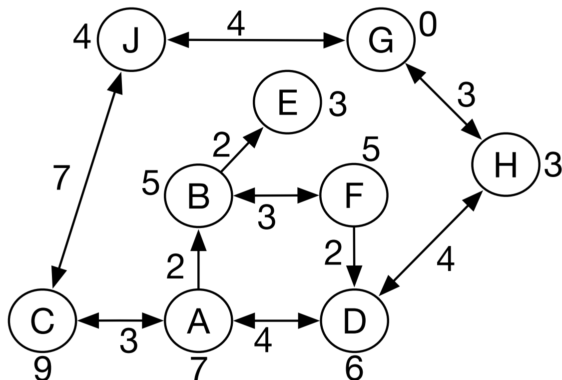
Start: A. Goal: G.



Heuristic value of a node is shown next to the node.

Example graph with heuristics

Start: A. Goal: G.



Heuristic value of a node is shown next to the node.

A^* Search Algorithm

- A^* is a mix of lowest-cost-first and best-first search.
- It treats the frontier as a priority queue ordered by $f(p)$.
- It always selects the node on the frontier with the lowest estimated distance from the start to a goal node constrained to go via that node.

Complexity of A^* Search

- Does A^* search guarantee to find the path with fewest arcs?
- Does A^* search guarantee to find the least-cost path?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of length of the path selected?
- What is the space complexity as a function of length of the path selected?
- How does the goal affect the search?

Admissibility of A^*

If there is a solution, A^* always finds an optimal solution – -the first path to a goal selected — if

- the branching factor is finite
- arc costs are bounded above zero (there is some $\epsilon > 0$ such that all of the arc costs are greater than ϵ), and
- $h(n)$ is nonnegative and an underestimate of the cost of the shortest path from n to a goal node:

$$0 \leq h(n) \leq \text{cost of shortest path from } n \text{ to a goal}$$

Why is A^* admissible?

- If a path p to a goal is selected from the frontier, can there be a lower cost path to a goal?
- $h(p) = 0$
- Suppose path p' is on the frontier. Because p was chosen before p', and $h(p) = 0$:

$$cost(p) \leq cost(p') + h(p').$$

- Because h is an underestimate:

$$cost(p') + h(p') \leq cost(p'')$$

  for any path p'' to a goal that extends p'.
- So $cost(p) \leq cost(p'')$ for any other path p'' to a goal.

Why is A^* admissible?

A^* can always find a solution if there is one:

- The frontier always contains the initial part of a path to a goal, before that goal is selected.
- A^* halts, as the costs of the paths on the frontier keeps increasing, and will eventually exceed any finite number.

How do good heuristics help?

Suppose c is the cost of an optimal solution. What happens to a path p from a start node, where

- $cost(p) + h(p) < c$
  It will be expanded
- $cost(p) + h(p) > c$
  It will not be expanded
- $cost(p) + h(p) = c$
  It might or might not be expanded.

How can a better heuristic function help?

Summary of Search Strategies

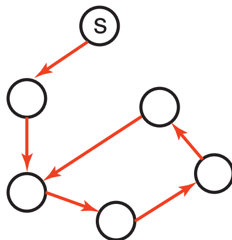
Strategy	Frontier Selection	Complete	Halts	Space
Depth-first	Last node added	No	No	Linear
Breadth-first	First node added	Yes	No	Exp
Heuristic depth-first	Local min $h(p)$	No	No	Linear
Best-first	Global min $h(p)$	No	No	Exp
Lowest-cost-first	Minimal $cost(p)$	Yes	No	Exp
A^*	Minimal $f(p)$	Yes	No	Exp

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

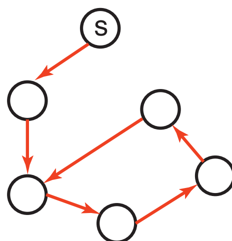


- A searcher can prune a path that ends in a node already on the path, without removing an optimal solution.

Graph searching with cycle pruning

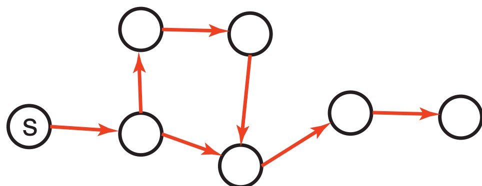
Input: a graph,
a set of start nodes,
Boolean procedure $goal(n)$ that tests if n is a goal node.
$frontier := \{\langle s \rangle : s \text{ is a start node}\}$
while *frontier* is not empty:
select and **remove** path $\langle n_0, \ldots, n_k \rangle$ from *frontier*
if $n_k \notin \{n_0, \ldots,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

- 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'.
- 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 $\langle m, n \rangle$.
- 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).
- 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.
- 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 $mb^{k/m} \ll b^k$.
- 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

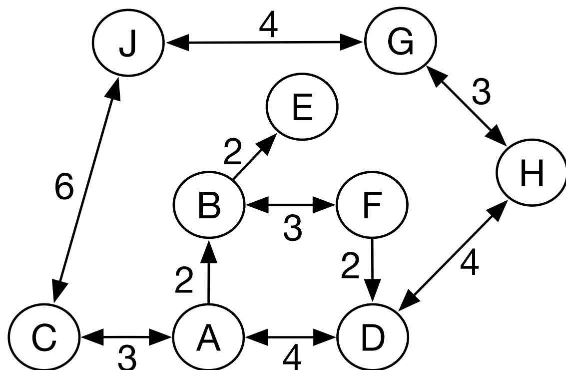
$$cost_to_goal(n) = \begin{cases} 0 & \text{if } goal(n), \\ \min_{\langle n,m\rangle \in A}(cost(\langle n,m\rangle) + cost_to_goal(m)) & \text{otherwise.} \end{cases}$$

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

- 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).

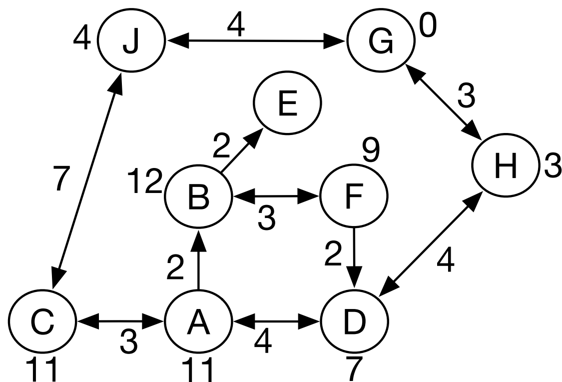
Example graph with heuristics

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
- 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

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

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.

Summary of Search Strategies

Strategy	Frontier	Complete	Halts	Space
Depth-first w/o CP	Last added	No	No	Linear
Depth-first w CP	Last added	No	Yes	Linear
Depth-first w MPP	Last added	No	Yes	Exp
Breadth-first w/o MPP	First added	Yes	No	Exp
Breadth-first w MPP	First added	Yes	Yes	Exp
Best-first w/o MPP	Min $h(p)$	No	No	Exp
Best-first w MPP	Min $h(p)$	No	Yes	Exp
A^* w/o MPP	Min $f(p)$	Yes	No	Exp
A^* w MPP	Min $f(p)$	Yes	Yes	Exp

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

Halts — on finite graph (perhaps with cycles).

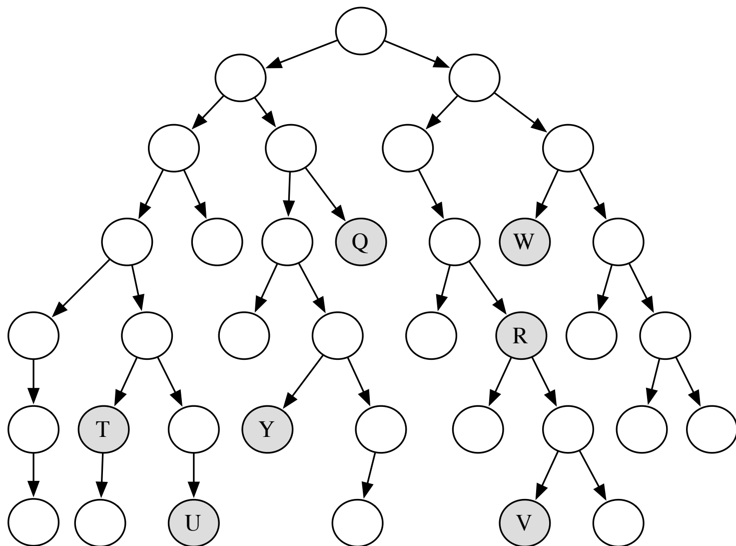
Space — as a function of the length of current path

Assume graph satisfies the assumptions of A^* proof + montonicity

Bounded Depth-first search

- A bounded depth-first search takes a bound (cost or depth) and does not expand paths that exceed the bound.
  - explores part of the search graph
  - uses space linear in the depth of the search.
- How does this relate to other searches?
- How can this be extended to be complete?

Which shaded goal will a depth-bounded search find first?



Iterative-deepening search

- Iterative-deepening search:
  - Start with a bound $b = 0$.
  - Do a bounded depth-first search with bound b
  - If a solution is found return that solution
  - Otherwise increment b and repeat.
- This will find the same first solution as what other method?
- How much space is used?
- What happens if there is no path to a goal?
- Surely recomputing paths is wasteful!!!

Iterative Deepening Complexity

Complexity with solution at depth k & branching factor b:

level	breadth-first	iterative deepening	# nodes
1	1	k	b
2	1	$k-1$	b^2
$\cdots$	$\cdots$	$\cdots$	$\cdots$
$k-1$	1	2	b^{k-1}
k	1	1	b^k
total	$\geq b^k$	$\leq b^k \left(\frac{b}{b-1}\right)^2$	

Depth-first Branch-and-Bound

- combines depth-first search with heuristic information.
- finds optimal solution.
- most useful when there are multiple solutions, and we want an optimal one.
- uses the space of depth-first search.

Depth-first Branch-and-Bound

Suppose we want to find a single optimal solution.

- Suppose *bound* is the cost of the lowest-cost path found to a goal so far.
- What if the search encounters a path p such that $cost(p) + h(p) \geq bound$?
  p can be pruned.
- What can we do if a non-pruned path to a goal is found?
  bound can be set to the cost of p, and p can be remembered as the best solution so far.
- Why should this use a depth-first search?
  Uses linear space.
- What can be guaranteed when the search completes?
  It has found an optimal solution.
- How should the bound be initialized?

© 2023 D. L. Poole and A. K. Mackworth

Depth-first Branch-and-Bound: Initializing Bound

- The bound can be initialized to ∞.
- The bound can be set to an estimate of the optimal path cost. After depth-first search terminates either:
  - A solution was found.
  - No solution was found, and no path was pruned
  - No solution was found, and a path was pruned.

Which shaded goals will be best solutions so far?