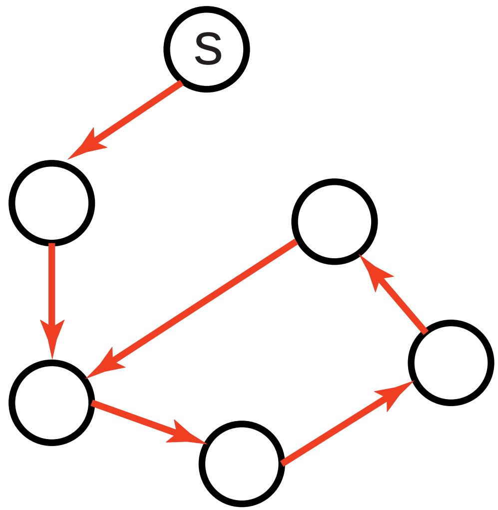
Summary of Search Strategies

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

Cycle Checking

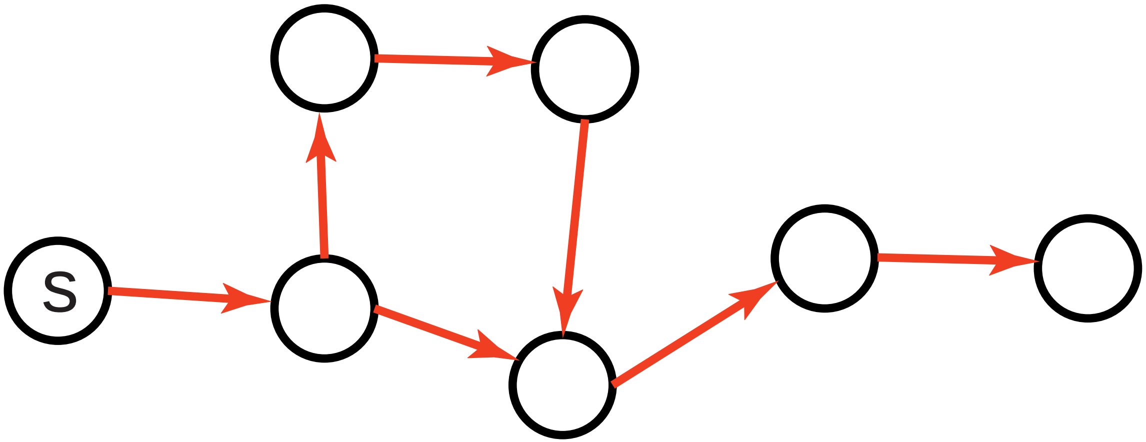


You can prune a node *n* that is on the path from the start node to *n*. This pruning cannot remove an optimal solution.

Using depth-first methods, with the graph explicitly stored, this can be done in constant time.

For other methods, the cost is linear in path length.

Multiple-Path Pruning



You can prune a node *n* that you have already found a path to.

Multiple-path pruning subsumes a cycle check.

This entails storing all nodes you have found paths to.

What if you want the shortest path, but a subsequent path found is shorter than than first path found?

Multiple-Path Pruning & Optimal Solution

Problem: what if a subsequent path to n is shorter that the first path to n?

- You can remove all paths from the frontier that use the longer path.
- You can change the initial segment of the paths on the frontier to use the shorter path.
- You can ensure this doesn't happen. You make sure that the shortest path to a node is found first.

Multiple-Path Pruning & A^*

Suppose node n was selected, but there was a shorter path to n. Suppose this shorter path was via node n' on the frontier.

$g(n) + h(n) \leq g(n') + h(n')$ because n was selected before n'.

$g(n') + d(n', n) < g(n)$ because the path to n via n' is shorter.

$$d(n', n) < g(n) - g(n') \leq h(n') - h(n).$$

You can ensure this doesn't occur if $|h(n') - h(n)| \leq d(n', n)$.

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

Iterative Deepening

- So far all search strategies that are guaranteed to halt use exponential space.
- Idea: let's recompute elements of the frontier rather than saving them.
- Look for proofs of depth 0, then 1, then 2, then 3, etc.
- You need a depth-bounded depth-first searcher.
- If a proof cannot be found at depth B, look for a proof at depth $B + 1$. Increase the depth-bound when the search fails unnaturally (depth-bound was reached).

Depth-bounded depth-first search

$dbsearch(N, DB, P)$ is true if path P is a path of length DB from N to a goal.

$$
\begin{aligned}
& dbsearch(N, 0, [N]) \leftarrow \\
& \qquad is\_goal(N). \\
& dbsearch(N, DB, [N|P]) \leftarrow \\
& \qquad DB > 0 \wedge \\
& \qquad neighbors(N, NNs) \wedge \\
& \qquad member(NN, NNs) \wedge \\
& \qquad DB_1 \text{ is } DB - 1 \wedge \\
& \qquad dbsearch(NN, DB_1, P).
\end{aligned}
$$

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
$k-1$	1	2	b^{k-1}
k	1	1	b^k
	$\geq b^k$	$\leq b^k \left(\frac{b}{b-1}\right)^2$	

Direction of Search

The definition of searching is symmetric: find path from start nodes to goal node or from goal node to start nodes.

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: sometimes when graph is dynamically constructed, you may not be able to construct the backwards graph.

Bidirectional Search

- You can 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 one breadth-first method that builds a set of locations that can lead to the goal. In the other direction another method can be used to find a path to these interesting locations.

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.

You can solve the subproblems using islands $\Longrightarrow$

hierarchy of abstractions.

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:

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

This can be used locally to determine what to do.

There are two main problems:

- You need enough space to store the graph.
- The *dist* function needs to be recomputed for each goal.