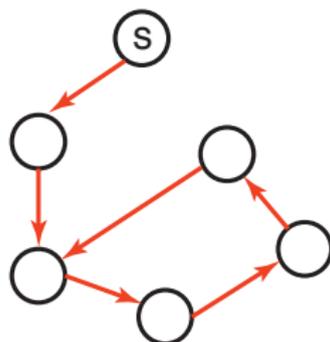


# Summary of Search Strategies

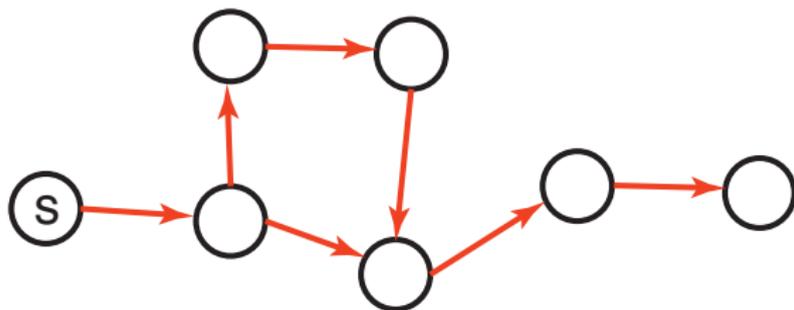
Strategy	Frontier Selection	Halts?	Space
Depth-first	Last node added	No	Linear
Breadth-first	First node added	Yes	Exp
Heuristic depth-first	Local min $h(n)$	No	Linear
Best-first	Global min $h(n)$	No	Exp
Lowest-cost-first	Minimal $cost(n)$	Yes	Exp
$A^*$	Minimal $f(n)$	Yes	Exp

# Cycle Checking



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



- Multiple path pruning: prune a path to node  $n$  that the searcher has already found a path to.
- Multiple-path pruning subsumes a cycle check.
- This entails storing all nodes it has found paths to.
- Want to guarantee that an optimal solution can still be found.

**Problem:** what if a subsequent path to  $n$  is shorter 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 shorter path.
- ensure this doesn't happen. Make sure that the shortest path to a node is found first.

## Multiple-Path Pruning & $A^*$

- Suppose path  $p$  to  $n$  was selected, but there is a shorter path to  $n$ . Suppose this shorter path is via path  $p'$  on the frontier.
- Suppose path  $p'$  ends at node  $n'$ .
- $cost(p) + h(n) \leq cost(p') + h(n')$  because  $p$  was selected before  $p'$ .
- $cost(p') + cost(n', n) < cost(p)$  because the path to  $n$  via  $p'$  is shorter.

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

You 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  $|h(m) - h(n)| \leq \text{cost}(m, n)$  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.
- This is a strengthening of the admissibility criterion.

- 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 paths of depth 0, then 1, then 2, then 3, etc.
- You need a depth-bounded depth-first searcher.
- If a path cannot be found at depth  $B$ , look for a path at depth  $B + 1$ . Increase the depth-bound when the search fails unnaturally (depth-bound was reached).

# Iterative-deepening search

```
Boolean natural_failure;  
Procedure dbsearch( $\langle n_0, \dots, n_k \rangle : path, bound : int$ ):  
    if goal( $n_k$ ) and  $bound = 0$  report path  $\langle n_0, \dots, n_k \rangle$ ;  
    if  $bound > 0$   
        for each neighbor  $n$  of  $n_k$   
            dbsearch( $\langle n_0, \dots, n_k, n \rangle, bound - 1$ );  
        else if  $n_k$  has a neighbor then natural_failure := false;  
end procedure dbsearch;  
Procedure idsearch( $S : node$ ):  
    Integer  $bound := 0$ ;  
    repeat  
        natural_failure := true;  
        dbsearch( $\langle s \rangle, bound$ );  
         $bound := bound + 1$ ;  
    until natural_failure;  
end procedure idsearch
```

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

# Depth-first Branch-and-Bound

- Way to combine 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

- Idea: maintain the cost of the lowest-cost path found to a goal so far, call this *bound*.
- If the search encounters a path  $p$  such that  $cost(p) + h(p) \geq bound$ , path  $p$  can be pruned.
- If a non-pruned path to a goal is found, it must be better than the previous best path. This new solution is remembered and *bound* is set to its cost.
- The search can be a depth-first search to save space.

# Depth-first Branch-and-Bound

- Idea: maintain the cost of the lowest-cost path found to a goal so far, call this *bound*.
- If the search encounters a path  $p$  such that  $cost(p) + h(p) \geq bound$ , path  $p$  can be pruned.
- If a non-pruned path to a goal is found, it must be better than the previous best path. This new solution is remembered and *bound* is set to its cost.
- The search can be a depth-first search to save space.
- How should the bound be initialized?

# Depth-first Branch-and-Bound: Initializing Bound

- The bound can be initialized to  $\infty$ .
- 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.

# 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.

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

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

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  $\implies$  **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:

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

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.