Uninformed (Blind) Search

- Many problems are amenable to attack by search methods
- We shall analyse a number of different search strategies
- We begin by examining search methods that have no problem-specific knowledge to use as guidance
- Theme: “In [search] space, no one can hear you scream.”
  
  [Aliens]

- References:
Breadth-First Search

Breadth-First Search — Analysis

- Complete
- Optimal — provided path cost is nondecreasing function of the depth of the node
- Maximum number of nodes generated: 
  \[ b + b^2 + b^3 + \ldots + b^d \]
  (where \( b \) = forward branching factor; \( d \) = path length to solution)
- Time and space requirements are the same \( O(b^d) \)

Uniform Cost Search

- Also known as Lowest-Cost-First search
- Shallower goal state may not be the least-cost solution
- Idea: Expand lowest cost (measured by path cost \( g(n) \)) node on the frontier
- Order nodes in the frontier in increasing order of path cost
- Breadth-first search \( \approx \) uniform cost search where \( g(n) = \text{depth}(n) \) (except breadth-first search stops when goal state generated)
- Include check that generated state has not already been explored
- Include test to ensure frontier contains only one node for any state — for path with lowest cost

% Figure 11.10  An implementation of breadth-first search.
% solve(Start, Solution):
% Solution is a path (in reverse order) from Start to a goal
solve(Start, Solution) :-
  breadthfirst([Start], Solution).
% breadthfirst([Path1, Path2, \ldots], Solution):
% Solution is an extension to a goal of one of paths
breadthfirst([\[Node|Path\]|\_], [\[Node|Path\]]) :-
goal(Node).
breadthfirst([\[Node|Path\]|\_], [\[Node|Path\]]),
  extend(Path, NewPaths),
  conc(Paths, NewPaths, Paths1),
  breadthfirst(Paths1, Solution).
extend([\[Node|Path\]|\_], NewPaths) :-
  bagof([\[NewNode, Node|Path\]|\_],
    (s(Node, NewNode), not member(NewNode, [\[Node|Path\]])),
    NewPaths), !.
extend([\_], []). % bagof failed: Node has no successor
Uniform Cost Search

- Complete
- Optimal — provided path cost does not decrease along path (i.e. $g(\text{successor}(n)) \geq g(n)$ for all $n$)
- Reasonable assumption when path cost is cost of applying operators along the path
- Performs like breadth-first search when $g(n) = \text{depth}(n)$
- If there are paths with negative cost we would need to perform an exhaustive search

Depth-First Search

- Idea: Always expand node at deepest level of tree and when search hits a dead-end return back to expand nodes at a shallower level
- Can be implemented by using a stack to store frontier nodes
- At any point depth-first search stores single path from root to leaf together with any remaining unexpanded siblings of nodes along path
- Stop when node with goal state is expanded
- Include check that generated state has not already been explored along a path – cycle checking
Depth-First Search

Figure 11.7 A depth-first search program that avoids cycling.

solve(Node, Solution):
  Solution is acyclic path (in reverse order) between Node and goal.

solve(Node, Solution) :-
  depthfirst([], Node, Solution).

% depthfirst(Path, Node, Solution):
% extending the path [Node|Path] to a goal gives Solution.

depthfirst(Path, Node, [Node|Path]) :-
  goal(Node).

depthfirst(Path, Node, Sol) :-
  s(Node, Node1),
  not member(Node1, Path), % Prevent a cycle
  depthfirst([Node|Path], Node1, Sol).

Depth-First Search — Analysis

- Storage: $O(bm)$ nodes (where $m$ = maximum depth of search tree)
- Time: $O(b^m)$
- In cases where problem has many solutions depth-first search may outperform breadth-first search because there is a good chance it will happen upon a solution after exploring only a small part of the search space.
- However, depth-first search may get stuck following a deep or infinite path even when a solution exists at a relatively shallow level.
- Therefore, depth-first search is not complete and not optimal.
- Avoid depth-first search for problems with deep or infinite paths.

Depth-Limited Search

- Idea: impose bound on depth of a path.
- In some problems you may know that a solution should be found within a certain cost (e.g., a certain number of moves) and therefore there is no need to search paths beyond this point for a solution.

Depth-Limited Search — Analysis

- Complete but not optimal (may not find shortest solution).
- However, if the depth limit chosen is too small a solution may not be found and depth-limited search is incomplete in this case.
- Time and space complexity similar to depth-first search (but relative to depth limit rather than maximum depth).

Iterative Deepening Search

- It can be very difficult to decide upon a depth limit for search.
- The maximum path cost between any two nodes is known as the diameter of the state space.
- This would be a good candidate for a depth limit but it may be difficult to determine in advance.
- Idea: try all possible depth limits in turn.
- Combines benefits of depth-first and breadth-first search.
Iterative Deepening Search

- Optimal; Complete; Space \( O(bd) \)
- Some states are expanded multiple times. Isn’t this wasteful?
- Number of expansions to depth \( d = 1 + b + b^2 + b^3 + \ldots + b^d \)
- Therefore, for iterative deepening, total expansions = \((d + 1)1 + (d)b + (d - 1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + b^d\)
- The higher the branching factor, the lower the overhead (even for \( b = 2 \), search takes about twice as long)
- Hence time complexity still \( O(b^d) \)
- May consider doubling depth limit at each iteration — overhead \( O(d \log d) \)
- In general, iterative deepening is the preferred search strategy for a large search space where depth of solution is not known

Bidirectional Search

- Idea: search forward from initial state and backward from goal state at the same time until the two meet
- To search backwards we need to generate predecessors of nodes (this is not always possible or easy)
- If operators are reversible successor sets and predecessor sets are identical
- If there are many goal states we can try a multi-state search (how would you do this in chess, say?)
- Need to check whether a node occurs in both searches — may not be easy
- Which is the best search strategy for each half?
Bidirectional Search — Analysis

- If solution exists at depth $d$ then bidirectional search requires time $O(2b^d) = O(b^{d/2})$ (assuming constant time checking of intersection).
- To check for intersection must have all states from one of the searches in memory, therefore space complexity is $O(b^{d/2})$.

Conclusion

- We have surveyed a variety of uninformed search strategies.
- All can be implemented within the framework of the general search procedure.
- There are other considerations we can make like trying to save time by not expanding a node which has already been seen on some other path.
- There are a number of techniques available and often use is made of a hash table to store all nodes generated.

Summary — Blind Search

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform Cost</th>
<th>Depth-First</th>
<th>Depth-Limited</th>
<th>Iterative Deepening</th>
<th>Bidirectional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>$b^d$</td>
<td>$b^d$</td>
<td>$b^m$</td>
<td>$b^d$</td>
<td>$b^d$</td>
<td>$b^{d/2}$</td>
</tr>
<tr>
<td>Space</td>
<td>$b^d$</td>
<td>$b^d$</td>
<td>$bm$</td>
<td>$bd$</td>
<td>$bd$</td>
<td>$b^{d/2}$</td>
</tr>
<tr>
<td>Optimal</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Complete</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes, if $l \geq d$</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

$b$ — branching factor
$d$ — depth of shallowest solution
$m$ — maximum depth of tree
$l$ — depth limit