CS 331: Artificial Intelligence
Uninformed Search

Simpler Search Problems

Search Problem Formulation

A search problem has 5 components:
1. A finite set of states $S$
2. A non-empty set of initial states $I \subseteq S$
3. A non-empty set of goal states $G \subseteq S$
4. A successor function $\text{suc}(s)$ which takes a state $s$ as input and returns as output the set of states you can reach from state $s$ in one step.
5. A cost function $\text{cost}(s, s')$ which returns the non-negative one-step cost of travelling from state $s$ to $s'$. The cost function is only defined if $s'$ is a successor state of $s$.

Example: Oregon

$S = \{\text{Coos Bay, Newport, Corvallis, Junction City, Eugene, Medford, Albany, Lebanon, Salem, Portland, McMinnville}\}$
$I = \{\text{Corvallis}\}$
$G = \{\text{Medford}\}$
$\text{suc}(\text{Corvallis}) = \{\text{Albany, Newport, McMinnville, Junction City}\}$
$\text{Cost}(s, s') = 1$ for all transitions

Real World Search Problems

Assumptions About Our Environment

- Fully Observable
- Deterministic
- Sequential
- Static
- Discrete
- Single-agent
Results of a Search Problem

- **Solution**
  Path from initial state to goal state

- **Solution quality**
  Path cost (3 in this case)

- **Optimal solution**
  Lowest path cost among all solutions (In this case, we found the optimal solution)

---

Search Tree

**Start with Initial State**

**Search Tree**

Is initial state the goal?
- Yes, return solution
- No, apply Successor() function

*Queue*

McMinnville
Albany
Junction City
Newport

**Search Tree**

Apply Successor() function

These nodes have not been expanded yet. Call them the fringe. We’ll put them in a queue.

**Search Tree**

Now remove a node from the queue. If it's a goal state, return the solution. Otherwise, call Successor() on it, and put the results in the queue. Repeat.

**Queue**

Albany
Junction City
Newport
Portland

Things to note:
- Order in which you expand nodes (in this example, we took the first node in the queue)
- Avoid repeated states
Tree-Search Pseudocode

```plaintext
function TREE-SEARCH(problem, fringe) returns a solution, or failure
fringe ← INSERT(Make-Nodo(INITIAL-STATE[problem]), fringe)
loop do
  if fringe is empty then return failure
  node ← REMOVE-FRONT(fringe)
  if GOAL-TEST(node) then return SOLUTION(node)
  fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

Note: Goal test happens after we grab a node off the queue.

Tree-Search Pseudocode

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
fringe ← INSERT(Make-Nodo(INITIAL-STATE[problem]), fringe)
loop do
  if fringe is empty then return failure
  node ← REMOVE-FRONT(fringe)
  if GOAL-TEST(node) then return SOLUTION(node)
  fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

Why are these parent node backpointers important?

Tree-Search Pseudocode

```
function EXPAND(node, problem) returns a set of nodes
successors ← the empty set
for each action, result in SUCCESSORS(F[problem][STATE[node]]) do
  s ← a new node
  PARENT-NODE[s] ← node
  ACTION[s] ← action
  STATE[s] ← result
  PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
  Depth[s] ← Depth[node] + 1
  add s to successors
return successors
```

Uninformed Search

- No info about states other than generating successors and recognizing goal states
- Later on we’ll talk about informed search – can tell if a non-goal state is more promising than another

Evaluating Uninformed Search

- Completeness
  Is the algorithm guaranteed to find a solution when there is one?
- Optimality
  Does it find the optimal solution?
- Time complexity
  How long does it take to find a solution?
- Space complexity
  How much memory is needed to perform the search

Complexity

1. Branching factor (b) – maximum number of successors of any node
2. Depth (d) of the shallowest goal node
3. Maximum length (m) of any path in the search space

Time Complexity: number of nodes generated during search
Space Complexity: maximum number of nodes stored in memory
Uninformed Search Algorithms

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative Deepening Depth-first Search
- Bidirectional search

Breadth-First Search

- Expand all nodes at a given depth before any nodes at the next level are expanded
- Implement with a FIFO queue

Breadth First Search Example

Evaluating BFS

<table>
<thead>
<tr>
<th></th>
<th>Complete?</th>
<th>Optimal?</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes provided branching factor is finite</td>
<td>Yes if step costs are identical</td>
<td>(b + b^2 + b^3 + \ldots + b^{d+1} - b) = (O(b^{d+1}))</td>
<td>(O(b^{d+1}))</td>
</tr>
</tbody>
</table>

Exponential time and space complexity make BFS impractical for all but the smallest problems.
### Uniform-cost Search

- What if step costs are not equal?
- Recall that BFS expands the shallowest node
- Now we expand the node with the lowest path cost
- Uses priority queues

Note: Gets stuck if there is a zero-cost action leading back to the same state.
For completeness and optimality, we require the cost of every step to be \( \geq \varepsilon \).

### Evaluating Uniform-cost Search

<table>
<thead>
<tr>
<th>Complete?</th>
<th>Optimal?</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes provided branching factor is finite and step costs ( \geq \varepsilon ) for small positive ( \varepsilon ).</td>
<td>Yes</td>
<td>( O(b^{1+\text{floor}(C*/\varepsilon)}) ) where ( C^* ) is the cost of the optimal solution</td>
<td>( O(b^{1+\text{floor}(C*/\varepsilon)}) ) where ( C^* ) is the cost of the optimal solution</td>
</tr>
</tbody>
</table>

### Depth-first Search

- Expands the deepest node in the current fringe of the search tree
- Implemented with a LIFO queue

### Evaluating Depth-first Search

<table>
<thead>
<tr>
<th>Complete?</th>
<th>Optimal?</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
</table>
Evaluating Depth-first Search

<table>
<thead>
<tr>
<th>Complete?</th>
<th>No (Might not terminate if it goes down an infinite path with no solutions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal?</td>
<td>No (Could expand a much longer path than the optimal one first)</td>
</tr>
<tr>
<td>Time Complexity</td>
<td>O(b^m)</td>
</tr>
<tr>
<td>Space Complexity</td>
<td>O(bm)</td>
</tr>
</tbody>
</table>

Depth-limited Search

- Solves infinite path problem by using predetermined depth limit \( l \)
- Nodes at depth \( l \) are treated as if they have no successors
- Can use knowledge of the problem to determine \( l \) (but in general you don’t know this in advance)

Evaluating Depth-limited Search

<table>
<thead>
<tr>
<th>Complete?</th>
<th>No (If shallowest goal node beyond depth limit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal?</td>
<td>No (If depth limit &gt; depth of shallowest goal node and we expand a much longer path than the optimal one first)</td>
</tr>
<tr>
<td>Time Complexity</td>
<td>O(b^l)</td>
</tr>
<tr>
<td>Space Complexity</td>
<td>O(b^l)</td>
</tr>
</tbody>
</table>

Iterative Deepening Depth-first Search

- Do DFS with depth limit 0, 1, 2, … until a goal is found
- Combines benefits of both DFS and BFS

Iterative Deepening Depth-first Search Example

IDDFS Example

Limit = 0

Limit = 1

Limit = 2

Limit = 3
IDDFS Example

Evaluating Iterative Deepening Depth-first Search

Complete? |
--- |
Optimal? |
Time Complexity |
Space Complexity |

Isn’t Iterative Deepening Wasteful?

- Actually, no! Most of the nodes are at the bottom level, doesn’t matter that upper levels are generated multiple times.
- To see this, add up the 4th column below:

<table>
<thead>
<tr>
<th>Depth</th>
<th># of nodes</th>
<th># of times generated</th>
<th>Total # of nodes generated at depth d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b</td>
<td>d</td>
<td>(b^d)</td>
</tr>
<tr>
<td>2</td>
<td>b^2</td>
<td>d-1</td>
<td>(d-1)b^2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>b^d</td>
<td>1</td>
<td>(1)b^d</td>
</tr>
</tbody>
</table>

In general, iterative deepening is the preferred uninformed search method when there is a large search space and the depth of the solution is not known

Bidirectional Search

- Run one search forward from the initial state
- Run another search backward from the goal
- Stop when the two searches meet in the middle
Bidirectional Search

- Needs an efficiently computable Predecessor() function
- What if there are several goal states?
  - Create a new dummy goal state whose predecessors are the actual goal states
- Difficult when the goal is an abstract description like “no queen attacks another queen”

Evaluating Bidirectional Search

<table>
<thead>
<tr>
<th>Complete?</th>
<th>Yes provided branching factor is finite and both directions use BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal?</td>
<td>Yes if the step costs are all identical and both directions use BFS</td>
</tr>
<tr>
<td>Time Complexity</td>
<td>$O(b^{d/2})$</td>
</tr>
<tr>
<td>Space Complexity</td>
<td>$O(b^{d/2})$ (At least one search tree must be kept in memory for the membership check)</td>
</tr>
</tbody>
</table>

Avoiding Repeated States

- Tradeoff between space and time!
- Need a closed list which stores every expanded node (memory requirements could make search infeasible)
- If the current node matches a node on the closed list, discard it (ie. discard the newly discovered path)
- We’ll refer to this algorithm as GRAPH-SEARCH
- Is this optimal? Only for uniform-cost search or breadth-first search with constant step costs.

GRAPH-SEARCH

Things You Should Know

- How to formalize a search problem
- How BFS, UCS, DFS, DLS, IDS and Bidirectional search work
- Whether the above searches are complete and optimal plus their time and space complexity
- The pros and cons of the above searches