Search II: Uninformed Search Strategies

AIMA 3.4

Iterative-Deepening Search
Bidirectional Search
Uniform-Cost Search

Example: Missionaries & Cannibals

Three missionaries and three cannibals come to a river. A rowboat that seats two is available. If the cannibals ever outnumber the missionaries on either bank of the river, the missionaries will be eaten. (problem 3.9)

How shall they cross the river?

Missionaries and Cannibals

States: (CL, ML, BL)
Initial 331  Goal 000

Actions:
Travel Across  Travel Back
-101  101
-201  201
-011  011
-021  021
-111  111

Review: Search Strategies

Dimensions for evaluation
- Completeness: always find the solution?
- Time complexity - # of nodes generated
- Space complexity - # of nodes in memory
- Optimality - finds a least cost solution (lowest path cost)?

Time/space complexity measurements
- b, maximum branching factor of search tree
- d, depth of the shallowest goal node
- m, maximum length of any path in the state space (potentially ∞)
Search Conundrum

- **Breadth-first**
  - Complete, Optimal
  - but uses $O(b^d)$ space
- **Depth-first**
  - Not complete unless $m$ is bounded
  - Not optimal
  - Uses $O(b^m)$ time; terrible if $m >> d$
  - but only uses $O(b * m)$ space

How can we get the best of both?

Depth-limited search: A building block

- **Depth-First search** *but with depth limit $\ell$*
  - i.e. nodes at depth $\ell$ have no successors.
  - No infinite-path problem!
- If $\ell = d$ (by luck!), then optimal
  - But:
    - If $\ell < d$ then incomplete
    - If $\ell > d$ then not optimal
- **Time complexity:** $O(b^\ell)$
- **Space complexity:** $O(b \ell)$

Iterative deepening search

- A general strategy to find best depth limit $\ell$
  - Key idea: use Depth-limited search as subroutine, with increasing $\ell$.
  - For $d = 0$ to $\infty$
    - depth-limited-search to level $d$
    - if it succeeds
      - then return solution
- Complete & optimal: Goal is always found at depth $d$, the depth of the shallowest goal-node.
  - *Could this possibly be efficient?*

Nodes constructed at each deepening

- Depth 0: 0 (Given the node, doesn’t construct it.)
- Depth 1: $b^1$ nodes
- Depth 2: $b$ nodes + $b^2$ nodes
- Depth 3: $b$ nodes + $b^2$ nodes + $b^3$ nodes
- ...

Total nodes constructed:

- Depth 0: 0 (Given the node, doesn’t construct it.)
- Depth 1: $b^1$ = $b$ nodes
- Depth 2: $b$ nodes + $b^2$ nodes
- Depth 3: $b$ nodes + $b^2$ nodes + $b^3$ nodes
- ...

Suppose the first solution is the last node at depth 3:
Total nodes constructed:
- $3 * b$ nodes + $2 * b^2$ nodes + $1 * b^3$ nodes

ID search, Evaluation II: Time Complexity

- More generally, the time complexity is
  - $N(\text{IDS}) = (d)b + (d-1)b^2 + … + (1)b^d = O(b^d)$
- As efficient in terms of $O(\ldots)$ as Breadth First Search:
  - $N(\text{BFS}) = b + b^2 + … + b^d = O(b^d)$
### ID search, Evaluation III

- Complete: YES (no infinite paths)
- Time complexity: \( O(b^d) \)
- Space complexity: \( O(bd) \)
- Optimal: YES if step cost is 1.

### Summary of algorithms

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Depth-First</th>
<th>Depth-limited</th>
<th>Iterative deepening</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Time</td>
<td>( h^d )</td>
<td>( b^m )</td>
<td>( b^l )</td>
<td>( b^d )</td>
</tr>
<tr>
<td>Space</td>
<td>( h^d )</td>
<td>( b^m )</td>
<td>( b^l )</td>
<td>( bd )</td>
</tr>
<tr>
<td>Optimal?</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>

### Very briefly: Bidirectional search

- Two simultaneous searches from start an goal.
  - Motivation: \( b^{d/2} + b^{d/2} < b^d \)
- Check whether the node belongs to the other frontier before expansion.
- Space complexity is the most significant weakness.
- Complete and optimal if both searches are Breadth-First.

### How to search backwards?

- The predecessor of each node must be efficiently computable.
  - Works well when actions are easily reversible.

### "Uniform Cost" Search

"In computer science, uniform-cost search (UCS) is a tree search algorithm used for traversing or searching a weighted tree, tree structure, or graph." - Wikipedia

### Motivation: Romanian Map Problem

- All our search methods so far assume step-cost = 1
- This is only true for some problems
**g(N): the path cost function**

- If all moves equal in cost:
  - Cost = # of nodes in path - 1
  - \( g(N) = \text{depth}(N) \) in the search tree
  - Equivalent to what we’ve been assuming so far

- Assigning a (potentially) unique cost to each step
  - \( N_0, N_1, N_2, N_3 \) = nodes visited on path \( p \) from \( N_0 \) to \( N_3 \)
  - \( C(i,j) \): Cost of going from \( N_i \) to \( N_j \)
  - If \( N_0 \) the root of the search tree,
    \[ g(N_3) = C(0,1)+C(1,2)+C(2,3) \]

**Uniform-cost search (UCS)**

- Extension of BF-search:
  - Expand node with lowest path cost
- Implementation:
  - frontier = priority queue ordered by \( g(n) \)
- Subtle but significant difference from BFS:
  - Tests if a node is a goal state when it is selected for expansion, not when it is added to the frontier.
  - Updates a node on the frontier if a better path to the same state is found.
  - So always enqueues a node before checking whether it is a goal.

**Complexity of UCS**

- Complete!
- Optimal! if the cost of each step exceeds some positive bound \( \epsilon \).
- Time complexity: \( O(b^{d+C*/\epsilon}) \)
- Space complexity: \( O(b^{d+C*/\epsilon}) \)
  where \( C^* \) is the cost of the optimal solution
  (if all step costs are equal, this becomes \( O(b^{d+1}) \))

**Summary of algorithms (for notes)**

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-cost</th>
<th>Depth-limited</th>
<th>Iterative deepening</th>
<th>Bidirectional search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Time</td>
<td>( b^d )</td>
<td>( b^{d+C^*} )</td>
<td>( b^n )</td>
<td>( b^d )</td>
<td>( b^{d+2} )</td>
</tr>
<tr>
<td>Space</td>
<td>( b^d )</td>
<td>( b^{d+C^*} )</td>
<td>( b_m )</td>
<td>( b^d )</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>
</tr>
</tbody>
</table>

**NOTE:**

- Dijkstra’s algorithm just UCS without goal

**Assumes \( b \) is finite**