Title: Solving Problems by Searching
AIMA: Chapter 3 (Sections 3.4)

Introduction to Artificial Intelligence
CSCE 476-876, Spring 2012
URL: www.cse.unl.edu/~choueiry/S12-476-876

Berthe Y. Choueiry (Shu-we-ri)
choueiry@cse.unl.edu, (402)472-5444
function GENERAL-SEARCH( problem, strategy) returns a solution, or failure
   initialize the search tree using the initial state of problem
   loop do
      if there are no candidates for expansion then return failure
      choose a leaf node for expansion according to strategy
      if the node contains a goal state then return the corresponding solution
      else expand the node and add the resulting nodes to the search tree
   end

Essence of search: which node to expand first?
   \[ \rightarrow \text{search strategy} \]

A strategy is defined by picking the order of node expansion
Types of Search

Uninformed: use only information available in problem definition

Heuristic: exploits some knowledge of the domain

Uninformed search strategies

1. Breadth-first search
2. Uniform-cost search
3. Depth-first search
4. Depth-limited search
5. Iterative deepening depth-first search
6. Bidirectional search
Search strategies

Criteria for evaluating search:

1. Completeness: does it always find a solution if one exists?
2. Time complexity: number of nodes generated/expanded
3. Space complexity: maximum number of nodes in memory
4. Optimality: does it always find a least-cost solution?

Time/space complexity measured in terms of:

- $b$: maximum branching factor of the search tree
- $d$: depth of the least-cost solution
- $m$: maximum depth of the search space (may be $\infty$)
**Breadth-first search (I)**

→ Expand root node
→ Expand *all* children of root
→ Expand *each* child of root
→ Expand successors of each child of root, etc.

---

→ Expands nodes at depth $d$ before nodes at depth $d + 1$
→ Systematically considers all paths length 1, then length 2, etc.
→ Implement: put successors at end of queue.. FIFO
Breadth-first search (2)
Breadth-first search (3)

→ One solution?
→ Many solutions? Finds shallowest goal first

1. Complete? Yes, if \( b \) is finite

2. Optimal? provided cost increases monotonically with depth, not in general (e.g., actions have same cost)

3. Time? \( 1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1}) \)

\[
O(b^{d+1}) \begin{cases} 
\text{branching factor } b \\ 
\text{depth } d 
\end{cases}
\]

4. Space? same, \( O(b^{d+1}) \), keeps every node in memory, big problem

can easily generate nodes at 10MB/sec so 24hrs = 860GB
Uniform-cost search (I)

→ Breadth-first does not consider path cost $g(x)$
→ Uniform-cost expands first lowest-cost node on the fringe
→ Implement: sort queue in decreasing cost order

When $g(x) = \text{Depth}(x)$ → Breadth-first ≡ Uniform-cost
Uniform-cost search (2)

1. Complete?
   Yes, if cost $\geq \epsilon$

2. Optimal?
   If the cost is a monotonically increasing function
   When cost is added up along path, an operator’s cost .......?

3. Time?
   $\#$ of nodes with $g \leq$ cost of optimal solution, $O(b^{[C^*/\epsilon]})$
   where $C^*$ is the cost of the optimal solution

4. Space?
   $\#$ of nodes with $g \leq$ cost of optimal solution, $O(b^{[C^*/\epsilon]})$
Depth-first search (I)

→ Expands nodes at deepest level in tree
→ When dead-end, goes back to shallower levels
→ Implement: put successors at front of queue.. LIFO

→ Little memory: path and unexpanded nodes
For $b$: branching factor, $m$: maximum depth, space ........?

\[ \begin{array}{c}
\text{Depth-first search (I)} \\
\text{→ Expands nodes at deepest level in tree} \\
\text{→ When dead-end, goes back to shallower levels} \\
\text{→ Implement: put successors at front of queue.. LIFO} \\
\text{→ Little memory: path and unexpanded nodes} \\
\text{For } b: \text{ branching factor, } m: \text{ maximum depth, space ........?} \\
\end{array} \]
Depth-first search (2)
Depth-first search (3)

Time complexity:

- We may need to expand all paths, $O(b^m)$
- When there are many solutions, DFS may be quicker than BFS
- When $m$ is big, much larger than $d$, $\infty$ (deep, loops), .. troubles

→ Major drawback of DFS: going deep where there is no solution..

Properties:

1. Complete? Not in infinite spaces, complete in finite spaces
2. Optimal?
3. Time? $O(b^m)$  \hspace{1cm} Woow..
   - terrible if $m$ is much larger than $d$, but if solutions are dense,
   - may be much faster than breadth-first
4. Space? $O(bm)$, linear!  \hspace{1cm} Woow..
Depth-limited search (I)

\[\rightarrow\] DFS is going too deep, put a threshold on depth!
For instance, 20 cities on map for Romania, any node deeper
than 19 is cycling. Don’t expand deeper!

\[\rightarrow\] Implement: nodes at depth \( l \) have no successor

Properties:

1. Complete?
2. Optimal?
3. Time? (given \( l \) depth limit)
4. Space? (given \( l \) depth limit)

Problem: how to choose \( l \)?
Iterative-deepening search (I)

→ DLS with depth = 0
→ DLS with depth = 1
→ DLS with depth = 2
→ DLS with depth = 3...

Limit = 0

Limit = 1

Limit = 2

Limit = 3

→ Combines benefits of DFS and BFS
Iterative-deepening search (2)
Iterative-deepening search (3)

→ combines benefits of DFS and BFS

Properties:

1. Time? \((d + 1).b^0 + (d).b + (d - 1).b^2 + \ldots + 1.b^d = O(b^d)\)
2. Space? \(O(bd)\), like DFS
3. Complete? like BFS
4. Optimal? like BFS (if step cost = 1)
Iterative-deepening search (4)

→ Some nodes are expanded several times, wasteful?

\[
N(\text{BFS}) = b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - d)
\]

\[
N(\text{IDS}) = (d)b + (d - 1)b^2 + \ldots + (1)b^d
\]

Numerical comparison for \(b = 10\) and \(d = 5\):

\[
N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450
\]

\[
N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100
\]

→ IDS is preferred when search space is large and depth unknown
Bidirectional search (I)

→ Given initial state and the goal state, start search from both ends and meet in the middle

→ Assume same $b$ branching factor, $\exists$ solution at depth $d$, time:

$$O\left(2b^{d/2}\right) = O\left(b^{d/2}\right)$$

$b = 10, d = 6$, DFS = 1,111,111 nodes, BDS = 2,222 nodes!
Bidirectional search (2)

In practice:

- Need to define predecessor operators to search backwards
  If operator are invertible, no problem

- What if $\exists$ many goals (set state)?
  do as for multiple-state search

- need to check the 2 fringes to see how they match
  need to check whether any node in one space appears in the other space (use hashing)
  need to keep all nodes in a half in memory $O(b^{d/2})$

- What kind of search in each half space?
## Summary

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes*</td>
<td>Yes*</td>
<td>No</td>
<td>Yes, if ( l \geq d )</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>( b^{d+1} )</td>
<td>( b[^{C^*}/\epsilon] )</td>
<td>( b^m )</td>
<td>( b^l )</td>
<td>( b^d )</td>
</tr>
<tr>
<td>Space</td>
<td>( b^{d+1} )</td>
<td>( b[^{C^*}/\epsilon] )</td>
<td>( bm )</td>
<td>( bl )</td>
<td>( bd )</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes*</td>
<td>Yes*</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

\( b \) branching factor  
\( d \) solution depth  
\( m \) maximum depth of tree  
\( l \) depth limit
**Loops:** Avoid repeated states (I)

Avoid expanding states that have already been visited

Valid for both infinite and finite trees

\[
\begin{align*}
&\quad m \text{ maximum depth} \\
&\quad m + 1 \text{ states} \\
&\quad 2^m \text{ possible branches (paths)}
\end{align*}
\]

Example:

![Diagram showing loops and tree structure]
**Loops:** (2)

Keep nodes in two lists:
- Open list: Fringe
- Closed list: Leaf and expanded nodes

Discard a current node that matches a node in the closed list

Tree-Search $\rightarrow$ Graph-Search

**Issues:**

1. Implementation: hash table, access is constant time
   - Trade-off cost of storing + checking vs. cost of searching

2. Losing optimality
   - when new path is cheaper/shorter of the one stored

3. DFS and IDS now require exponential storage
Summary

**Path:** sequence of actions leading from one state to another

**Partial solution:** a path from an initial state to another state

**Search:** develop a sets of partial solutions

- Search tree & its components (node, root, leaves, fringe)
- Data structure for a search node
- Search space vs. state space
- Node expansion, queue order
- Search types: uninformed vs. heuristic
- 6 uninformed search strategies
- 4 criteria for evaluating & comparing search strategies