Navigating through a search tree

Unexpanded nodes: the fringe

Tree search

function TREE-SEARCH(problem, strategy) return a solution or failure
Initialize search tree to the initial state of the problem
repeat
  if no candidates for expansion then return failure
  choose leaf node for expansion according to strategy
  if node contains goal state then return solution
  else expand the node and add resulting nodes to the search tree
endrepeat

At every point in the search process we keep track of a list of nodes that haven’t been expanded yet: the fringe
function TREE-SEARCH(problem, strategy) return a solution or failure
Initialize search tree to the initial state of the problem
do
  if no candidates for expansion then return failure
  choose leaf node for expansion according to strategy
  if node contains goal state then return solution
  else expand the node and add resulting nodes to the search tree
enddo

What’s in a node
- State
- Parent
- Action (the action that got us from the parent)
- Depth
- Path-Cost

Tree search
function TREE-SEARCH(problem, fringe) return a solution or failure
fringe ← INSERT(MAKE-NODE(INITIAL-STATE(problem)), fringe)
loop do
  if EMPTY(fringe) then return failure
  node ← REMOVE-FIRST(fringe)
  if GOAL-TEST(problem) applied to STATE(node) succeeds then return SOLUTION(node)
  fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
enddo

Metrics for comparing search strategies
- A strategy is defined by the order of node expansion.
- Problem-solving performance is measured in four ways:
  - Completeness: Does it always find a solution if one exists?
  - Optimality: Does it always find the least-cost solution?
  - Time Complexity: Number of nodes generated/expanded.
  - Space Complexity: Number of nodes stored in memory during search.
- Time and space complexity are measured in terms of:
  - b - maximum branching factor of the search tree
  - d - depth of the least-cost solution
  - m - maximum depth of the state space (may be ∞)

Uninformed search strategies
- a.k.a. blind search = use only information available in problem definition.
- When strategies can determine whether one non-goal state is better than another → informed search.
- Search algorithms are defined by the node expansion method:
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search.
  - Bidirectional search

Breadth-First Search (BFS)
- Expand all nodes at depth d before proceeding to depth d+1.
- Implementation: queue (FIFO).
Evaluation of BFS

- Completeness:
  - Does it always find a solution if one exists?
  - YES (if shallowest goal node is at some finite depth d)

- Time complexity:
  - Assume a state space where every state has b successors.
  - Assume solution is at depth d
  - Worst case: expand all but the last node at depth d
  - Total number of nodes expanded:
    \[ b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = \mathcal{O}(b^{d+1}) \]

- Space complexity:
  - Same, if each node is retained in memory

BFS evaluation

- Memory requirements are a bigger problem than its execution time.
- Exponential complexity search problems cannot be solved by uninformed search methods for any but the smallest instances.

<table>
<thead>
<tr>
<th>DEPTH</th>
<th>NODES</th>
<th>TIME</th>
<th>MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1000</td>
<td>0.51 seconds</td>
<td>1 megabyte</td>
</tr>
<tr>
<td>4</td>
<td>111100</td>
<td>11 seconds</td>
<td>100 megabytes</td>
</tr>
<tr>
<td>6</td>
<td>$10^2$</td>
<td>19 minutes</td>
<td>20 gigabytes</td>
</tr>
<tr>
<td>8</td>
<td>$10^3$</td>
<td>35 hours</td>
<td>1 terabyte</td>
</tr>
<tr>
<td>10</td>
<td>$10^4$</td>
<td>129 days</td>
<td>10 terabytes</td>
</tr>
<tr>
<td>12</td>
<td>$10^5$</td>
<td>35 years</td>
<td>10 petabytes</td>
</tr>
<tr>
<td>14</td>
<td>$10^6$</td>
<td>3523 years</td>
<td>1 exabyte</td>
</tr>
</tbody>
</table>

Time and memory requirements for BFS for b=10, 10,000 nodes/sec; 1000 bytes/node

Uniform cost search

- Extension of BFS:
  - Expand node with lowest path cost
- Implementation: fringe = queue ordered by path cost.
- Same as BFS when all step-costs are equal.
Uniform cost search

- Completeness:
  - YES, if step-cost > ε (small positive constant)
- Time complexity:
  - Assume C* the cost of the optimal solution.
  - Assume that every action costs at least ε
  - Worst-case: $O(k^{C^*})$
- Space complexity:
  - Same as time complexity
- Optimality:
  - nodes expanded in order of increasing path cost.
  - YES, if complete.

Depth First Search (DFS)

Expand deepest unexpanded node
Depth First Search (DFS)

Expand deepest unexpanded node

Implementation: fringe is a stack (LIFO)

DFS evaluation

- Completeness:
  - NO unless search space is finite.
- Time complexity:
  - Terrible if \( m \) (depth of search space) is much larger than \( d \) (depth of optimal solution)
  - But if many solutions, then faster than BFS

- Space complexity: \( O(b^m) \)
  - Possible to use even less (expand one successor instead of all \( b \)).
DFS evaluation
- Completeness: NO unless search space is finite.
- Time complexity: $O(b^m)$
- Space complexity: $O(bm)$
- Optimality: No
  - Same issues as completeness

Depth-limited search
- DFS with depth limit $l$.
  - i.e. nodes at depth $l$ have no successors.
  - Problem knowledge can be used.
  - Solves the infinite-path problem.
  - If $l < d$ (depth of least cost solution) then incomplete
  - If $l > d$ then not optimal.
- Time complexity: $O(b^l)$
- Space complexity: $O(bl)$

Iterative Deepening Search (IDS)
- A strategy to find best depth limit $l$.
- Depth-Limited Search to depth 1, 2, ...
- Expands from the root each time.
- Appears very wasteful, but combinatorics can be counter intuitive:
  - $N(DLS) = b + b^2 + \ldots + b^{l+1} = O(b^l)$
  - $N(IDS) = db + (d-1)b^2 + \ldots + 2b^{l+1} + b^l = O(b^l)$
  - $N(BFS) = b + b^2 + \ldots + b^d + b^{d+1} = O(b^d)$
- Example: For $b = 10$ and $d = 5$
  - $N(DLS) = 111,111$
  - $N(IDS) = 123,456$
  - $N(BFS) = 1,111,100$.  

Iterative deepening search
- function iterative_deepening_search(problem) return a solution or failure
- inputs: problem
- for depth = 0 to $\infty$
  - result = DEPTH-LIMITED_SEARCH(problem, depth)
  - if result = cutoff then return result

Evaluation of IDS
- Completeness:
  - YES (no infinite paths)

Evaluation of IDS
- Completeness:
  - YES
- Time complexity: $O(b^l)$
Evaluation of IDS

- Completeness:
  - YES
- Time complexity: \(O(b^d)\)
- Space complexity: \(O(bd)\)
- Same as DFS

Optimality:
- YES if step cost is 1.

Iterative Deepening Search

- Analogous to BFS: explores a complete layer of nodes before proceeding to the next one.
- Combines benefits of DFS and BFS.

Bidirectional Search

- Two simultaneous searches from start and goal.
- Motivation: \(b^{d/2} + b^{d/2}\) much less than \(b^d\)
- Before a node is expanded it is checked if it is in the fringe of the other search (can be done in constant time using a hash table).
- Time complexity: \(O(b^{d/2})\).
- Space complexity: same.
- Complete and optimal (for uniform step costs) if both searches are BFS.

Comparison of search strategies

<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>Completes?</td>
<td>YES*</td>
<td>YES*</td>
<td>NO</td>
<td>YES, if (d \geqslant d)</td>
<td>YES</td>
</tr>
<tr>
<td>Time</td>
<td>(b^d)</td>
<td>(b^{d/2})</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/2})</td>
<td>(bd)</td>
<td>(bd)</td>
<td>(b^{d/2})</td>
</tr>
<tr>
<td>Optimality?</td>
<td>YES*</td>
<td>YES*</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>

Issues in applying:
- The predecessor of each node should be efficiently computable.
- When actions are easily reversible.
- Goal node: sometimes not known explicitly (e.g. in chess).
When the search graph is not a tree:

- Need to avoid repeated states!
- Happens in problems with reversible operators
- Examples: missionaries and cannibals problem, sliding blocks puzzles, route finding problems.
- Detection: compare a node to be expanded to those already expanded. Those are kept in the closed list.
- Increases memory requirements (especially for DFS): bounded by the size of the state space.

Graph Search

```
function GRAPH-SEARCH(problem, fringe) return a solution or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if EMPTY?(fringe) then return failure
    node ← REMOVE-FIRST(fringe)
    if GOAL-TEST[problem] applied to STATE[node] succeeds
      then return SOLUTION(node)
    if STATE[node] is not in closed
      then add STATE[node] to closed
      fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```