Uninformed Search

Russell and Norvig chap. 3

Following this, the pong paddle went on a mission to destroy Atari headquarters and, due to a mixup, found himself inside the game The Matrix Reloaded. Boy, was THAT ever hard to explain to him.
Navigating through a search tree
Navigating through a search tree
Navigating through a search tree
Navigating through a search tree
Navigating through a search tree
Navigating through a search tree
Navigating through a search tree
Navigating through a search tree
Navigating through a search tree
Navigating through a search tree
Navigating through a search tree
Navigating through a search tree
At every point in the search process we keep track of a list of nodes that haven’t been expanded yet: the frontier.
Tree search

function TREE-SEARCH(problem) return a solution or failure
  Initialize frontier using the initial state of problem
  do
    if the frontier is empty then return failure
    choose leaf node from the frontier
    if node is a goal state then return solution
    else expand the node and add resulting nodes to the frontier
function TREE-SEARCH(problem) return a solution or failure
   Initialize frontier using the initial state of problem
   do
      if the frontier is empty then return failure
      choose leaf node from the frontier
      if node is a goal state then return solution
      else expand the node and add resulting nodes to the frontier
What’s in a node

- **State**
- **Parent**
- **Action** (the action that got us from the parent)
- **Depth**
- **Path-cost** (total cost to get to the node)

Why do we need the parent and action information?
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.
- Increases memory requirements (especially for DFS): bounded by the size of the state space.
function GRAPH-SEARCH(problem) return a solution or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
loop do
  if the frontier is empty then return failure
  choose a node from the frontier
  if node is a goal state then return the corresponding solution
  add the node to the explored set
  expand the node, adding the resulting nodes to the frontier
  (only if not in the frontier or explored set)

Algorithms that forget their history are doomed to repeat it
Search strategies

function GRAPH-SEARCH(problem) return a solution or failure

initialize the frontier using the initial state of problem
initialize the explored set to be empty
loop do
    if the frontier is empty then return failure
    choose a node from the frontier
    if node is a goal state then return the corresponding solution
    add the node to the explored set
    expand the node, adding the resulting nodes to the frontier
        (only if not in the frontier or explored set)

Search strategies differ in how a node is chosen from the frontier
Show code for DFS/BFS tree/graph search
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
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 $\infty$)*
Breadth-First Search (BFS)

- Expand all nodes at depth $d$ before proceeding to depth $d+1$. Return the first goal node found.
- Implementation: queue (FIFO).
Evaluation of BFS

- Completeness:
  - Does it always find a solution if one exists?
Evaluation of BFS

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

- Completeness:
  - YES

- Time complexity:
  - Assume a state space where every state has $b$ successors.
    - Assume solution is at depth $d$
    - Worst case: generate all but the last node at depth $d$
  - Total number of nodes generated:

$$b + b^2 + b^3 + \ldots + b^d = O(b^d)$$
Evaluation of BFS

- Completeness:
  - YES

- Time complexity:
  - Total number of nodes generated:
    \[ b + b^2 + b^3 + \ldots + b^d = O(b^d) \]

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

- Completeness:
  - **YES**

- Time complexity:
  - Total number of nodes generated:
    
    \[ b + b^2 + b^3 + ... + b^d = O(b^d) \]

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

- Optimality:
  - *Does it always find the least-cost solution?*
    
    YES (if all actions have the same cost)
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>1100</td>
<td>0.11 seconds</td>
<td>1 megabyte</td>
</tr>
<tr>
<td>4</td>
<td>111100</td>
<td>11 seconds</td>
<td>106 megabytes</td>
</tr>
<tr>
<td>6</td>
<td>$10^7$</td>
<td>19 minutes</td>
<td>10 gigabytes</td>
</tr>
<tr>
<td>8</td>
<td>$10^9$</td>
<td>31 hours</td>
<td>1 terabyte</td>
</tr>
<tr>
<td>10</td>
<td>$10^{11}$</td>
<td>129 days</td>
<td>101 terabytes</td>
</tr>
<tr>
<td>12</td>
<td>$10^{13}$</td>
<td>35 years</td>
<td>10 petabytes</td>
</tr>
<tr>
<td>14</td>
<td>$10^{15}$</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* = priority queue ordered by path cost.

- Same as BFS when all step-costs are equal.

- What’s the relationship to Dijkstra’s algorithm?
Uniform cost search

- Completeness:
  - YES

- Time complexity:
  - Assume $C^*$ the cost of the optimal solution.
  - Assume that every action costs at least $\varepsilon$
  - Worst-case:
    $$O(b^{C^*/\varepsilon})$$

- Space complexity:
  - Same as time complexity

- Optimality:
  - YES (nodes expanded in order of increasing path cost)
Uniform cost search

function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
frontier ← a priority queue ordered by PATH-COST, with node as the only element
explored ← an empty set

loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier) /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
            add child.STATE to explored
            frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
            replace that frontier node with child
    end for
end loop

Note the use of a priority queue and the extra check in case a shorter path to a frontier state is discovered.
Depth First Search (DFS)

Expand deepest unexpanded node
Depth First Search (DFS)

Expand deepest unexpanded node
Depth First Search (DFS)

Expand deepest unexpanded node
Depth First Search (DFS)

Expand deepest unexpanded node
Depth First Search (DFS)

Expand deepest unexpanded node
Depth First Search (DFS)

Expand deepest unexpanded node
Depth First Search (DFS)

Expand
deepest
unexpanded
node
Depth First Search (DFS)

Expand deepest unexpanded node
Depth First Search (DFS)

Expand deepest unexpanded node

Implementation: frontier is a stack (LIFO)
DFS evaluation

- Completeness:
  - *Does it always find a solution if one exists?*
DFS evaluation

- Completeness:
  - *Does it always find a solution if one exists?*
    - NO
  - *unless search space is finite (also beware of loops if using tree search)*
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 might be faster than BFS
DFS evaluation

- Completeness:
  - NO (unless search space is finite).
- Time complexity: $O(b^m) - m$ is maximum depth of graph
- Space complexity: $O(bm)$
  - Assumes we are not keeping track of explored nodes
DFS evaluation

- Completeness:
  - NO (unless search space is finite).
- Time complexity: $O(b^m)$
- Space complexity: $O(bm)$
- Optimality: No
Depth-limited search

- DFS with depth limit $l$.
  - Treat nodes at depth $l$ as if they have no successors.
- Solves the infinite-path problem.
- If $l < d$ (depth of least cost solution) then incomplete.
- 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 $1, 2, \ldots$
- Expands from the root each time.
- Appears very wasteful, but combinatorics can be counter intuitive:

\[
N(\text{DLS}) = b + b^2 + \ldots + b^{d-1} + b^d = O(b^d)
\]
\[
N(\text{IDS}) = db + (d-1)b^2 + \ldots + 2b^{d-1} + b^d = O(b^d)
\]
\[
N(\text{BFS}) = b + b^2 + \ldots + b^d = O(b^d)
\]
Iterative deepening search

function ITERATIVE_DEEPENING_SEARCH(problem) return a solution or failure

for depth = 0 to ∞ do
    result ← DEPTH-LIMITED_SEARCH(problem, depth)
    if result ≠ cutoff then return result

Note: depth-limited_search returns cutoff when it has reached the given depth without finding a solution
Evaluation of IDS

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

- Completeness:
  - YES

- Time complexity: $O(b^d)$
Evaluation of IDS

- Completeness: YES
- Time complexity: $O(b^d)$
- Space complexity: $O(bd)$
  - Same as DFS
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.
- Need to keep at least one of the fringes in memory
- Time complexity: $O(b^{d/2})$
- Space complexity: same.
- Complete and optimal (for uniform step costs) if both searches are BFS
Bidirectional Search

Issues in applying:

- The predecessor of each node should be efficiently computable.
  - When actions are easily reversible.
- Goal node: sometimes not unique or known explicitly (e.g. in chess).
- Memory consumption
## Comparison of search strategies

<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 search</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>
<td>YES*</td>
</tr>
<tr>
<td>Time</td>
<td>(b^{d+1})</td>
<td>(b^{C*/c})</td>
<td>(b^m)</td>
<td>(b^l)</td>
<td>(b^d)</td>
<td>(b^{d/2})</td>
</tr>
<tr>
<td>Space</td>
<td>(b^{d+1})</td>
<td>(b^{C*/c})</td>
<td>(b^m)</td>
<td>(b^l)</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>
<td>YES</td>
</tr>
</tbody>
</table>