Search Strategies

Reading: Russell’s Chapter 3
Search strategies

A search strategy is defined by picking the **order of node expansion**

Strategies are evaluated along the following dimensions:

- **completeness**: does it always find a solution if one exists?
- **time complexity**: number of nodes generated/expanded
- **space complexity**: maximum number of nodes in memory
- **optimality**: does it always find a least-cost solution?

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 $\infty$)
Uninformed search strategies use only the information available in the problem definition

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search
Breadth-first search

Expand shallowest unexpanded node

Implementation:
– *fringe* is a FIFO queue, i.e., new successors go at end
Breadth-first search

Expand shallowest unexpanded node

Implementation:

– fringe is a FIFO queue, i.e., new successors go at end
Breadth-first search

Expand shallowest unexpanded node

Implementation:

- *fringe* is a FIFO queue, i.e., new successors go at end
Breadth-first search

Expand shallowest unexpanded node

**Implementation:**

– *fringe* is a FIFO queue, i.e., new successors go at end
Properties of breadth-first search

Complete? Yes (if $b$ is finite)

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

Space? $O(b^{d+1})$ (keeps every node in memory)

Optimal? Yes (if cost = 1 per step)

Space is the bigger problem (more than time)
BFS; evaluation

Two lessons:

– 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>

Table is based on: \(b = 10; 10000/\text{second}; 1000 \text{ bytes/node}\)
Uniform-cost search

Extension of BFS:
- Expand node with lowest path cost

Implementation: fringe = queue ordered by path cost.

UCS is the same as BFS when all step-costs are equal.
Uniform-cost search

Completeness:
- YES, if step-cost > \( \varepsilon \) (small positive constant)

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:
- The same as time complexity

Optimality:
- nodes expanded in order of increasing path cost.
- YES, if complete.
Depth-first search

Expand deepest unexpanded node

Implementation:
– fringe = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:
- fringe = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:

– *fringe* = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:
– \textit{fringe} = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:

– fringe = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:
– fringe = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:
- fringe = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:
- *fringe* = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:
- \textit{fringe} = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:
– *fringe* = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

**Implementation:**
- `fringe` = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:
- fringe = LIFO queue, i.e., put successors at front
Properties of depth-first search

**Complete?** No: fails in infinite-depth spaces, spaces with loops

- Modify to avoid repeated states along path
  → complete in finite spaces

**Time?** $O(b^m)$: terrible if $m$ is much larger than $d$

- but if solutions are dense, may be much faster than breadth-first

**Space?** $O(bm)$, i.e., linear space!

**Optimal?** No
Depth-limited search

= depth-first search with depth limit /
i.e., nodes at depth / have no successors

Recursive implementation:

```python
def Depth-Limited-Search(problem, limit)
    return Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)

def Recursive-DLS(node, problem, limit)
    cutoff-occurred? ← false
    if Goal-Test[problem](State[node]) then return Solution(node)
    else if Depth[node] = limit then return cutoff
    else for each successor in Expand(node, problem) do
        result ← Recursive-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    end for
    if cutoff-occurred? then return cutoff else return failure
```
Depth-limited search

Is 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 \( l < d \) then incompleteness results.

If \( l > d \) then not optimal.

Time complexity: \( O(b^l) \)

Space complexity: \( O(bl) \)
Iterative deepening search

```plaintext
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  inputs: problem, a problem
  for depth ← 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
```
Iterative deepening search

Figure 3.16  Four iterations of iterative deepening search on a binary tree.
Properties of iterative deepening search

**Complete?** Yes

**Time?** \( (d+1)b^0 + d b^1 + (d-1)b^2 + \ldots + b^d = O(b^d) \)

**Space?** \( O(bd) \)

**Optimal?** Yes, if step cost = 1
ID search, evaluation

Completeness:
- YES (no infinite paths)

Time complexity: $O(b^d)$
- Algorithm seems costly due to repeated generation of certain states.
- Node generation:
  - level $d$: once
  - level $d-1$: 2
  - level $d-2$: 3
  - ...
  - level 2: $d-1$
  - level 1: $d$

$$N(IDS) = (d)b + (d-1)b^2 + ... + (1)b^d$$
$$N(BFS) = b + b^2 + ... + b^d + (b^{d+1} - b)$$

Num. Comparison for $b=10$ and $d=5$ solution at far right
$$N(IDS) = 50 + 400 + 3000 + 20000 + 100000 = 123450$$
$$N(BFS) = 10 + 100 + 1000 + 10000 + 100000 + 999990 = 1111100$$
# Summary of algorithms

<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>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Time O(b^{d+1})</td>
<td>O(b^{C*/\epsilon})</td>
<td>O(b^m)</td>
<td>O(b^l)</td>
<td>O(b^d)</td>
<td></td>
</tr>
<tr>
<td>Space O(b^{d+1})</td>
<td>O(b^{C*/\epsilon})</td>
<td>O(bm)</td>
<td>O(bl)</td>
<td>O(bd)</td>
<td></td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Repeated states

Failure to detect repeated states can turn a linear problem into an exponential one!
Graph search

function GRAPH-SEARCH( problem, fringe) returns a solution, or failure

    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)

loop do

    if fringe is empty then return failure

    node ← REMOVE-FRONT(fringe)

    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)

    if STATE[node] is not in closed then

        add STATE[node] to closed

        fringe ← INSERTALL(EXPAND(node, problem), fringe)

    end if

end loop

Bidirectional search

Two simultaneous searches from start and goal.

- Motivation:
  \[ b^{d/2} + b^{d/2} \neq b^d \]

Check whether the node belongs to the other fringe before expansion. Space complexity is the most significant weakness. Complete and optimal if both searches are BFS.
How to search backwards?

The predecessor of each node should be efficiently computable.

- When actions are easily reversible.