Uninformed Search Strategies

AIMA 3.3-3.4
Review: Formulating search problems

- **Formulate search problem**
  - **States**: configurations of the puzzle (9! configurations)
  - **Actions**: Move one of the movable pieces (≤4 possible)
  - **Performance measure**: minimize total moves

- **Formulate goal**
  - Pieces to end up in order as shown…

- **Find solution**
  - Sequence of pieces moved: 3,1,6,3,1,…
Review: a problem is defined by:

1. **States**: a set $S$
2. An *initial state* $s_i \in S$
3. **Actions**: a set $A$
   — $\forall s \text{ Actions}(s) = \text{the set of actions that can be executed in } s$, that are *applicable* in $s$.
4. **Transition Model**: $\forall s \forall a \in \text{Actions}(s) \text{ Result}(s, a) \rightarrow s_r$
   — $s_r$ is called a *successor* of $s$
   — $\{s_i\} \cup \text{Successors}(s_i)^* = \text{state space}$
5. **Path cost (Performance Measure)**: Must be additive
   — e.g. sum of distances, number of actions executed, ... 
   — $c(x,a,y)$ is the step cost, assumed $\geq 0$
     — (where action $a$ goes from state $x$ to state $y$)
6. **Goal test**: $\text{Goal}(s)$
   — Can be implicit, e.g. *checkmate*(s)
   — $s$ is a *goal state* if $\text{Goal}(s)$ is *true*
Review: Solutions & Optimal Solutions

• A *solution* is a sequence of *actions* from the *initial state* to a *goal state*.

• *Optimal Solution*: A solution is *optimal* if no solution has a lower *path cost*. 
Outline for today’s lecture

• **Search Fundamentals (AIMA 3.3)**

• **Introduction to Uninformed Search**
  • Review of Breadth first and Depth-first search

• **Iterative deepening search**
  • Strange Subroutine: Depth-limited search
  • Depth-limited search + iteration = WIN!!
Useful Concepts

- **State space**: the set of all states reachable from the initial state by *any* sequence of actions
  - *When several operators can apply to each state, this gets large very quickly*
  - *Might be a proper subset of the set of configurations*
- **Path**: a sequence of actions leading from one state \( s_j \) to another state \( s_k \)
- **Frontier**: those states that are available for *expanding* (for applying legal actions to)
- **Solution**: a path from the initial state \( s_i \) to a state \( s_f \) that satisfies the goal test
Basic search algorithms: *Tree Search*

- Generalized algorithm to solve search problems (Review from Data Structures)
  - *Enumerate in some order all possible paths from the initial state*
  - Here: search through *explicit tree generation*
    - ROOT= initial state.
    - Nodes in search tree generated through *transition model*
function TREE-SEARCH(*problem, strategy*) return a solution or failure

Initialize frontier to the initial state of the *problem*

do

if the frontier is empty then return *failure*
choose leaf node n for expansion according to strategy & remove from frontier
if n contains goal state then return *solution*
else EXPAND n and add resulting nodes to the frontier
8-Puzzle: States and Nodes

- A **state** is a (representation of a) **physical configuration**
- A **node** is a data structure constituting **part of a search tree**
  - Here **node** = `<state, parent-node, children, action, path-cost, depth>`
- States do not have parents, children, depth or path cost!
- The **EXPAND** function
  - uses the Actions and Transition Model to create the corresponding states
    — creates new nodes from these states
    — fills in the various fields of the new node data structure
8-Puzzle **Search Tree**

- (Nodes show state, parent, children - leaving *Action, Cost, Depth* Implicit)

(Suppressing useless “backwards” moves)
Tree Search Flaw: Repeated states

- Failure to detect *repeated states* can turn a linear problem into an *exponential* one!
Solution: Graph Search!

- **Simple Mod to tree search:** Check to see if a node has been visited before adding to search queue
  - must keep track of all possible states
    - can use a lot of memory
  - e.g., 8-puzzle problem, we have $9!/2 \approx 182K$ states

State Space

Search Tree
Graph Search vs Tree Search

function TREE-SEARCH(problem) returns a solution, or failure
initialize the frontier using the initial state of problem
loop do
  if the frontier is empty then return failure
  choose a leaf node and remove it from the frontier
  if the node contains a goal state then return the corresponding solution
  expand the chosen node, adding the resulting nodes to the frontier

function GRAPH-SEARCH(problem) returns 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 leaf node and remove it from the frontier
  if the node contains a goal state then return the corresponding solution
  add the node to the explored set
  expand the chosen node, adding the resulting nodes to the frontier
  only if not in the frontier or explored set

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.
Outline for today’s lecture

- Search Fundamentals

- Introduction to Uninformed Search (AIMA 3.4.1-3)
  - Review of Breadth first and Depth-first search

- Iterative deepening search
  - Strange Subroutine: Depth-limited search
  - Depth-limited search + iteration = WIN!!
Uninformed search strategies:

- AKA “Blind search”
- Uses only information available in problem definition

Informally:

- **Uninformed search**: All non-goal nodes in frontier look equally good
- **Informed search**: Some non-goal nodes can be ranked above others.
Search Strategies

- **Review: Strategy** = order of tree expansion
  - Implemented by different queue structures (LIFO, FIFO, priority)

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

- **Time/space complexity variables**
  - \(b\), \(maximum\ \text{branching factor}\) of search tree
  - \(d\), \(depth\) of the shallowest goal node
  - \(m\), maximum length of any path in the state space (potentially \(\infty\))
Introduction to space complexity

- You know about:
  - “Big O” notation
  - *Time complexity*

- *Space complexity* is analogous to time complexity

- Units of space are arbitrary
  - Doesn’t matter because Big O notation ignores constant multiplicative factors
  - Plausible Space units:
    - One Memory word
    - Size of any fixed size data structure
      - eg Size of fixed size node in search tree
Review: Breadth-first search

• **Idea:**
  • Expand *shallowest* unexpanded node

• **Implementation:**
  • *frontier* is FIFO (First-In-First-Out) Queue:
    —Put successors at the *end* of *frontier* successor list.
Breadth-first search (simplified)

function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node <- a node with STATE = problem.INITIAL-STATE, PATH-COST=0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier <- a FIFO queue with node as the only element
    loop do
        if EMPTY?(frontier) then return failure
        node <- POP(frontier) \(\parallel\) chooses the shallowest node in frontier
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child <- CHILD-NODE(problem, node, action)
            if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
            frontier <- INSERT(child, frontier)

From Figure 3.11 Breadth-first search (ignores loops, repeated nodes)
Properties of breadth-first search

- **Complete?** Yes (if $b$ is finite)
- **Time Complexity?** $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$
- **Space Complexity?** $O(b^d)$ (keeps every node in memory)
- **Optimal?** Yes, if cost = 1 per step
  (not optimal in general)

$b$: maximum branching factor of search tree
$d$: depth of the least cost solution
$m$: maximum depth of the state space ($\infty$)
Exponential Space (and time) Not Good...

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

<table>
<thead>
<tr>
<th>DEPTH</th>
<th>NODES</th>
<th>TIME</th>
<th>MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>110</td>
<td>0.11 milliseconds</td>
<td>107 kilobytes</td>
</tr>
<tr>
<td>4</td>
<td>11110</td>
<td>11 milliseconds</td>
<td>10.6 megabytes</td>
</tr>
<tr>
<td>6</td>
<td>$10^6$</td>
<td>1.1 seconds</td>
<td>1 gigabyte</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>2 minutes</td>
<td>103 gigabytes</td>
</tr>
<tr>
<td>10</td>
<td>$10^{10}$</td>
<td>3 hours</td>
<td>10 terabytes</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>13 days</td>
<td>1 petabyte</td>
</tr>
<tr>
<td>14</td>
<td>$10^{14}$</td>
<td>3.5 years</td>
<td>99 petabytes</td>
</tr>
</tbody>
</table>

Fig 3.13 Assumes b=10, 1M nodes/sec, 1000 bytes/node
Review: Depth-first search

• **Idea:**
  • Expand *deepest* unexpanded node

• **Implementation:**
  • *frontier* is LIFO (Last-In-First-Out) Queue:
    —Put successors at the *front* of *frontier* successor list.
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?** \(O(b^m)\), i.e., linear space!
- **Optimal?** No

\(b\): maximum branching factor of search tree
\(d\): depth of the least cost solution
\(m\): maximum depth of the state space (\(\infty\))
Depth-first vs Breadth-first

- **Use depth-first if**
  - *Space is restricted*
  - There are many possible solutions with long paths and wrong paths are usually terminated quickly
  - Search can be fine-tuned quickly

- **Use breadth-first if**
  - *Possible infinite paths*
  - Some solutions have short paths
  - Can quickly discard unlikely paths
Outline for today’s lecture

- Formulating Search Problems – An Example

- Search Fundamentals

- Introduction to Uninformed Search
  - Review of Breadth first and Depth-first search

- Iterative deepening search (AIMA 3.4.4-5)
  - Strange Subroutine: Depth-limited search
  - Depth-limited search + iteration = WIN!!
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* $l$.
  - i.e. nodes at depth $l$ have no successors.
  - No infinite-path problem!

- If $l = d$ (by luck!), then optimal
  - But:
    - If $l < d$ then incomplete 😞
    - If $l > d$ then not optimal 😞

- Time complexity: $O(b^l)$
- Space complexity: $O(bl)$ 😊
Iterative deepening search

- A general strategy to find best depth limit $l$.
  - Key idea: use *Depth-limited search* as subroutine, with increasing $l$.

  For $l = 0$ to $\infty$ do
  
  depth-limited-search to level $l$
  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
  • \((d)b + (d-1)b^2 + \ldots + (1)b^d = O(b^d)\)

• As efficient in terms of \(O(\ldots)\) as Breadth First Search:
  • \(b + b^2 + \ldots + b^d = O(b^d)\)
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>$b^d$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$b^d$</td>
</tr>
<tr>
<td>Space</td>
<td>$b^d$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$b^d$</td>
</tr>
<tr>
<td>Optimal?</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>