Search Fundamentals & Uninformed Search

AIMA 3.2-4
Outline for today’s lecture

• Formulating Search Problems (AIMA 3.2)
• Search Fundamentals (AIMA 3.3)
• Introduction to Uninformed Search (AIMA 3.4)
  • Review of Breadth first and Depth-first search

• Next Time
  • (Depth-limited search)
  • Iterative deepening search
  • Uniform-cost search
  • Bidirectional search
Review: 8-puzzle search problem

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

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

- **Find solution**
  - Sequence of pieces moved: 3,1,6,3,1,…
Review: Holiday in Romania

- Currently *in Arad*
  - Flight leaves tomorrow *from Bucharest*
- Formulate *goal*
  - Be in Bucharest
- Formulate *search problem*
  - States: various cities
  - Actions: drive between cities
  - Performance measure: minimize distance
- Find *solution*
  - Sequence of cities; e.g. Arad, Sibiu, Fagaras, Bucharest, …
Review: a problem is defined by:

1. A set of states $S$
2. An initial state $s_i \in S$
3. A set of actions $A$
   - $\forall s, \text{Actions}(s) =$ 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. Goal test $\text{Goal}(s)$
   - Can be implicit, e.g. $\text{checkmate}(x)$
   - $s$ is a goal state if $\text{Goal}(s)$ is true
6. Path cost (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 $y$)
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.
Hard subtask: Selecting a state space

- Real world is absurdly complex
  State space must be *abstracted* for problem solving

- (abstract) **State** = set (equivalence class) of real world states

- (abstract) **Action** = complex combination of real world actions
  - e.g. *Arad* → *Zerind* represents a complex set of possible routes, detours, rest stops, etc
  - The abstraction is valid if the path between two states is reflected in the real world

- (abstract) **Solution** = set of abstract paths that are solutions in the abstract space

- Each abstract action should be “easier” than the real problem
Formulating a Search Problem

Decide:

- Which properties matter & how to represent
  - Initial State, Goal State, Possible Intermediate States
- Which actions are possible & how to represent
  - Operator Set: Actions and Transition Model
- Which action is next
  - Path Cost Function
Example: 8-puzzle

- States??
- Initial state??
- Actions??
- Transition Model??
- Goal test??
- Path cost??
Example: 8-puzzle

- States?? List of 9 locations- e.g., [7,2,4,5,-,6,8,3,1]
- Initial state?? [7,2,4,5,-,6,8,3,1]
- Actions?? \{Left, Right, Up, Down\}
- Transition Model?? ...
- Goal test?? Check if goal configuration is reached
- Path cost?? Number of actions to reach goal
Example: Missionaries & Cannibals

Three missionaries and three cannibals come to a river. A rowboat that seats two is available. If the cannibals ever outnumber the missionaries on either bank of the river, the missionaries will be eaten.  (problem 3.9)

How shall they cross the river?
Formulation: Missionaries & Cannibals

- **How to formalize:**
  - **Initial state:** all M, all C, and boat on one bank
  - **Actions:** ??
  - **Transition Model??**
  - **Goal test:** True if all M, all C, and boat on other bank
  - **Cost:** ??

**Remember:**

- **Representation:**
  - **States:** Which properties matter & how to represent
  - **Actions & Transition Model:** Which actions are possible & how to represent
  - **Path Cost:** Deciding which action is next
Search Fundamentals

AIMA 3.3
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.*

- **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 UG Data Structures)
  - Enumerate in some order all possible paths from the initial state
  - Here: search through explicit tree generation
    - ROOT = initial state.
    - Nodes and leafs generated through transition model
  - In general search generates a graph (same state through multiple paths), but we’ll just look at trees in lecture
    - Treats different paths to the same node as distinct
  - (Extension to graphs: just check to see if any node has been visited before adding to search queue)
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 for expansion according to strategy & remove from frontier
  if node contains goal state then return solution
  else expand the node and add resulting nodes to the frontier

Determines search process!!
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*
  - Also includes *parent, children, depth, path cost* \( g(x) \)
  - Here **node** = \(<\text{state, parent-node, action, path-cost, depth}>\)
- 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,
    - fills in the various fields
8-Puzzle Search Tree

- (Leaving Action, Cost, Depth Implicit)

- Suppressing useless “backwards” moves
Problem: Repeated states

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

- **Graph search**
  - Optimal but memory inefficient
  - check to see if any node has been visited before adding to search queue
    - must keep track of all possible states (uses a lot of memory)
    - e.g., 8-puzzle problem, we have $9!/2 \approx 182K$ states
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
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

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.
Search 1: Uninformed Search Strategies

AIMA 3.4
Search Strategies

- **Strategy** = order of expansion

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

- **Time/space complexity measurements**
  - \( b \), *maximum 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
  - Space units:
    — One Memory word
    — Size of fixed Data structure
Uninformed search strategies

- (a.k.a. Blind Search) = use only information available in problem definition.

- Informed search.
  - When strategies can determine whether one non-goal state is better than another

- Search categories defined by expansion strategy:
  - Review: Breadth-first search
  - Review: Depth-first search
  - (Depth-limited search) → Iterative deepening search
  - Uniform-cost search

- Bidirectional search
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*) // 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*)

*Position within queue of new items determines search strategy*

*Subtle: Node inserted into queue only after testing to see if it is a goal state*

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

CIS 521 - 2015
Properties of breadth-first search

- Complete?
- Time Complexity?
- Space Complexity?
- Optimal?
Properties of breadth-first search

- **Complete?** Yes (if \( b \) is finite)
- **Time?** \( 1 + b + b^2 + b^3 + \ldots + b^d = O(b^d) \)
- **Space?** \( O(b^d) \) (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 search problems cannot be solved by uninformed search methods 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
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.
Uninformed search strategies

- Categories defined by expansion strategy:
  - Breadth-first search: FIFO queue
  - Depth-first search: LIFO queue
  - Uniform-cost search: Priority queue

- (Depth-limited search) → Iterative deepening search
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 can be detected quickly
  - Search can be fine-tuned quickly

- **Use breadth-first if**
  - Possible infinite paths
  - Some solutions have short paths
  - Can quickly discard unlikely paths