Lecture 2
Uninformed Search

CS 520: Intro AI
Jingjin Yu | Rutgers
Solving Problems through Search: An Example

Search for a route in Romania
Components of a Search Problem

- **State space** $S$: in this case, an edge-weighted graph
- **Initial** (start) and **goal** (final) states: $x_I$ and $x_G$
  - There can be more than one start/goal state: solve one side of a Rubik’s cube
- **Action**: in this case, moving from one state to a nearby state
- **Transition model**: tuples $(s_1, a, s_2)$ that are valid
  - Sometimes written as $T(s_1, a) = s_2$
  - There are usually costs/rewards associated with a transition, $R(s_1, a)$
- **Solution**: valid transitions connecting $x_I$ and $x_G$
  - Optimal solution: solution with lowest cost (e.g., length of the path)
Example: Vacuum-Cleaner World

State space: \( \{A, B\} \times \{A_{\text{dirty}}, A_{\text{clean}}\} \times \{B_{\text{dirty}}, B_{\text{clean}}\} \)

State space size: \( 2 \times 2 \times 2 = 8 \)

Action: \( \{\text{left, right, suck}\} \)

Transition example: \( (A, A_{\text{clean}}, B_{\text{dirty}}) \xrightarrow{\text{right}} (B, A_{\text{clean}}, B_{\text{dirty}}) \)

Initial state: can be an arbitrary state

Goal states: \( \{A, B\} \times \{A_{\text{clean}}\} \times \{B_{\text{clean}}\} \)

Cost: can be the number of actions in a solution
State Space of the Vacuum-Cleaner World
Example: 8-Queens

⇒ State space: possible locations of 8 queens
  ⇒ State space size: \( C(64, 8) = \frac{64 \times 63 \times \cdots \times 57}{8!} \approx 4 \times 10^9 \)

⇒ Action/transition: place or move a queen

⇒ Initial state: can be an arbitrary state

⇒ Goal states: a placement of queens in which no two queens attacking

⇒ Cost: no clear cost
Example: 8-puzzle

- State space: arrangements of the 8 pieces
  - State space size: $9! = 362880$
- Action/transition: shifting a piece to the empty cell
- Initial state: an arbitrary state
- Goal state: an arbitrary fixed state
- Cost: number of actions (moves)
Applications of Search Algorithms

- Navigation
- Robot Motion planning
- Competitive Chess
- Game AI
Uninformed Search v.s. Informed Search

- Uninformed search only uses information from problem definition
- Example of uninformed and informed search: maze
  - Note that the distinction is a bit vague...

Searching for a hole with unknown location in a maze
Searching for an exit at south
Search Algorithms: An Illustration

Search algorithms generally builds a search tree.
State in State Space ≠ Search State!

 dụ A state in state space: where an agent is at a certain time

 dụ A search state: represents a path on the search tree from the root, which is a valid sequence of actions from $x_i$
Tree Search: Structure

input: $S$, $x_i$, $G(·)$

AddToQueue($x_i, Queue$);  // Add $x_i$ to a queue of nodes to be expanded

while(!IsEmpty($Queue$))

    $x$ ← Front($Queue$);  // Retrieve the front of the queue

    if($G(x)$) return solution;  // Return if goal is reached

    for each successor $n_i$ of $x$  // Add all neighbors to the queue
        AddToQueue($n_i, Queue$)

return failure;  // Expanding a search node

⇒ Different search algorithms use different AddToQueue

⇒ BFS: FIFO queue – first in first out
⇒ DFS: LIFO queue – last in first out
⇒ Uniform-cost: Priority queue – node with smallest cost in the front

⇒ Tree search CANNOT tell whether a node has been seen before
A Word on Implementation Details

⇒ The “algorithm” covered in the class may not be the most efficient.
  ⇒ For example, DFS algorithms only needs space of $O(m)$ in stead of $O(bm)$.
  ⇒ Our DFS uses $O(bm)$ because we add all successors before working on the next node.
  ⇒ In a standard DFS algorithm, only one successor of a node is added to the queue.
  ⇒ Our purpose here is to provide a general framework that covers all these different uninformed (and later informed) tree/graph search methods.
Search Algorithms: Important Properties

- **Completeness**: a complete algorithm returns a correct answer when there is one
- **Optimality**: whether the solution is the best possible according to some cost function
- **Time complexity**: the number of basic operations incurred by the algorithm
- **Space complexity**: the amount of memory needed to run the algorithm
  - In our case, the maximum size of the queue
Breadth-First Search (BFS)

AddToQueue  adds newly discovered node at the end of queue

[Diagram of tree traversal with queue updates]

Completeness  Yes, for finite branching factor $b$
Optimality     Optimal when edges have uniform length
Time complexity $O(b^d)$  $b$: branching factor, $d$: max solution depth
Space complexity $O(b^d)$
Uniform-Cost Search (UC)

- `AddToQueue` orders nodes by cost

Completeness: Yes, for finite branching factor
Optimality: Yes
Time complexity: $O(b^{1+C^*/\epsilon})$
Space complexity: $O(b^{1+C^*/\epsilon})$

Can be worse than BFS
Depth-First Search (DFS)

⇒ AddToQueue inserts in the front of the queue

Completeness: No
Optimality: No
Time complexity: $O(b^m)$ with $m$ being the maximum depth
Space complexity: $O(bm)$, makes DFS more attractive than BFS
Why Tree Search can be Bad

Assume:
The first neighbor of A is B
The first neighbor of C is also B

DFS search tree generated by tree search
Non-Optimality of DFS

$x_I = A, x_G = B$

DFS search tree
Depth-Limited Search

Disallow for handling the case when there are infinite depth paths
Disallow same as DFS but limit the maximum depth to some $\ell < m$

<table>
<thead>
<tr>
<th>Completeness</th>
<th>No if $\ell &lt; d$, otherwise still no (same as DFS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimality</td>
<td>No – Same as DFS</td>
</tr>
<tr>
<td>Time complexity</td>
<td>$O(b^\ell)$</td>
</tr>
<tr>
<td>Space complexity</td>
<td>$O(b\ell)$</td>
</tr>
</tbody>
</table>
Iterative Deepening Depth-First Search

Choosing the correct $\ell$ can be tricky – start with 0, 1, ...

Completeness: Yes, for finite branching factor

Optimality: Yes, for unit edge costs

Time complexity: $O(b^d)$

Space complexity: $O(bd)$
Bidirectional Search

- Run in parallel two BFS searches, one from start and one from goal
- Stop when the two trees meet

Completeness: Same as BFS
Optimality: Same as BFS
Time complexity: $O(b^{d/2})$
Space complexity: $O(b^{d/2})$
Graph Search

input: \( S, x_i, G(\cdot) \)

AddToQueue \((x_i, \text{Queue})\); // Add \( x_i \) to a queue of nodes to be expanded

while(!IsEmpty(\text{Queue}))
    \( x \leftarrow \text{Front(\text{Queue})}; \) // Retrieve the front of the queue
    \( \text{if}(x.\text{expanded} == \text{true}) \text{ continue;} \) // Do not expand a node twice
    \( \text{if}(G(x)) \text{ return solution;} \) // Return if goal is reached
    \( x.\text{expanded} = \text{true}; \) // Mark \( x \) as expanded
    for each successor \( n_i \) of \( x \) // Add all neighbors of \( x \) to the queue
        \( \text{if}(n_i.\text{expanded} == \text{false}) \text{ AddToQueue}(n_i, \text{Queue}) \)

return failure;

\( \Rightarrow \) Keeps track of expanded nodes – each node is expanded once

\( \Rightarrow \) AddToQueue can be more efficient

\( \Rightarrow \) Recall that tree search cannot tell nodes apart but graph search can

\( \Rightarrow \) For priority queue, only a single copy of \( n_i \) with smallest cost needs to be in the queue
Tree Search versus Graph Search

- Both tree and graph search build a **search tree**
  - Graph search’s search tree is part of the state space graph
  - The queue forms a **frontier** on the graph, separating the explored nodes from nodes that are not explored

- Graph search expands the same state once – graph search DFS is complete on finite graph

- If one cannot distinguish the states, then graph search cannot be applied – use tree search in this case

- Another reason for using tree search is space – graph search needs extra memory for tracking expanded nodes
BFS Tree Search and Graph Search Example

Problem graph

Running BFS tree and graph search

Tree search

Graph search

More compact search tree!

Adjacency list
S: A, B
A: B, C, G
B: C
C: G
G:
### Summary on Uninformed (Tree) Search

<table>
<thead>
<tr>
<th>Method</th>
<th>Complete?</th>
<th>Optimal?</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>Yes, for ( b &lt; \infty )</td>
<td>Yes, for uniform edge costs</td>
<td>( O(b^d) )</td>
<td>( O(b^d) )</td>
</tr>
<tr>
<td>Uniform cost</td>
<td>Yes, for ( b &lt; \infty ) and ( \epsilon &gt; \text{fixed } \delta &gt; 0 )</td>
<td>Yes</td>
<td>( O(b^{1+\frac{C^*}{\epsilon}}) )</td>
<td>( O(b^{1+\frac{C^*}{\epsilon}}) )</td>
</tr>
<tr>
<td>DFS</td>
<td>No</td>
<td>No</td>
<td>( O(b^m) )</td>
<td>( O(bm) )</td>
</tr>
<tr>
<td>Depth limited DFS</td>
<td>No</td>
<td>No</td>
<td>( O(b^\ell) )</td>
<td>( O(b^\ell) )</td>
</tr>
<tr>
<td>Iterative deepening DFS</td>
<td>Yes, for ( b &lt; \infty )</td>
<td>Yes, for uniform edge costs</td>
<td>( O(b^d) )</td>
<td>( O(bd) )</td>
</tr>
<tr>
<td>Bidirectional</td>
<td>Yes, for BFS with ( b &lt; \infty )</td>
<td>Yes, for uniform edge costs</td>
<td>( O(b^{\frac{d}{2}}) )</td>
<td>( O(b^{\frac{d}{2}}) )</td>
</tr>
</tbody>
</table>

- \( b \): branching factor (average)
- \( d \): max. solution depth (i.e., depth of \( x_G \))
- \( m \): max. depth of any node from \( x_I \)
- \( \ell \): depth limit
- \( C^* \): optimal solution cost
- \( \epsilon \): smallest step (edge) cost, must be finite

⇒ For graph search on finite state space, DFS is complete and depth limited DFS is complete when \( \ell \geq d \)
Search from AI and Algorithms Perspectives

⇒ AI perspective
  ⇒ The state space may be infinite
  ⇒ Often the state space is not explicitly given
  ⇒ Does not assume \( x_G \) is always reachable
    ⇒ That is why “goal detector” is provided
  ⇒ There may not even be an \( x_G \)
  ⇒ Therefore, we use branching factor and search depth to measure time and space complexity

⇒ Algorithms perspective
  ⇒ The state space (a graph) is often finite
    ⇒ Otherwise it is not possible to provide running time analysis
  ⇒ State space (graph) is often explicitly given though not always
  ⇒ Goal is usually given as a known state
  ⇒ Therefore, generally does not discuss “tree search”
Use the Romania road map, pick a random start/goal pair and carry out the various search strategies by hand (draw the search tree)

*Read CLRS algorithm book’s coverage on search algorithms