Outline of today’s lecture

1. Initial state, goal state, transition model

2. General algorithm for solving search problems
   1. First data structure: a frontier queue
   2. Second data structure: a search tree
   3. Third data structure: explored set
   4. Fourth data structure: explored dict

3. Breadth-first search (BFS) and Depth-first search (DFS)
   1. Completeness
   2. Optimality
   3. Time Complexity
   4. Space Complexity
Search

• We will consider the problem of designing goal-based agents in fully observable, deterministic, discrete, static, known environments

• Environment is sequential: agent’s action changes its state

• Agent must plan the best sequence of actions to achieve a goal
Search problem components

- **Initial state**
- **Actions**
- **Transition model**
  - What *successor state* results from performing a given *action* in a given *predecessor state*?
- **Goal state**
- **Path cost**
  - Assume that it is a sum of nonnegative *step costs*

- The **optimal solution** is the sequence of actions that gives the *lowest* path cost for reaching the goal
Knowledge Representation: State

• State = description of the world
  • Must have enough detail to decide whether or not you’re currently in the initial state
  • Must have enough detail to decide whether or not you’ve reached the goal state
  • Often but not always: “defining the state” and “defining the transition model” are the same thing
Example of state definition: Romania

• On vacation in Romania; currently in Arad
• Flight leaves tomorrow from Bucharest

• **state** = name of the city

• **Path cost**
  • Sum of edge costs (total distance traveled)
Example of state definition: Maze solving

- **State** = \((x,y)\), current position of the agent
Example of state definition: Traveling salesman problem

- Goal: visit every city in the United States
- Path cost: total miles traveled
- Initial state: Champaign, IL
- Action: travel from one city to another
- Transition model: when you visit a city, mark it as “visited.”
Example of state definition: Traveling salesman problem

- **state** = (agent, goals)
  
  - **agent** = (agent_x, agent_y) is current position of the agent
  
  - **goals** = [goal[0], goal[1], ...] lists the goals that have not yet been reached
    
    - **goal[i]** = (goal_x, goal_y) tells the location of the i’th remaining goal

Outline of today’s lecture

1. Initial state, goal state, transition model

2. General algorithm for solving search problems
   1. First data structure: a frontier queue
   2. Second data structure: a search tree
   3. Third data structure: explored set
   4. Fourth data structure: explored dict

3. Breadth-first search (BFS) and Depth-first search (DFS)
   1. Completeness
   2. Optimality
   3. Time Complexity
   4. Space Complexity
How does this problem differ from every problem you’ve ever seen before?

• Search differs from most Computer Science problems in that the state space might be infinite. We don’t assume, in advance, that we can enumerate every possible configuration of the world.

• Traditional definition of Dijkstra’s algorithm:
  • First, list all of the possible states in the “not explored” list
  • Then, move them to the “explored” list after we visit them

• Modifying Dijkstra’s algorithm for the infinite-world assumption:
  • Instead of a list of all possible states, we have a method
    \((\text{next}_\text{state}, \text{cost}) = \text{Transition}_\text{Model} (\text{current}_\text{state}, \text{action})\)
  • Instead of an infinite “not explored” list, we have a finite “frontier.”
First data structure: Frontier

• Frontier = set of nodes that you know how to reach, but you haven’t yet tested to see what comes next after those states

• node = ( state, parent_node, path_cost )

• Initialize: frontier = { (initial_state, None, 0) }

• Iterate, until goal is reached:
  • Set current_state to some node from the frontier, remove it from the frontier.
  • Expand current_state:
    • If it’s the goal, then you’re done! Return the corresponding path.
    • If not, then find its children, and transition costs, using (next_state,cost)=Transition_Model( current_state, action), and add them to the frontier.
Example: Romania

• On vacation in Romania; currently in Arad
• Flight leaves tomorrow from Bucharest

• **Initial state**
  • Arad

• **Actions**
  • Go from one city to another

• **Transition model**
  • If you go from city A to city B, you end up in city B

• **Goal state**
  • Bucharest

• **Path cost**
  • Sum of edge costs (total distance traveled)
Search step 0

Frontier: \{ Arad \}

Tree: 

Arad, 0
Search step 1

Frontier: \{ Sibiu, Zerind, Timisoara \}

Tree:

- Arad, 0
- Sibiu, 140
- Timisoara, 118
- Zerind, 75
Tree Search: Basic idea

1. **SEARCH** for an optimal solution
   - Maintain a **frontier** of unexpanded states
   - At each step, pick a state from the frontier to **expand**:
     - Check to see whether or not this state is the goal state. If so, DONE!
     - If not, then list all of the states you can reach from this state, add them to the frontier, and add them to the tree

2. **BACK-TRACE**: go back up the tree; list, in reverse order, all of the actions you need to perform in order to reach the goal state.

3. **ACT**: the agent reads off the sequence of necessary actions, in order, and does them.
Nodes vs. States

• **State** = description of the world
  • Must have enough detail to decide whether or not you’re currently in the initial state
  • Must have enough detail to decide whether or not you’ve reached the goal state
  • Often but not always: “defining the state” and “defining the transition model” are the same thing

• **Node** = a point in the search tree
  • Knows the ID of its STATE
  • Knows the ID of its PARENT NODE
  • Knows the COST of the path
Search step 1

Frontier: \{ Sibiu, Zerind, Timisoara \}

Tree:

- Arad, 0
  - Sibiu, 140
  - Timisoara, 118
  - Zerind, 75
Search step 2
Expand Sibiu

Frontier: \{ Sibiu, Zerind, Timisoara \}

Tree:

- Arad, 0
- Timisoara, 118
- Zerind, 75

Sibiu, 140
Search step 2
Expanded Sibiu

Frontier: \{ Zerind, Timisoara, Oradea, Arad, Rimnicu Vilcea, Fagaras \}

Tree:

- **Arad, 0**
  - **Sibiu, 140**
  - **Timisoara, 118**
  - **Zerind, 75**
- **Oradea, 291**
- **Arad, 280**
- **Ramnicu Valcea, 220**
- **Fagaras, 239**
Tree Search: Computational Complexity

- $b =$ “branching factor” = max # states you can reach from any given state
- $d =$ “depth” = # layers in the tree (# moves that you have made)
- Complexity of Tree Search = $O\{b^d\}$

If the world is infinite (there are an infinite number of possible states), then $O\{b^d\}$ is a reasonable cost to pay.

But what if (as in the Romania example) the world is finite? What if there are only $N$ cities, where $O\{N\} < O\{b^d\}$? ... it’s foolish to suffer $O\{b^d\}$ complexity for a tree search, when an exhaustive search would be only $O\{N\}$.
Third data structure: Explored set

How to limit complexity to $O\{\min(N, b^d)\}$: use an explored set

When you expand a state, do the following for each child state.
• Check to see whether it’s already been explored.
• If so:
  • Skip it.
• If not:
  • Add it to the frontier
  • Add it to the tree
  • Add it to the explored set
Search step 0

Frontier: \{ Arad \}
Explored: \{ Arad \}

Tree:

Arad, 0
Search step 1: expand Arad

Frontier: { Sibiu, Zerind, Timisoara }
Explored: { Arad, Sibiu, Zerind, Timisoara }

Arad, 0

Sibiu, 140
Timisoara, 118
Zerind, 75
Search step 2: expand Sibiu

Frontier: { Zerind, Timisoara, Oradea, Ramnicu Valcea, Fagaras }

Explored: { Arad, Sibiu, Zerind, Timisoara, Oradea, Ramnicu Valcea, Fagaras }

Oradea, 291
Sibiu, 140
Ramnicu Valcea, 220
Timisoara, 118
Fagaras, 239
Zerind, 75
Arad, 0
Search step 3: expand Zerind

Frontier: \{ Timisoara, Oradea, Ramnicu Valcea, Fagaras \}
Explored: \{ Arad, Sibiu, Zerind, Timisoara, Oradea, Ramnicu Valcea, Fagaras \}
- We don’t add Oradea to the frontier again, b/c already in explored set.
- But our new path to Oradea is shorter – only 146km, instead of 291km!
- Can we go back and fix our mistake?
Fourth data structure: Explored Dictionary

Explored = dictionary mapping from state ID to path cost

• If a child state is in the explored dict, and our new path has HIGHER COST, then
  • Skip it.
• If a child state is in the explored dict, but our new path has LOWER COST, then:
  • Update the dict: explored[state] = new_cost
  • Put the new (state, parent, cost) tuple into the frontier and the tree
Search step 3: expanded Zerind

Frontier: { Timisoara, Oradea, Rimnicu Vilcea, Fagaras }
Frontier: { Timisoara, Oradea, Ramnicu Valcea, Fagaras }
Explored: { Arad:0, Sibiu:140, Zerind:75, Timisoara:118, Oradea:146, Rimnicu Vilcea:220, Fagaras:239 }
Outline of today’s lecture

1. Initial state, goal state, transition model
2. General algorithm for solving search problems
   1. First data structure: a frontier queue
   2. Second data structure: a search tree
   3. Third data structure: a “visited states” dict
3. Breadth-first search (BFS) and Depth-first search (DFS)
   1. Completeness
   2. Optimality
   3. Time Complexity
   4. Space Complexity
In which order should you pick nodes from the frontier?

• LIFO (last-in, first-out) = Depth-First Search (DFS):
  • the next node you expand will always be the one most recently added to the frontier.
• FIFO (first-in, first-out) = Breadth-First Search (BFS):
  • the next node you expand will always be the one least recently added to the frontier.
Depth-first search (DFS)

Expand frontier in LIFO order (last in, first out).

Result: most recently discovered path is pursued, all the way to the end.
Analysis of search strategies

• Strategies are evaluated along the following criteria:
  • **Completeness:** does it always find a solution if one exists?
  • **Optimality:** does it always find a least-cost solution?
  • **Time complexity:** number of nodes generated
  • **Space complexity:** maximum number of nodes in memory

• Time and space complexity are measured in terms of
  • $b$: maximum branching factor of the search tree
  • $d$: depth of the optimal solution
  • $m$: maximum length of any path in the state space (may be infinite)
Depth-first search (DFS)

Incomplete: If there are an infinite number of states, DFS might go down a path of infinite length, and might never find a solution.

Suboptimal: DFS returns the first path it finds, which might not be the shortest path.

Time Complexity: $O\{b^m\}$, where $m$ is the longest possible path length.

Space Complexity: only $O\{m\}$! Once you’ve finished a path, you can delete it from the tree!

Depth-first-search. CC-BY-SA 3.0, Mre, 2009
https://commons.wikimedia.org/wiki/File:Depth-First-Search.gif
Breadth-first search (BFS)

Expand the frontier in FIFO order (first-in, first-out).

Result: all paths of length \( d \) are explored, then all paths of length \( d+1 \), and so on.
Breadth-first search (BFS)

Complete: if a finite-length path exists, BFS will find it.

Optimal: BFS returns the first solution it finds, which is always the shortest path.

Time Complexity: $O\{b^d\}$, where $d$ is the length of the best path. This is usually much less than $O\{b^m\}$, because $d < m$.

Space Complexity: $O\{b^d\}$. No part of the tree can be deleted until you’ve found the solution.

Animated-BFS. CC-SA 3.0, Blake Matheny, 2007
https://commons.wikimedia.org/wiki/File:Animated_BFS.gif
BFS: How to do it

• Notice that BFS searches in exactly the same order as Dijkstra’s algorithm.

• BFS is the normal way you would implement Dijkstra’s algorithm for a possibly-infinite search space.

Dijkstra’s progress, CC-BY 3.0, Subh83, 2011
https://commons.wikimedia.org/wiki/File:Dijkstras_progress_animation.gif
Outline of today’s lecture

1. Initial state, goal state, transition model
2. General algorithm for solving search problems
   1. First data structure: a frontier queue
   2. Second data structure: a search tree
   3. Third data structure: a “visited states” dict
3. Breadth-first search (BFS) and Depth-first search (DFS)