1 Introduction

1.1 Agents

An agent is an entity that perceives and acts. A **rational agent**, an entity that has goals or preferences and tries to perform a series of actions that yield the best/optimal expected outcome given these goals. Rational agents exist in an environment, which is specific to the given instantiation of the agent. A rational agent selects actions that maximize its (expected) utility. A **reflex agent** chooses actions only based on the current percept. It is one that does not consider the future consequences of its actions, but rather selects an action based solely on the current state of the world. It considers how the world is. **Planning agents** make decisions based on (hypothesized) consequences of actions. A planning agent has a a model of how the world evolves in response to actions.

2 Search Problems

Through a search problem, we try to find a new state that the agent can reach from its current state that gets it closer to its goal in the most optimal way. A search problem consists of 4 things:

- A **state space** - The set of all possible states that are possible in your given world
- A **successor function** - A function that takes in a state and an action and computes the cost of performing that action as well as the successor state, the state the world would be in if the given agent performed that action
- A **start state** - The state in which an agent exists initially
- A **goal test** - A function that takes a state as input, and determines whether it is a goal state

A solution is a sequence of actions (a plan) which transforms the start state to a goal state. At the start state, the agent explores the state space using the successor function, iteratively computing successors of various states until it arrives at a goal state at which point it will have determined a path from the start state to the goal state, which is a **plan**.

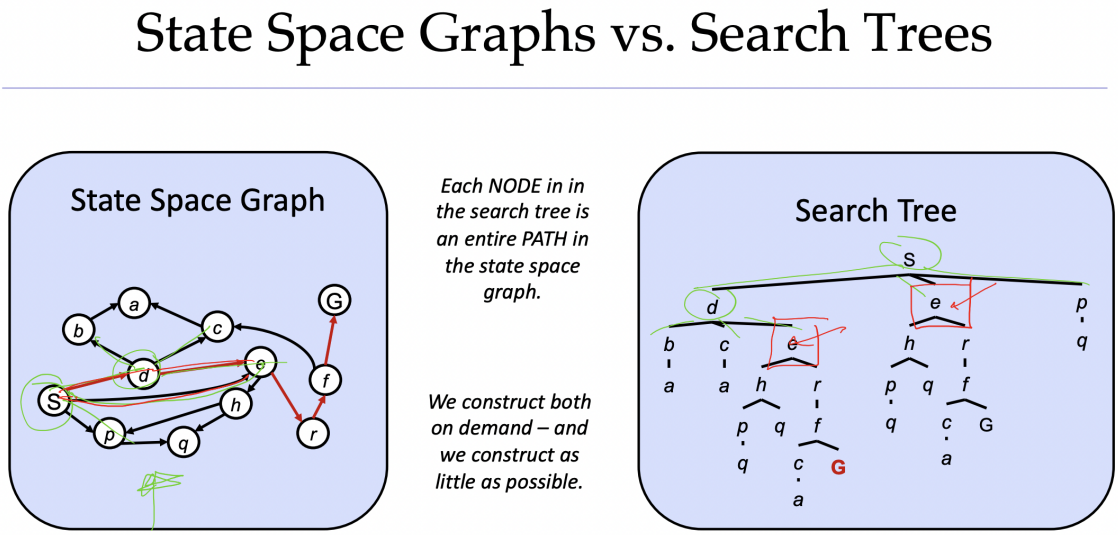
A world state contains all information about a given state, whereas a search state contains only the information about the world that's necessary for planning.

3 State Space Graphs

A graph is defined by a set of nodes and a set of edges connecting various pairs of nodes. These edges may also have weights associated with them. A state space graph is constructed with states representing nodes, with directed edges existing from a state to its successors. The arcs represent successors (action results) and the goal test is a set of goal nodes (maybe only one). In a state space graph, each state is represented **exactly once**. State space graph is a mathematical concept, and we do not explicitly build and store the state graph in memory as they are often huge.

4 Search Trees

In a search tree, on the other hand, a state can appear multiple times. This is because though search trees are also a class of graph with states as nodes and actions as edges between states, each state/node encodes not just the state itself, but the entire path (or plan) from the start state to the given state in the state space graph. Since there often exist multiple ways to get from one state to another, states tend to show up multiple times in search trees.



Search problems are solved using search trees, where we very carefully store a select few nodes to observe at a time, iteratively replacing nodes with their successors until we arrive at a goal state. There exist various methods by which to decide the order in which to conduct this iterative replacement of search tree nodes.

5 Search Algorithms and Uninformed Search

For finding a plan to get from the start state to a goal state, we maintain an outer fringe of partial plans derived from the search tree. We continually expand our fringe by removing a node (which is selected using our given strategy) corresponding to a partial plan from the fringe, and replacing it on the fringe with all its children.

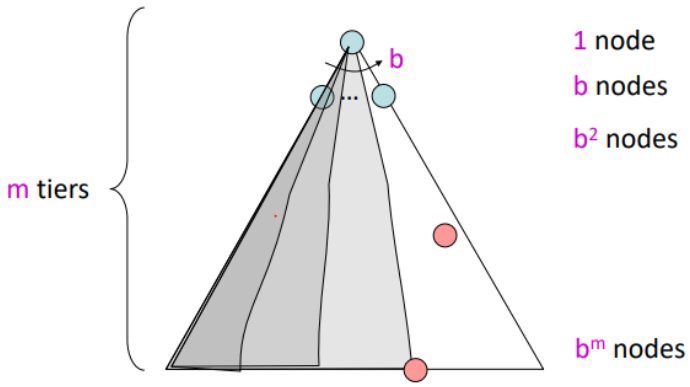
To enable comparisons between different search algorithms, some search algorithm properties we will be discussing are as follows:
Complete: Is the search algorithm guaranteed to find a solution if one exists?
Optimal: Is it guaranteed to find the least cost path?
Time Complexity: What is the runtime of this algorithm?
Space Complexity: How much space (memory) does this algorithm need?

Since we run search algorithms on search trees, it is also important to understand features of a search tree:
The **branching factor** b - The increase in the number of nodes on the frontier each time a frontier node is dequeued and replaced with its children is $O(b)$. At depth k in the search tree, there exists $O(b^k)$ nodes.
The maximum depth m.
The depth of the shallowest solution s.

We will be discussing three search algorithms in this section: Depth-First Search (DFS), Breadth-First Search (BFS), and Uniform-Cost Search (UCS).

5.1 DFS

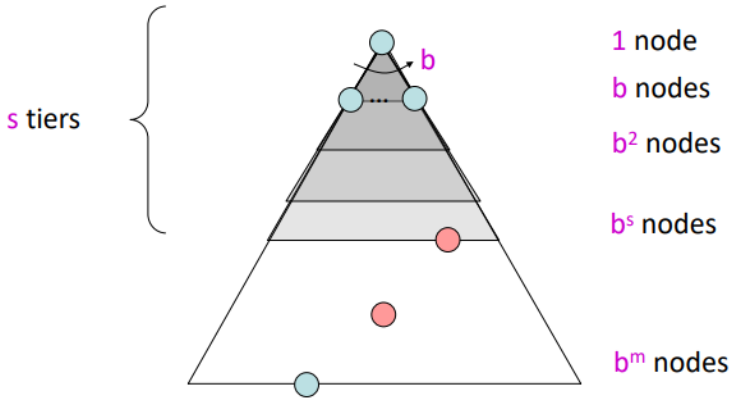
The strategy with DFS is to expand a deepest node first (dig into the search tree) and is implemented by treating the fringe as a Last-In-First-Out stack.



Complete: No, since m could be infinite. Removing cycles in the search tree can fix this problem. With small modification to the original algorithm (ensure a node is only visited once), DFS can be complete.
Optimal: No, since we could find a goal node that is deeper than s (the shallowest solution).
Time Complexity: In the worst case, we go through the whole search tree. So, the runtime is $O(b^m)$ (assuming m is finite).
Space Complexity: $O(bm)$, since we only keep track of siblings on the path from the root node.

5.2 BFS

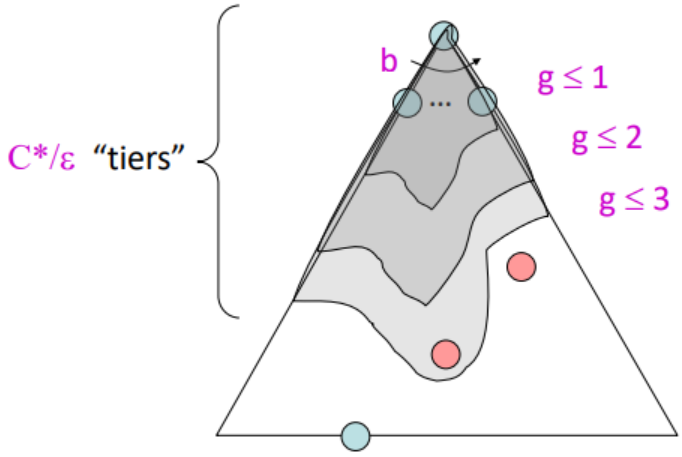
The strategy with BFS is to expand a shallowest node first (sweep the search tree) and is implemented by treating the fringe as a First-In-First-Out queue.



Complete: Yes, since we find a solution if it exists.
Optimal: Yes, if we assume all edges have equal weight.
Time Complexity: $O(b^s)$, where s is the shallowest solution. In the worst case (no solution), we sweep through the whole tree.
Space Complexity: $O(b^s)$, since we keep track of roughly the next layer, where s is the depth of the shallowest goal.

5.3 UCS

The algorithms covered so far do not account for the weight of edges, which can likely change the lowest-cost solution. UCS helps with this problem by accounting for the weights of edges. The strategy with UCS is to expand a cheapest cost node first and is implemented by treating the fringe as a First-In-First-Out priority queue.



Let C* be the cost of the cheapest solution and let ϵ be the lowest cost of the edges. Then, the "effective depth" is roughly C*/ϵ.
Complete: Yes, since we find a solution if it exists.
Optimal: Yes, because we keep track of edge weights.
Time Complexity: $O(b^{C^*/\epsilon})$.
Space Complexity: $O(b^{C^*/\epsilon})$, since we keep track of roughly the last cost tier.

What is the problem with these search algorithms?
They explore options in a methodical, predetermined manner and lack information about the goal node. We will look at algorithms that overcome this limitation in the next section.