Uninformed Search

Table of Contents

[Reflex and Planning Agents 1](#_Toc121328599)

[Mapping Real World Problems 2](#_Toc121328600)

[State Spaces 3](#_Toc121328601)

[Search Trees 5](#_Toc121328602)

[Search Tree Algorithms 7](#_Toc121328603)

[Algorithm Properties 8](#_Toc121328604)

[Depth First Search 9](#_Toc121328605)

[Properties 9](#_Toc121328606)

[Breadth First Search 9](#_Toc121328607)

[Properties 10](#_Toc121328608)

[Iterative Deeping 10](#_Toc121328609)

[Cost Sensitive Search 11](#_Toc121328610)

[Properties 11](#_Toc121328611)

[UCS Issues 12](#_Toc121328612)

## Reflex and Planning Agents

A **Reflex Agent** is an agent which does not plan. It takes actions based on its current perception of the world and perhaps with the help of some memory of past experiences. It does not consider what the consequences of its actions will be and how the surrounding environment will react to its actions. Reflex agents are good for quick decision-making. They are not always rational agents in that their action is not necessarily the rational one, but if their action happens to be the optimal solution for a particular situation, they can be considered rational.

A **Planning Agent** is the exact opposite. They consider the consequences of all the possible actions. This guarantees that they will come up with the optimal solution, but it also makes them infuriatingly slow. To be able to consider the consequences, planning agents need a model of how their environment works, e.g., understanding the concept of gravity to understand that pushing something will make it fall over. They must also have an end goal which, when achieved, they will stop planning. Otherwise, they will keep planning forever.

Planning agents can be further divided into two categories, **masterminds**, which consider all the possible actions before choosing the optimal one, and **replanning agents**, which take the best actions based on the current situation and immediate consequences, ignoring whether this is the best solution in the long run. Replanning agents are basically the equivalent of the greedy approach.

When creating agents, we need to keep two metrics in mind:

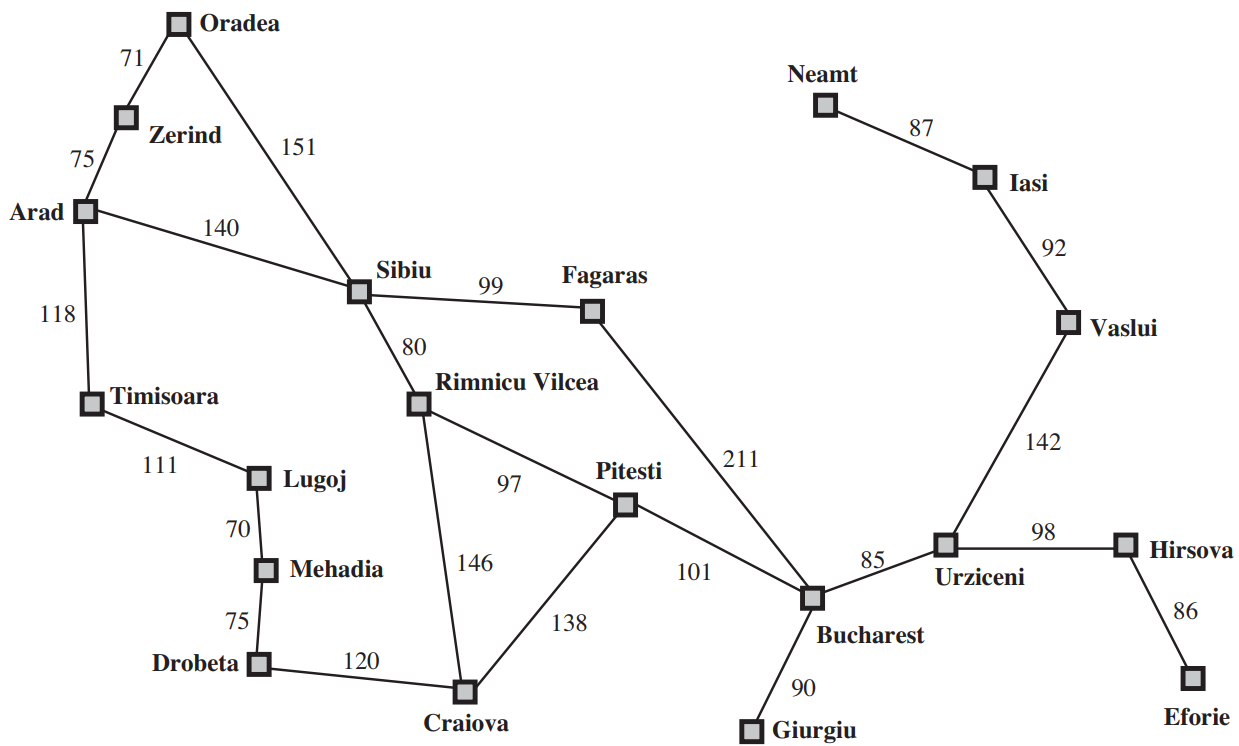
* **Optimality** – Whether the agent manages to minimize the cost of reaching the goal
* **Completeness** – Whether the agent is always able to find a solution, given that one exists

## Mapping Real World Problems

We will now consider how we can create a **search problem** out of a real world one, specifically, a game of Pac-Man. When considering any problem, we first need to define a few things:

* **State Space** – This consists of all the possible configurations of our environment. For our example, this consists of all possible positions of dots and Pac-Man.
* **Successor Function** – This is a function which takes a state and an action as input and outputs what the next state will be based on that input. Thus, if the current state shows Pac-Man in one position and the player takes an action, the output will be the new position of Pac-Man. The function might also take a cost associated with taking an action as a parameter.
* **Start State** – The initial state.
* **Goal Test** – This is a function which checks if the goal has been achieved or not. The goal will be one or more of the states in the state space, so the function just checks if the current state is one of the goal states.
* **Solution** – This is the complete sequence of actions (called a plan) which takes the player from the start state to any of the goal states.

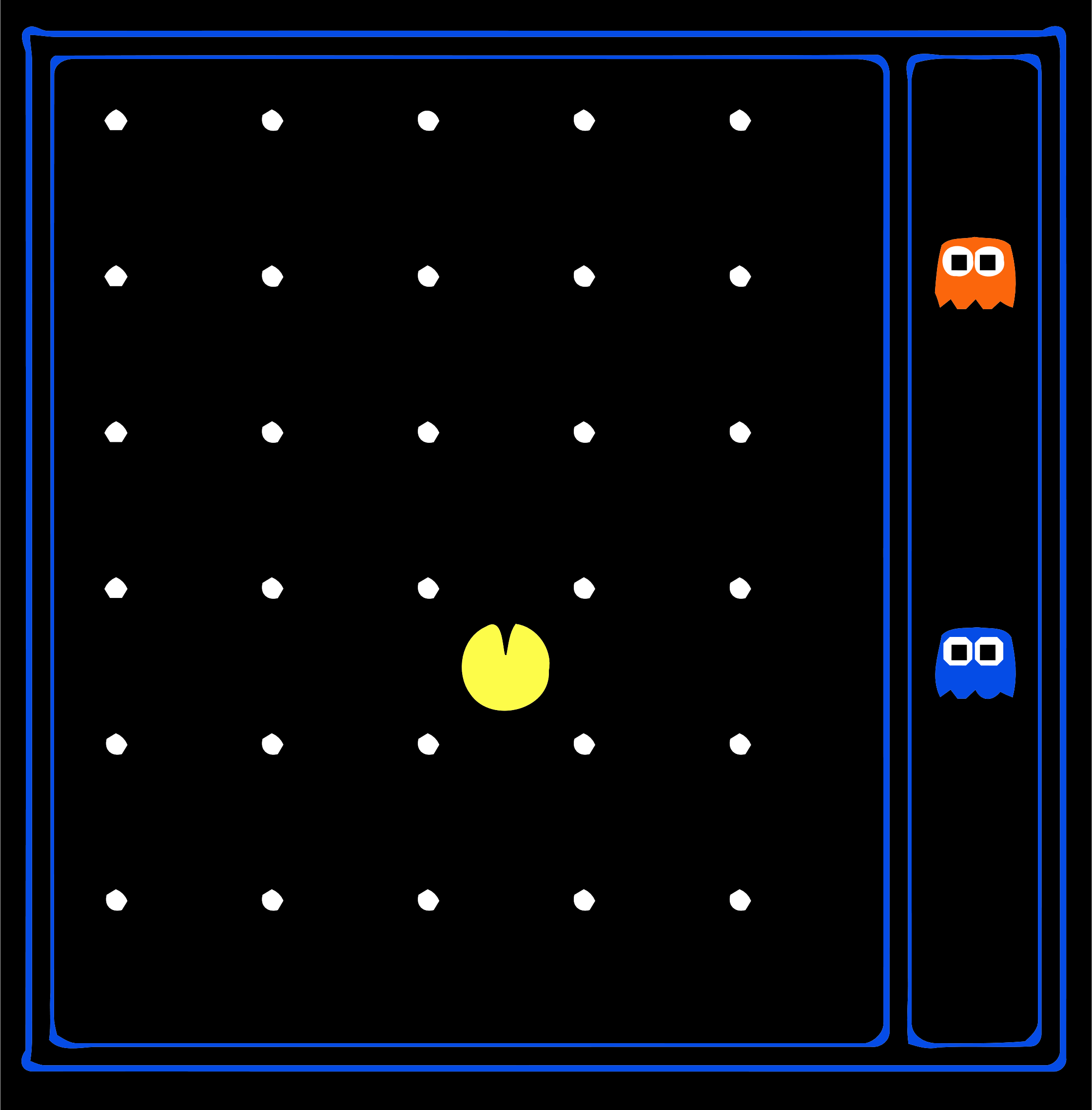
A search problem is just a model. The better we can model the real world based on the above categories, the more optimized our solution will be.



Consider the map of Romania above. Suppose our goal is to get from one city to another. The state space would include all the cities, since those are the possible positions for us to be in. The successor function would take the route to be used as a parameter to decide what state we should be taken to. The start state would be the initial city we are in. The goal test would check if our current city is the city we were trying to reach. The solution would be the path from the initial city to the city we were trying to reach.

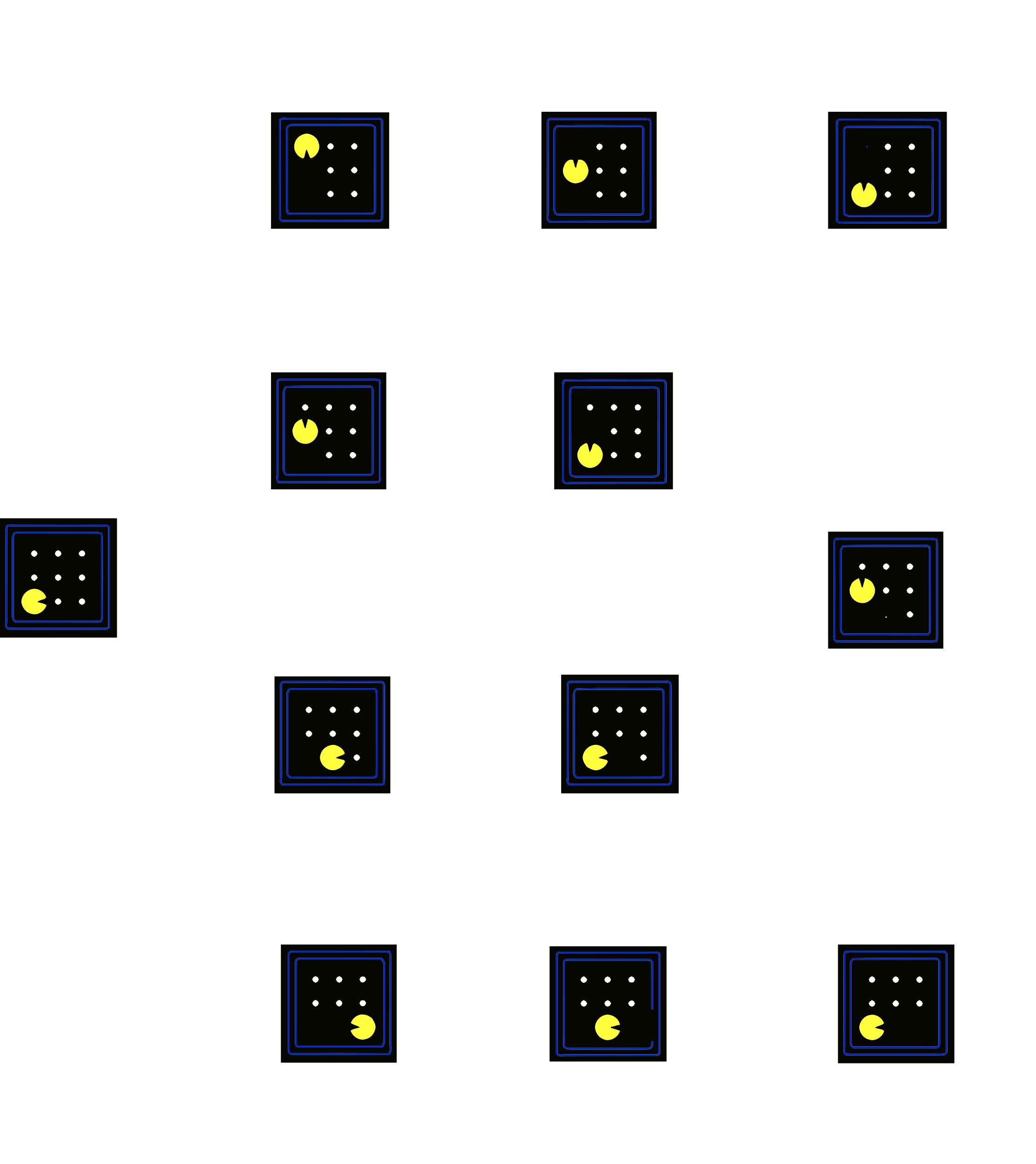
## State Spaces

There are two types of state spaces, the world state space and the search state space. The **world state space** contains all possible configurations of the environment. For Pac-Man, this means all possible positions of Pac-Man, dots, and ghosts. The **search state space** would only include states that are relevant to finding the solution. An example should make this clearer.



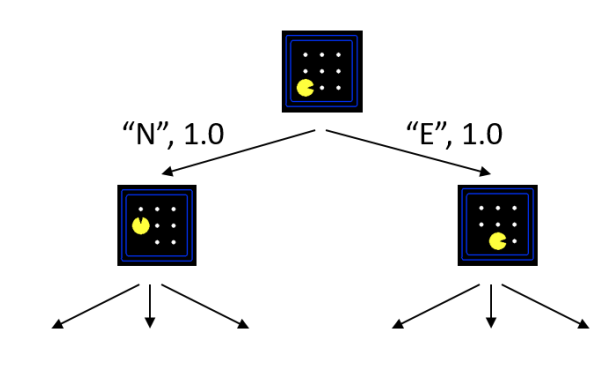
For the diagram above, the world state would include the 120 positions reachable by Pac-Man, the 230 eaten and not eaten states of the 30 dots, the 122 positions of the 2 ghosts and the 4 directions in which Pac-Man can face. Thus, the world state includes states.

On the other hand, the search state will have a different size depending on the problem we want to solve. Suppose the problem we want to solve is to get Pac-Man from point A to point B. In this case, the states of the dots and the positions of the ghosts are not relevant to us. Thus, the search state would have states. On the other hand, if the problem to solve is to eat all the dots, then the search state has states. This is because the ghosts cannot reach Pac-Man and can thus be ignored, and we are assuming that the direction Pac-Man faces is irrelevant to its movements and ability to eat the dots.



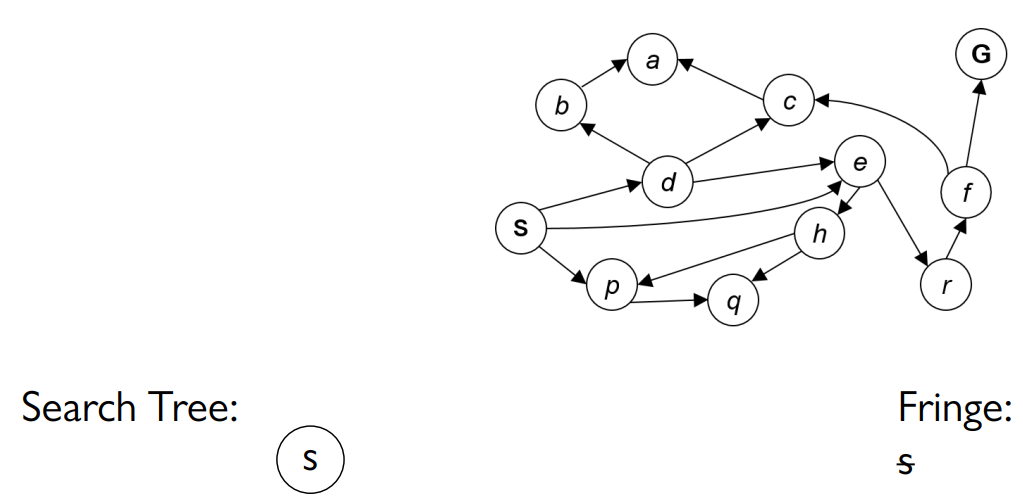
A **state space graph** such as the one shown above can be created from the search problem. Each **node** on the graph represents a different search state. The **arcs** or edges represent an action which takes us from one state to another. The goal test would still check the current node against the goal nodes. We can now solve this using a graph search algorithm.

## Search Trees



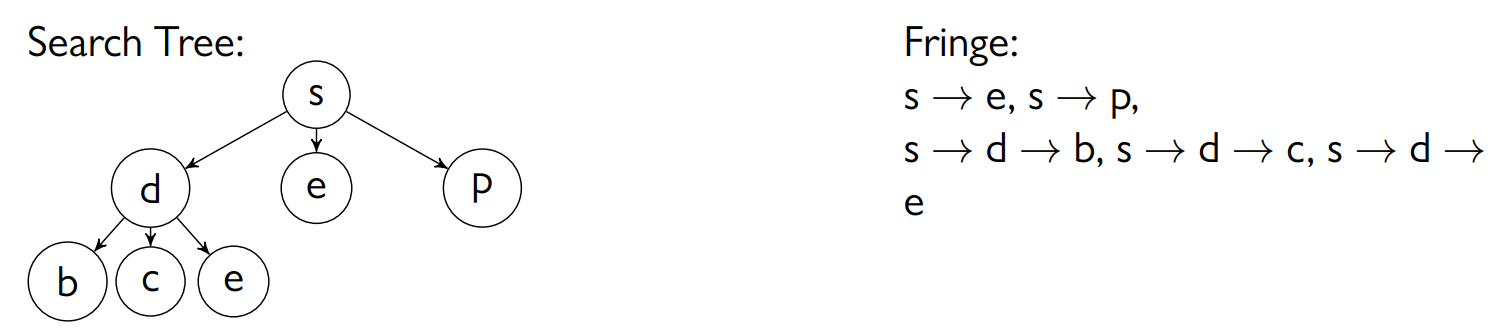
A **search tree** starts at a root node and shows all the possible futures based on the actions we take. For most problems, we cannot build the tree because it would become too large.

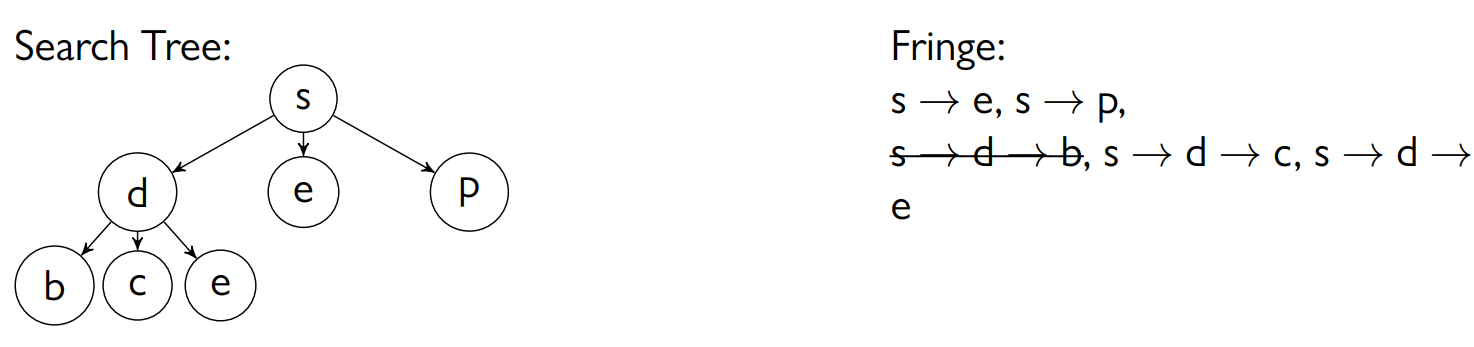
Each node shows the state we are in, but it also shows the **plan** we are making to reach the node from the start state. We just need to execute the plan, which is a series of actions, at the start node and we will end up at the current node.



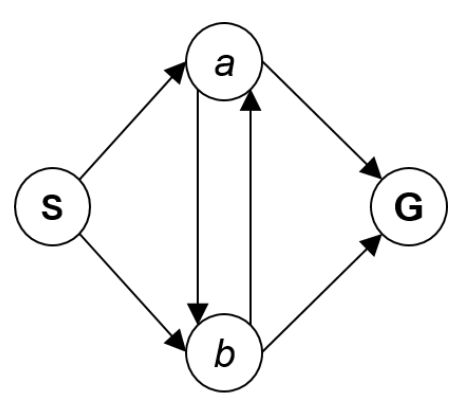








Search trees have more repetition than an equivalent state space graph. Also, if the graph contains any cycles, we can end up in an infinitely long search tree.



## Search Tree Algorithms

The general steps for all the search tree algorithms we will be seeing follow three rules:

1. Expand potential plans
2. Keep track of the potential plans using a fringe (a list)
3. Try to expand as few nodes as possible

The algorithm will start at the root node and keep expanding potential plans from the fringe until it finds an overall solution which it returns.

Initialize search tree

Loop

If no candidates for expansion

return failure

Choose a node based on the exploration strategy

If goal is reached

return plan

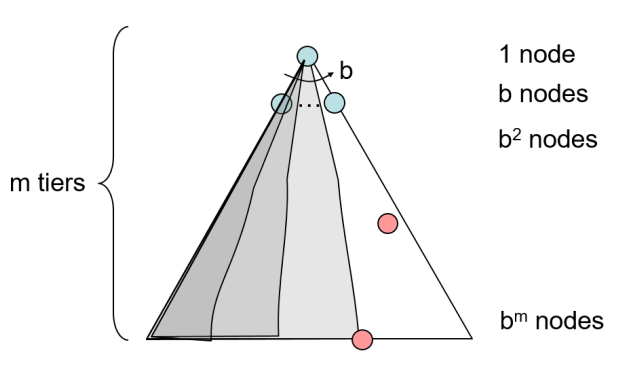
else

expand node and add to list

## Algorithm Properties

When trying to determine which algorithm to use, we take a few properties of the algorithm into account. Namely, these are completeness, optimality, time complexity and space complexity.

From any given state, we have a maximum of possible moves. This is called the **branching factor**. For Pac-Man, the branching factor is 4. The number of nodes increases exponentially as we go down the tree due to this branching factor.



At depth 0 (the root node), there is 1 node. At depth 1, there are nodes. At depth 2, there are nodes. Continuing in this fashion, at depth , the maximum possible depth, there are nodes. gives us the total number of nodes. Since we are dealing with complexity, this can be written as . Thus, the time complexity for expanding all the nodes in the worst case is .

## Depth First Search

**Depth First Search** involves expanding the deepest node first. While we expand, the fringe used is a **LIFO Stack**. When expanding the nodes, the order of expanding nodes at the same depth is not specified by the DFS algorithm, so we can expand them in any order.

### Properties

* The worst-case time complexity of DFS is .
* At any given moment of time, there are other nodes for each depth in the fringe. Thus, the space complexity is .
* If there are no cycles in the state space graph, then the algorithm is complete.
* The algorithm is not optimal.

## Breadth First Search

**Breadth First Search** involves expanding the shallowest node first. It uses a **FIFO Queue** as a fringe.

### Properties

* The worst-case time complexity if . However, since BFS travels layer by layer, it is very likely that we will find a solution by the time we reach some layer where . Thus, we can say that the time complexity is .
* The space complexity of BFS is worse than that of DFS, since unlike DFS we are having to store all possible paths until the current layer. Thus, the space complexity is .
* The algorithm is complete given that there are no cycles in the state space graph.
* The algorithm is optimal assuming that the cost for each action is uniform.

## Iterative Deeping

The **Iterative Deepening** algorithm is basically DFS, but with a maximum limit set for the depth, meaning the algorithm will not explore further than the specified depth. This depth is increased iteratively. This gives us the benefit of the space complexity of DFS combined with the optimality of BFS.

One issue with this algorithm is that we are repeatedly checking the nodes towards the top of the tree. However, it is better to check the few nodes towards the top repeatedly than to check the nodes near the bottom even once.

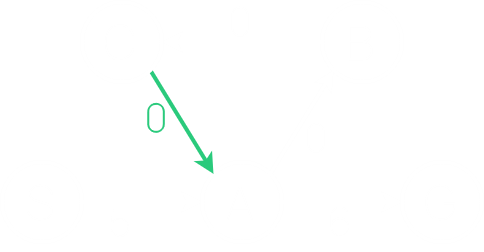
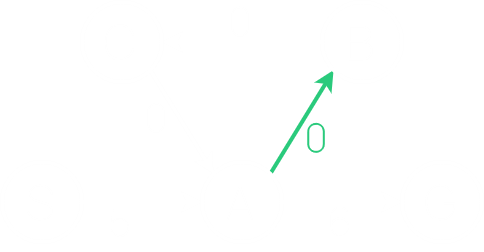
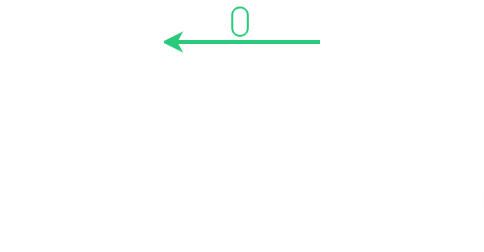
## Cost Sensitive Search

So far, the algorithms we have explored assumed the cost for each action is uniform. However, if there are costs attached to the actions, the optimization factor would shift to the cost instead of the path size.

The most naïve approach to this is to just expand the cheapest paths first. In this case, the fringe is **priority queue**.

### Properties

* Suppose the optimal path for this search algorithm takes steps at a cost of on average for each step. The best-case cost is thus . is called the **effective depth**. Thus, the time complexity is . If , this becomes the BFS algorithm.
* Like the above, the space complexity is also .
* The algorithm is complete given that the cost is finite, and all arc costs are positive. Otherwise, we can end up in situations like the one below, where the algorithm gets stuck in a loop.

* The algorithm is optimal because it attempts to find solutions by checking costs in an ascending order, i.e., it checks paths with cost 1 first, then 2 and so on until it finds the path with the least cost. The proof for this will be covered in a later lecture.

### UCS Issues

UCS explores options in every direction. It does not have any information about which direction the goal is in. Because of this, it can find the optimal solution, but it will take a long time. We will explore how to do this faster using **informed search**.