CS 188 Introduction to Artificial Intelligence
Fall 2022 Note 1

These lecture notes are heavily based on notes originally written by Nikhil Sharma.

Last updated: September 2, 2022

Agents

In artificial intelligence, the central problem at hand is that of the creation of 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. Agents use sensors to interact with the environment and act on it using actuators. As a very simple example, the environment for a checkers agent is the virtual checkers board on which it plays against opponents, where piece moves are actions. Together, an environment and the agents that reside within it create a **world**.

A **reflex agent** is one that doesn't think about the consequences of its actions, but rather selects an action based solely on the current state of the world. These agents are typically outperformed by **planning agents**, which maintain a model of the world and use this model to simulate performing various actions. Then, the agent can determine hypothesized consequences of the actions and can select the best one. This is simulated "intelligence" in the sense that it's exactly what humans do when trying to determine the best possible move in any situation - thinking ahead.

To define the task environment we use the **PEAS** (**P**erformance Measure, **E**nvironment, **A**ctuators, **S**ensors) description. The performance measure describes what utility the agent tries to increase. The environment summarizes where the agent acts and what affects the agent. The actuators and the sensors are the methods with which the agent acts on the environment and receives information from it.

The **design** of an agent heavily depends on the type of environment the agents acts upon. We can characterize the types of environments in the following ways.

- In *partially observable* environments, the agent does not have full information about the state and thus the agent must have an internal estimate of the state of the world. This is in contrast to *fully observable* environments, where the agent has full information about their state.
- *Stochastic* environments have uncertainty in the transition model, i.e. taking an action in a specific state may have multiple possible outcomes with different probabilities. This is in contrast to *deterministic* environments, where taking an action in a state has a single outcome that is guaranteed to happen.
- In *multi-agent* environments the agent acts in the environments along with other agents. For this reason the agent might need to randomize its actions in order to avoid being "predictable" by other agents.
- If the environment does not change as the agent acts on it, then this environment is called *static*. This is contrast to *dynamic* environments that change as the agent interacts with it.

- If an environment has *known physics*, then the transition model (even if stochastic) is known to the agent and it can use that when planning a path. If the *physics are unknown* the agent will need to take actions deliberately to learn the unknown dynamics.

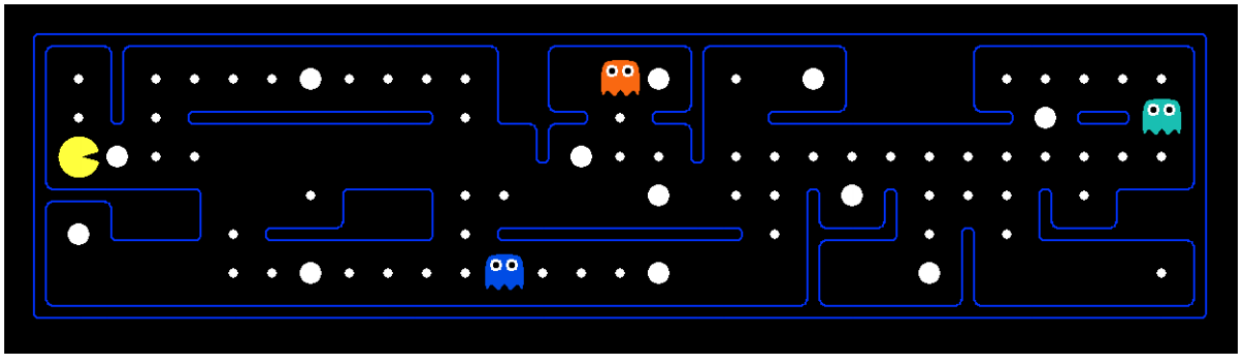
State Spaces and Search Problems

In order to create a rational planning agent, we need a way to mathematically express the given environment in which the agent will exist. To do this, we must formally express a **search problem** - given our agent's current **state** (its configuration within its environment), how can we arrive at a new state that satisfies its goals in the best possible way? A search problem consists of the following elements:

- A **state space** - The set of all possible states that are possible in your given world
- A set of **actions** available in each state
- A **transition** model - Outputs the next state when a specific action is taken at current state
- An action **cost** - Incurred when moving from one state to another after applying an 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

Fundamentally, a search problem is solved by first considering the start state, then exploring the state space using the action and transition and cost methods, iteratively computing children of various states until we arrive at a goal state, at which point we will have determined a path from the start state to the goal state (typically called a **plan**). The order in which states are considered is determined using a predetermined **strategy**. We'll cover types of strategies and their usefulness shortly.

Before we continue with how to solve search problems, it's important to note the difference between a **world state**, and a **search state**. 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 (primarily for space effiency reasons). To illustrate these concepts, we'll introduce the hallmark motivating example of this course - Pacman. The game of Pacman is simple: Pacman must navigate a maze and eat all the (small) food pellets in the maze without being eaten by the malicious patrolling ghosts. If Pacman eats one of the (large) power pellets, he becomes ghost-immune for a set period of time and gains the ability to eat ghosts for points.



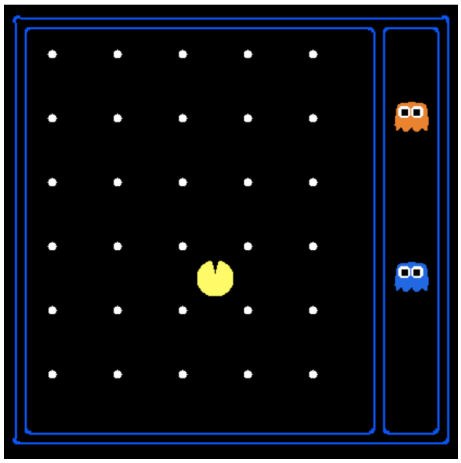
Let's consider a variation of the game in which the maze contains only Pacman and food pellets. We can pose two distinct search problems in this scenario: pathing and eat-all-dots. Pathing attempts to solve the problem of getting from position (x_1, y_1) to position (x_2, y_2) in the maze optimally, while eat all dots attempts to solve the problem of consuming all food pellets in the maze in the shortest time possible. Below, the states, actions, transition model, and goal test for both problems are listed:

- **Pathing**
  - States: (x,y) locations
  - Actions: North, South, East, West
  - Transition model (getting the next state): Update location only
  - Goal test: Is (x,y)=END?
- **Eat-all-dots**
  - States: (x,y) location, dot booleans
  - Actions: North, South, East, West
  - Transition model (getting the next state): Update location and booleans
  - Goal test: Are all dot booleans false?

Note that for pathing, states contain less information than states for eat-all-dots, because for eat-all-dots we must maintain an array of booleans corresponding to each food pellet and whether or not it's been eaten in the given state. A world state may contain more information still, potentially encoding information about things like total distance traveled by Pacman or all positions visited by Pacman on top of its current (x,y) location and dot booleans.

State Space Size

An important question that often comes up while estimating the computational runtime of solving a search problem is the size of the state space. This is done almost exclusively with the **fundamental counting principle**, which states that if there are n variable objects in a given world which can take on x_1, x_2, ..., x_n different values respectively, then the total number of states is $x_1 \cdot x_2 \cdot \ldots \cdot x_n$. Let's use Pacman to show this concept by example:



Let's say that the variable objects and their corresponding number of possibilities are as follows:

- *Pacman positions* - Pacman can be in 120 distinct (x, y) positions, and there is only one Pacman
- *Pacman Direction* - this can be North, South, East, or West, for a total of 4 possibilities
- *Ghost positions* - There are two ghosts, each of which can be in 12 distinct (x, y) positions

- *Food pellet configurations* - There are 30 food pellets, each of which can be eaten or not eaten

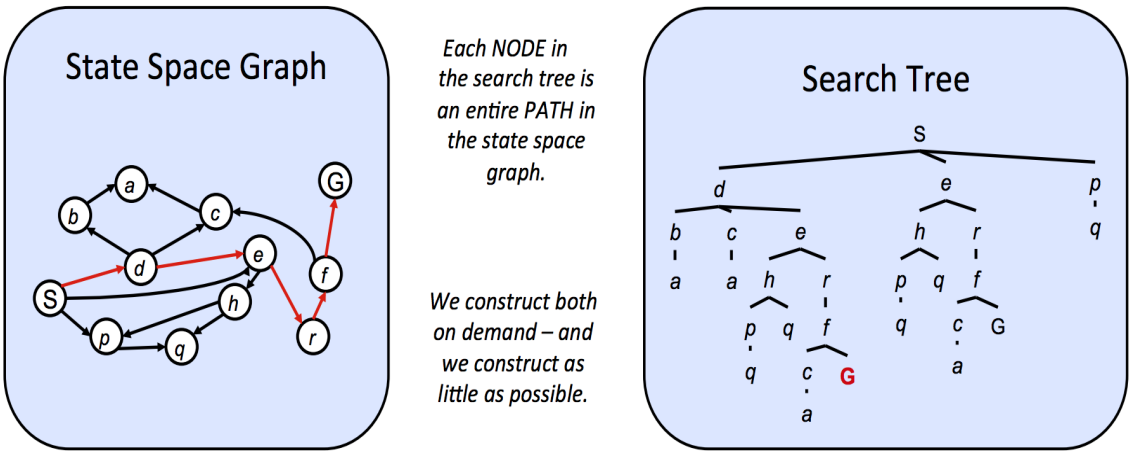
Using the fundamental counting principle, we have 120 positions for Pacman, 4 directions Pacman can be facing, $12 \cdot 12$ ghost configurations (12 for each ghost), and $2 \cdot 2 \cdot \ldots \cdot 2 = 2^{30}$ food pellet configurations (each of 30 food pellets has two possible values - eaten or not eaten). This gives us a total state space size of $\boxed{120 \cdot 4 \cdot 12^2 \cdot 2^{30}}$.

State Space Graphs and Search Trees

Now that we've established the idea of a state space and the four components necessary to completely define one, we're almost ready to begin solving search problems. The final piece of the puzzle is that of state space graphs and search trees.

Recall that 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 children. These edges represent actions, and any associated weights represent the cost of performing the corresponding action. Typically, state space graphs are much too large to store in memory (even our simple Pacman example from above has $\approx 10^{13}$ possible states, yikes!), but they're good to keep in mind conceptually while solving problems. It's also important to note that in a state space graph, each state is represented exactly once - there's simply no need to represent a state multiple times, and knowing this helps quite a bit when trying to reason about search problems.

Unlike state space graphs, our next structure of interest, **search trees**, have no such restriction on the number of times a state can appear. 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. Observe the state space graph and corresponding search tree below:



The highlighted path (S $\rightarrow$ d $\rightarrow$ e $\rightarrow$ r $\rightarrow$ f $\rightarrow$ G) in the given state space graph is represented in the corresponding search tree by following the path in the tree from the start state *S* to the highlighted goal state *G*. Similarly, each and every path from the start node to any other node is represented in the search tree by a path from the root *S* to some descendant of the root corresponding to the other node. Since there often exist multiple ways to get from one state to another, states tend to show up multiple times in search trees. As a result, search trees are greater than or equal to their corresponding state space graph in size.

We've already determined that state space graphs themselves can be enormous in size even for simple problems, and so the question arises - how can we perform useful computation on these structures if they're too big to represent in memory? The answer lies in how we compute the children of a current state - we only store states we're immediately working with, and compute new ones on-demand using the corresponding getNextState, getAction, and getActionCost methods. Typically, 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 children 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, and we'll present these methods now.

Uninformed Search

The standard protocol for finding a plan to get from the start state to a goal state is to maintain an outer **frontier** of partial plans derived from the search tree. We continually **expand** our frontier by removing a node (which is selected using our given **strategy**) corresponding to a partial plan from the frontier, and replacing it on the frontier with all its children. Removing and replacing an element on the frontier with its children corresponds to discarding a single length n plan and bringing all length $(n+1)$ plans that stem from it into consideration. We continue this until eventually removing a goal state off the frontier, at which point we conclude the partial plan corresponding to the removed goal state is in fact a path to get from the start state to the goal state.

Practically, most implementations of such algorithms will encode information about the parent node, distance to node, and the state inside the node object. This procedure we have just outlined is known as **tree search**, and the pseudocode for it is presented below:

```
function TREE-SEARCH(problem, frontier) return a solution or failure
    frontier← INSERT(MAKE-NODE(INITIAL-STATE[problem]), frontier)
    while not IS-EMPTY(frontier) do
        node ← POP(frontier)
        if problem.IS-GOAL(node.STATE) then return node
        end if
        for each child-node in EXPAND(problem, node) do
            add child-node to frontier
        end for
    end while

    return failure
```

The `EXPAND` function appearing in the pseudocode returns all the possible nodes that can be reached from a given node by considering all available actions. The pseudocode of the function is as follows:[1]

```
function EXPAND(problem, node) yields nodes
    s← node.STATE
    for each action in problem.ACTIONS(s) do
        s′ ← problem.RESULT(s, action)
        yield NODE(STATE=s′, PARENT=node, ACTION=action)
    end for
```

When we have no knowledge of the location of goal states in our search tree, we are forced to select our strategy for tree search from one of the techniques that falls under the umbrella of **uninformed search**.

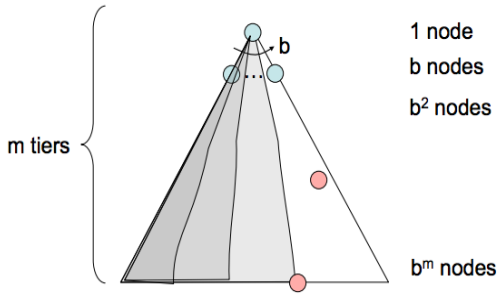
[1] For definition of **yield**, refer to Appendix B.2 of the textbook.

We'll now cover three such strategies in succession: **depth-first search**, **breadth-first search**, and **uniform cost search**. Along with each strategy, some rudimentary properties of the strategy are presented as well, in terms of the following:

- The **completeness** of each search strategy - if there exists a solution to the search problem, is the strategy guaranteed to find it given infinite computational resources?
- The **optimality** of each search strategy - is the strategy guaranteed to find the lowest cost path to a goal state?
- 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.

Depth-First Search

- *Description* - Depth-first search (DFS) is a strategy for exploration that always selects the *deepest* frontier node from the start node for expansion.
- *Frontier Representation* - Removing the deepest node and replacing it on the frontier with its children necessarily means the children are now the new deepest nodes - their depth is one greater than the depth of the previous deepest node. This implies that to implement DFS, we require a structure that always gives the most recently added objects highest priority. A last-in, first-out (LIFO) stack does exactly this, and is what is traditionally used to represent the frontier when implementing DFS.

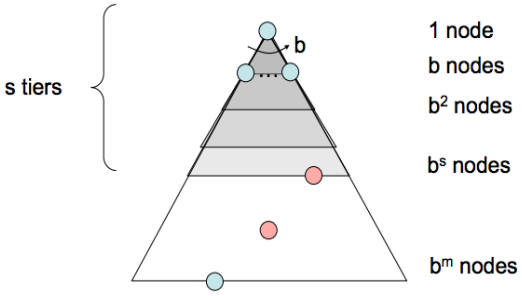


- *Completeness* - Depth-first search is not complete. If there exist cycles in the state space graph, this inevitably means that the corresponding search tree will be infinite in depth. Hence, there exists the possibility that DFS will faithfully yet tragically get "stuck" searching for the deepest node in an infinite-sized search tree, doomed to never find a solution.
- *Optimality* - Depth-first search simply finds the "leftmost" solution in the search tree without regard for path costs, and so is not optimal.
- *Time Complexity* - In the worst case, depth first search may end up exploring the entire search tree. Hence, given a tree with maximum depth m, the runtime of DFS is $O(b^m)$.

- *Space Complexity* - In the worst case, DFS maintains b nodes at each of m depth levels on the frontier. This is a simple consequence of the fact that once b children of some parent are enqueued, the nature of DFS allows only one of the subtrees of any of these children to be explored at any given point in time. Hence, the space complexity of DFS is $O(bm)$.

Breadth-First Search

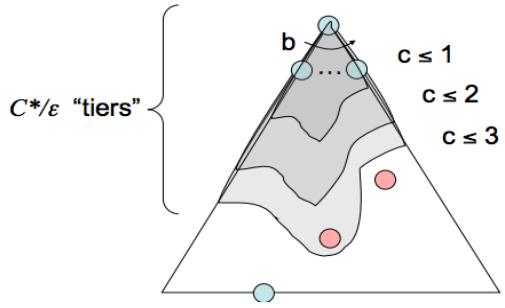
- *Description* - Breadth-first search is a strategy for exploration that always selects the *shallowest* frontier node from the start node for expansion.
- *Frontier Representation* - If we want to visit shallower nodes before deeper nodes, we must visit nodes in their order of insertion. Hence, we desire a structure that outputs the oldest enqueued object to represent our frontier. For this, BFS uses a first-in, first-out (FIFO) queue, which does exactly this.



- *Completeness* - If a solution exists, then the depth of the shallowest node s must be finite, so BFS must eventually search this depth. Hence, it's complete.
- *Optimality* - BFS is generally not optimal because it simply does not take costs into consideration when determining which node to replace on the frontier. The special case where BFS is guaranteed to be optimal is if all edge costs are equivalent, because this reduces BFS to a special case of uniform cost search, which is discussed below.
- *Time Complexity* - We must search $1 + b + b^2 + ... + b^s$ nodes in the worst case, since we go through all nodes at every depth from 1 to s. Hence, the time complexity is $O(b^s)$.
- *Space Complexity* - The frontier, in the worst case, contains all the nodes in the level corresponding to the shallowest solution. Since the shallowest solution is located at depth s, there are $O(b^s)$ nodes at this depth.

Uniform Cost Search

- *Description* - Uniform cost search (UCS), our last strategy, is a strategy for exploration that always selects the *lowest cost* frontier node from the start node for expansion.
- *Frontier Representation* - To represent the frontier for UCS, the choice is usually a heap-based priority queue, where the priority for a given enqueued node v is the path cost from the start node to v, or the *backward cost* of v. Intuitively, a priority queue constructed in this manner simply reshuffles itself to maintain the desired ordering by path cost as we remove the current minimum cost path and replace it with its children.



- *Completeness* - Uniform cost search is complete. If a goal state exists, it must have some finite length shortest path; hence, UCS must eventually find this shortest length path.
- *Optimality* - UCS is also optimal if we assume all edge costs are nonnegative. By construction, since we explore nodes in order of increasing path cost, we're guaranteed to find the lowest-cost path to a goal state. The strategy employed in Uniform Cost Search is identical to that of Dijkstra's algorithm, and the chief difference is that UCS terminates upon finding a solution state instead of finding the shortest path to all states. Note that having negative edge costs in our graph can make nodes on a path have decreasing length, ruining our guarantee of optimality. (See Bellman-Ford algorithm for a slower algorithm that handles this possibility)
- *Time Complexity* - Let us define the optimal path cost as C^* and the minimal cost between two nodes in the state space graph as ε. Then, we must roughly explore all nodes at depths ranging from 1 to C^*/ε, leading to an runtime of $O(b^{C^*/\varepsilon})$.
- *Space Complexity* - Roughly, the frontier will contain all nodes at the level of the cheapest solution, so the space complexity of UCS is estimated as $O(b^{C^*/\varepsilon})$.

As a parting note about uninformed search, it's critical to note that the three strategies outlined above are fundamentally the same - differing only in expansion strategy, with their similarities being captured by the tree search pseudocode presented above.