
CS 444 Notes – Search Problems and Uninformed Search ¹

Learning Objectives for Week

The learning outcomes/objectives for this week are:

- define a problem given its 5 components (section 3.1.1 in the textbook)
- define a world state and given a problem, define the minimum required world state representation
- Define the difference between a state space graph and a search tree. Given a problem, be able to draw a state space graph and/or define the size of the state space.
- Define completeness, optimality, branching factor, solution depth, goal state, and successor function with respect to search problems
- Characterize the DFS, BFS, UCS, DLS, and IDS algorithms using these properties: completeness, optimality, running time, and space complexity. Be able to trace the execution of each of these algorithms given a problem.

State Space 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? Formulating such a problem requires four 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

Fundamentally, a search problem is solved by first considering the start state, then exploring the state space using the successor function, iteratively computing successors 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 efficiency 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.

¹These notes are heavily based on work by Nikhil Sharma and others Berkley instructors

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, successor function, and goal test for both problems are listed:

- **Pathing**

- States (x,y) locations
- Actions: North, South, East, West
- Successor: Update location only
- Goal test: Is $(x,y) = \text{END}$?

- **Eat-all-dots**

- States: (x, y) location, dot booleans
- Actions: North, South, East, West
- Successor: 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, \dots, x_n different values respectively, then the total number of states is $x_1 \cdot x_2 \cdot \dots \cdot x_n$. 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 $2 \cdot 2 \cdot \dots \cdot 2 = 2^{30}$ 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 $120 \cdot 4 \cdot 12^2 \cdot 2^{30}$.

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 successors. 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,

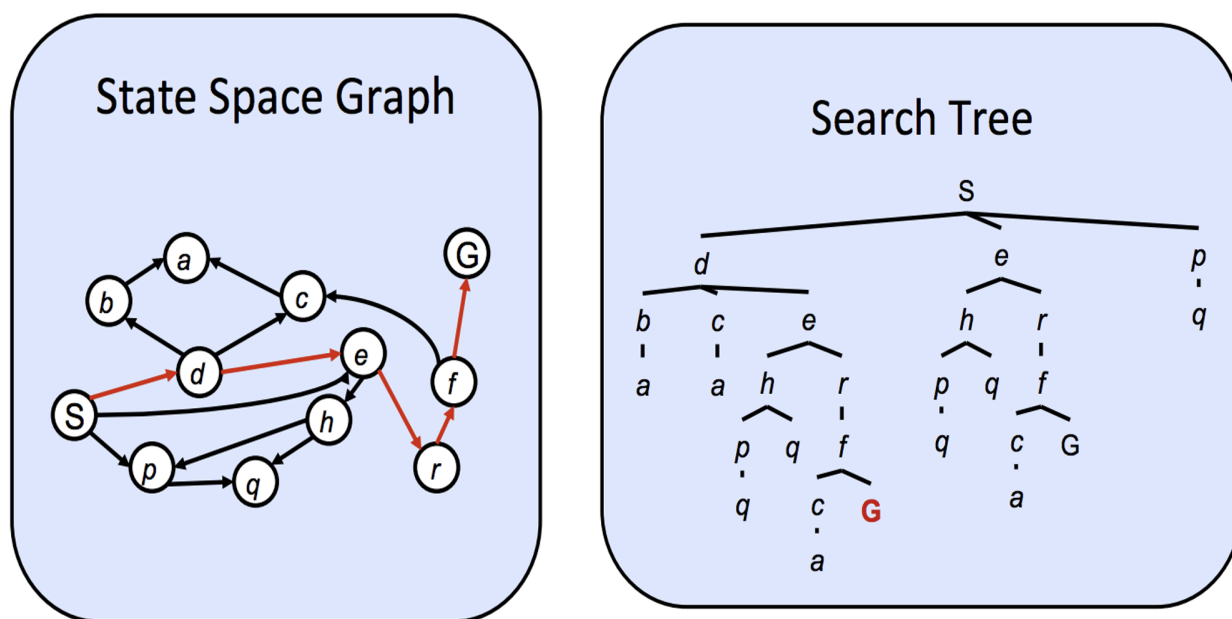


Figure 1: Each node in the search tree is an entire **PATH** in the state space graph. We construct both **on demand** - and we construct as little as possible.

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.

Uninformed Search

The standard protocol for finding a plan to get from the start state to a goal state is to 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. Removing and replacing an element on the fringe 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 fringe, 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:

Algorithm 1 Tree-Search

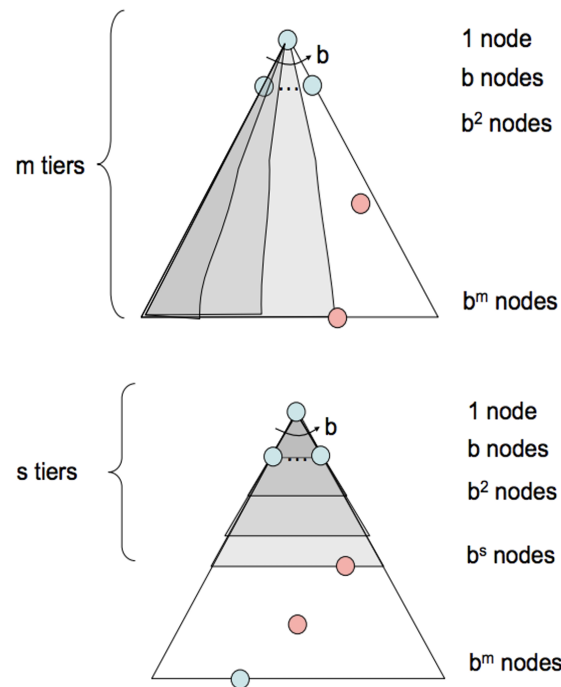
```
1: function TREE-SEARCH(problem) returns a solution or failure
2:   fringe  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
3:   loop
4:     if fringe is empty then return failure
5:     end if
6:     node  $\leftarrow$  REMOVE-FRONT(fringe)
7:     if GOAL-TEST(problem, STATE[node]) then return node
8:     end if
9:     for child-node in EXPAND(STATE[node], problem) do
10:       fringe  $\leftarrow$  INSERT(child-node, fringe)
11:     end for
12:   end loop
13: end function
```

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. 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 fringe each time a fringe node is dequeued and replaced with its children is $\mathcal{O}(b)$. At depth k in the search tree, there exists $\mathcal{O}(b^k)$ nodes.
- The maximum depth m .
- The depth of the shallowest solution s (sometimes referred to as d as well).

Depth-First Search

- *Description* - Depth-first search (DFS) is a strategy for exploration that always selects the deepest fringe node from the start node for expansion.
- *Fringe representation* - Removing the deepest node and replacing it on the fringe 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 fringe 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 $\mathcal{O}(b^m)$.



- *Space Complexity* - In the worst case, DFS maintains b nodes at each of m depth levels on the fringe. 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 $\mathcal{O}(bm)$.

Breadth-First Search

- *Description* - Breadth-first search is a strategy for exploration that always selects the *shallowest* fringe node from the start node for expansion.
- *Fringe 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 fringe. 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 fringe. 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 $1 + b + b^2 + \dots + b^s$ nodes in the worst case, since we go through all nodes at every depth from 1 to s . Hence, the time complexity is $\mathcal{O}(bs)$.
- *Space Complexity* - The fringe, 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 $\mathcal{O}(b^s)$ nodes at this depth.

Depth Limited Search – DLS

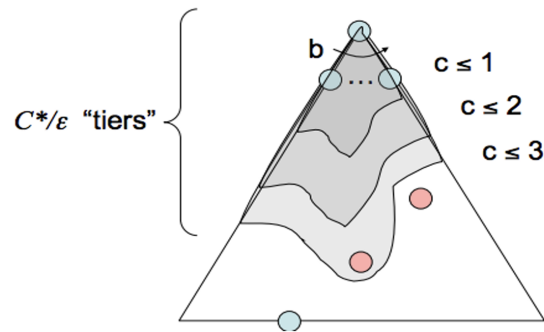
- *Description* - Perform a DFS but impose a limit ℓ that defines the maximum depth to explore. You can think of DFS is just DLS with $\ell = \infty$.
- *Fringe representation* - Just like DFS, we require a structure that always gives the most recently added objects highest priority, so a LIFO stack is used. Once a depth of ℓ is reached, the node is treated as a leaf (it has no successor states).
- *Completeness* - The max depth of the tree to be searched is ℓ . If $\ell < m$ (where m is the depth of the shallowest solution), DLS is incomplete.
- *Optimality* - DLS is not guaranteed to be optimal in the cases where $\ell > m$.
- *Time Complexity* - We must search $1 + b + b^2 + \dots + b^\ell$ nodes in the worst case, since we go through all nodes at every depth from 1 to ℓ . Hence, the time complexity is $\mathcal{O}(b^\ell)$.
- *Space Complexity* - In the worst case, DLS maintains b nodes at each of ℓ depth levels on the fringe. This is a simple consequence of the fact that once b children of some parent are enqueued, the nature of DLS 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 DLS is $\mathcal{O}(bm)$.

Iterative Deepening search(IDS)

- *Description* - Perform a DLS, and keep increasing the depth limit ℓ by one until a solution is found. This may seem wasteful, but, given that most of the nodes are at the bottom level of the tree, and that the next to bottom level is only generated twice (and the top of the tree does not contain that many nodes), it works quite well in practice.
- *Fringe representation* - Same as DLS.
- *Completeness* - IDS is complete (given a finite b). If a goal state exists, it must have some finite path, and therefore will be found. IDS addresses the infinite-path problem by always enforcing a max depth limit via DLS.
- *Optimality* - IDS is optimal IF the costs of each edge are the same.
- *Time Complexity* - For a solution a depth d , those nodes are generated once (and there are b^d of these nodes). The nodes at the prior level are generated $(2) * b^{d-1}$, and working our way back to the root node, which is generated d times. We get the series: equation $N(IDS) = d(b) + (d-1)b^2 + (d-2)b^3 + \dots + 2b^{d-1} + b^d$ equation Which gives the same big-Oh complexity, \mathcal{O}^d (the same as BFS).
- *Space Complexity* - Same as DLS, maintains at most $\mathcal{O}(bd)$ nodes (where d is the shallowest depth of a solution).

Uniform First Search

- *Description* - Uniform cost search (UCS) is a strategy for exploration that always selects the *lowest cost* fringe node from the start node for expansion.
- *Fringe representation* - To represent the fringe for UCS, the choice is usually a heap-based priority queue, where the weight 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 a runtime of $\mathcal{O}(b^{C^*/\epsilon})$.
- *Space Complexity* - Roughly, the fringe will contain all nodes at the level of the cheapest solution, so the space complexity of UCS is estimated as $\mathcal{O}(b^{C^*/\epsilon})$.

Summary on Uninformed Search

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.

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional
Complete?	Yes ^A	Yes ^A	No	No	Yes ^A	Yes ^{A,D}
Time	$\mathcal{O}(b^d)$	$\mathcal{O}(b^{1+\lceil C^*/\epsilon \rceil})$	$\mathcal{O}(b^m)$	$\mathcal{O}(b^\ell)$	$\mathcal{O}(b^d)$	$\mathcal{O}(b^{d/2})$
Space	$\mathcal{O}(b^d)$	$\mathcal{O}(b^{1+\lceil C^*/\epsilon \rceil})$	$\mathcal{O}(bm)$	$\mathcal{O}(b\ell)$	$\mathcal{O}(bd)$	$\mathcal{O}(b^{d/2})$
Optimal	Yes ^C	Yes	No	No	Yes ^C	Yes ^{C,D}

Figure 2: Figure from Russell/Norvig. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; ℓ is the depth limit. Footnote explanations: ^A - complete if b is finite; ^B - complete if step costs $\geq \epsilon$ for positive ϵ ; ^C - optimal if step costs are all identical; ^D - if both directions use breadth-first search