Harvard CS 182: Artificial Intelligence September 7, 2022

Section 1: Uninformed Search

Lecturer: Stephanie Gil *Authors: Lauren Cooke and Chris Lee*

Search Problems

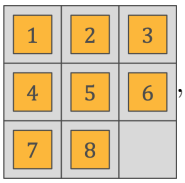
A *search problem* has the following components:

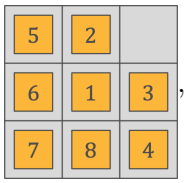
- A set of *states*, where each state represents a different possible moment in our problem world.
- A *start state*, or the state representing our problem space when we begin our search.
- At least one *goal state*, or a state that meets all the constraints of the problem which will be the end point of our search algorithm.
- A *successor function* that maps any given state to its possible successor states. For each possible successor state, the successor function also tells us which action a searcher would take to get to this next state and the cost of doing so by returning an action, successor state, cost triplet.

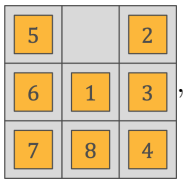
Generally, we will represent a search problem as a graph $G = (V, E)$, where each state is a node $v \in V$ and the successor function f gives the (directed) edges; in particular, there exist states $v_1, v_2 \in V$, an action a, and a cost c such that $(a, v_2, c) \in f(v_1)$ if and only if $(v_1, v_2) \in E$. This general framework will allow us to solve a wide range of different problems.

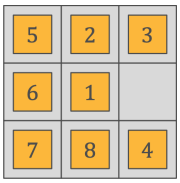
Example 1 *In the 8-puzzle, we have a 3-by-3 grid of squares. 8 of the squares are filled with tiles numbered 1 to 8, and the last is empty. During each move, the empty square can be swapped with an adjacent (not diagonal) tile. The game starts with the tiles shuffled randomly, and the goal of the game is to perform a series of moves to sort the numbers in ascending order from left to right and top to bottom.*

Consider the following four game states:

$t =$ 

$s =$ 

$u =$ 

$v =$ 

The goal state is given by t, and s is an example of a start state. Then, let $\mathcal{U}$, $\mathcal{D}$, $\mathcal{L}$, and $\mathcal{R}$ be the actions of moving a tile up, down, left, or right into the empty spot, and let our cost be the number of moves we make. The successor function f would then give

$$f(s) = \{(\mathcal{R}, u, 1), (\mathcal{U}, v, 1)\}.$$

Problem 1 *Consider the Tower of Hanoi. In this puzzle, a player is given a set of three vertical rods and n disks of different sizes. When starting the puzzle, all n disks sit on one rod ordered with the largest disk at the bottom and the smallest at the top. The goal of the puzzle is to move this stack of disks from the original rod to another rod in the same configuration by moving each disk one at a time such that a disk can never be placed on top of a smaller disk.*

Suppose we want to write an algorithm to solve this puzzle. Describe a state, start state, goal state, and successor function for this problem.

General Types of Search Algorithms

We explore two types of search algorithms: tree search and graph search. The key difference between the two algorithms is that graph search keeps track of all previously visited nodes and ensures they are never explored again.

In both search algorithm types, we will be keeping track of a frontier set, which will be all the unexplored neighbors of previously visited nodes. We will subsequently be relying on the *separation property*, which states that every path from the initial state to any unexplored state must pass through this frontier.

Tree Search

In *tree search*, we do not keep track of nodes that we have visited before, meaning that in the process of looking for a solution, we can come across and re-examine an identical node. We start with a set of frontier nodes that initially only contains the start state and loop the following steps:

1. return failure if the set of frontier nodes is empty
2. choose frontier node to expand based on some strategy
3. if a node has a goal state return the solution
4. otherwise expand node, remove it from the frontier, and add newly discovered nodes from this expansion to frontier set

Graph Search

In *graph search*, we repeat a similar process to tree search, except upon expanding a node, we add it to a separate *explored set* (in step 2). We check any node we want to add to the frontier against this explored set (in step 4) to avoid repeating an expansion later on.

Tree Search vs. Graph Search

In tree search, it is possible that we will explore exponentially or even infinitely more states before we are able to find a goal state. In particular, it is possible to get stuck

in (potentially infinite) loops of already-visited nodes. Thus, tree search performs best in tree-like structures (e.g. connected and directed acyclic graphs) where such loops do not exist.

Graph search can be applied to a wider range of search problems since it avoids the problems with tree search outlined above. However, its main bottleneck is memory – it is often necessary to keep track of exponentially many visited nodes, quickly running out of memory for large search spaces.

Problem 2 *Should the 8-puzzle be solved using tree search or graph search? What about the Tower of Hanoi?*

Uninformed Search Algorithms

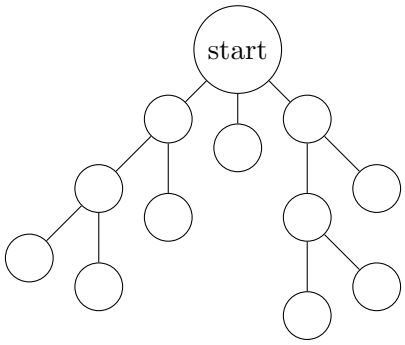
In uninformed search, we only have the information given in the problem statement about our goal state. In particular, we cannot identify any characteristics about each state other than whether or not it is a goal state. Thus, our uninformed strategies for finding a solution will systematically iterate through the problem space.

Breadth First Search (BFS)

For *BFS*, we expand the shallowest frontier node first. This algorithm is complete, as we will go through all nodes eventually. This algorithm will only be optimal if the path cost is a non-decreasing function of the depth of the graph (e.g. uniform costs). BFS can be be implemented by using a FIFO queue to store the frontier.

If there is only one goal state, then we can also implement bidirectional search, where we perform BFS starting from both the start and goal states simultaneously, stopping when their frontiers intersect.

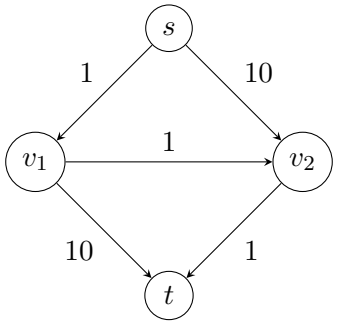
Problem 3 *Number the nodes in this graph starting from* 1 *in the order that they will be traversed by BFS given that we choose to expand the leftmost node first where applicable.*



Uniform Cost Search (UCS)

To perform UCS, we expand the frontier node with the lowest path cost first as given by a function $g(x)$ (this will be relevant for next week's section on informed search!). We apply a goal test when a node is selected for expansion, meaning we need to update the cost of nodes in the frontier. UCS can be implemented using a priority queue structure such as a heap. In algorithmic analysis, Uniform Cost Search is also known as Dijkstra's Algorithm.

Example 2 *Consider the following graph of a search problem, with start state s and goal state t:*



We seek to find the lowest-cost path from s to t. Because this search problem is represented by a directed acyclic graph, we will implement UCS as a tree search algorithm. The frontier will contain node, cost tuples, where the cost is the total cost from the start to the relevant node, and where nodes with lower cumulative cost have higher priority. Then UCS would thus proceed as follows:

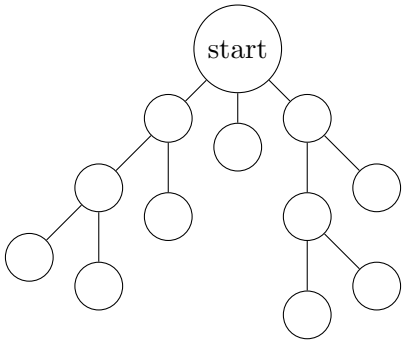
Iteration	*Current node*	*Current cost*	*Frontier*	*Goal?*
0	$\varnothing$	0	$\{(\mathbf{s}, \mathbf{0})\}$	*No*
1	s	0	$\{(\mathbf{v_1}, \mathbf{1}), (v_2, 10)\}$	*No*
2	v_1	1	$\{(v_2, 10), (\mathbf{v_2}, \mathbf{2}), (t, 11)\}$	*No*
3	v_2	2	$\{(v_2, 10), (t, 11), (\mathbf{t}, \mathbf{3})\}$	*No*
4	t	3	$\{(v_2, 10), (t, 11)\}$	*Yes*

Thus, UCS outputs the path $s \to v_1 \to v_2 \to t$ with cost 3, which we can easily verify by inspection is the path with the lowest cost.

Depth First Search (DFS)

For DFS, we expand the deepest unexpanded frontier node first. DFS can be implemented using a stack to represent the frontier or with recursion.

Problem 4 *Number the nodes in this graph starting from 1 in the order that they will be traversed by DFS, given that we choose to expand the leftmost node first where applicable.*



Iterative Deepening Search (IDS)

In IDS, we also expand the deepest unexpanded node first, but with a depth limit, where said limit increases by 1 after exploring all possible paths.

Search Algorithm Performance

We focus on four metrics with which we measure the performance of our algorithms:

- *Completeness*: Is the algorithm guaranteed to find a solution whenever one exists?
- *Optimality*: Will the algorithm find the best (lowest cost) solution?
- *Time*: What is the algorithm's running time?
- *Space*: What is the algorithm's space complexity?

We can summarize the uninformed search algorithms using these metrics as in the following table:

Algorithm	**Complete?**	**Optimal?**	**Time**	**Space**
BFS	Yes	Not Really	$\Theta(b^d)$	$\Theta(b^d)$
UCS	Sort Of	Yes	$\Theta(b^{\frac{C^*}{\epsilon}})$	$\Theta(b^{\frac{C^*}{\epsilon}})$
DFS	No	No	$\Theta(b^m)$	$\Theta(bm)$
IDS	Yes	No	$\Theta(b^d)$	$\Theta(bd)$

Note that b represents the branching factor of graph and gives number of available ends, d represents the distance from start node, C^* represents the cost of an optimal solution, ϵ represents the cost per action, and m represents the maximum depth of state space.

Problem 5 *When is UCS not complete? When is DFS not complete? When is BFS not optimal?*

Problem 6 *In Sudoku, we are given a partially filled 9-by-9 grid of numbers, as shown below. The goal state is a completed grid of numbers such that the numbers 1 through 9 each appear exactly once in each row, column, and each of the nine 3-by-3 sub-grids. To create a Sudoku solver, which search algorithm (BFS, UCS, DFS, or IDS) should we use?*

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9