COMP 560: Artificial Intelligence
February 10, 2025

Assignment 1
Due date: Mar 6, 2025

Question 1. Web Wanderings & Jar Jigsaw: Navigating Uninformed (25 points)

a). You have a set of *n* Internet webpages. Each webpage contains hyperlinks, which allow you to jump to other particular webpages. For example, assume your set of webpages are A,B,C,D and page A may contain hyperlinks to B and C. Then starting from A, you can move to B or C.

On every step, you are on a particular webpage, and you can click on a hyperlink to move to another webpage. You start on one particular webpage, and your goal is to visit each all of the *n* webpages at least (not exactly) once. *NOTE: Assume you can find such a path.*

i) (4 points) Give a formulation of this search problem by describing the state space, start state, successor function, and goal state. *NOTE: a previous version of this document said "Also, give an upper bound on the size of the state space." You do not need to do this now.*

State Space: any combination of the websites you can visit through the hyperlinks
Start State: The webpage from which the user starts
Successor Function: clicking the hyperlink to navigate to a new unvisited webpage
Goal State: visit all webpages at least once

ii) (2 points) Suppose you want to find the shortest number of steps required to solve the above problem. Will BFS or DFS be more suitable, and why?

BFS would be more suitable because it explores all possible paths level by level, ensuring that the shortest path is found first. DFS, on the other hand, may take a longer route due to backtracking, especially in cases where cycles or dead-ends exist. This makes BFS the better choice for finding the shortest number of steps to visit all web pages at least once.

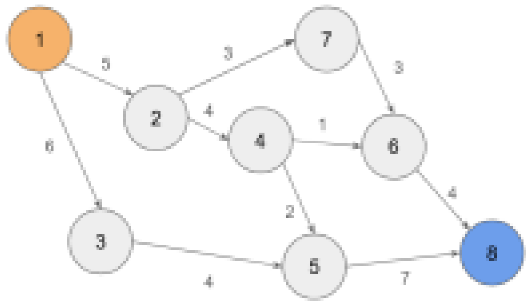
iii) (2 points) If there are a large number of webpages, then the above chosen algorithm may not work well. Explain why that may be.

If there are a large number of webpages, BFS may require excessive memory because it stores all nodes at a given level before expanding to the next. As the graph grows deeper, the number of nodes at each level increases exponentially, leading to high space complexity of $O(b^s)$. This makes BFS impractical for very large search spaces.

iv) (4 points) To avoid this problem, we could use beam search and iterative deepening search. Discuss the relative advantages and disadvantages of these algorithms.

Beam search has a set frontier size to limit the number of nodes expanded, so it can navigate large search spaces more efficiently, but it may not return the optimal result. Iterative deepening has DFS's memory efficiency and BFS's time complexity advantages, as well as efficiently finding the optimal solution to shallow graphs. The downside to iterative deepening is that it isn't efficient in very deep graphs.

b). Consider the following state-space graph with edge costs.



We wish to find a path from node 1 to node 8. Write the full list of nodes, in the order in which they are expanded, when using each of the following algorithms. Assume we use the graph search algorithms (so we do not visit states twice). Give the found path. (If multiple states are successors from an expanded node, then visit the states by increasing order. For example, if node 1 has successors 2 and 3, then we visit 2 then 3.)

i) (2 points) Depth-first search - 1, 2, 4, 5, 8 is expansion, path returned is 1, 2, 4, 5, 8

ii) (2 points) Breadth-first search - 1, 2, 3, 4, 7, 5, 6, 8 is expansion, path returned is 1, 3, 5, 8

iii) (4 points) Uniform-cost search - 1, 2, 3, 4, 5, 6, 7, 8 is expansion, path found is 1, 2, 4, 6, 8

If a graph node has multiple successors, then visit the successors in ascending order on the label. For DFS and BFS, ignore the edge costs.

c). (5 points) Formulate the following as a search problem by identifying the state space, the start state, the successor function, and the goal states: You are provided with three empty jars that have capacities for 3 cups, 5 cups, and 17 cups. Using these and a tap with an unlimited supply of water, you want to measure out 1 cup of water in a jar.

State Space - The different possible amounts of water in each jar

Start Space - 3 Empty jars and an unlimited supply of water in the tap

Successor Function - Legal state from one of the following operations: Filling a jar to full capacity, emptying a jar completely, transfer water from one jar to another until either source jar is empty or destination jar is full

Goal State: Any state where one of the jars has 1 cup of water in it

Question 2. A*stounding Heights (25 points)

a Answer the following questions about A* search, with justifications for your conclusions:

i) (3 points) Will Depth-first search always expand at least as many nodes as A* search with an admissible heuristic?

The A* search with an admissible heuristic expands nodes in order of the lowest estimated cost f(n) = g(n) + h(n), so explore the most promising paths first, whereas DFS will explore the leftmost full path to its depth limit and backtrack to try the next one until it finds a valid path (even if it's not optimal). Thus, A* expands nodes in a way that focuses on paths with lower cost estimates, whereas DFS explores blindly, so DFS will always expand at least as many nodes as A*.

ii) (3 points) A chess rook can traverse any number of squares vertically or horizontally but cannot leap over other pieces. Is the Manhattan distance an admissible heuristic for moving the rook from square A to square B in the fewest moves?

Understanding that Manhattan distance heuristic (manhattan(A, B)= | X1-X2 | + |Y1 -Y2|) calculates the sum of absolute differences in row and column positions, we can say it works well for things like king or a pawn moving step by step but the rook moves in straight lines across the board in one move. The most moves a rook needs to reach a target square is two,while the Manhattan distance can give a much higher number. Since a heuristic needs to never overestimate, Manhattan distance isn't admissible here.

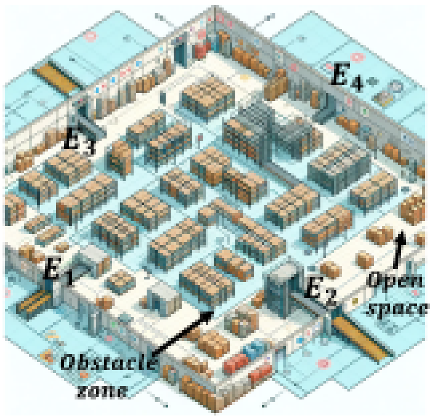
iii) (3 points) For an admissible function h, and a function $f(n) = w \cdot g(n) + (1 - w)h(n)$, where $0 \leq w \leq 1$, assess whether A* search is certain to find an optimal solution when $w = 1/2$. A* is guaranteed to find an optimal solution if the evaluation function f(n) is based on both cost-so-far (g(n)) and an admissible heuristic (h(n)) while ensuring non-overestimation of costs. The given function blends g(n) and h(n) in a weighted manner with w=1/2, ensuring a balanced combination. Since h(n) is admissible and remains a factor in guiding the search, the solution remains optimal. So yes it still works.

iv) (3 points) Does combining two admissible heuristics by taking their maximum at each node always result in an admissible heuristic? (3 points)

If h1(n) and h2(n) are both admissible heuristics, then their maximum at any node is hmax(n)=max(h1(n),h2(n)) will also be admissible. This is because neither heuristic overestimates, so taking their maximum still ensures non-overestimation. Additionally, using the maximum often results in a more informed heuristic, making A* search more efficient without sacrificing optimality.

v) (3 points) Is IDA* search guaranteed to use less memory than A* while still ensuring optimality? Yes, and that's why it exists! A* keeps every explored node in memory, which can eat up space fast. IDA* gets around this by only keeping track of the current path and re exploring nodes when needed. This makes it way more memory-efficient while still guaranteeing the best solution. The tradeoff is that it might repeat work, but at least it doesn't run out of memory like A* can on large problems.

b) Consider a multi-level warehouse where autonomous robots are tasked with retrieving items. The warehouse consists of multiple floors, each with a grid-like layout including open spaces, storage racks, and obstacles (see illustration). The robots can move in the cardinal directions (up, down, left, right) within a floor but cannot traverse diagonally or through storage racks. Movement through obstacles is possible but at half the standard speed. Vertical movement between floors is facilitated by fixed elevators at known locations. The objective is for a robot to efficiently reach a designated item retrieval location from its starting point, potentially across different floors.



Variables:

- F: Number of floors in the warehouse.
- E_k: Set of k elevators, with each elevator's location denoted as $(s_1, s_2, \ldots, s_k)$. • V : Standard speed of the robot in open spaces.
- (s_G, E_G): Location of the goal for item retrieval.

- (s_S, E_S): Starting location of the robot.
- V_E: Maximum speed of elevators (minimum time to traverse a vertical floor). **i) (3 points)** Identify constraints in the multi-level warehouse that can be relaxed to develop a heuristic for the A* algorithm.

To make A* more efficient in a multi-level warehouse, we can relax some constraints to simplify calculations. Some of them are

- Obstacle Movement Penalty: To make cost calculations simpler, reduce or ignore the penalty for moving through obstacles

- Elevator wait time: Assume that there are never any delays and that elevators are always accessible allowing instant vertical movement

- Warehouse grid structure: In order to enable smoother path estimation, treat movement as continuous rather than strictly following the grid.

- Storage Racks as Barriers: Ignore storage racks and treat them as open spaces to simplify movement calculation

- Varying speeds: assume that the robot does not adapt to different types of terrain but rather goes at a fixed pace everywhere.

We can develop an easier to calculate heuristic that still provides a reliable estimate of the shortest path by loosening these constraints.

ii) (5 points) Based on your response to part (a), propose a heuristic reflecting the constraint relaxation. Explain why this heuristic is admissible, i.e., it does not overestimate the true minimum cost to reach the goal. Provide a logical argument or mathematical proof for the admissibility of your heuristic in the context of the multi-level warehouse layout.

Based on the previous answer a good heuristic for this problem is them Manhattan Distance, adjusted for different floors, h(s)= | Sg -Ss | + |Eg - Es | + | Fg - Fs | / Ve

This formula adds up both: | Sg -Ss | + |Eg - Es | the Manhattan distance on the same floor.
| Fg - Fs | / Ve the minimum time required for vertical movement, assuming an instant elevator availability.

This is heuristic admissible because Underestimation - Property: The heuristic ignores obstacles, assuming open paths. It assumes immediate elevator access, which may not be true in reality. It does not factor in slower movement through obstacle zones. Since it underestimates the actual cost, it remains admissible.

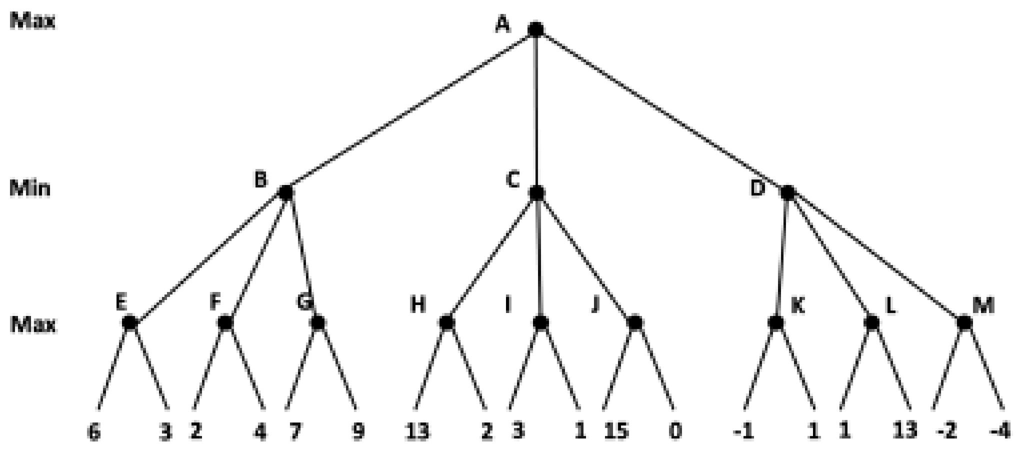
- Triangle Inequality and Consistency: Any move from one state to another adds at least the actual movement cost. The estimated cost function never overestimates the real cost, ensuring that A* remains optimal.

iii) (2 points) Discuss possible challenges of implementing your heuristic in a real-world scenario. Consider factors such as changes in the warehouse layout, varying item locations, and potential elevator wait times.

Dynamic Warehouse Layout Changes:
-Storage racks and obstacles may shift due to restocking or new item placements.
-The heuristic assumes static obstacles, which may lead to suboptimal routes in real-time.
Item Location Variability:
-The goal location may change frequently, requiring re-computation of paths.
-Implementing a dynamic heuristic that adapts to changing item locations can be computationally expensive.
Elevator Delays and Congestion:
-The heuristic assumes instant elevator availability, which may not be true in a real-world scenario.
-Robots may need to wait for elevators, leading to unexpected delays.
-A more advanced heuristic should incorporate elevator wait time estimation.
Traffic from Other Robots:
-The warehouse may have multiple autonomous robots, causing congestion in certain areas.
- A heuristic that considers dynamic traffic flow might be required for better efficiency.

To overcome these challenges, the heuristic could be improved by incorporating real time updates, traffic management, and adaptive pathfinding. However, even in its basic form, it provides a solid foundation for efficient navigation.

Question 3. Bet ya can do alpha-beta: Adversarial Search (20 points) In this question, you will work through an example of the minimax algorithm with alpha-beta pruning. Consider the game search tree in the figure. We will assume that the first player is a max player, and the given values at the root of the tree correspond to the utility of those outcomes. The first player wants to maximize the achieved utility, while the opponent wants to minimize it.



a) (5 points) Compute the minimax values for each of the nodes in the search tree (without alpha-beta pruning).

We will start from the lowest nodes and move towards the highest node. The minimax values with some work for each node are as follows:

Lowest Level (Max)	Middle Level (Min)	Highest Level (Max)
M = Max[-2, -4] = **-2**		
L = Max[1, 13] = **13**	⟶ D = Min[K, L, M] = **-2**	
K = Max[-1, 1] = **1**		
J = Max[15, 0] = **15**		
I = Max[3, 1] = **3**	⟶ C = Min[H, I, J] = **3**	⟶ A = Max[B, C, D] = **4**

H = Max[13, 2] = **13**		
G = Max[7, 9] = **9**	⟶ B = Min[E, F, G] = **4**	
F = Max[2, 4] = **4**		
E = Max[6, 3] = **6**		

b) (2 points) What is the move that the first player should choose? What is the trajectory of moves the game will move through if both players act rationally?

The first player is trying to maximize the utility of outcomes and the second player is the opponent aiming to minimize the utility outcomes. The starting node is at A and the Max player goes first. The node chosen will be B. Then it is the Min player's turn. The node chosen from this will be F and the game is now done.

In short the first move chosen is B and the trajectory of moves is as follows: A → B → F.

c) (8 points) Now, we will be using the alpha-beta pruning algorithm to traverse the game tree, and visiting successors of any node from left to right. What are the values of alpha and beta at each examined node?

Similarly to (a) we will store the final values of α and β for each node. If the node was pruned we will denote it as such.

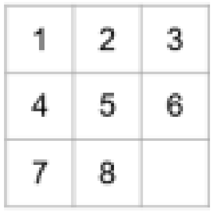
Lowest Level	Middle Level	Highest Level
M: Pruned	D: $\alpha = 4, \beta = 1$	A: $\alpha = 4, \beta = \infty$
L: Pruned		
K: $\alpha = 4, \beta = \infty$		
J: Pruned	C: $\alpha = 4, \beta = 3$	
I: $\alpha = 4, \beta = 13$		
H: $\alpha = 13, \beta = \infty$		
G: $\alpha = 7, \beta = 4$	B: $\alpha = -\infty, \beta = 4$	
F: $\alpha = 4, \beta = 6$		
E: $\alpha = 6, \beta = \infty$		

c) (5 points) In part (c), what nodes in the tree get pruned and do not need to be examined with alpha-beta pruning?

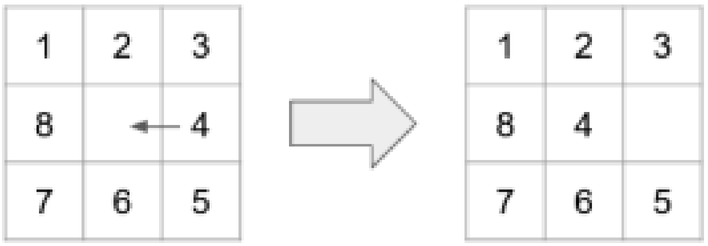
The nodes in the tree that were pruned are J, L, and M.

Question 4. Solving the 8-Puzzle (30 points)

The goal of this problem is to run different search algorithms to solve the 8-puzzle game, which we also discussed in class. In the game, there are 8 tiles numbered 1 to 8 (inclusive) and one empty tile within a 3 x 3 grid. The goal state is shown below.



On each movement, we can slide a single tile into the empty space. An example of a movement is shown below.



You will be implementing different search methods to solve the 8-puzzle. For this, you need to extend the starter code we provide in the Google Colab file at https://colab.research.google.com/drive/16CyYq9EoPCN_kdMor03Z9Y3JBMaV-Lx1?usp=sharing. There is a search Node class that can record information about the parent node, the current state (the 3 x 3 configuration) and the current depth in the search. The state should be represented as a 3 x 3 python list where 0 denotes the empty space. For example, the goal state is represented as [[1, 2, 3], [4, 5, 6], [7, 8, 0]]. A Priority Queue class has also been provided.

a). (5 points) Implement the is goal function which checks if the parameter is the goal state (returns a boolean) and implement the successors function to output the successor states as a list. If there are multiple nodes with equal priority in your search strategy, then order them with the following preference order: move the tile below the empty tile; move the tile on the right; move the tile above; move the tile on the left. You do not need to return the cost as each move is assumed to have a cost of 1.

b). (5 points) Implement the Breadth-First Search (BFS) function in BfsSearcher. For all parts of this question, you should implement the graph search version of the search algorithm. This means that a state that has already been expanded should not be expanded again (check for state repetition).

Run the algorithm on on both the *Easy* and *Hard* start states provided,and return (1) the solution returned by your search, (2) the number of expanded nodes in self.expanded nodes count, (3) the execution time of your code (the code has been provided for this part).

NOTE: You cannot insert a Python list into a Python set. Instead, you can wrap it in the provided State class type beforehand.

c). (5 points) Implement the Iterative Deepening Search (IDS) function. Run the algorithm on the example start states, and return (1) the solution returned by your search, (2) the number of expanded nodes in self.expanded nodes count, (3) the execution time of your code.

NOTE: With IDS (and for any depth-limited search), we need to take special care when looking at state repetitions. This is because even if we have expanded a state at a deeper depth, we might still want to expand it again if we find a path to it at a shallower depth. Thus, when adding a node to the closed set, store the corresponding depth value in a map. When expanding a node, you must check if the state has been visited before and the current depth is less than the stored depth. If so, then expand the node and update the depth value stored in the map.

NOTE: If a node has been expanded multiple times, you should increase the counter every time a node has been expanded.

d). (10 points) Now, we will implement A* search. For this, you first need to implement the Manhattan distance and misplaced tiles heuristics for the 8-puzzle game that were discussed in class. Then, implement the A* search function and run A* search using each heuristic. For each case, return (1) the solution returned by your search, (2) the number of expanded nodes in self.expanded nodes count, (3) the execution time of your code

e). (5 points) Comment on the relative performance of BFS, IDS, and A* Search using both heuristics.

The number of nodes expanded by BFS and IDS grows exponentially with the depth of the solution, quickly making them very expensive for deeper puzzles. On the other hand, A* with an admissible heuristic, particularly Manhattan distance, greatly reduces the number of node expansions. Even though Manhattan distance performs better than the Misplaced Tiles heuristic, both versions of A* still outperform BFS and IDS by a notable margin.

Please share your Google Colab link (make sure permissions are set to everyone with the link) or upload the ipynb file as an assignment solution. On Google Colab, you can download the ipynb by clicking File / Download / Download .ipynb.

Colab Link:
https://drive.google.com/file/d/1VZoICqUpF2LPWNUTbQaJKLKsss92Vjv2/view?usp=sharing
Github Repo:
https://github.com/kensho-pilkey/560Homework1

BONUS: For the 8-puzzle problem, is it possible to reach the goal state from any start state? Briefly justify your answer.

We will show that it is not possible to reach the goal state from any start state. Please note that if it is possible to go from one start state to the goal state then it must also be possible to go from the goal state to that start state. Due to this, we will show that from the goal state it is not possible to reach any random start state, which is equivalent to the original statement.

To do this, consider the goal state shown to the right. Now for each square systematically check how many times the relative order is broken when comparing to the squares to the left of the current square. We will call the total number of times for the puzzle the score of the puzzle.

1	2	3
8	4	
7	6	5

Demonstration

We will demonstrate this algorithm for the puzzle on the top right. We will start from the top left to bottom right. (Square 1) there are no squares to the left of it, so the score is 0. (Square 2) the square to the left is 1 but the relative order is kept because 2 should come after 1, so the score is still 0. (Square 3) same as square 2, so the score stays 0. (Square 4) The relative order compared to 1, 2, and 3 is kept because 8 comes after those numbers, so the score is still 0. (Square 5) The relative order is kept with respect to 1, 2, and 3, but 4 does not come after 8 so the score is now updated to 1. (Square 6) This is blank go to the next one. (Square 7) The relative order is kept with respect to every square before it besides Square 4 so the score is updated to 2. (Square 8) The relative order is broken with respect to Square 4 and Square 7 so the score is now updated to 4. (Square 9) The relative order is broken with respect to Square 4, Square 7, and Square 8, so the final score is 7.

Back to Justification

Now note that for the desired final state of a puzzle the score will necessarily be 0 because the relative order is never broken. Additionally note that when you make a valid swap (i.e swapping the blank square with a square to the left, right, top, or bottom to it) there are only a few outcomes regarding the score of the puzzle after the swap. If the swap occurs horizontally (left or right) the score will not change because no relative orders change when you swap with a blank row to the left or right. However, if we do a vertical swap there are 3 potential outcomes: no change in score, score+2, score-2. These are the only outcomes because when you swap vertically there are only 2 tiles in between the original location and the new location. Thus we have these cases: the numbered tile moves down and it now comes after 2 tiles that it previously messed up (-2), the numbered tile moves up and it now comes before 2 tiles that it previously didn't mess up, or the move either up or down causes one tile to get messed up and one gets fixes (+0).

Now we will use the fact that if it is possible to go from one start state to the goal state then it must also be possible to go from the goal state to that start state. This is useful because we know that at the goal state the score is 0, and since any valid swap changes the score by either 0 or 2, any state that follows from valid moves from the goal state must have a score that is even. As a result, we can conclude that we can not reach the goal state from a start state that has an odd score. Note we know an odd score is possible from the demonstration above.