CSE 415 Spring 2026 Assignment 3

Last name:_____ First name:_____ UWNetID: _____

Due Wednesday night April 22 via Gradescope at 11:59 PM. You may turn in either of the following types of PDFs: (1) Scans of these pages that include your answers (handwriting is OK, if it's clear), or (2) Documents you create with the answers, saved as PDFs. When you upload to Gradescope, you'll be prompted to identify where in your document your answer to each question lies.

Do all four exercises. These are intended to take 30-60 minutes each if you know how to do them. Each is worth 25 points. If any corrections have to be made to this assignment, these will be posted in ED.

This is an *individual-work* assignment. Do not collaborate on this assignment. You may use AI to assist you with this assignment. If you do, you must explain why in your learning diary entry (LDE) for this assignment. Also, read the AI prompting guidelines in ED.

Whether you use AI or not, submit a learning diary entry (LDE) at the end of your PDF. Choose 2 of the 4 A3 questions to focus on within your learning diary entry answers. Identify those two questions in the first section of the LDE ("Goals"). If you used AI, include your reasons in this section, too. In each section (including the Goals section), use two separate paragraphs to represent your answer to that item for each A3 question you selected. Points will be deducted for missing or insubstantial answers, up to half the points for each of the two A3 questions you have chosen to focus on. If you do not use AI, then you can omit including prompts and other LLM details.

Prepare your answers in a neat, easy-to-read PDF. Our grading rubric will be set up such that when a question is not easily readable or not correctly tagged or with pages repeated or out of order, then points will be deducted. However, if all answers are clearly presented, in proper order, and tagged correctly when submitted to Gradescope, we will award a 5-point bonus. (Thus the maximum points for A3 will be 105).

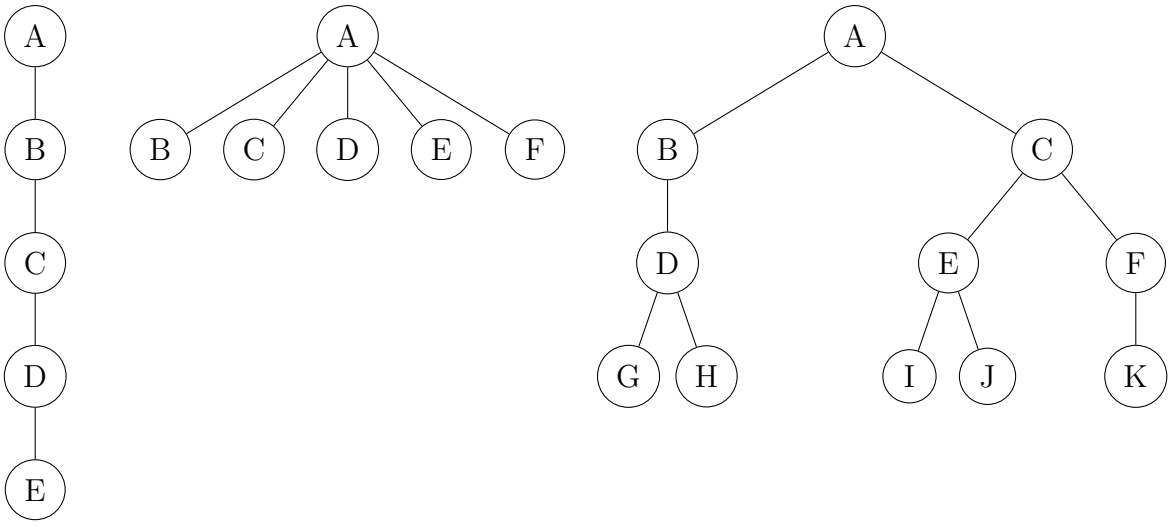
If you choose to typeset your answers in Latex using the template file for this document, please put your answers in blue while leaving the original text black.

Version of 26-04-15. Any updates to this document this will be announced in ED.

1 Basic Search

Use the following tree to answer the questions below comparing Breadth First Search, Depth First Search, and Iterative-Deepening Depth First Search. Assume that children are visited from left to right.

Tree 1: **Tree 2:** **Tree 3:**



Please use **Tree 3** to answer questions (a - c).

(a) (2 points) Write out the order that the nodes are expanded in using Breadth First Search, starting from A and searching to J.

(b) (2 points) Write out the order that the nodes are expanded in using Depth First Search, starting from A and searching to J.

(c) (2 points) Write out the order that the nodes are expanded in using Iterative-Deepening Depth First Search, starting from A and searching to J. If a node is repeated, make sure to include it each time it is expanded.

(d) (2 points) How does increasing the depth d affect the maximum open list size for BFS versus DFS, assuming a fixed branching factor b?

(e) (2 point) How does the total number of nodes expanded by IDDFS compare between **Tree 1** and **Tree 2** when searching for node E? What property of each tree causes this difference?

(f) (2 points) How does the maximum open list size for BFS compare between **Tree 1** and **Tree 2**? What property of each tree causes this difference?

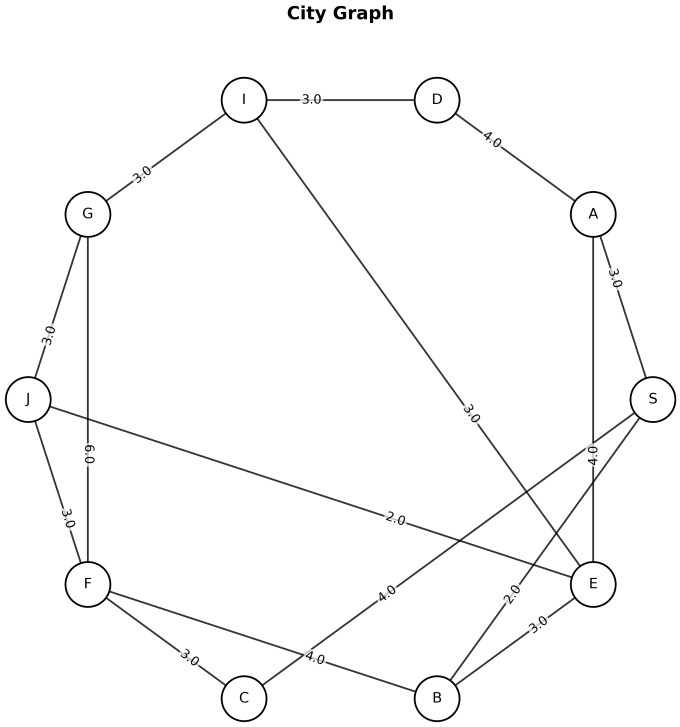
(g) (1 point) (True/False) IDDFS has a worse asymptotic runtime than BFS because it re-expands nodes at every iteration.

(h) (4 points) What are the advantages of using IDDFS over DFS and BFS for very large or infinite graphs in terms of (i) memory usage, (ii) completeness, and (iii) computation time?

(i) (3 points) Suppose you are searching a tree where the goal might be at any depth, including very deep or potentially infinite depth. Which of the three search algorithms (BFS, DFS, IDDFS) would guarantee to find the goal if it exists? Justify your answer with a 2 - 3 sentence explanation.

2 A* Operation

(25 points) This exercise is intended to reinforce your understanding of the mechanics of graph-search algorithms with and without heuristics. Consider the following graph, which represents cities and the distances between them. Each circle represents a city, and the edges between them represent roads from one city to another. The number next to an edge indicates the travel distance along that edge. For example, in the graph, the edge from city S to city A is labeled with 3, which means the travel distance from city S to city A is 3 units.



The problem is to find a lowest-distance path from city S to city G. You'll solve this problem first using Uniform Cost Search (UCS) and then solve it again using A* Search. For the A* search, you will find your own heuristic values to satisfy the given conditions.

(a) **Uniform Cost Search (UCS)**: Apply UCS to find a lowest-distance path from city S to city G.

- (i) Show how the OPEN list (a.k.a. the "fringe") evolves during this search below, starting with the version before the beginning of the first iteration of the UCS main

loop that is shown already with starting city S and its cumulative distance 0. Show the new version of the OPEN list at the end of each iteration. Be sure to include each city's cumulative distance (g-value). Within each list, show these pairs in order of increasing distance (even though a real priority queue doesn't have to use a sorted list in its implementation). Breaking any ties using alphabetical ordering (3 points)

[(S, 0)]

- (ii) List the nodes in the order they are added to the CLOSED list (1 point)

- (iii) Provide the final lowest-distance path found and its total distance (1 point)

(b) **A* Search**: Apply A* search with your own heuristic values to satisfy each of the conditions.

- (iv) Devise non-admissible heuristic values that **lead to a FAILURE to return an optimal path** (4 points).

city s:	S	A	B	C	D	E	F	G	I	J
heuristic $h(s)$:										

- (v) Show how the OPEN list (a.k.a. the "fringe") evolves during this search, but use the cities' f values instead of their g values. As with UCS, show each snapshot of the OPEN list with cities ordered by associated values, and break any ties using

alphabetical ordering. (2 points)

[(S,)]

- (vi) List the nodes in the order they are added to the CLOSED list (1 point)

- (vii) Provide the final path and distance (1 point)

- (viii) Devise non-admissible heuristic values that **still outperform UCS in efficiency AND return an optimal path** (4 points).

city s:	S	A	B	C	D	E	F	G	I	J
heuristic $h(s)$:										

- (ix) Show how the OPEN list (a.k.a. the "fringe") evolves during this search, but use the cities' f values instead of their g values. As with UCS, show each snapshot of the OPEN list with cities ordered by associated values, and break any ties using alphabetical ordering. (2 points)

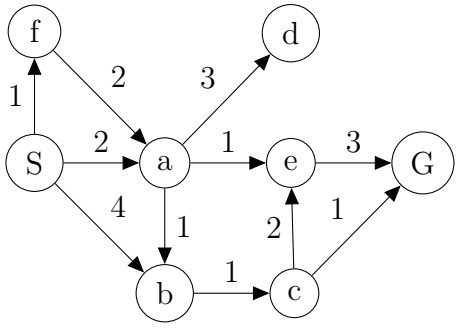
[(S,)]

- (x) List the nodes in the order they are added to the CLOSED list (1 point)

- (xi) Provide the final path and distance (1 point)

- (xii) Explain why these non-admissible heuristic values can achieve better efficiency than UCS and find an optimal path? (4 points)

3 Properties of Heuristics



For the following questions, consider three heuristics h_1, h_2, h_3. The table below indicates the estimated cost to goal, G, for each of the heuristics for each node in the search graph.

state (s)	S	a	b	c	d	e	f	G
heuristic $h_1(s)$	4	3	3	2	4	2	6	0
heuristic $h_2(s)$	4	3	2	1	1	1	1	0
heuristic $h_3(s)$	5	3	2	1	6	3	5	0

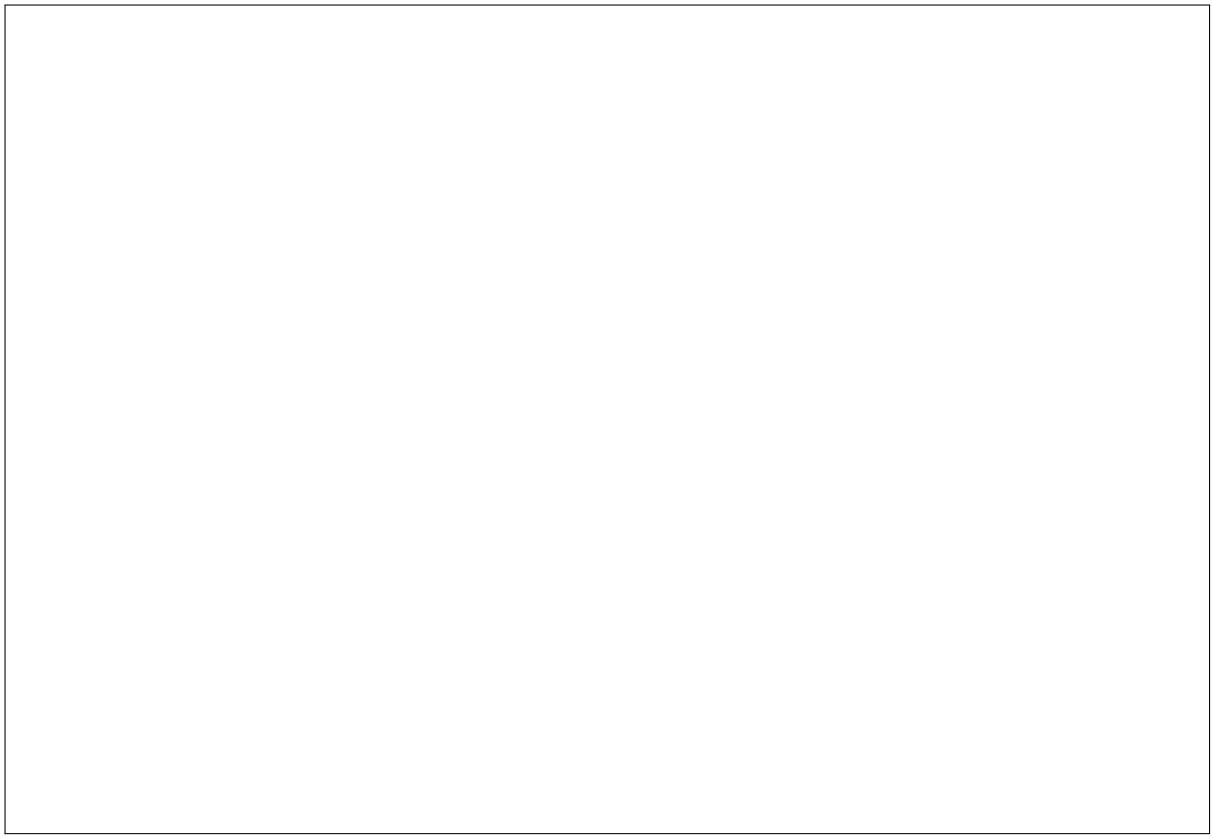
(a) (5 points) Consider heuristic h_1 - is this heuristic "admissible"? If **yes**, explain why. If **no**, explain why not and change only the values needed to make the heuristic admissible.

(b) (5 points) Consider heuristic h_2 - is this heuristic "consistent"? If **yes**, explain why. If **no**, explain why not and change only the values needed to make the heuristic consistent.

(c) (5 points) Consider your revised h_1 and h_2 heuristics, along with heuristic h_3. Which of these three heuristics would you select as the "dominant" heuristic and why?

(d) (5 points) Creating a "good" heuristic is something of an art. Typically, a number of factors need to be considered and sometimes compromises need to be made. So far in this question, we have just worked with the heuristic values given without taking the cost of coming up with those values into account. For this question, consider that each increase of 1 in the value of the heuristic requires 1 hour of calculation time (assume each heuristic value starts at 0). Unfortunately, you can only afford to spend 20 hours on determining the heuristic values. Which heuristic (if any) would you choose, and why?

(e) (5 points) Imagine a search graph that is much larger than the small example we've been considering so far. You have several heuristics at your disposal: **heuristic a** can only be trusted to quickly get within 75% of the true cost before it starts producing overestimates, **heuristic b** can get very close to the true cost, but each additional percentage point closer takes incrementally more calculation time, **heuristic c** can get quickly calculate heuristic values that get within 90% of the true cost, but tends to generate heuristics that have issues with consistency. Which heuristic do you think would be the best one to use and what are some of the trade-offs you'd consider?



4 Adversarial Search and Alpha-Beta Pruning

Consider the Tic-Tac-Toe game tree shown in the figure.

(a) (4 points) Compute the static values of the leaf nodes in the given tree using the function $v(s)$, where v is defined below, as a weighted sum of three features.

$$v(s) = 1 \cdot f_1(s) + 10 \cdot f_2(s) + 100 \cdot f_3(s)$$

Here $f_1(s)$ counts the number times that there is a line containing an X alone minus the number of times there is a line containing an O alone. Also, $f_2(s)$ counts the number of unblocked instances of two Xs in a row, minus the number of unblocked instances of two Os in a row. Finally, $f_3(s)$ is the number of 3-in-a-row instances of Xs, minus the similar number of Os. (This is equivalent to the static evaluation function used in lecture and a worksheet.)

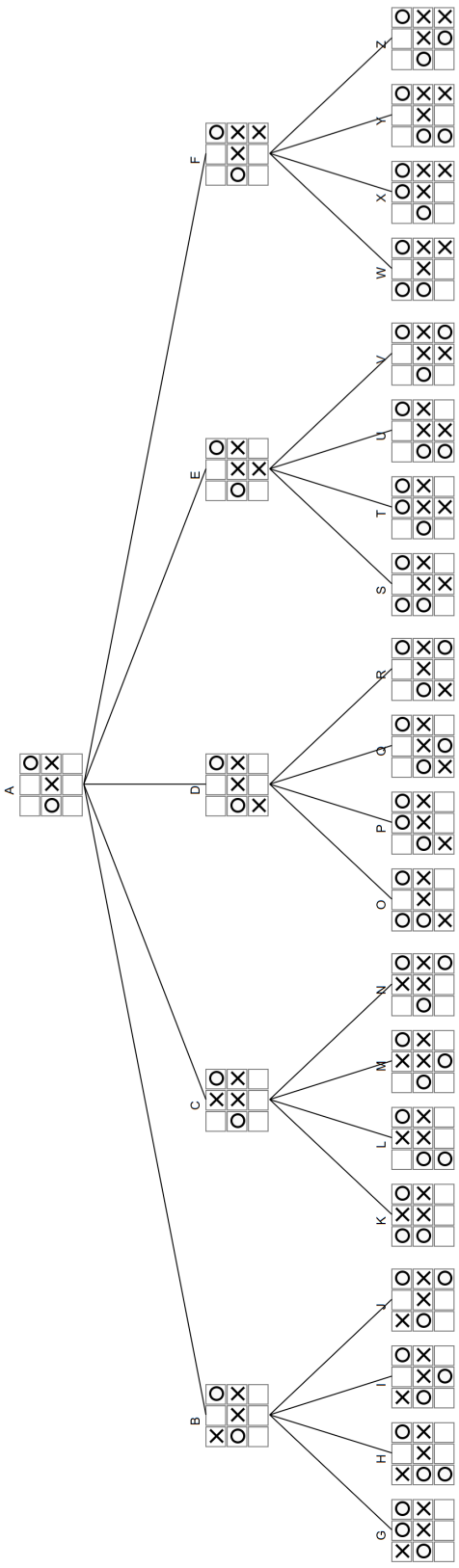
Show these values underneath states G through Z in the diagram.

(b) (3 points) Use straight minimax to determine the values of the internal nodes in the tree, and write those values in the table below.

node	Minimax value
A	
B	
C	
D	
E	
F	

Table 4.1: Minimax values for internal nodes

(c) (5 points) Next, redo the adversarial search using alpha-beta pruning on the same tree. Fill in the values in Table 4.2 below as you go. For α and β values, write the values that are passed from the state's parent to that state. If a state does not have to be evaluated, do not write any values for it in the table.


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z

state	value	α	β
A		N/A	N/A
B			
C			
D			
E			
F			
G			
H			
I			
J			
K			
L			
M			
N			
O			
P			
Q			
R			
S			
T			
U			
V			
W			
X			
Y			
Z			

Table 4.2: State values and parameter values for α and β.

(d) (4 points) Suppose that O knows that X moves randomly choosing among the legal moves with equal probability for each choice. In order for O to choose the best move, O decides to choose the move corresponding to the minimum of the "expected values" of the resulting states. Redo the computation of backed up values of the internal nodes from (b) by using the statistical expectation of the values of states where it is O's turn to move, instead of the maximum value of its children. For each internal node B through F, show how you compute the expected value from its children's values. Assume the uniform probability distribution for four items: $[1/4, 1/4, 1/4, 1/4]$. Put your expressions and values in Table 4.3.

node	expression	expected value
B		
C		
D		
E		
F		

Table 4.3: Expected values for chance nodes.

(e) (4 points) **Ordering for Alpha-Beta cutoffs.** One way to improve the likelihood of cutoffs in Alpha-Beta search is to statically evaluate a node's successors *before* expanding them, and then expand in the order most likely to trigger a cutoff. Specifically: at any node being expanded, compute the static value $v(s)$ of all successors, *except* that if a successor is a leaf and produces a cutoff, do not statically evaluate its remaining siblings.

Let each call to $v(s)$ cost one "evaluation unit." For the tree in the figure, assume successors are ordered so that the child with the *largest* v is explored first at MAX nodes, and the child with the *smallest* v is explored first at MIN nodes (ties broken by the original left-to-right order).

(i) List the order in which the children of A (i.e., B, C, D, E, F) and the children of each of B, C, D, E, F (i.e., the leaves G–Z) are visited under this ordering scheme.

(ii) Re-run alpha-beta pruning using this new visitation order and report the *total* number of static evaluations performed (including those needed for the pre-expansion orderings), compared to the plain left-to-right alpha-beta from part (c). Does the ordering reduce the total work once the cost of computing v is included? Justify briefly.

(f) (3 points) **Static values as an ordering heuristic at INTERNAL nodes.** At an internal (non-leaf) node, we do not know the true backed-up minimax value until after its expansion. Describe, in 3–5 sentences, how you would use the static evaluation function $v(s)$ to *order* the successors of an internal node for expansion. State one advantage and one disadvantage of using v (rather than a more expensive look-ahead) for this purpose. Your answer should reference the tree above. For node A, which of its children (B, C, D, E, or F) would be expanded first, under your scheme, and why?

(g) (2 points) **Transposition table for Tic-Tac-Toe.** In Tic-Tac-Toe, many distinct move sequences lead to the same board state, so the same state s can be visited many times across the lookahead trees of different turns. To avoid recomputing $v(s)$ from scratch each time, maintain a *transposition table*: a key–value store whose **keys** are states the search has already evaluated and whose **values** are pairs ($v(s)$, d), where $v(s)$ is the

value previously assigned to s and d records the *depth-quality* of that value:

$d = 0$ if $v(s)$ is purely the static evaluation;
$d = 1$ if $v(s)$ is the best-of-children (1 ply of look-ahead);
$d = k$ if $v(s)$ was backed up from k plies of look-ahead.

Importantly, the table stores only *values* for states that have already been examined; it does not store the future states themselves, and it does not change which states the search expands. Whenever the search reaches a state s that already has an entry in the table, it consults the entry to decide whether the cached $v(s)$ is good enough to use as-is, or whether s should be reconsidered to obtain a higher quality result (i.e., based on more ply of look-ahead).

(i) Describe the rule your search should use, when it encounters a state s with a cached entry (v, d), to decide whether to *reuse* the cached value or to *recompute* it at higher quality. Your rule should depend on the ply budget remaining from s on the current turn.

(ii) On turn t, suppose the agent's search explores some hypothetical position s^* (a state several plies below the current state at turn t), backs up a value for s^* from 2 plies of further look-ahead, and stores (s^*, v, d=2) in the transposition table. The agent then makes its move and the opponent replies, advancing the actual state to turn t+1. During turn t+1's search, the earlier-seen state s^* is visited again as a descendant of the new root, and the ply budget remaining from s^* downward is now 3. (Note: s^* is not the current state at turn t; it is a state inside turn t's search tree, which is why the same s^* can naturally reappear inside turn t+1's search tree – typically at a shallower depth, since one real move has been played.) What does your rule from (i) do in this situation, and why?