COSC 4550/5550 - Introduction to Artificial Intelligence
Fall 2023
University of Wyoming
Homework #1

Due: Saturday, September 23, 2023, 11:59 p.m.

Instructions: Answer all of the following questions. This homework has two parts: a written part and a programming part. For the written exercises, prepare a PDF document named **written.pdf** with your answers. If you show your work there is a chance for partial credit. Feel free to use a calculator (or better yet, the Python interpreter!) to compute the final numeric answers. Show at least 4 decimal places in your answers. Be sure to use floating point division in the Python interpreter (one way to make sure is write things like 1./2 instead of 1/2).

For the programming part, you will answer it by completing the implementation in code_hw1.py. To test that you implemented it correctly, you can run **tests_hw1.py**. There will also be other tests, not included in **tests_hw1.py**, which will also be used to grade your submission. When you are finished, submit the completed files **written.pdf** and **code_hw1.py**.

Part A: Written instructions

1. Suppose **Algorithm 1** below is used to generate a search tree. Suppose each tree node has at most b children, and that when the algorithm terminates, the deepest leaves of the tree are at depth D, where the initial node is at depth $d = 0$, its children are at depth $d = 1$, etc. Use the numeric values of b and D assigned to your NetID for question 2 in **individualized.pdf**.

   (a) If a FIFO (first-in, first-out) queue is used, then in the worst case, what is the largest number of nodes simultaneously stored in the queue at any given time?

   (b) Repeat the previous question if a LIFO (last-in, first-out) queue is used instead of FIFO.

   (c) Whichever type of queue is used, in the worst case, what is the largest number of nodes popped off the frontier before the algorithm terminates?

Algorithm 1 Basic Queue search

```
explored ← {}
frontier ← empty queue
Push initial state node on to frontier
while Frontier is not empty do
    node ← pop(frontier)
    if node state is a goal then exit
    explored ← explored ∪{node state}
    for Each child state of node state, in alphabetical order do
        if child state ∉ explored then
            Push child node on to frontier
        end if
    end for
end while
```

2. Suppose you run a uniform cost search on a graph where each step costs at least ε, each node has at most b children, and the cost of the optimal path is $C*$. Look up the values of ε, b, and $C*$ for your NetID in **individualized.pdf**. What is the worst-case asymptotic running time in your case?

3. Consider the following puzzle called "Moving Magic Square". It is played on a 3×3 table containing each of the numbers 1 to 9. The number 9 is the "movable number". You can move 9 in four directions (up/down/left/right), and swap 9 with the number in that direction. The initial state is shown in the following table. As the player, we want to move 9 to reach a final state such that the sum of the three numbers on every row, column, and diagonal is 15. There are multiple states that satisfy this condition, and you can stop your answer when you find the first satisfied state. (Hint: You can define the operations in order of up/down/left/right, which might be helpful for fewer steps to reach the satisfied state.)

6	9	8
7	1	3
2	5	4

   (a) Formulate this as a search problem. What are the states, actions, initial state, and goal condition?
   (b) Is this state space a graph or a tree?
   (c) Draw the portion of the state space that would be generated by the Breadth-First Search (BFS) procedure and mark the order in which each state is expanded with the numbers 1, 2, 3... Do not create multiple copies of states that are identical, and identify the states that are goal states.
   (d) Draw the portion of the state space that would be generated by the Depth-First Search (DFS) procedure and mark the order in which each state is expanded with lower-case letters a, b, c... Do not create multiple copies of states that are identical, and identify the states that are goal states.
   (e) Draw the portion of the state space that would be generated by the Iterative Deepening Search (IDS) procedure and mark the order in which each state is expanded with upper-case letters A, B, C... Do not create multiple copies of states that are identical, and identify the states that are goal states.

Part B: Programming

4. For this question, you will finish the implementation of code_hw1.py. This question will perform a depth-first search on a variant of tic-tac-toe. In this variant, the players never stop until the board is completely full. In addition, there are different scores for different types of three-in-a-row, according to four weights: w_0, w_1, w_2, and w_3. The weights for your NetID are listed in order in **individualized.pdf**. Specifically:

   – For every column where player X has three x marks, the value for player X goes up by $+w_0$. For every column where player O has three o marks, the value for player X goes down by $-w_0$.

   – For every row where player X has three x marks, the value for player X goes up by $+w_1$. For every row where player O has three o marks, the value for player X goes down by $-w_1$.

- If Player X has three x marks on the diagonal from top-left to bottom-right, the value for player X goes up by $+w_2$. If Player O has three o marks on the diagonal from top-left to bottom-right, the value for player X goes down by $-w_2$. Assume that at every turn, both players choose actions uniformly at random.

- If Player X has three x marks on the diagonal from bottom-left to top-right, the value for player X goes up by $+w_3$. If Player O has three o marks on the diagonal from bottom-left to top-right, the value for player X goes down by $-w_3$.

Assume that at every turn, both players choose actions uniformly at random.

- Implement the **score** function using the weights assigned to your NetID in **individualized.pdf**.

- Modify the **dfs** function, whose inputs are the current game state and player, so that it returns a tuple with two elements as output. The first element should be the number of leaves below the current game state where the score for player X is strictly greater than 1. The second element should be the expected final score, when starting game play from the current game state.