Artificial Intelligence
Take home Quiz 01

NAME:
DATE:

This homework quiz is meant to help you prepare for the mid-term exams. The mid-term will cover concepts from Agents, Environments, Search (both informed and un-informed), Adversarial Search and Constraint Satisfaction Problems..

1. Prove each of the following statements, or give a counterexample (6 points)
   a. Breadth-first search is a special case of uniform-cost search.

   When all step costs are equal, g(n) $\propto$ depth(n), so uniform-cost search reproduces breadth-first search.

   b. Depth-first search is a special case of best-first tree search.

   Breadth-first search is best-first search with f(n) = depth(n);
   depth-first search is best-first search with f(n) = −depth(n);
   uniform-cost search is best-first search with f(n) = g(n).

   c. Uniform-cost search is a special case of A∗ search.

   Uniform-cost search is A∗ search with h(n) = 0

2. Decide whether the following statements are true or false. If true, explain why. If false, give a contradicting example. Recall that B is the average branching factor and L is the length of the shortest path from start to goal. (8 points)

   a) Bi-directional BFS is always faster than BFS when B >= 2 and L >= 4.

   False.
   Consider searching on a tree from the leaf to the root, and transitions always move towards the root. BFS is fast because branching factor is always 1, while BIBFS waste time on the half of the reverse part tracing a lot of branches.

   Note that time complexity is not the same as actual timing or number of nodes expanded. O (...) are worst case, and under a constant factor. On the other hand, in cases like A being more efficient than any optimal search given a heuristic, we really mean ALWAYS -- it has been proved that for any search space no other optimal search can expand fewer nodes. Know the difference. Also note that*

treating B as a constant is just an approximation, since actually the branching factor is different between nodes. Finally, you need to understand the assumptions underlying the analysis that BIBFS is faster than BFS.

b) A* search always expands fewer nodes than DFS does.

False.
No optimal search algorithm can be more efficient than A, but DFS is not optimal and it can be lucky in searching goals. For example, on the Sudoku search problem, DFS almost always expand fewer nodes than A*, because A* need to expand L-1 levels.*

c) For any search space, there is always an admissible and consistent A* heuristic.

True. For example h (s) = 0.

d) IDA* does not need a priority queue as in A*, but can use the program stack in a recursive implementation as in DFS.

True. Note that the inner loop in IDA is DFS, not A*.*

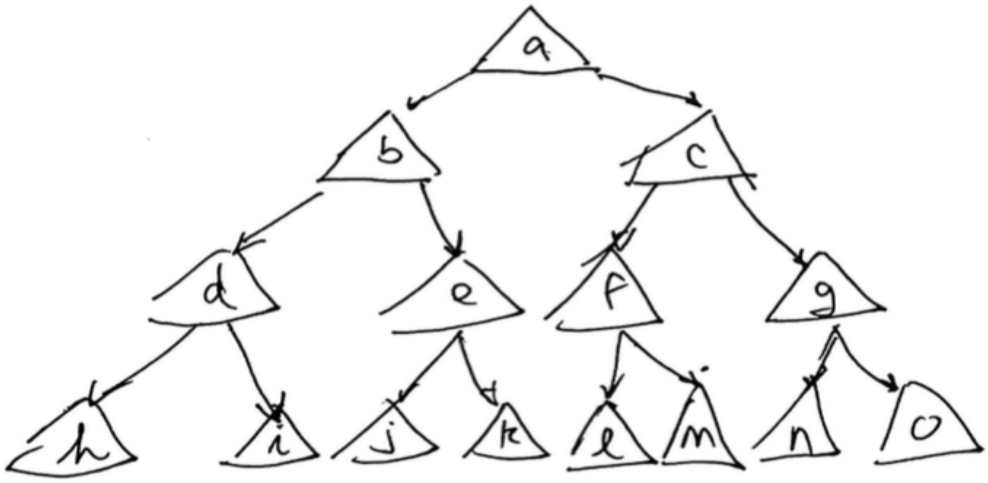
3. Tree search algorithm
Tree search algorithm is applicable to searches where there is no worry about re-visiting prior states. It is simpler than graph-search because there is no need to maintain the closed set (of previously visited states). Use tree search to solve the following problems

NOTES
"The intuitive idea behind generic search algorithm, given a graph, a set of start nodes, and a set of goal nodes, is to incrementally explore paths from the start nodes. This is done by maintaining a **frontier** (or **fringe**) of paths from the start node that have been explored. The frontier contains all of the paths that could form initial segments of paths from a start node to a goal node. Initially, the frontier contains trivial paths containing no arcs from the start nodes. As the search proceeds, the frontier expands into the unexplored nodes

until a goal node is encountered. To expand the frontier, the searcher selects and removes a path from the frontier, extends the path with each arc leaving the last node, and adds these new paths to the frontier. A search strategy defines which element of the frontier is selected at each step."

Refer to the reference books (Artificial Intelligence: Foundations of Computation Agents - Chapter 3 (It was in our reading materials))

A. Breadth-first search using FIFO queue



Frontier (fill from L–R):

Using the tree search algorithm, perform the search for **goal state *o*** starting from initial state *a*.

Expand new state/action pairs from left to right. Draw nodes in the Fringe as they are expanded. When they get removed for goal-testing, cross them out. (4 points)

Draw nodes as a circle with a letter inside. This is to visually distinguish them from states (triangles).

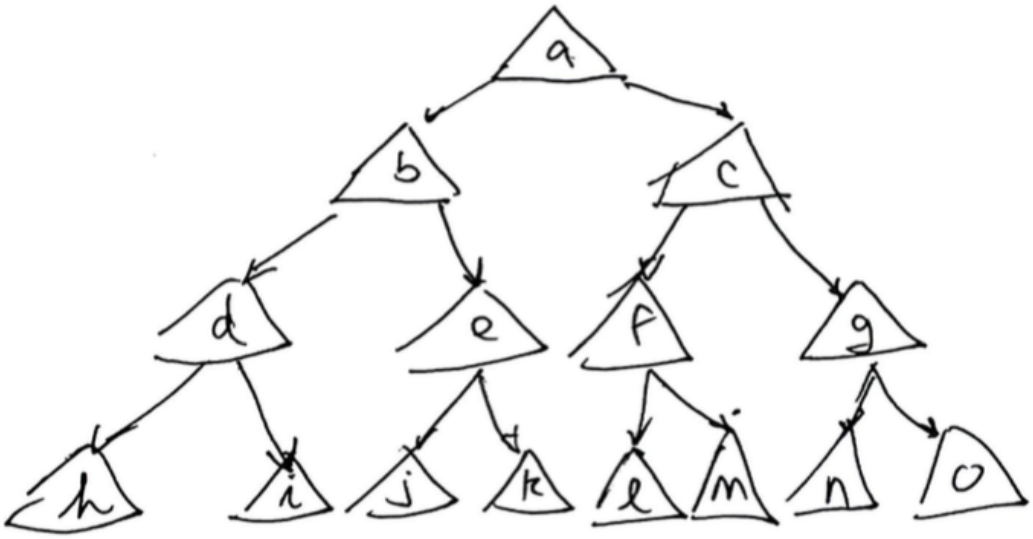
Treat the Fringe as a FIFO queue. This should produce breadth-first search.

a. What goal node is returned by the algorithm? (2 points)

b. How many states were expanded in total? (2 points)

c. At most, how many nodes were actively in the fringe at one time during the algorithm? (2 points)

B) Depth-first search using LIFO stack



.

Frontier (fill from L–R):

Using the tree search algorithm, perform the search for **goal state *h*** starting from initial state *a*.

Expand new state/action pairs from left to right. Draw nodes in the Fringe as they are expanded. When they get removed for goal-testing, cross them out. (4 points)

Draw nodes as a circle with a letter inside. This is to visually distinguish them from states (triangles).

Treat the Fringe as a LIFO stack. This should produce depth-first search.

When done:

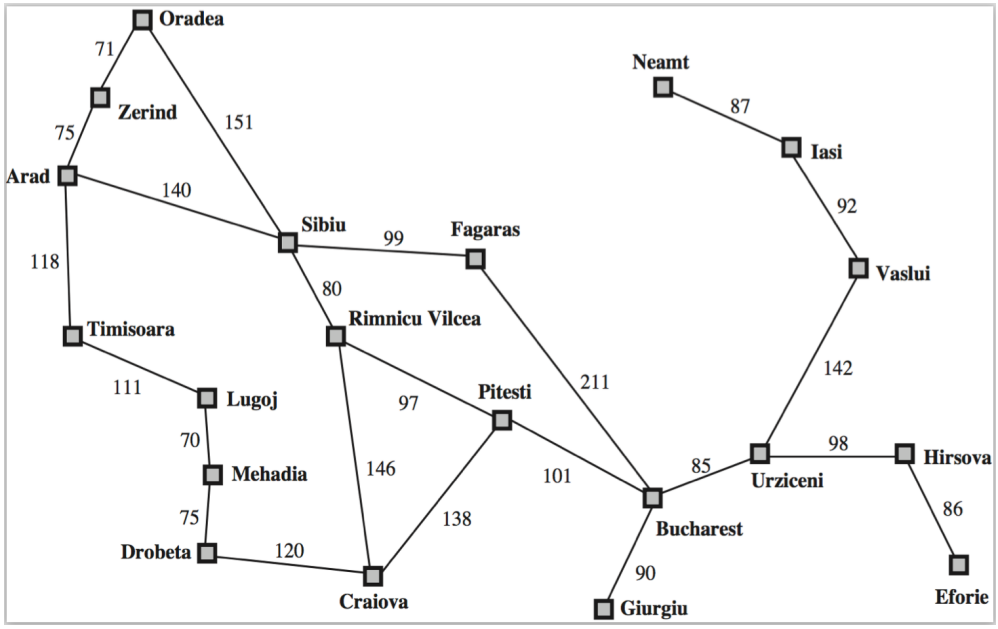
a. What goal node is returned by the algorithm? (2 points)

b. How many states were expanded in total? (2 points)

c. At most, how many nodes were actively in the fringe at one time during the algorithm? (2 points)

4. Uniform cost search

Uniform cost search (UCS) is a breath-first search where the node to be expanded is the one with the lowest accumulated path cost in the frontier. Let's use it to search for the best path from Arad to Giurigu. "Best path" is defined as

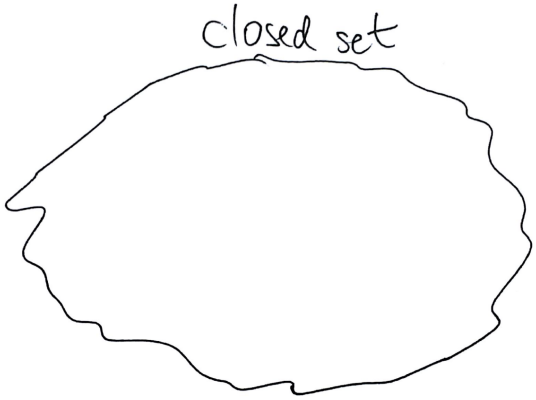


shortest distance.

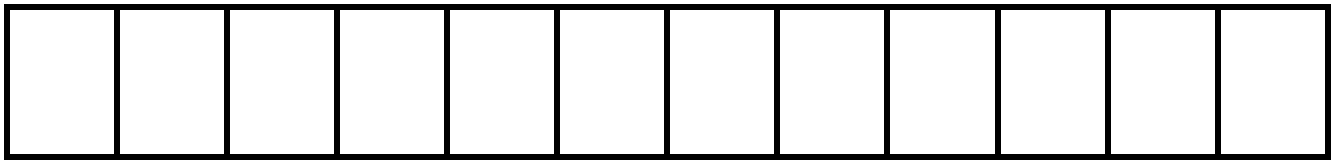
We will use **Graph Search**

```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

Uniform Cost Search using priority queue



Frontier (fill from L–R):



Start state is Arad.
Goal state is Giurgiu.

Draw nodes in the Fringe as they are expanded. When they get removed for goal-testing, cross them out and put them in the closed set as states.

Draw nodes as a circle with a letter inside, and include the total path cost. (4 points)

Treat the Fringe as a priority queue. When performing "Remove-Front," take the node with the smallest accumulated cost.

a. What path is returned by the algorithm? (2 points)

b. What is its path cost (total distance)? (2 points)

c. What did you observe that was interesting about the algorithm? (2 points)

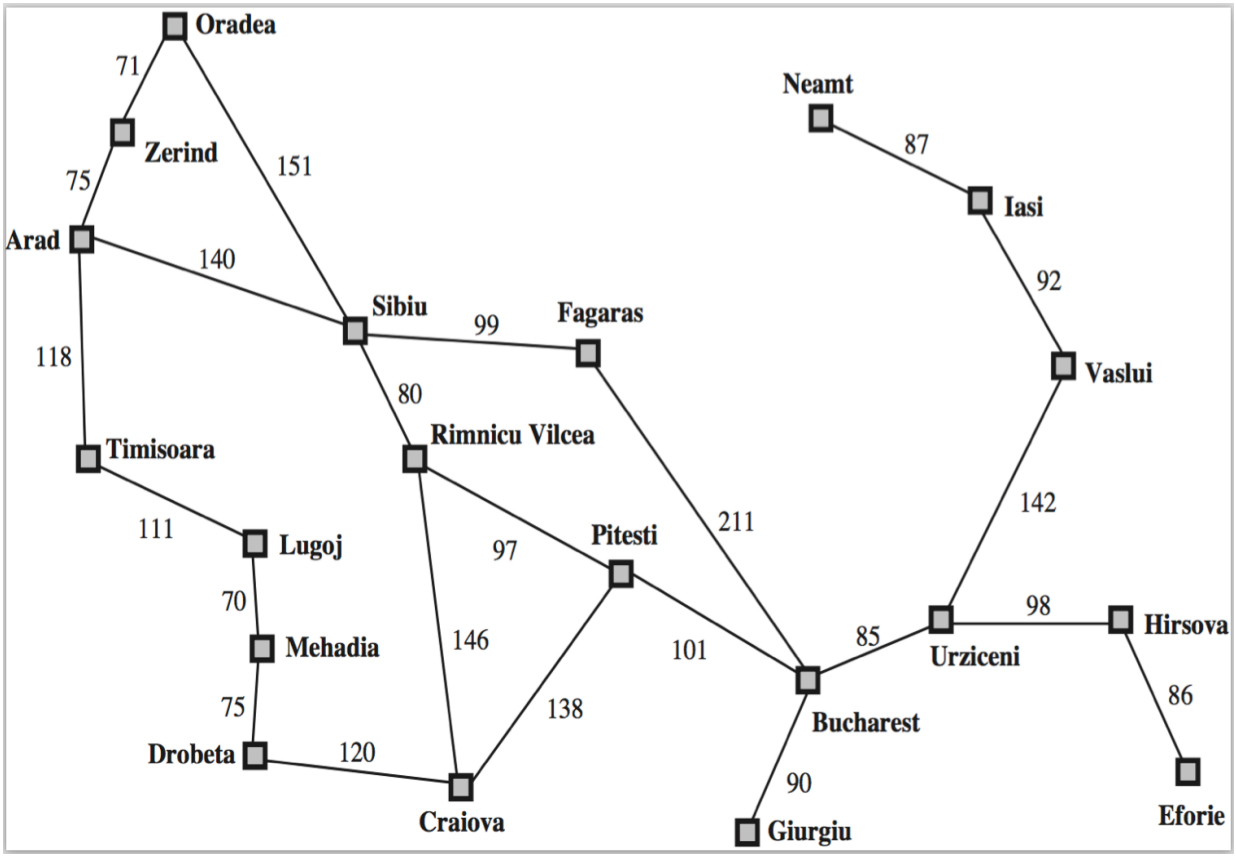
5. A* Search

A* search (pronounced "a star") is a breath-first search where the node to be expanded is the one with the best (lowest) value of (accumulated path cost in the frontier) + (heuristic distance to the goal). In other words, the priority function *f* for a node *n* is:

$$f(n) = g(h) + h(n)$$

The heuristic function we will use is straight line distance (h_{sld}), which is guaranteed to be admissible for Euclidean spaces.

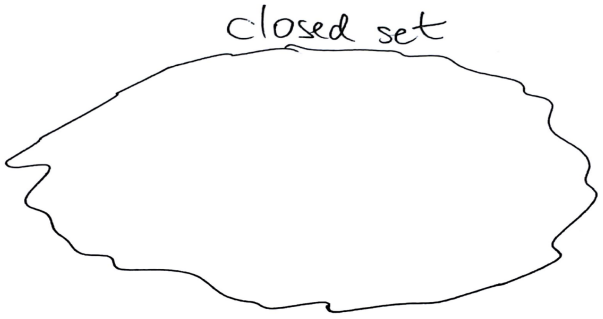
The actual path cost may be equal to or greater than this heuristic value, but it cannot be less than it.



Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

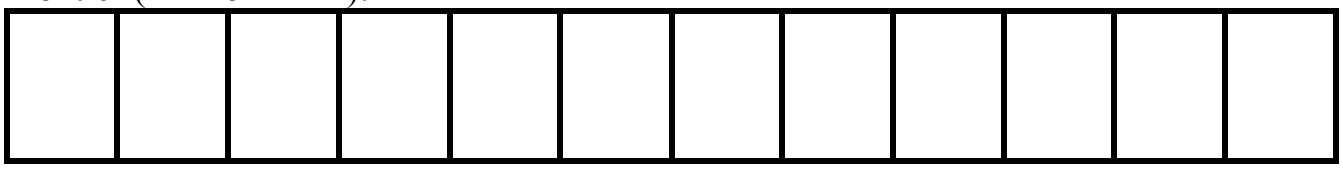
Perform A* search using the priority function $f(n) = g(h) + h(n)$. Start state is Arad. Goal state is Giurgiu. Draw nodes in the Fringe as they are expanded. Include their f value at the point of insertion. (4 points)

When nodes get removed for goal-testing, cross them out and put them in the closed set as states.



Treat the Fringe as a priority queue. When performing "Remove-Front," take the node with the smallest $f=g+h$ cost.

Frontier (fill from L–R):



a. What path is returned by the algorithm? (2 points)

b. What is its path cost (total distance)? (2 points)

c. What did you observe that was interesting about the algorithm, compared to UCS? (2 points)

6. More A*

a) Suppose we have two admissible heuristic functions, *h1* and *h2*, where for all search states s, $h1(s) < h2(s)$. In other words:

$$\forall(s): h1(s) < h2(s)$$

What will be the impact on the search process? Specifically:

i) Optimality: Will the search return the best solution when using *h1*? What about when using *h2*? (3 points)

Both are admissible, so using a tree search, the A utilizing them will find optimal solution.*

ii) Efficiency: Will more nodes be expanded when using *h1* or *h2*? The same? (3 points)

-h2 dominates h1 which makes it more efficient

b) Suppose we use a heuristic *h3* that is *not* admissible. (Sometimes, it produces values that are larger than the actual best path to the goal.)

i) What will be the optimality and efficiency implications of this? Consider separately for the three cases below. Also, *answer for the worst case. (3 points)*

Not admissible means that the path to the solution will be suboptimal albeit with a faster convergence (less nodes expanded)

b) Only one solution exists, and there is only one path to it.

i) Optimality: Will *h3* find the solution? Why or why not? (2 points)

-ONLY one path to the goal means that the Algorithm will find the solution, at which point it will be the optimal one.

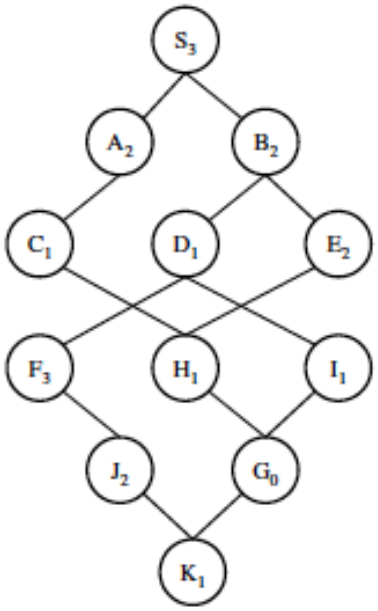
ii) Efficiency: Will *h3* cause more, fewer, or the same number of nodes to be expanded vs. an admissible heuristic? Why? (3 points)

If there were many paths to the solution, then h3 will cause fewer nodes to be expanded because it dominates an admissible heuristic. In this case the path is sub-optimal.
Since there is only one path to this problem, then is likely to expand more nodes looking for the solution.

NB: Inadmissible DOES NOT mean incorrect.

7. Consider the problem of moving a knight on a 3x4 board, with start and goal states l labeled as S and G in figure 3. The search space can be translated into the following graph. The letter in each node is its name and the subscript digit is the heuristic value. All transitions have cost 1.

S_3	H_1	D_1	K_1
I_1	J_2	A_2	E_2
C_1	B_2	G_0	F_3



Make the following assumptions:

- All the algorithms do not generate paths with loops.
- Nodes are selected in alphabetical order when the algorithm finds a tie.
- A node is visited when that node is at the front of the search queue.

Write the sequence of nodes in the order visited by the specified methods.

Note: You may find it useful to draw the search tree corresponding to the graph above.

(a) Depth-first search. (4 points)

Solution: Nodes are selected in alphabetical order when the algorithm finds a tie.

S A C H E B D F J K G

(b) Breadth-first search. (4 points)

Solution: ***S A B C D E H F I G***

(c) A* search. In addition to the sequence of nodes visited, you are also to attach the estimated total cost f(s)=g(s)+h(s) for each node visited and return the final path found and its length. (4points)

Solution: g(s) - cost to reach the node
h(s) - cost to get from the node to the goal

State	f(s)	Observation
S	f(S)=0+3=3	
A	f(A)=1+2=3, f(B)= 1+2=3	A will be selected because nodes are selected in alphabetical order when the algorithm finds a tie.
B	f(C)=2+1=3	Between B and C with $f(B) = f(C) = 3$, B will be selected
C	f(D)=2+1=3, f(E)=2+2=4	Between D and C with $f(C) = f(D) = 3$, C will be selected
D	f(H)=3+1=4	D is the only node visited with the lowest value for $f(D) = 3$
E	f(F)=4+3=7, f(I)=4+1=5	E is the only node visited with the lowest value for $f(E) = 4$
H	f(H)=4	already computed
G	f(G)=4+0=4	

Nodes visited: S A B C D E H G
Possible final path: S A C H G