CSE 415 Spring 2021 Assignment 4

Last name:____________________ First name:____________________________

Due Wednesday night May 5 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 the following five exercises. These are intended to take 20-25 minutes each if you know how to do them. Each is worth 20 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.

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 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.

If you choose to 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.

---

1 Blind Search with the Towers of Hanoi

The 2-disk version of the Towers of Hanoi is a trivial puzzle for humans and machines alike. However, it's nice and simple context for comparing different algorithms.

Several aspects of this problem will come up again in Assignment 5, and so this problem will not only help you get more familiar with certain details of the search algorithms, but it will provide some insight into the the Towers of Hanoi problem space.

Let us assume that the problem is formulated with the following state representation and operators.

```
Initial state:
Left: 1,2
Middle:
Right:

Goal state:
Left:
Middle:
Right: 1,2
```

Operator ϕ_0: "Move a disk from Left to Middle."

Operator ϕ_1: "Move a disk from Left to Right."

Operator ϕ_2: "Move a disk from Middle to Left."

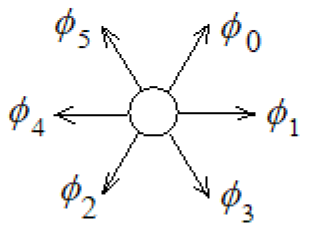
Operator ϕ_3: "Move a disk from Middle to Right."

Operator ϕ_4: "Move a disk from Right to Left."

Operator ϕ_5: "Move a disk from Right to Middle."

Hand simulate DFS (Depth-First Search, BFS (Breadth-First Search) and IDDFS (Iterative-Deepening Depth-First Search) on this problem, in order to determine the state visitation orderings and compare them.

The problem-space graph for this formulation can be laid out as in the diagrams for sub-questions a-c below. Note that the operators always take a specific direction, as shown in this diagram:



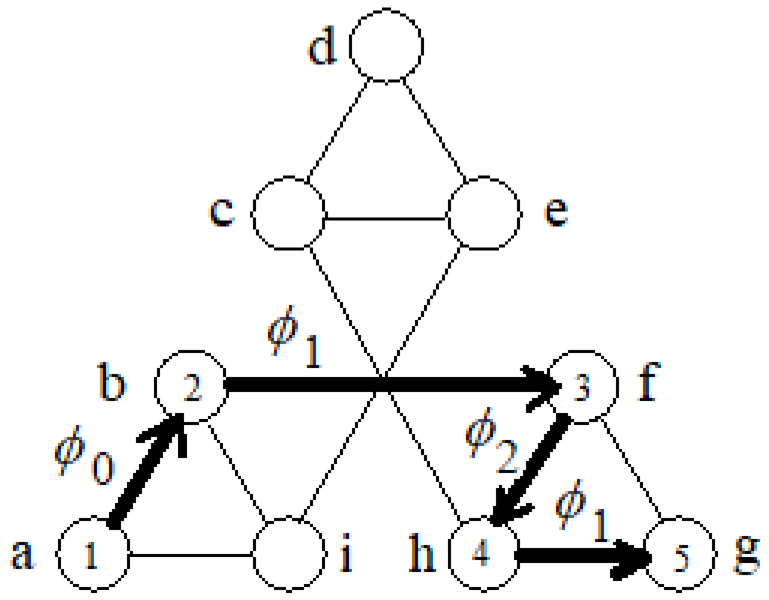
Use the copies of the problem space graph below to show the progress of each search algorithm. When doing IDDFS, you'll use a separate copy of the graph for each outer iteration.

For each algorithm stop when the goal node (g) is selected as the current state.

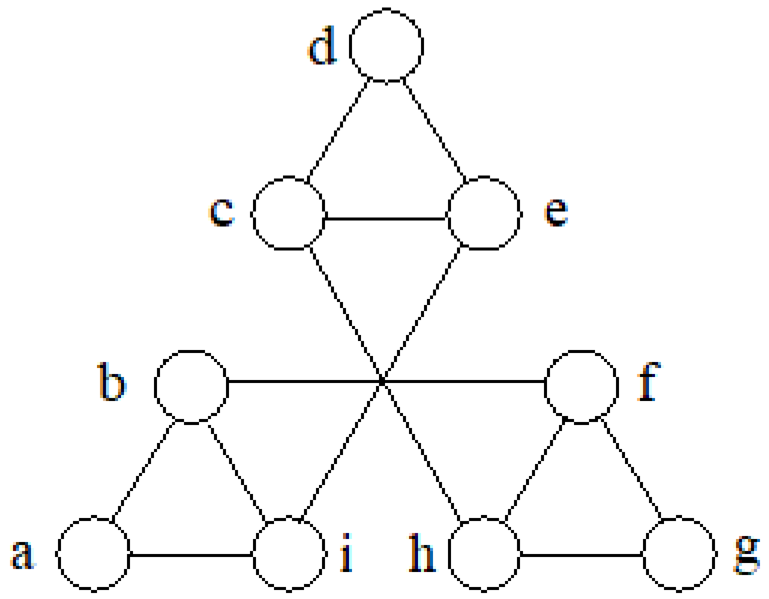
Number the nodes as they are visited (i.e., as they become the current state). The initial state should get numbered 1 when each algorithm starts. However, during IDDFS, it should get multiple numbers since it will be visited multiple times.

In order to get the correct numberings, it is very important to follow the pseudocode in the DFS, and BFS algorithms, and to generate the successors by using the operators in their given order: phi 0, phi 1, phi 2, phi 3, phi 4, phi 5.

(a) (0 points) Hand-simulate DFS and put the node visitation order on the graph. Note that the answers to this part are done for you as an example.

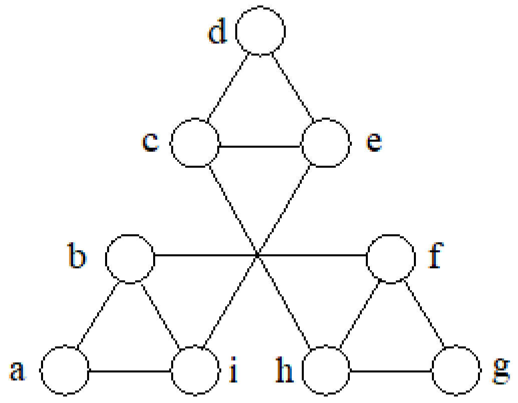


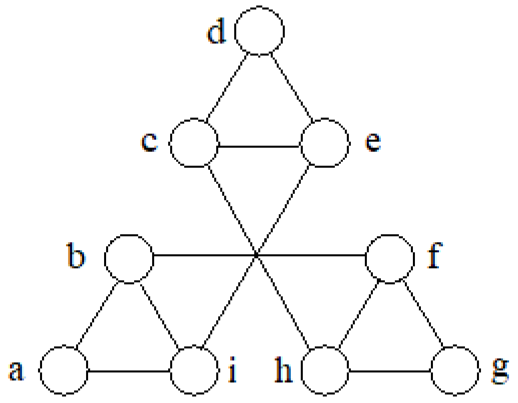
(b) (8 points) Hand-simulate BFS and put the node visitation order on the graph. As in the given example, also show the moves using arrows and operator identifiers (ϕ_0, etc.).



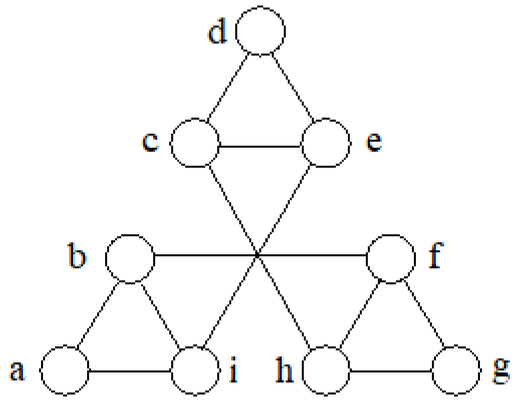
(c) (12 points) Hand-simulate IDDFS and put the node visitation order onto the four graphs below. Use one graph copy for each iteration of IDDFS. The first graph should have only one node (for the initial state) visited. on the graph. The second graph's initial state should have visitation number 2 (since this state is visited again). Note that within one iteration of IDDFS, some node(s) may be reached multiple times along different paths from the initial state. This is OK, and each such repeat visit should be counted as a node visitation and shown in your results. However, there should be no more than one visitation of any given node along the current path between the initial state and current state. That is, the search path must never be allowed to loop back on itself.

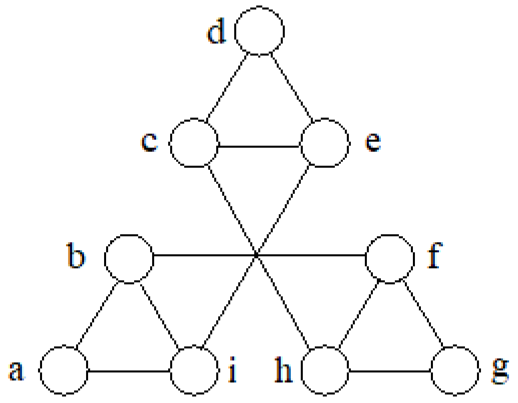
First and second iterations:





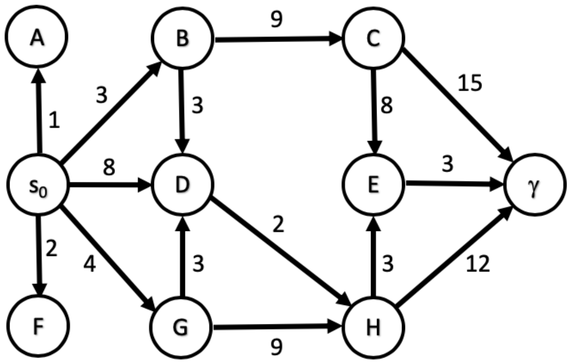
Third and fourth iterations:





2 Heuristic Search

(a) (5 points) Consider the following statement: *The best heuristic is always the one that gives you the estimate closest to the true cost.* Is this always true? Explain why you agree or disagree with the statement.

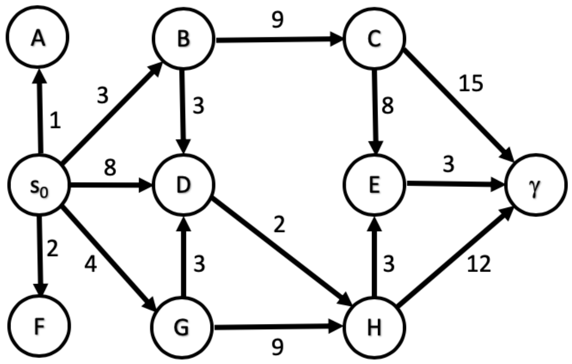


state (s)	s_0	A	B	C	D	E	F	G	H	γ
heuristic $h_1(s)$	14	16	10	10	7	2	20	10	5	0
heuristic $h_2(s)$	9	16	7	9	5	2	20	7	4	0
heuristic $h_3(s)$	10	16	8	10	6	2	20	8	4	0

(b) (5 points) Which heuristics (h_1, h_2, h_3) shown above are admissible?

(c) (5 points) Which heuristics (h_1, h_2, h_3) shown above are consistent?

(d) (5 points) Which of the 3 heuristics above would you select as the best one to use with A* search? Why? Refer to both consistency and admissibility in your justification.



state (s)	s_0	A	B	C	D	E	F	G	H	γ
heuristic $h_4(s)$	7	28	5	5	4	1	24	5	3	0

(e) (5 points) Referring back to the graph again, trace out the path that would be followed in an A* search, given the heuristics provided above. As you trace the path, complete the table below, indicating which nodes are on the open and closed lists, along with their 'f' values:

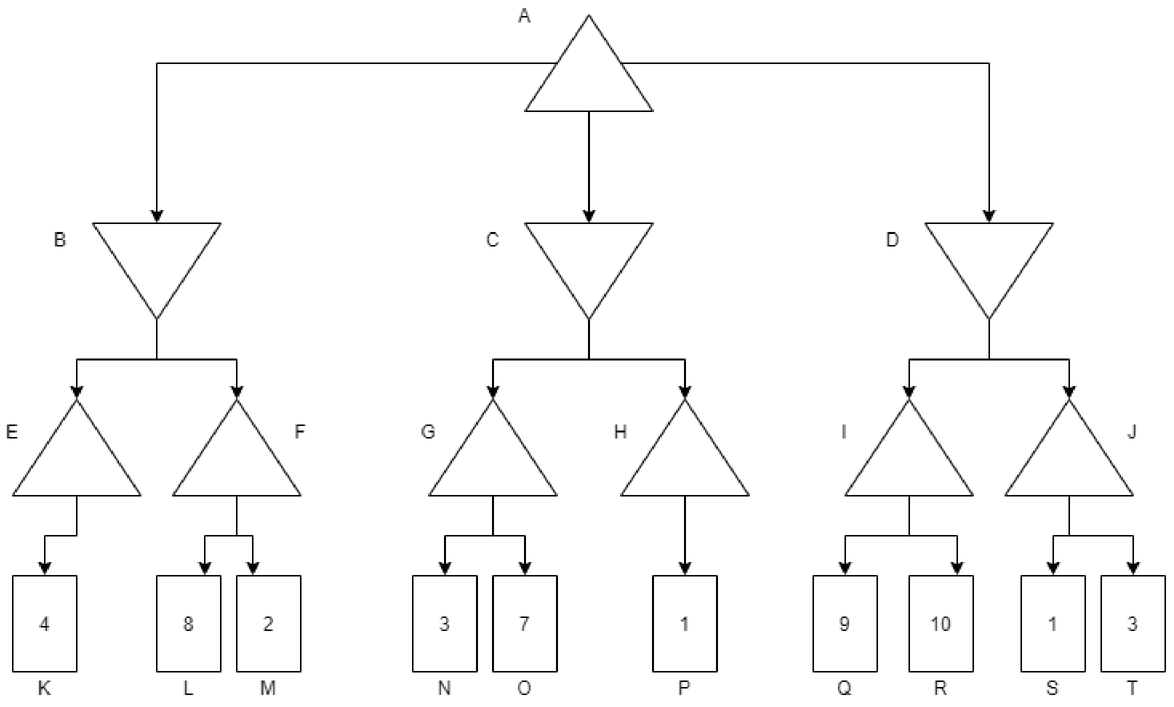
	Open	Closed
Starting A^* search	$[s_0, 7]$	empty
s_0		

3 Adversarial Search

Minimax game-tree search finds a best move under the assumption that both players play rationally, to either maximize or minimize the value of the same static-evaluation function to a certain ply limit. (It can also do well even when those assumptions are relaxed somewhat.) However, the quality of a move usually improves as the maximum ply is increased. That usually makes minimax take a lot longer. Alpha-beta pruning is a method for speeding up minimax by eliminating any subtrees from the search that can be identified as not able to contribute to the outcome. However, alpha-beta pruning's success is dependent upon the order in which the successors of a state are analyzed.

For each of the four following methods, determine the number of cutoffs, the number of leaf nodes statically evaluated and the total number of states that would have to be generated. Note that this example has a maximizing node at the root; that means we are computing the value of the best move for the maximizing player and computing the maximizing player's best move, as the overall objective in this problem.

For parts (c) and (d) there are blank tree diagrams you should complete to show the new order in which the space is searched.



(a) (7 points) Straight minimax search, depth-first, left-to-right. Show the backed-up values at each internal node. Then fill in the table. Total number of states generated should include the root, and should include those leaf nodes that had to be statically evaluated.

Number of leaf nodes cutoff:	0
Number of leaf nodes processed:	
Total number of states generated:	

(b) (3 points) Alpha-beta pruning, left-to-right, on the given tree. Mark where cutoffs occur on the tree, and fill out the table:

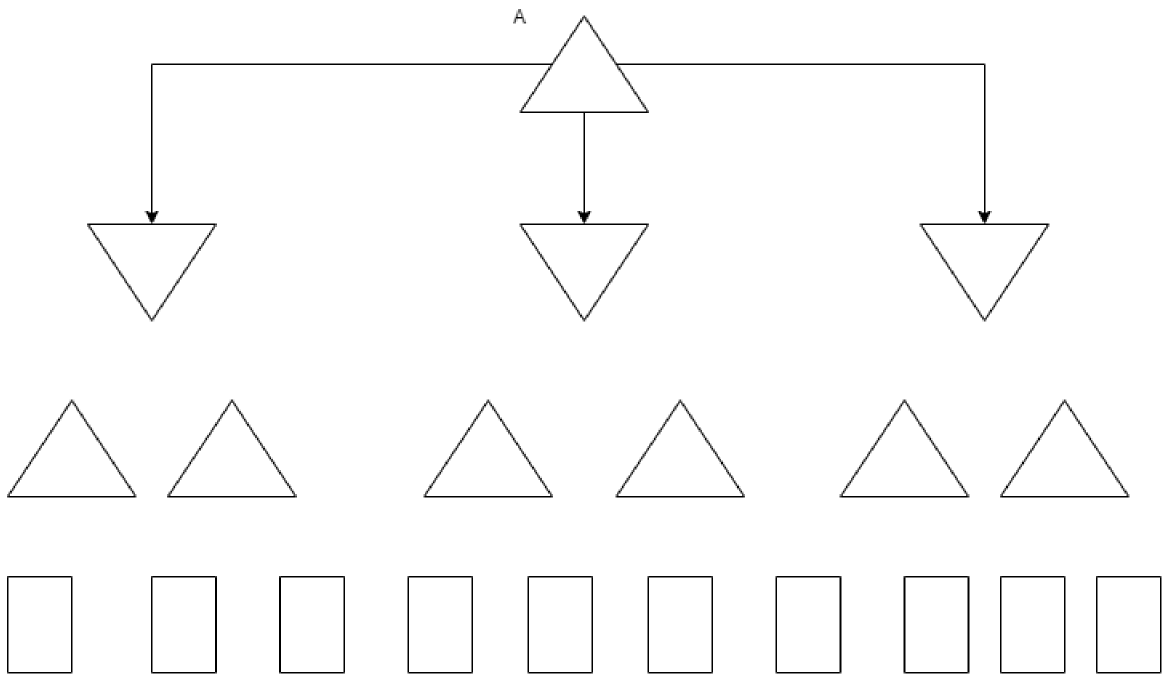
Number of leaf nodes cutoff:	
Number of leaf nodes processed:	
Total number of states generated:	

(c) (10 points) Alpha-beta pruning, using a secondary evaluation function $f_2(s)$. Rather than going depth-first, use the following method. Start at the root, A; call this S_c for current state. To process S_c, check whether it is a leaf node (i.e., at the maximum ply). If so, return its static value (the normal static evaluation, the values are given in the rectangles in the diagram); otherwise, S_c is an internal node, so generate all its successors (its immediate children, but not their children, etc.). Apply f_2 to each of the children, and sort them into best-first order. (If S_c is a maximizing node, then highest is best. Else lowest is best.) Process these children in this best-first order, using the regular alpha-beta method, by first calling recursively on the best child, then the next best child (unless a cutoff happens and the rest of the children of S_c can be ignored). Return the best value found among those children not cut off.

For this part, use f_2 as given in this table. Leave nodes K through T in the same relative order as in the original diagram. (In practice, an agent designer might use a single static evaluation function to serve both the usual purpose of evaluating leaf nodes and the new purpose of pre-evaluating internal nodes, but one point of this exercise is to show that they can be different functions, possibly investing more or fewer computational resources into finding a best ordering for the successors of a state prior to alpha-beta pruning.)

Node s:	A	B	C	D	E	F	G	H	I	J
$f_2(s)$:		7	6	3	2	4	5	4	1	2

Complete the diagram of the re-ordered tree, labeling each internal node with the letter for the appropriate state in the original diagram. Draw in the missing edges, since the tree's shape may now be a little different. Mark where cutoffs occur on the tree, and fill out the table:



Number of leaf nodes cutoff:	
Number of leaf nodes processed:	
Total number of states generated:	

Note that the total number of states generated must be sure to include all the children of any internal node that did not get cut off, since we assume that f_2 cannot be applied to them unless they are created.

4 Markov Decision Processes

Consider the following game. You have two coins - one **gold** and one **silver**, that are tossed independently of one another. The gold coin has higher value - if you get heads on the gold coin, you obtain a score of 2, and 0 otherwise. Getting heads on the silver coin gives a score of 1 or 0 for heads or tails respectively. However, the gold coin is biased - there is only 1/3 probability of getting heads, while the silver coin is unbiased.

On every turn, you set in motion a series of coin flips. At the beginning of each turn, you flip both coins once. The gold coin is the driver; if it gives tails, your turn ends. However, if you get heads, then you flip the 2 coins again. You keep repeating the coin flips till either the gold coin gives tails, or you **go bust** (see description below). You record a **turn score**, which is the sum of scores of both coins across all the coin flips in that turn. Once the turn ends, you may start off a new turn.

You also keep a **running score** that accumulates the turn scores obtained across all the turns so far.

Just before any turn starts, you can either choose to toss the coins or stop if the running score is less than 6. In the middle of a turn, if the sum of your running score and turn score accrued so far in the turn **reaches or exceeds 6**, you "go bust" and go to the final state, accruing zero reward.

When in any state other than the final state, you are allowed to stop. When you stop, you reach the final state and your reward is the running score (which is less than 6).

Note: there is no direct reward from tossing the coins (or we could say that there is a reward but it's always 0). The only non-zero reward comes from explicitly taking the stop action. Discounting or not should not matter in the MDP for this game, but for the record, we assume no discounting (i.e., $\gamma = 1$).

This game can be formulated in the form of an MDP. Let the MDP be (S, A, T, R) where:
$S = \{s_0, s_1, \cdots\}$,
$A = \{a_0, a_1, \cdots\}$,
$T : S \times A \times S \to [0, 1]$ and
$R : S \times A \times S \to \mathbb{R}$.

(a) (4 points) Write down the states $s_i \in S$ and actions $a_j \in A$ for this MDP. (Hint: there are 7 states in total and each should correspond to a numeric value except the final state)

(b) (10 points) Give the full transition function $T(s, a, s')$ where s is a current state, a is an action, and s' is a possible next state when action a is taken in s. In other words, for any triple $(s, a, s') \in S \times A \times S$, you need to provide the value of the transition function. For convenience, you may give the values in the form of a table (corresponding to each action), where the s states are the rows, and s' states are the columns.

(c) (2 points) Give the full reward function $R(s, a, s')$ i.e., for any triple (s, a, s'), you need to give the value of the reward function. Note: You don't need to tabulate these values, one way you can write them is the following:
$R(s_0, a_0, s_0) = \langle value1 \rangle$
$R(s_0, a_0, s_1) = \langle value2 \rangle$
$\vdots$
$R(s, a, s') = 0$, otherwise

(d) (4 points) What is the optimal policy? You may describe it in words. Give a brief explanation as to how you chose that policy.

5 Computing MDP State Values and Q-Values

Recently, Jim has been working on building an intelligent agent to help a friend solve a problem that can be modeled using an MDP. In this environment, there are 3 possible states $S = \{s_1, s_2, s_3\}$ and at each state the agent always has 2 available actions $A = \{f, g\}$. Applying any action a from any state s has a probability $T(s, a, s')$ of moving the agent to one of the *other two* states but will never result in the agent staying at the original state. The rewards for this environment represent the cost/reward of an action, and thus are only dependent on the original state and action taken ($\forall s' \in S, R(s, a, s') = R(s, a)$), not where the agent ended up.

(a) (2 points) Write down the problem-specific Bellman update equations for each of the 3 states ($V(s) =?$) in this particular MDP. (Use the names of the specific states and actions.)

(b) (18 points) One fateful day, while Jim was running a VI-based MDP solver on this problem, a mistake in specifying arguments caused the file that recorded the transition probability table $T(s, a, s')$ to be overwritten with the output solution. Now, Jim has the solution $V^*(s)$ and optimal policy $\pi^*(s)$ but has lost the transition probabilities for the problem.

s	a	$R(s,a)$	$V^*(s)$	$\pi^*(s)$
s_1	f	-3	-1.4	g
s_1	g	-4	-1.4	g
s_2	f	3	3	f
s_2	g	3	3	f
s_3	f	1.92	1.4	f
s_3	g	1.7	1.4	f

After looking at the command line history and noting that a discount of $\gamma = 1$ was specified, Jim muses that it may be possible to recover some parts of the transition probability table $T(s, a, s')$. Using the information above, fill in the values in table below that you can recover. For the entries where it is impossible to determine the value, figure out the upper bound and the lower bound for this entry and write down the range of values.

s	a	$T(s,a,s_1)$	$T(s,a,s_2)$	$T(s,a,s_3)$
s_1	f	0		
s_1	g	0		
s_2	f		0	
s_2	g		0	
s_3	f			0
s_3	g			0