CS 188 Spring 2013 Introduction to Artificial Intelligence Midterm 1

- You have approximately 2 hours.
- The exam is closed book, closed notes except your one-page crib sheet.
- Please use non-programmable calculators only.
- Mark your answers ON THE EXAM ITSELF. If you are not sure of your answer you may wish to provide a *brief* explanation. All short answer sections can be successfully answered in a few sentences AT MOST.

First name	
Last name	
SID	
edX username	

First and last name of student to your left	
First and last name of student to your right	

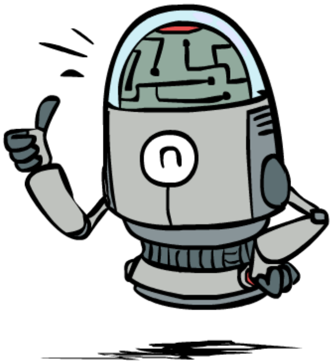
For staff use only:

Q1.	Warm-Up	/1
Q2.	CSPs: Midterm 1 Staff Assignments	/17
Q3.	Solving Search Problems with MDPs	/11
Q4.	X Values	/10
Q5.	Games with Magic	/23
Q6.	Pruning and Child Expansion Ordering	/10
Q7.	A* Search: Parallel Node Expansion	/28
	Total	/100

THIS PAGE IS INTENTIONALLY LEFT BLANK

Q1. [1 pt] Warm-Up

Circle the CS188 mascot



Q2. [17 pts] CSPs: Midterm 1 Staff Assignments

CS188 Midterm I is coming up, and the CS188 staff has yet to write the test. There are a total of 6 questions on the exam and each question will cover a topic. Here is the format of the exam:

- q1. Search
- q2. Games
- q3. CSPs
- q4. MDPs
- q5. True/False
- q6. Short Answer

There are 7 people on the course staff: Brad, Donahue, Ferguson, Judy, Kyle, Michael, and Nick. Each of them is responsible to work with Prof. Abbeel on one question. (But a question could end up having more than one staff person, or potentially zero staff assigned to it.) However, the staff are pretty quirky and want the following constraints to be satisfied:

(i) Donahue (D) will not work on a question together with Judy (J).

(ii) Kyle (K) must work on either Search, Games or CSPs

(iii) Michael (M) is very odd, so he can only contribute to an odd-numbered question.

(iv) Nick (N) must work on a question that's before Michael (M)'s question.

(v) Kyle (K) must work on a question that's before Donahue (D)'s question

(vi) Brad (B) does not like grading exams, so he must work on True/False.

(vii) Judy (J) must work on a question that's after Nick (N)'s question.

(viii) If Brad (B) is to work with someone, it cannot be with Nick (N).

(ix) Nick (N) cannot work on question 6.

(x) Ferguson (F) cannot work on questions 4, 5, or 6

(xi) Donahue (D) cannot work on question 5.

(xii) Donahue (D) must work on a question before Ferguson (F)'s question.

(a) [2 pts] We will model this problem as a constraint satisfaction problem (CSP). Our variables correspond to each of the staff members, J, F, N, D, M, B, K, and the domains are the questions 1, 2, 3, 4, 5, 6. After applying the unary constraints, what are the resulting domains of each variable? (The second grid with variables and domains is provided as a back-up in case you mess up on the first one.)

B	1	2	3	4	5	6
D	1	2	3	4	5	6
F	1	2	3	4	5	6
J	1	2	3	4	5	6
K	1	2	3	4	5	6
N	1	2	3	4	5	6
M	1	2	3	4	5	6

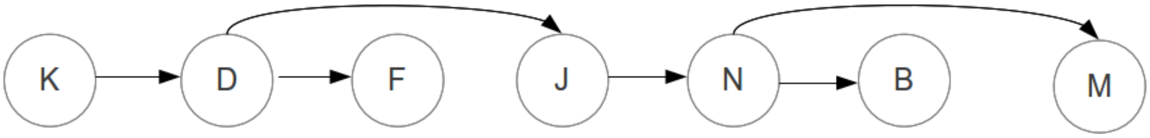
B	1	2	3	4	5	6
D	1	2	3	4	5	6
F	1	2	3	4	5	6
J	1	2	3	4	5	6
K	1	2	3	4	5	6
N	1	2	3	4	5	6
M	1	2	3	4	5	6

(b) [2 pts] If we apply the Minimum Remaining Value (MRV) heuristic, which variable should be assigned first?

(c) [3 pts] Normally we would now proceed with the variable you found in (b), but to decouple this question from the previous one (and prevent potential errors from propagating), let's proceed with assigning Michael first. For value ordering we use the Least Constraining Value (LCV) heuristic, where we use *Forward Checking* to compute the number of remaining values in other variables domains. What ordering of values is prescribed by the LCV heuristic? Include your work—i.e., include the resulting filtered domains that are different for the different values.

(d) Realizing this is a tree-structured CSP, we decide not to run backtracking search, and instead use the efficient two-pass algorithm to solve tree-structured CSPs. We will run this two-pass algorithm after applying the unary constraints from part (a). Below is the linearized version of the tree-structured CSP graph for you to work with.

(i) [6 pts] **First Pass: Domain Pruning.** Pass from *right to left* to perform Domain Pruning. Write the values that remain in each domain below each node in the figure above.



1	1	1	1	1	1	1
2	2	2	2	2	2	2
3	3	3	3	3	3	3
4	4	4	4	4	4	4
5	5	5	5	5	5	5
6	6	6	6	6	6	6

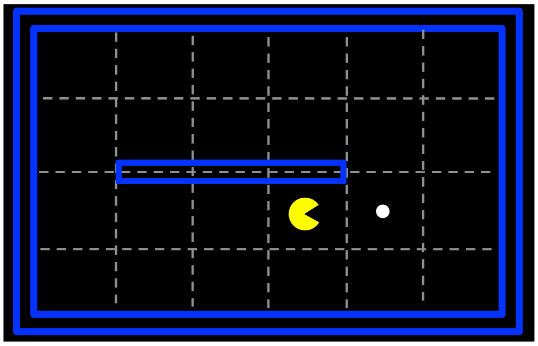
(ii) [4 pts] **Second Pass: Find Solution.** Pass from *left to right*, assigning values for the solution. If there is more than one possible assignment, choose the highest value.

Q3. [11 pts] Solving Search Problems with MDPs

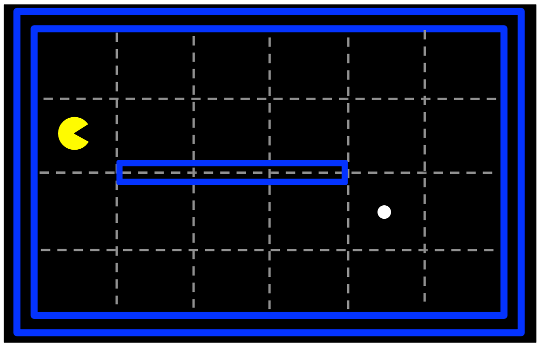
The following parts consider a Pacman agent in a deterministic environment. A goal state is reached when there are no remaining food pellets on the board. Pacman's available actions are $\{N, S, E, W\}$, but Pacman **can not** move into a wall. Whenever Pacman eats a food pellet he receives a reward of +1.

Assume that pacman eats a food pellet as soon as he occupies the location of the food pellet—i.e., the reward is received for the transition into the square with the food pellet.

Consider the particular Pacman board states shown below. Throughout this problem assume that $V_0(s) = 0$ for all states, s. Let the discount factor, $\gamma = 1$.



State A



State B

(a) [2 pts] What is the optimal value of state A, $V^*(A)$?

(b) [2 pts] What is the optimal value of state B, $V^*(B)$?

(c) [2 pts] At what iteration, k, will $V_k(B)$ first be non-zero?

(d) [2 pts] How do the optimal q-state values of moving W and E from state A compare? (*choose one*)

○ $Q^*(A, W) > Q^*(A, E)$ ○ $Q^*(A, W) < Q^*(A, E)$ ○ $Q^*(A, W) = Q^*(A, E)$

(e) [3 pts] If we use this MDP formulation, is the policy found guaranteed to produce the shortest path from pacman's starting position to the food pellet? If not, how could you modify the MDP formulation to guarantee that the optimal policy found will produce the shortest path from pacman's starting position to the food pellet?

Q4. [10 pts] X Values

Instead of the Bellman update equation, consider an alternative update equation, which learns the X value function. The update equation, assuming a discount factor $\gamma = 1$, is shown below:

$$X_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \max_{a'} \sum_{s''} T(s',a',s'') \left[R(s',a',s'') + X_k(s'') \right] \right]$$

(a) [6 pts] Assuming we have an MDP with two states, S_1, S_2, and two actions, a_1, a_2, draw the expectimax tree rooted at S_1 that corresponds to the alternative update equation.

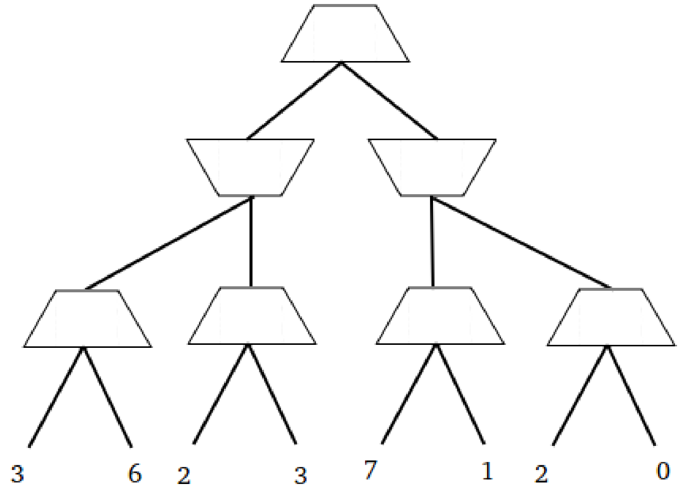
(b) [4 pts] Write the mathematical relationship between the X_k-values learned using the alternative update equation and the V_k-values learned using a Bellman update equation, or write *None* if there is no relationship.

Q5. [23 pts] Games with Magic

(a) Standard Minimax

(i) [2 pts] Fill in the values of each of the nodes in the following Minimax tree. The upward pointing trapezoids correspond to maximizer nodes (layer 1 and 3), and the downward pointing trapezoids correspond to minimizer nodes (layer 2). Each node has two actions available, Left and Right.

(ii) [1 pt] Mark the sequence of actions that correspond to Minimax play.



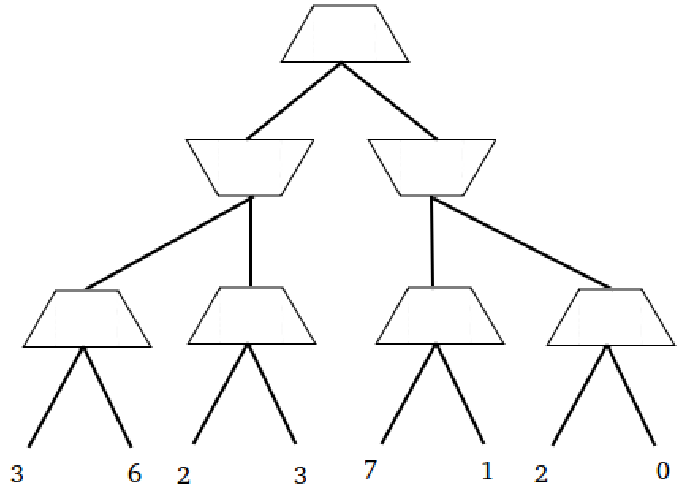
(b) Dark Magic

Pacman (= maximizer) has mastered some dark magic. With his dark magic skills Pacman can take control over his opponent's muscles while they execute their move — and in doing so be fully in charge of the opponent's move. But the magic comes at a price: every time Pacman uses his magic, he pays a price of c—which is measured in the same units as the values at the bottom of the tree.

Note: For each of his opponent's actions, Pacman has the *choice* to either let his opponent act (optimally according to minimax), or to take control over his opponent's move at a cost of c.

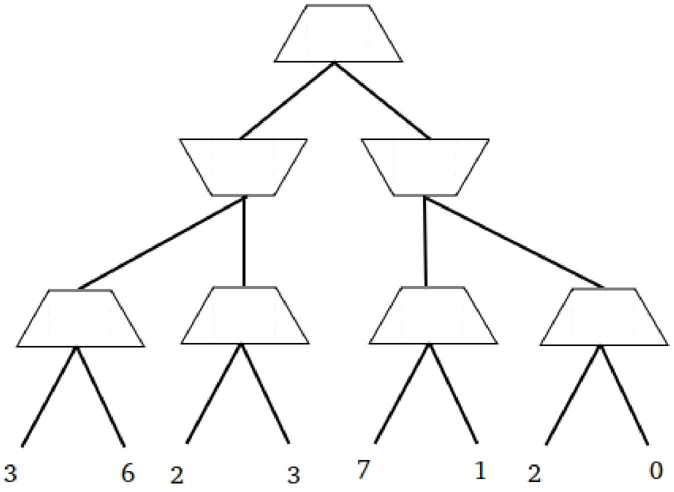
(i) [3 pts] **Dark Magic at Cost** $c = 2$

Consider the same game as before but now Pacman has access to his magic at cost $c = 2$. Is it optimal for Pacman to use his dark magic? If so, mark in the tree below where he will use it. Either way, mark what the outcome of the game will be and the sequence of actions that lead to that outcome.



(ii) [3 pts] **Dark Magic at Cost** $c = 5$

Consider the same game as before but now Pacman has access to his magic at cost $c = 5$. Is it optimal for Pacman to use his dark magic? If so, mark in the tree below where he will use it. Either way, mark what the outcome of the game will be and the sequence of actions that lead to that outcome.



(iii) [7 pts] **Dark Magic Minimax Algorithm**

Now let's study the general case. *Assume that the minimizer player has no idea that Pacman has the ability to use dark magic at a cost of c. I.e., the minimizer chooses their actions according to standard minimax.* You get to write the pseudo-code that Pacman uses to compute their strategy. As a starting point / reminder we give you below the pseudo-code for a standard minimax agent. Modify the pseudo-code such that it returns the optimal value for Pacman. Your pseudo-code should be sufficiently general that it works for arbitrary depth games.

```
function MAX-VALUE(state)
    if state is leaf then
        return UTILITY(state)
    end if
    $v \leftarrow -\infty$
    for successor in SUCCESSORS(state) do
        $v \leftarrow \max(v, \text{MIN-VALUE}(successor))$
    end for
    return v
end function

function MIN-VALUE(state)
    if state is leaf then
        return UTILITY(state)
    end if
    $v \leftarrow \infty$
    for successor in SUCCESSORS(state) do
        $v \leftarrow \min(v, \text{MAX-VALUE}(successor))$
    end for
    return v
end function
```

(iv) [7 pts] **Dark Magic Becomes Predictable**

The minimizer has come to the realization that Pacman has the ability to apply magic at cost c. Hence the minimizer now doesn't play according the regular minimax strategy anymore, but accounts for Pacman's magic capabilities when making decisions. Pacman in turn, is also aware of the minimizer's new way of making decisions.

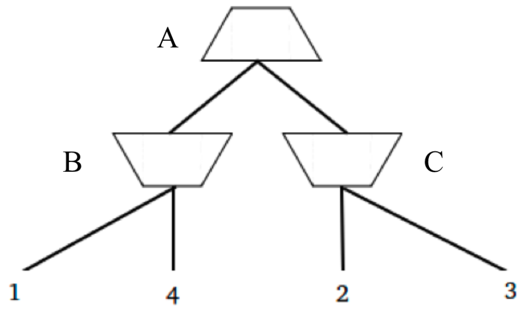
You again get to write the pseudo-code that Pacman uses to compute his strategy. As a starting point / reminder we give you below the pseudo-code for a standard minimax agent. Modify the pseudocode such that it returns the optimal value for Pacman.

```
function MAX-VALUE(state)
    if state is leaf then
        return UTILITY(state)
    end if
    $v \leftarrow -\infty$
    for successor in SUCCESSORS(state) do
        $v \leftarrow \max(v, \text{MIN-VALUE}(successor))$
    end for
    return $v$
end function
```

```
function MIN-VALUE(state)
    if state is leaf then
        return UTILITY(state)
    end if
    $v \leftarrow \infty$
    for successor in SUCCESSORS(state) do
        $v \leftarrow \min(v, \text{MAX-VALUE}(successor))$
    end for
    return $v$
end function
```

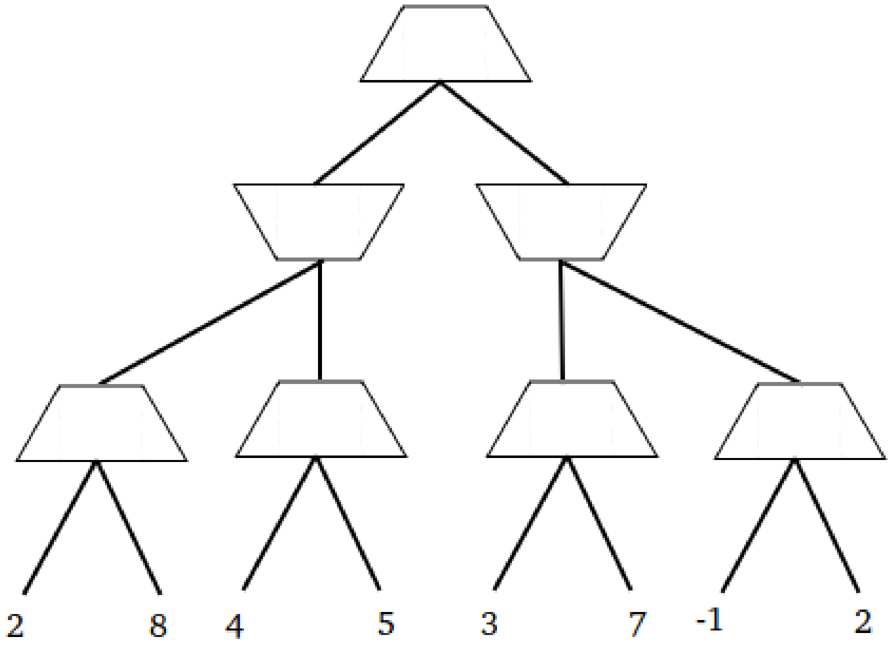
Q6. [10 pts] Pruning and Child Expansion Ordering

The number of nodes pruned using alpha-beta pruning depends on the order in which the nodes are expanded. For example, consider the following minimax tree.



In this tree, if the children of each node are expanded from left to right for each of the three nodes then no pruning is possible. However, if the expansion ordering were to be first Right then Left for node A, first Right then Left for node C, and first Left then Right for node B, then the leaf containing the value 4 can be pruned. (Similarly for first Right then Left for node A, first Left then Right for node C, and first Left then Right for node B.)

For the following tree, give an ordering of expansion for each of the nodes that will maximize the number of leaf nodes that are never visited due the search (thanks to pruning). ***For each node, draw an arrow indicating which child will be visited first. Cross out every leaf node that never gets visited.***
Hint: Your solution should have three leaf nodes crossed out and indicate the child ordering for 6 of the 7 internal nodes.



Q7. [28 pts] A* Search: Parallel Node Expansion

Recall that A* graph search can be implemented in pseudo-code as follows:

```
 1: function A*-Graph-Search(problem, fringe)
 2:     closed ← an empty set
 3:     fringe ← Insert(Make-Node(Initial-State[problem]), fringe)
 4:     loop do
 5:         if fringe is empty then return failure
 6:         node ← Remove-Front(fringe)
 7:         if Goal-Test(problem, State[node]) then return node
 8:         if State[node] is not in closed then
 9:             add State[node] to closed
10:             child-nodes ← Expand(node, problem)
11:             fringe ← Insert-All(child-nodes, fringe)
```

You notice that your successor function (Expand) takes a very long time to compute and the duration can vary a lot from node to node, so you try to speed things up using parallelization. You come up with A*-Parallel, which uses a "master" thread which runs A*-Parallel and a set of $n \geq 1$ "workers", which are separate threads that execute the function Worker-Expand which performs a node expansion and writes results back to a shared fringe. The master thread issues non-blocking calls to Worker-Expand, which dispatches a given worker to begin expanding a particular node.[1] The Wait function called from the master thread pauses execution (sleeps) in the master thread for a small period of time, e.g., 20 ms. The *fringe* for these functions is in shared memory and is always passed by reference. Assume the shared *fringe* object can be safely modified from multiple threads.

A*-Parallel is best thought of as a modification of A*-Graph-Search. In lines 5-9, A*-Parallel first waits for some worker to be free, then (if needed) waits until the fringe is non-empty so the worker can be assigned the next node to be expanded from the fringe. If all workers have become idle while the fringe is still empty, this means no insertion in the fringe will happen anymore, which means there is no path to a goal so the search returns failure. (This corresponds to line 5 of A*-Graph-Search). Line 16 in A*-Parallel assigns an idle worker thread to execute Worker-Expand in lines 17-19. (This corresponds to lines 10-11 of A*-Graph-Search.) Finally, lines 11-13 in the A*-Parallel, corresponding to line 7 in A*-Graph-Search is where your work begins. Because there are workers acting in parallel it is not a simple task to determine when a goal can be returned: perhaps one of the busy workers was just about to add a really good goal node into the fringe.

```
 1: function A*-Parallel(problem, fringe, workers)
 2:     closed ← an empty set
 3:     fringe ← Insert(Make-Node(Initial-State[problem]), fringe)
 4:     loop do
 5:         while All-Busy(workers) do Wait
 6:         while fringe is empty do
 7:             if All-Idle(workers) and fringe is empty then
 8:                 return failure
 9:             else Wait
10:         node ← Remove-Front(fringe)
11:         if Goal-Test(problem, State[node]) then
12:             if Should-Return(node, workers, fringe) then
13:                 return node
14:         if State[node] is not in closed then
15:             add State[node] to closed
16:             Get-Idle-Worker(workers).Worker-Expand(node, problem, fringe)

17: function Worker-Expand(node, problem, fringe)
18:     child-nodes ← Expand(node, problem)
19:     fringe ← Insert-All(child-nodes, fringe)
```

[1] A non-blocking call means that the master thread continues executing its code without waiting for the worker to return from the call to the worker.

Consider the following possible implementations of the Should-Return function called before returning a goal node in A*-Parallel:

```
I    function Should-Return(node, workers, fringe)
         return true

II   function Should-Return(node, workers, fringe)
         return All-Idle(workers)

III  function Should-Return(node, workers, fringe)
         fringe ← Insert(node, fringe)
         return All-Idle(workers)

IV   function Should-Return(node, workers, fringe)
         while not All-Idle(workers) do Wait
         fringe ← Insert(node, fringe)
         return F-Cost[node] == F-Cost[Get-Front(fringe)]
```

For each of these, indicate whether it results in a complete search algorithm, and whether it results in an optimal search algorithm. Give a brief justification for your answer (answers without a justification will receive zero credit). Assume that the state space is finite, and the heuristic used is consistent.

(a) **(i)** [4 pts] **Implementation I**

Optimal? Yes / No. Justify your answer:

Complete? Yes / No. Justify your answer:

(ii) [4 pts] **Implementation II**

Optimal? Yes / No. Justify your answer:

Complete? Yes / No. Justify your answer:

(iii) [4 pts] **Implementation III**

Optimal? Yes / No. Justify your answer:

Complete? Yes / No. Justify your answer:

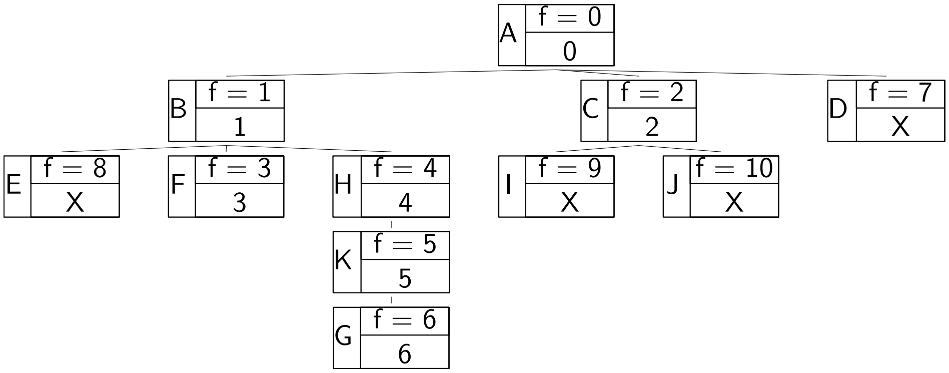
(iv) [4 pts] **Implementation IV**

Optimal? Yes / No. Justify your answer:

Complete? Yes / No. Justify your answer:

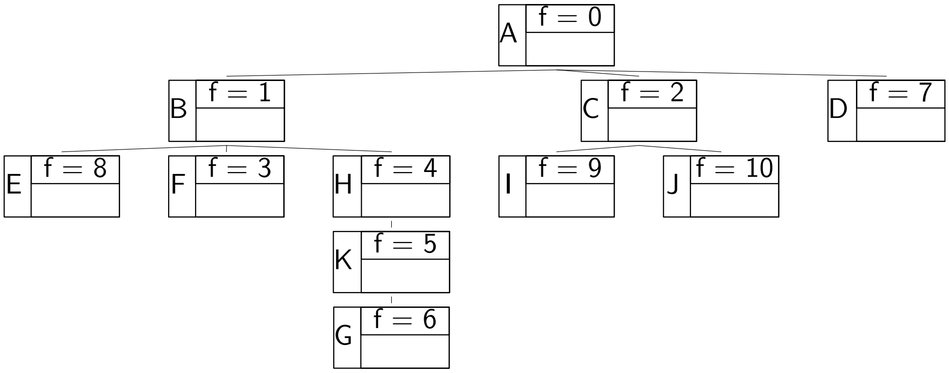
(b) Suppose we run A*-Parallel with implementation **IV** of the Should-Return function. We now make a new, additional assumption about execution time: Each worker takes exactly one time step to expand a node and push all of the successor nodes onto the fringe, independent of the number of successors (including if there are zero successors). All other computation is considered instantaneous for our time bookkeeping in this question.

A*-Parallel with the above timing properties was run with a single (1) worker on a search problem with the search tree in the diagram below. Each node is drawn with the state at the left, the f-value at the top-right ($f(n) = g(n) + h(n)$), and the time step on which a worker expanded that node at the bottom-right, with an 'X' if that node was not expanded. G is the unique goal node. In the diagram below, we can see that the start node A was expanded by the worker at time step 0, then node B was expanded at time step 1, node C was expanded at time step 2, node F was expanded at time step 3, node H was expanded at time step 4, node K was expanded at time step 5, and node G was expanded at time step 6. Nodes D, E, I, J were never expanded.

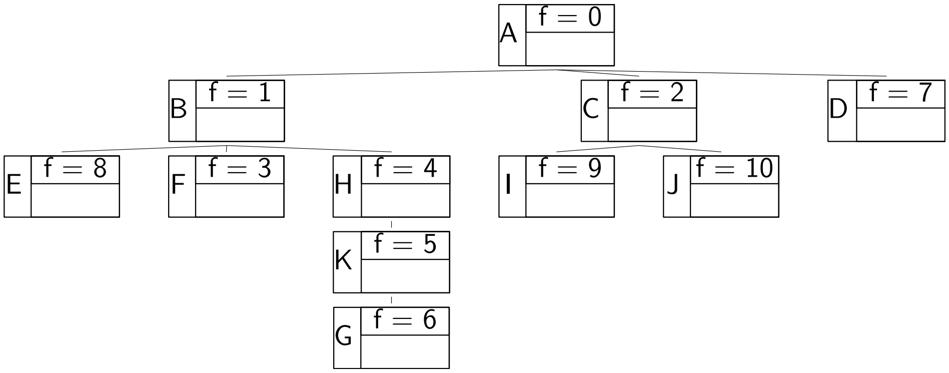


In this question you'll complete similar diagrams by filling in the node expansion times for the case of two and three workers. Note that now multiple nodes can (and typically will!) be expanded at any given time.

(i) [6 pts] Complete the node expansion times for the case of two workers and fill in an 'X' for any node that is not expanded.



(ii) [6 pts] Complete the node expansion times for the case of three workers and fill in an 'X' for any node that is not expanded.


A
f = 0
B
f = 1
C
f = 2
D
f = 7
E
f = 8
F
f = 3
H
f = 4
I
f = 9
J
f = 10
K
f = 5
G
f = 6