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	

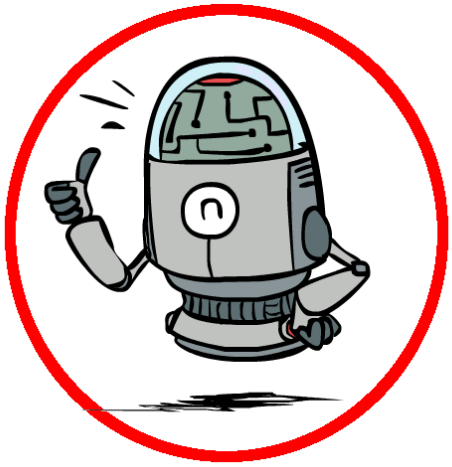
For staff use only:

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

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

Brad – because he has the least values left in his domain.

(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.

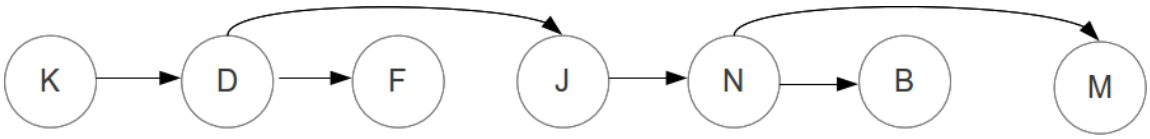
Michael's value will be assigned as 5, 3, 1, in that order.

Why these variables? They are the only feasible variables for Michael. Why this order? This is the increasing order of the number of constraints on each variable.

The only binary constraint incolving Michael is "Nick (N) must work on a question that's before Michael (M)'s question." So, only Nick's domain is affected by forward checking on these assignments, and it will change from {1, 2, 3, 4, 5} to {1, 2, 3, 4}, {1, 2}, and { } for the assignments 5, 3, 1, respectively.

(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.



K	D	F	J	N	B	M
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

Remaining values in each domain after the domain pruning right-to-left pass:
Kyle: 1
Donahue: 1,2
Ferguson: 1,2,3
Judy: 2,3,4,5,6
Nick: 1,2,3,4
Brad: 5
Michael: 1,3,5

(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.

Assigned Values after the left-to-right pass:

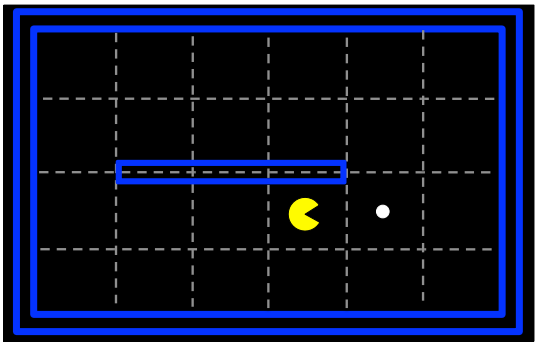
Kyle: 1
Donahue: 2
Ferguson: 3
Judy: 6
Nick: 4
Brad: 5
Michael: 5

Q3. [11 pts] Solving Search Problems with MDPs

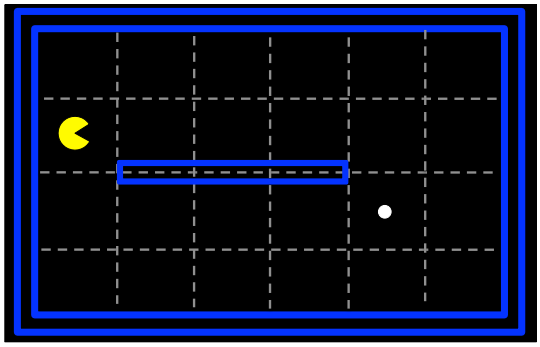
The following parts consider a Pacman agent in a deterministic environment. A goal state is reached when there are no remaining food pellets on the board. Pacman's available actions are $\{N, S, E, W\}$, but Pacman **can not** move into a wall. Whenever Pacman eats a food pellet he receives a reward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

1

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

1

The reason the answers are the same for both (b) and (a) is that there is no penalty for existing. With a discount factor of 1, eating the food at any future step is just as valuable as eating it on the next step. An optimal policy will definitely find the food, so the optimal value of any state is always 1.

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

5

The value function at iteration k is equivalent to the maximum reward possible within k steps of the state in question, B. Since the food pellet is exactly 5 steps away from Pacman in state B, $V_5(B) = 1$ and $V_{K<5}(B) = 0$.

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

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

Once again, since $\gamma = 1$, the optimal value of every state is the same, since the optimal policy will eventually eat the food.

(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?

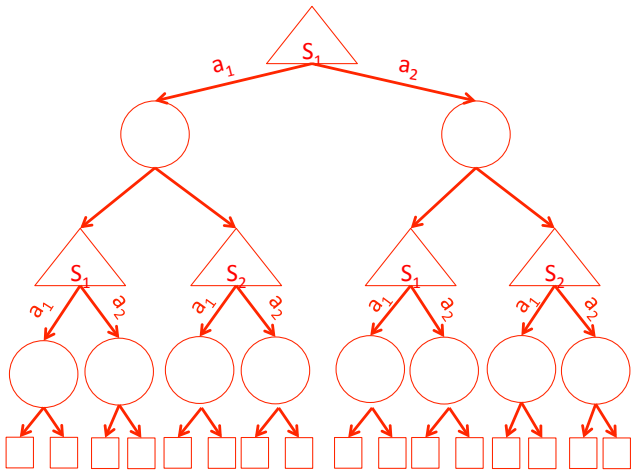
No. The Q-values for going *West* and *East* from state A are equal so there is no preference given to the shortest path to the goal state. Adding a negative living reward (example: -1 for every time step) will help differentiate between two paths of different lengths. Setting $\gamma < 1$ will make rewards seen in the future worth less than those seen right now, incentivizing Pacman to arrive at the goal as early as possible.

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.



The leaf nodes above will be the values of the previous iteration of the alternate update equation. Namely, if the value of the tree is $X_{k+1}(S_1)$, then the leaf nodes from left to right correspond to $X_k(S_1)$, $X_k(S_2)$, $X_k(S_1)$, $X_k(S_2)$, etc.

(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.

$X_k(s) = V_{2k}(s), \forall s$

The thing to demonstrate here is that X is doing two-step lookahead relative to V.

Why?

$$X_0(s) = V_0(s)$$

Run an iteration to update X. This is the same as updating V for two iterations. Hence,

$$X_1(s) = V_2(s)$$

Run another iteration to update X. This is the same as updating V for two iterations. Hence,

$$X_2(s) = V_4(s)$$

.

. . .

Hence,

$$X_k(s) = V_{2k}(s)$$

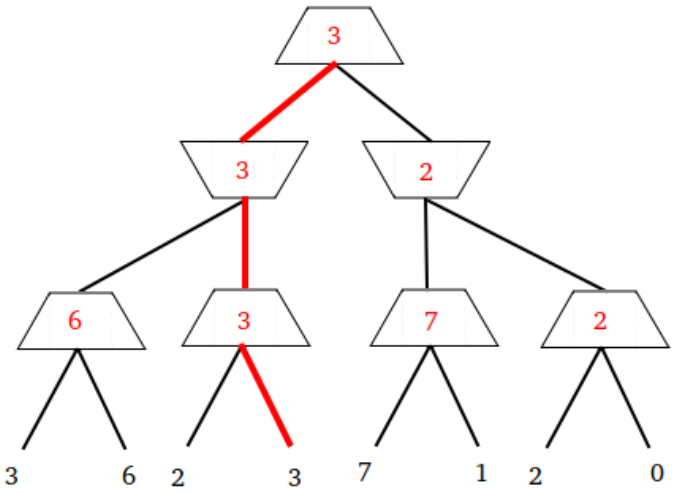
.

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.



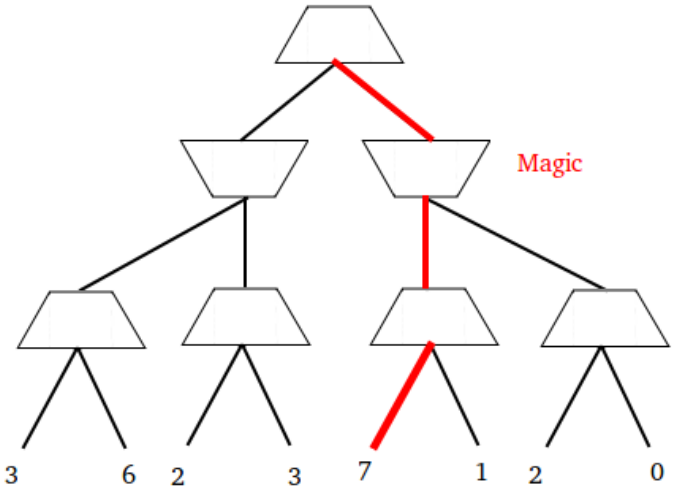
(b) Dark Magic

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** $c = 2$

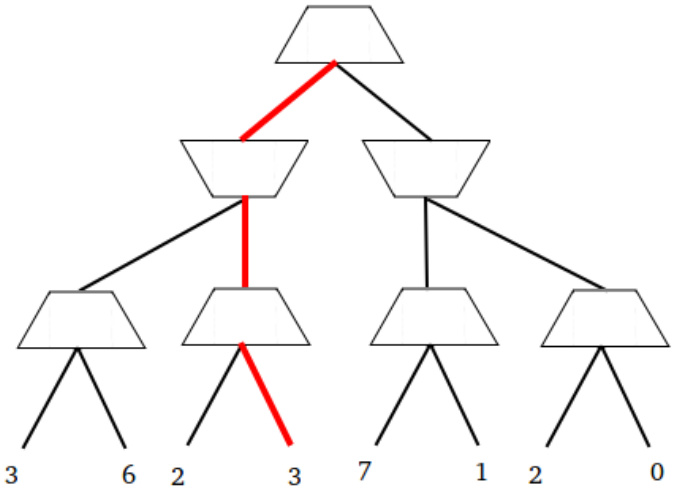
Consider the same game as before but now Pacman has access to his magic at cost $c = 2$. Is it optimal for Pacman to use his dark magic? If so, mark in the tree below where he will use it. Either way, mark what the outcome of the game will be and the sequence of actions that lead to that outcome.



Pacman goes right and uses dark magic to get 7-2=5. Not using dark magic would result in the normal minimax value of 3. Going left and using dark magic would have resulted in 6-2=4. So, in either case using magic benefits Pacman, but using it when going right is best.

(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.



Pacman doesn't use dark magic. Going left and using dark magic would result in 6-5=1, and going right and using dark magic would result in 7-5=2, while not using dark magic results in 3.

(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), UTILITY(state))
    end if
    v_min ← −∞
    v_max ← −∞
    for successor in SUCCESSORS(state) do
        vNext_min, vNext_max ← MIN-VALUE(successor)
        v_min ← max(v_min, vNext_min)
        v_max ← max(v_max, vNext_max)
    end for
    return (v_min, v_max)
end function
```

```
function MIN-VALUE(state)
    if state is leaf then
        return (UTILITY(state), UTILITY(state))
    end if
    v_min ← ∞
    min_move_v_max ← −∞
    v_magic_max ← −∞
    for state in SUCCESSORS(state) do
        vNext_min, vNext_max ← MAX-VALUE(successor)
        if v_min > vNext_min then
            v_min ← vNext_min
            min_move_v_max ← vNext_max
        end if
        v_magic_max ← max(vNext_max, v_magic_max)
    end for
    v_max ← max(min_move_v_max, v_magic_max − c)
    return (v_min, v_max)
end function
```

```
function MAX-VALUE(state)
    if state is leaf then
        return UTILITY(state)
    end if
    v ← −∞
    for successor in SUCCESSORS(state) do
        v ← max(v, MIN-VALUE(successor))
    end for
    return v
end function
```

```
function MIN-VALUE(state)
    if state is leaf then
        return UTILITY(state)
    end if
    v ← ∞
    for successor in SUCCESSORS(state) do
        v ← min(v, MAX-VALUE(successor))
    end for
    return v
end function
```

The first observation is that the maximizer and minimizer are getting different values from the game. The maximizer gets the value at the leaf minus c*(number of applications of dark magic), which we denote by v_{max}. The minimizer, as always, tries to minimize the value at the leaf, which we denote by v_{min}.

In $Max - Value$, we now compute two things.
(1) We compute the max of the children's v_{max} values, which tells us what the optimal value obtained by the maximizer would be for this node.

(2) We compute the max of the children's v_{min} values, which tells us what the minimizer thinks would happen in that node.

In $Min - Value$, we also compute two things.
(1) We compute the min of the children's v_{min} values, which tells us what the minimizer's choice would be in this node, and is being tracked by the variable v_{min}. We also keep track of the value the maximizer would get if the minimizer got to make their move, which we denote by $min\_move\_v_{max}$.

(2) We keep track of a variable v_{magic_max} which computes the maximum of the children's v_{max}.

If the maximizer applies dark magic he can guarantee himself $v_{magic\_max} - c$. We compare this with the $min\_move\_v_{max}$ from (1) and set v_{max} to the maximum of the two.

(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.

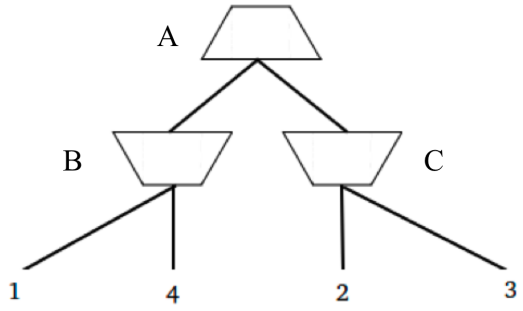
```
function MAX-VALUE(state)
    if state is leaf then
        return UTILITY(state)
    end if
    v ← −∞
    for successor in SUCCESSORS(state) do
        v ← max(v, MIN-VALUE(successor))
    end for
    return v
end function
```

```
function MIN-VALUE(state)
    if state is leaf then
        return UTILITY(state)
    end if
    v ← ∞
    for successor in SUCCESSORS(state) do
        v ← min(v, MAX-VALUE(successor))
    end for
    return v
end function
```

```
function MIN-VALUE(state)
    if state is leaf then
        return UTILITY(state)
    end if
    v ← ∞
    v_m ← −∞
    for state in SUCCESSORS(state) do
        temp ← MAX-VALUE(successor)
        v ← min(v, temp)
        v_m ← max(v_m, temp)
    end for
    return max(v, v_m − c)
end function
```

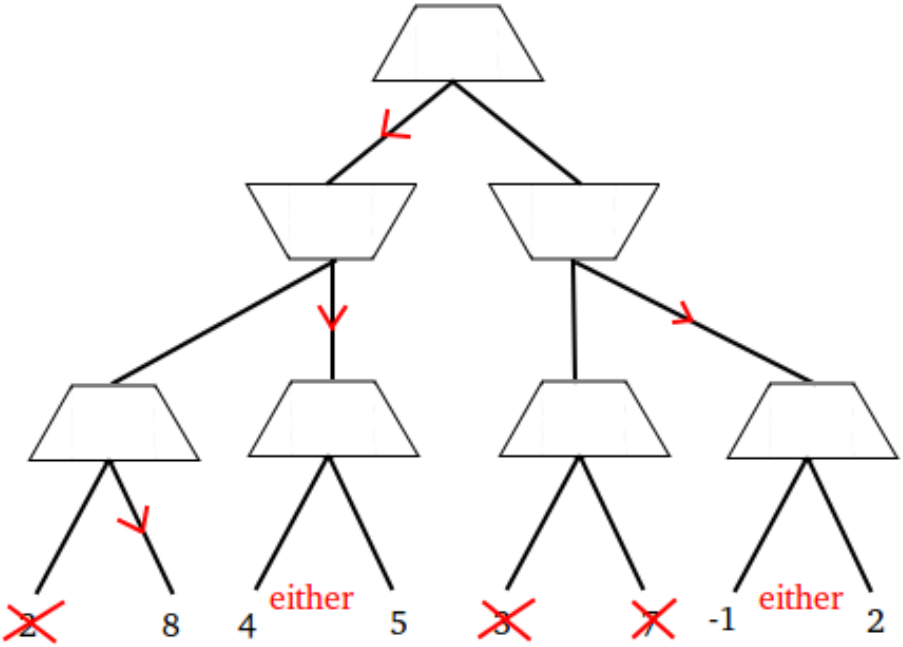
Q6. [10 pts] Pruning and Child Expansion Ordering

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.



The thing to understand here is how pruning works conceptually. A node is pruned from under a max node if it "knows" that the min node above it has a better – smaller – value to pick than the value that the max node just found. Similarly, a node is pruned from under a min node if it knows that the max node above it has a better – larger – value to pick than the value that the min node just found.

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:

Suppose we have a search problem with two paths to the single goal node. The first path is the optimal path, but nodes along this path take a really long time to expand. The second path is suboptimal and nodes along this path take very little time to expand. Then this implementation will return the suboptimal solution.

Complete? [Yes] / No. Justify your answer:

PARALLEL-A* will keep expanding nodes until either (a) all workers are idle (done expanding) and the fringe is empty, or (b) a goal node has been found and returned (this implementation of SHOULD-RETURN returns a goal node unconditionally when found). So, like standard A*-GRAPH-SEARCH, it will search all reachable nodes until it finds a goal.

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

Optimal? Yes / [No]. Justify your answer:

Not complete (see below), therefore not optimal.

Complete? Yes / [No]. Justify your answer:

Suppose there is just one goal node and it was just popped off the fringe by the master thread. At this time a worker can still be busy expanding some other node. When this happens this implementation returns false and we've "lost" this goal node because we've already pulled it off the fringe, and a goal node will never be returned since this was the only one.

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

Optimal? Yes / [No]. Justify your answer:

Optimality is not guaranteed. Suppose there is just a single node on the fringe and it is a suboptimal goal node. Suppose further that a single worker is currently working on expanding the parent of an optimal goal node. Then the master thread reaches line 10 and pulls the suboptimal goal node off the fringe. It then begins running GOAL-TEST in line 11. At some point during the execution of GOAL-TEST, the single busy worker pushes the optimal goal node onto the fringe and finishes executing WORKER-EXPAND, thereby becoming idle. Since it was the only busy worker when it was expanding, we now have ALL-IDLE(*workers*) and when the master thread finishes executing the goal test and runs SHOULD-RETURN, the ALL-IDLE check will pass and the suboptimal goal node is returned.

Complete? [Yes] / No. Justify your answer:

All goal nodes will be put back into the fringe, so we never throw out a goal node. Because the state space is finite and we have a closed set, we know that all workers will eventually be idle. Given these two statements and the argument from completeness of Implementation I that all reachable nodes will be searched (until a goal node is returned), we can guarantee a goal node will be returned.

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

Optimal? [Yes] / No. Justify your answer:

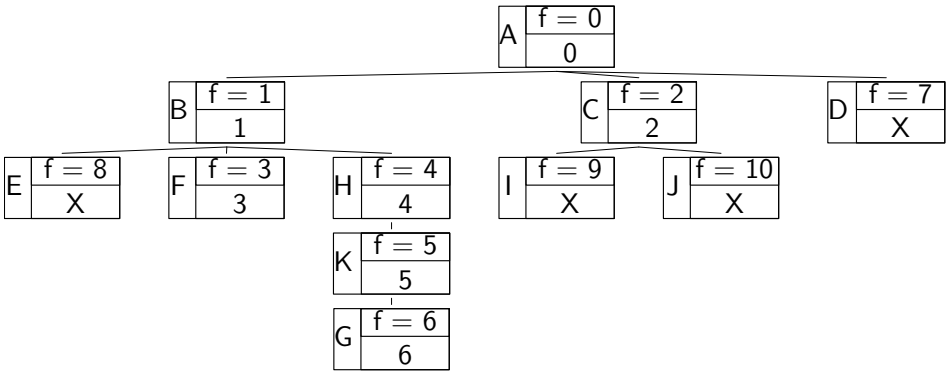
This implementation guarantees that an optimal goal node is returned. After WAITing for all the workers to become idle, we know that if there are any unexpanded nodes with lower F-cost than the goal node we are currently considering returning they will now be on the fringe (by the consistent heuristic assumption). Then, we re-insert the node into the fringe and return it only if it has F-cost equal to the node with the lowest F-cost in the fringe after the insertion. Note that even if it was *not* the lowest F-cost node in the fringe this time around, this might still be the optimal goal node. But not to worry; we have put it back into the fringe ensuring that it can still be returned once we have expanded all nodes with lower F-cost.

Complete? [Yes] / No. Justify your answer:

Optimal (see above), therefore complete.

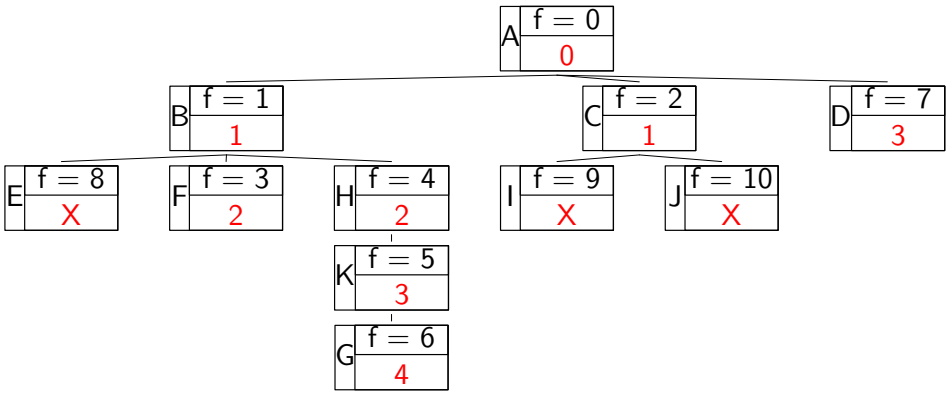
(b) Suppose we run A*-Parallel with implementation **IV** of the Should-Return function. We now make a new, additional assumption about execution time: Each worker takes exactly one time step to expand a node and push all of the successor nodes onto the fringe, independent of the number of successors (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.