

# yz709-AI-sup1

## 3 Search

[Question 3.1](#)

[Question 3.2](#)

[Question 3.3](#)

[Question 3.4](#)

[Question 3.5](#)

[Question 3.6](#)

[Question 3.7](#)

[Question 3.8](#)

## 4 Games

[Question 4.1](#)

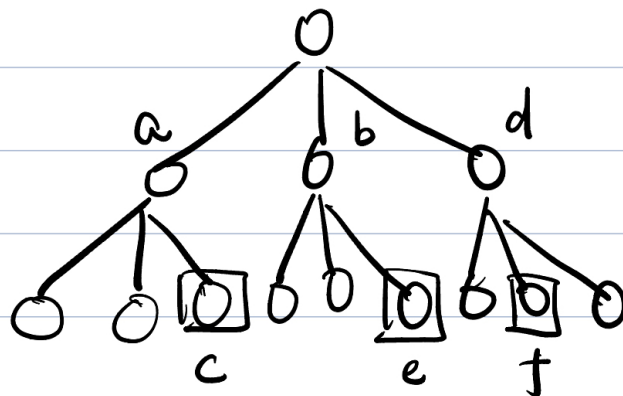
[Question 4.2](#)

[Question 4.3](#)

## 3 Search

### Question 3.1

1. Explain why breadth-first search is optimal if path-cost is a non-decreasing function of node-depth.
  - Optimality means the strategy could guarantee to find the best solution.
  - BFS searches level by level; it starts searching from the root node and expands all successor nodes at the current level before moving to the nodes at the next level.
  - Suppose we arrive at a goal node, but it is not the optimal solution, then there must be some nodes with a smaller path cost that haven't yet been explored. Since all path-cost is non-decreasing, all nodes that haven't been explored would have a path cost greater than the current node path cost; hence we have a contradiction. This implies that BFS would terminate at the optimal goal node.
  - e.g., in the following diagram, if the path cost to node  $A$  is smaller than node  $B$  and node  $D$ , then it will be explored first, and if the path cost to node  $C$  is smaller than both node  $B$  and  $D$ , then that means all child nodes of node  $B$  and  $D$  would have a path cost greater than node  $C$ ; thus node  $C$  is the optimal solution.



□ goals



Comments:

## Question 3.2

2. Iterative deepening depends on the fact that *the vast majority of the nodes in a tree are in the bottom level.*

- Denote by  $f_1(b, d)$  the total number of nodes appearing in a tree with branching factor  $b$  and depth  $d$ . Find a closed-form expression for  $f_1(b, d)$ .
- Denote by  $f_2(b, d)$  the total number of nodes generated in a complete iterative deepening search to depth  $d$  of a tree having branching factor  $b$ . Find a closed-form expression for  $f_2(b, d)$  in terms of  $f_1(b, d)$ .
- How do  $f_1(b, d)$  and  $f_2(b, d)$  compare when  $b$  is large?
- Assume the tree is complete and in the worst case, the goal is the last node at depth  $d$ .
- Define  $f_1(b, d)$ :
  - Since we have  $b^0$  at depth 0,  $b^1$  at depth 1, and  $b^d$  at depth  $d$ , the total number of nodes in the tree would be  $b^0 + b^1 + \dots + b^d = \frac{b^{d+1}-1}{b-1}$
- Define  $f_2(b, d)$ :
  - In a complete iterative deepening, the nodes at depth  $d$  would be generated once, and nodes at depth  $d - 1$  would be generated twice, nodes at depth 0 would be generated  $d + 1$  times.
  - Hence the total number of nodes generated would be:

$$\begin{aligned}
 f_2(b, d) &= \\
 f_1(b, 0) + f_1(b, 1) + \dots + f_1(b, d) &= \frac{1}{b-1}(b-1 + b^2 - 1 + \dots + b^{d+1} - 1) \\
 &= \frac{1}{b-1}\left(\frac{b^{d+1} - 1}{b-1} - d - 1 + b^{d+1} - 1\right) \\
 &= \frac{1}{b-1}(f_1(b, d) - d - 1) + f_1(b, d)
 \end{aligned}$$

- When  $b$  is large, the term  $\frac{1}{b-1}(f_1(b, d) - d - 1)$  would be negligible, hence  $f_2(b, d) \approx f_1(b, d)$ .



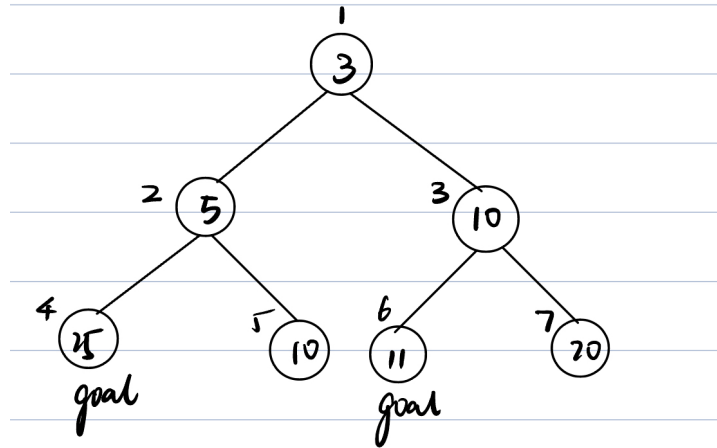
Comments:

### Question 3.3

3. The  $A^*$  tree-search algorithm does not perform a goal test on any state *until it has selected it for expansion*. We might consider a slightly different approach: namely, each time a node is expanded check all of its descendants to see if they include a goal.

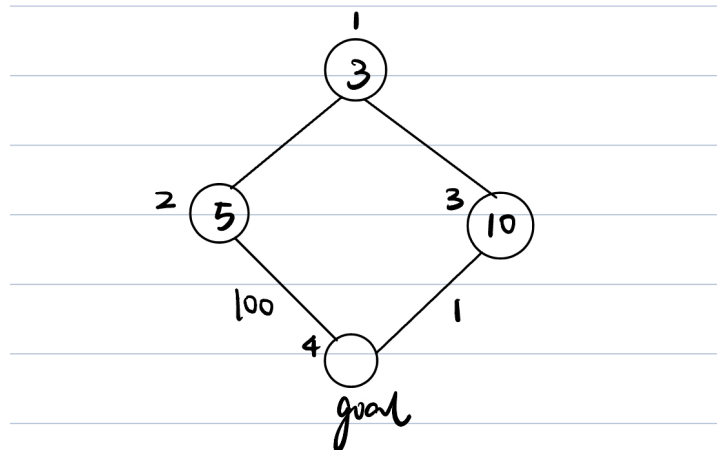
Give two reasons why this is a misguided idea, where possible illustrating your answer using a specific example of a search tree for which it would be problematic.

- (1) Not the optimal goal node: we cannot guarantee that nodes that haven't yet been explored have a greater  $f$  cost than the goal node we got.
  - For example, we have two goal nodes in the following diagram; because node 2 has a smaller  $f$  cost compared to node 3, we expand node 2 and check all its descendants, and we got the goal node that has an  $f$  cost of 25, however, one of the child nodes of node 3 is a goal node that has smaller  $f$  cost.



Assume the value labelled is the  $f$  cost of the node,  $f(s) = p(s) + h(s)$  where  $p(s)$  is the path cost of the starting node to this node and  $h(s)$  is the heuristic from this node to the goal state.

- (2) Not the optimal path to the goal node: in a graph, there may be multiple routes to a goal state; if we directly apply the goal test to the descendants, then we cannot guarantee the path is optimal.
  - For example, we have one goal node in the following example but two possible routes to it; when we expand node 2 and check to see if node 4 is the goal state, the  $f$  cost to node 4 is 105, but the path via node 3 to node 4 has a much smaller  $f$  cost, thus more optimal.



Comments:

### Question 3.4

4. The  $f$ -cost is defined in the usual way as

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

where  $n$  is any node,  $p$  denotes path cost and  $h$  denotes the heuristic. An admissible heuristic is one for which, for any  $n$

$$h(n) \leq \text{actual distance from } n \text{ to the goal}$$

and a heuristic is monotonic if for consecutive nodes  $n$  and  $n'$  it is always the case that

$$f(n') \geq f(n).$$

- Prove that  $h$  is monotonic if and only if it obeys the triangle inequality, which states that for any consecutive nodes  $n$  and  $n'$

$$h(n) \leq c_{n \rightarrow n'} + h(n')$$

where  $c_{n \rightarrow n'}$  is the cost of moving from  $n$  to  $n'$ .

- Prove that if a heuristic is monotonic then it is also admissible.
- Is the converse true? (That is, are all admissible heuristics also monotonic?) Either prove that this is the case or provide a counterexample.
- (i) Prove  $h$  is monotonic if and only if it obeys the triangle inequality, for any consecutive nodes  $n$  and  $n'$ , we have  $h(n) \leq c_{n \rightarrow n'} + h(n')$ :

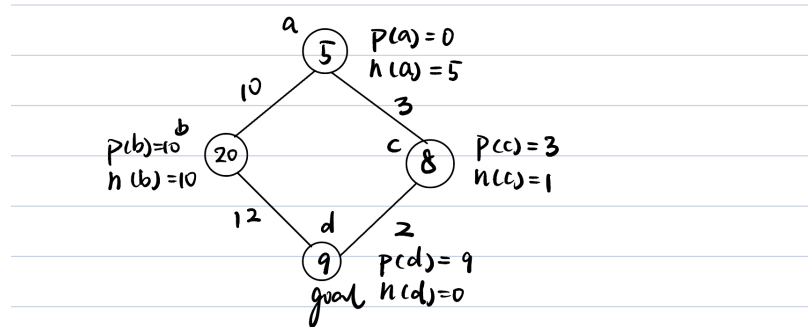
$h$  is monotonic, for any consecutive nodes  $n$  and  $n'$ ,  $f(n') \geq f(n)$

$$\iff p(n') + h(n') \geq p(n) + h(n)$$

$$\iff h(n') + p(n') - p(n) \geq h(n)$$

$$\iff h(n) + c_{n \rightarrow n'} \geq h(n')$$

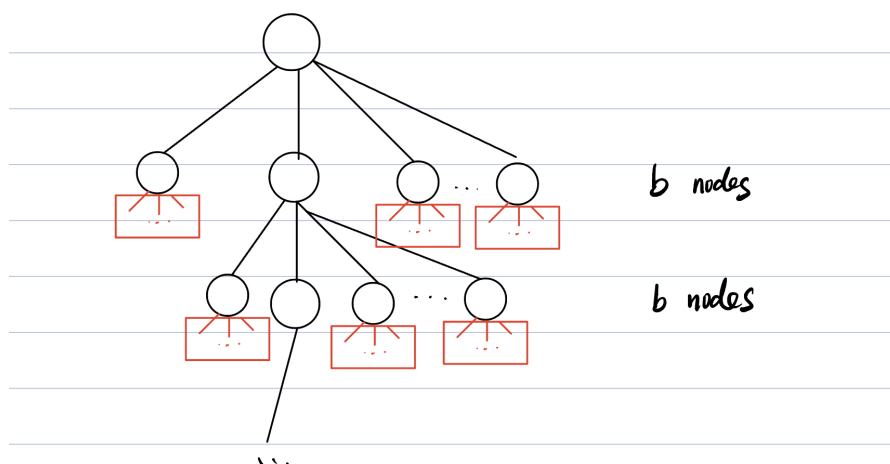
- (ii) Let  $\text{dist}(n, \text{goal})$  be the actual distance from node  $n$  to the goal node;
  - Base case: when  $n$  is the goal state  $h(\text{goal}) = 0 \leq \text{dist}(\text{goal}, \text{goal}) = 0$
  - Inductive case: (Induction hypothesis) assume the node  $n'$  has  $h(n') \leq \text{dist}(n', \text{goal})$ , without loss of generality, assume node  $n'$  is on the optimal path from node  $n$  to the goal node.
    - Since  $h$  is monotonic,  $f(n') \geq f(n)$  and obeys the triangle inequality  $h(n) \leq c_{n \rightarrow n'} + h(n')$ , hence we have  $h(n) \leq c_{n \rightarrow n'} + \text{dist}(n', \text{goal}) = \text{dist}(n, \text{goal})$
- (iii) The converse is not true. In the following diagram, all nodes have an admissible heuristic  $h$  because we ensure  $h(s) \leq \text{dist}(s, \text{goal})$ , but the heuristic  $h$  is not monotonic because  $h(a) = 5 > c_{a \rightarrow c} + h(c) = 3 + 1 = 4$



Comments:

### Question 3.5

- In RBFS we are replacing  $f$  values every time we backtrack to explore the current best alternative. This seems to imply a need to remember the new  $f$  values for all the nodes in the path we're discarding, and this in turn suggests a potentially exponential memory requirement. Why is this not the case?
- RBFS only remembers the new  $f$  value of the root of the subtree rather than all nodes under the subtree, this  $f$  value represents how good a discarded path could be so that we can easily return to it later.
  - So in each recursive function call of RBFS, it only preserves the nodes on the path currently in exploration and the siblings of the nodes on the path, thus the space requirement is  $O(bd)$





Comments:

## Question 3.6

6. In some problems we can simultaneously search:

- *forward* from the *start* state
- *backward* from the *goal* state

until the searches meet. This seems like it might be a very good idea:

- If the search methods have complexity  $O(b^d)$  then...
- ...we are converting this to  $O(2b^{d/2}) = O(b^{d/2})$ .

(Here, we are assuming the branching factor is  $b$  in both directions.) Why might this not be as effective as it seems?

- Not efficient when we have multiple goal states and starting states: if we have  $n$  goal states and  $m$  starting states, then pairing states up gives us  $nm$  ways of carrying out the bidirectional search. It would be more efficient to just begin from the starting state and search for the optimal goal state. This gives us  $m$  ways of carrying out the search.
- This bidirectional search requires us to know the goal states, hence cannot be used under scenarios where goal states are unknown.
- We have to design a robust termination state where the two routes meet; this is challenging because the two routes are run in parallel.



Comments:

## Question 3.7

7. Suggest a method for performing depth-first search using only  $O(d)$  space.

- Store siblings of a node in a linked list, and in the fringe, we only need to keep a pointer to the first node in the linked list.
- Previously we store  $O(bd)$  nodes, but now each depth only requires storing one node, so the space complexity would be  $O(d)$ .

```

fringe = [s0] # fringe: stack
while true:
    if fringe.empty(): return None
    s = fringe.pop() # pop most recent node
    if s.visited = True: pass
    if s.next != None: fringe.add(s.next) # add sibling nodes
    if goal(s): return (Some s)
    s.visited = True
    fringe.add(expand(s)) # assume expand(s) gives a LinkedList of children

```



Comments:

## Question 3.8

8. One modification to the basic local search algorithm suggested in the lectures was to make steps probabilistically, but only if the value of  $f$  is *improved*.

- (a) Would it be a good idea to also allow steps that move to a state with a *worse* value for  $f$ ?
- (b) Suggest an algorithm for achieving this, such that you have some control over the balance between steps that increase or decrease  $f$ .

- (a):
  - Yes, because that would benefit us in finding a global maximum instead of getting stuck in a local maximum; between the global maximum and the local maximum, there may have nodes with a lower  $f$  cost which prevents us from getting to the global maximum.
  - However, this approach also generates more searching, and we don't have a criterion to stop; hence we have to specify some iteration limit. Due to the manually controlled iteration limit, we cannot guarantee an optimal algorithm that can get the global maximum.
- (b): The probability depends on the difference in  $f$  cost between two nodes, where  $N(s)$  denotes the neighbour of  $s$ , constants  $\kappa_1$  and  $\kappa_2$  can be controlled to balance between uphill and downhill, the total probability should be normalised to 1.

$$\begin{aligned}
 N^+(s) &= \{s' \in N(s) \mid f(s') \geq f(s)\} \\
 N^-(s) &= \{s' \in N(s) \mid f(s') < f(s)\} \\
 Pr(s') &= \begin{cases} \kappa_1(f(s) - f(s')) & \text{if } s' \in N^-(s) \\ \kappa_2(f(s') - f(s)) & \text{otherwise} \end{cases}
 \end{aligned}$$



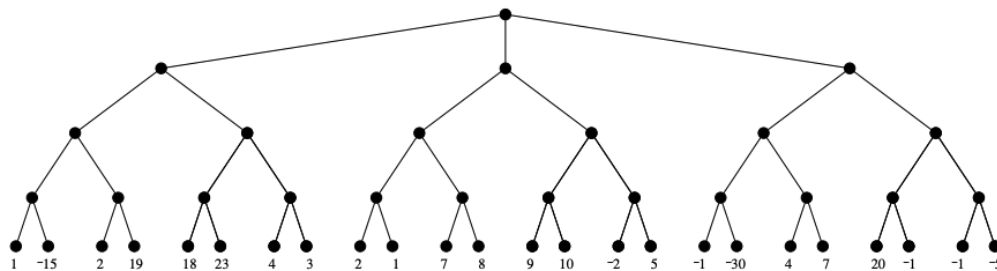


Comments:

## 4 Games

### Question 4.1

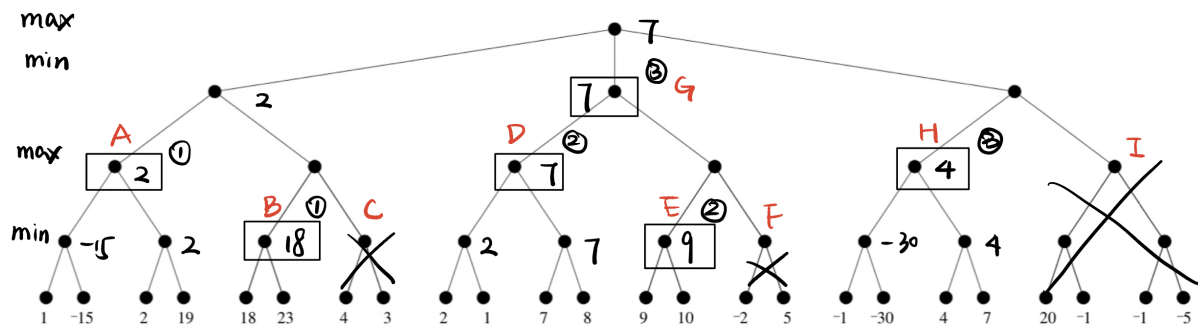
1. Consider the following game tree:



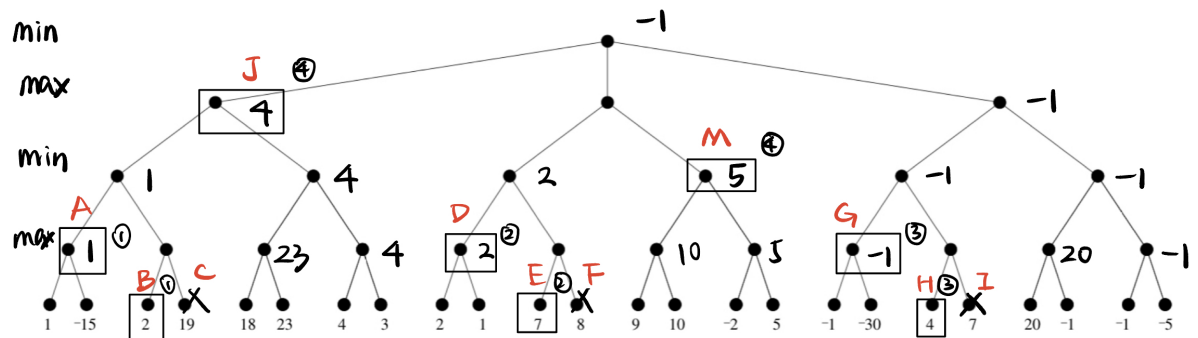
Large outcomes are beneficial for Max. How is this tree pruned by  $\alpha - \beta$  minimax if Max moves first? (That is, Max is the root.) How is it pruned if Min is the root, and therefore moves first?

*This is the initial attempt, I found out my mistakes after doing question 4.2.*

- Max starts first: we would prune three subtrees rooted at  $C$ ,  $F$  and  $I$ .
  - (1) Min can force a score of 2 from the subtree rooted at  $A$ , since the value of the subtree rooted at  $B$  is  $18 > 2$ , hence there is no need to further explore the siblings as Min has a better choice earlier.
  - (2) Min can force a score of 7 from the subtree rooted at  $D$ , since the value of the subtree rooted at  $E$  is  $9 > 7$ , hence there is no need to further explore the siblings as Min has a better choice earlier.
  - (3) Max can force a score of 7 from the subtree rooted at  $G$ , since the value of the subtree rooted at  $H$  is  $4 < 7$ , hence there is no need to further explore the siblings as Max has a better choice earlier.



- Min moves first: we would prune the subtree rooted at  $C$ ,  $F$  and  $I$ .
  - (1) Min can force a score of 1 from the subtree rooted at  $A$ , since the value of the subtree rooted at  $B$  is  $2 > 1$ , hence there is no need to further explore the siblings as Min has a better choice earlier.
  - (2) Min can force a score of 2 from the subtree rooted at  $D$ , since the value of the subtree rooted at  $E$  is  $7 > 2$ , hence there is no need to further explore the siblings as Min has a better choice earlier.
  - (3) Min can force a score of -1 from the subtree rooted at  $G$ , since the value of the subtree rooted at  $H$  is  $4 > -1$ , hence there is no need to further explore the siblings as Min has a better choice earlier.
  - (4) Min can force a score of 4 from the subtree rooted at  $J$ , since the value of the subtree rooted at  $M$  is  $5 > 4$ , hence there is no need to further explore the siblings as Min has a better choice earlier, but there are no siblings for this subtree, so we don't have to prune it.



Comments:


## Question 4.2

2. Implement the  $\alpha - \beta$  pruning algorithm and use it to verify your answer to the previous problem.

Implement in Java and package as a jar file to include both source and compiled code.

PartIB-coursework/AI at main · KyraZzz/PartIB-coursework

Coursework in one place!! Contribute to KyraZzz/PartIB-coursework development by creating an account on GitHub.

 <https://github.com/KyraZzz/PartIB-coursework/tree/main/AI>

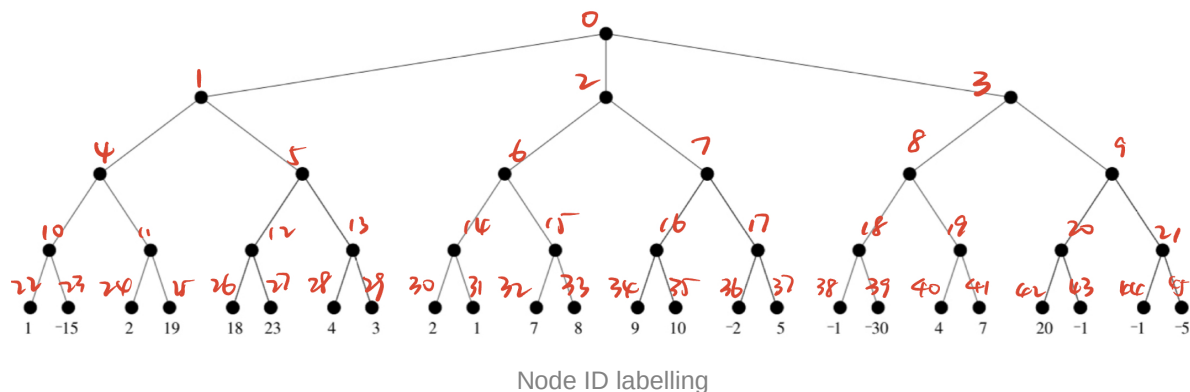
KyraZzz/**PartIB-coursework**

Coursework in one place!!



 1 Contributor
  0 Issues
  0 Stars
  0 Forks

The Jar file could be downloaded from the Github Link

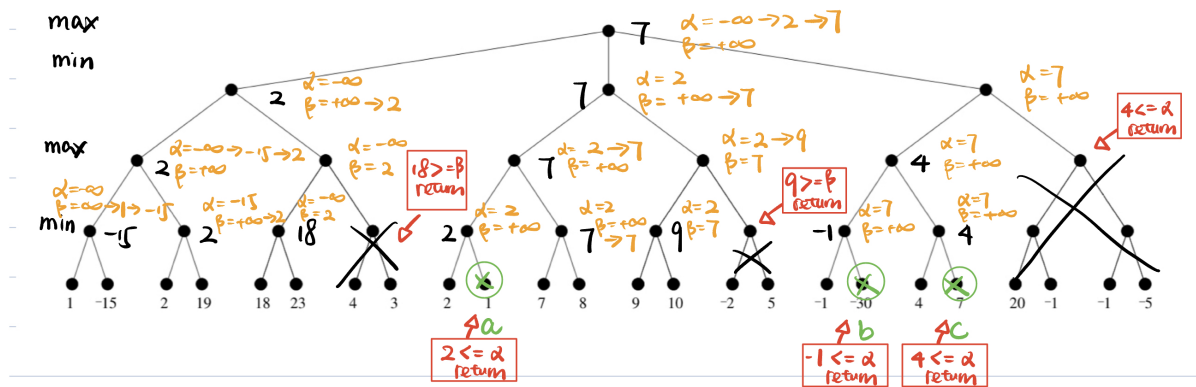


```

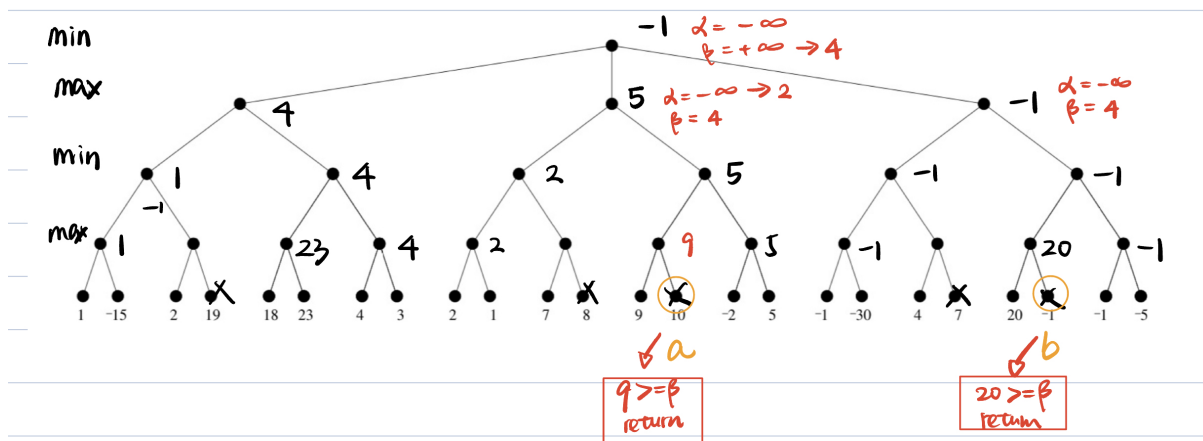
Max goes first:7
nodes not visited:
[9, 13, 17, 20, 21, 28, 29, 31, 36, 37, 39, 41, 42, 43, 44, 45]
Min goes first:-1
nodes not visited:
[25, 33, 35, 41, 43]

```

- When **Max** goes first, I didn't spot nodes 31, 39 and 41. These subtrees are pruned due to the fact that the previous sibling node has a value  $\leq \alpha$ ; therefore, player Maxima will never move here.



- When **Min** goes first, I didn't spot nodes 35 and 43. These subtrees are pruned due to the fact that the previous sibling node has a value  $\geq \beta$ ; therefore, player Minima will never move here.



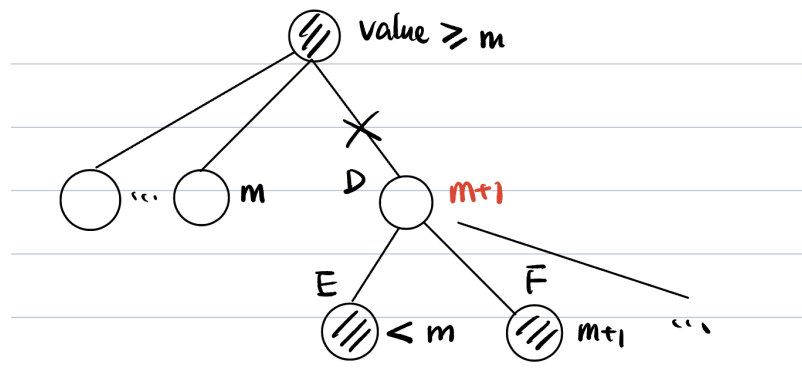
Comments:

### Question 4.3

3. Is the minimax approach to playing games optimal against an imperfect opponent? Either prove this is the case or give a counterexample.

- The Minimax approach with the complete tree is optimal against an imperfect opponent. Assume we are Max and the opponent is Min; if Min chooses a value  $> m$ , we can update the parent node to have that value; hence the result is still optimal.
- If we applied  $\alpha$ - $\beta$  pruning, as shown below: when the subtree rooted at node  $E$  has a value  $< m$ , since Max has a better choice earlier in the game, we would stop

searching through the child node of the subtree rooted at  $D$ . Under this scenario, If Min chooses some value  $> m$  at node  $D$ , but Max has already pruned the subtree rooted at node  $D$ , then we no longer have an optimal result.



Comments: