William Dahl
ICSI 435 Artificial Intelligence
HW 2
November 2nd, 2018

1.) **Informed Search**

i.) **Greedy Search Description**

- First I check if the current vertex we are at is the Goal.
- If it is, I return from the method
- If it is not, I loop through the neighbors of the current vertex
- If the vertex has not already been visited
  - I then compare the value of the heuristic function for the current vertex to the vertex I am currently going to go to next (initially initialized to the first neighbor in the graph).
- If the current vertex's heuristic is smaller than the one I am currently about to go to I set the current vertex to be the one I am most likely to go to.
- After iterating through all the neighbors I add the one that I will be going to next to the path and visited Nodes.
- I then recursively call the search function using the next vertex as the starting point

ii.)**Results of Greedy Search**

- Nodes expanded: ['S', 'E', 'R', 'F', 'G']
- Path returned: ['S', 'E', 'R', 'F', 'G']

iii.) **A* Search Description**

- while there are known un-expanded nodes
- get the node with the smallest total cost
- if that node is the goal return the path and expanded nodes
- else
- remove node from the un-expanded list and add it to the expanded list
- loop through the neighbors of the current vertex
- if the neighbor has already been expanded skip it
- else
- sum the total edge weight of the current node with the weight of the edge to its neighbor
- if the neighbor is not in the set of known nodes add it
- else check if the current total edge wight is greater than or equal to the total edge weight of the neighbors
- if so skip it
- else
- set the parent of the neighbor to he current node
- set the total edge weight for the neighbor to the current edge weight
- set the total weight for the neighbor to the total edge weight of the neighbor + the heuristic of the neighbor
- delete the current node from the total weight dictionary
- to get the path
- loop through the parent dictionary created when going through the graph
- set the current node to the parent of the current node
- add the new current node to the path
- reverse the path so it is the right way

iv.) **Results of A* Search**

- Nodes expanded: ['S', 'D', 'E', 'R', 'B', 'F', 'A', 'G']
- Path returned: ['S', 'D', 'E', 'R', 'F', 'G']

v.) Yes, h is admissible. Every heuristic in the graph is less than or equal to the actual distance between the node and the goal.

vi.) Yes, h is constant because for every pair of nodes A and C, h(A) – h(C) <= cost(Ato C).

2.) Constraint satisfaction Problems

i.) After a value is assigned for A and forward checking is done for A the domains for B, D, and E will change.

ii.) After forward checking is done for A and then a value is assigned to D and forward checking is done for D the domains that might change would be E and F

iii.) After assigning a value to A and enforcing arc consistency the domains that may be changed are D and E.

iv.) After a value is assigned to A and and arc constancy is enforced and then a value is assigned to D and Arc Consistence is enforced there are no domains that will be effected.

v.)No, there is not a valid solution, because the nodes can only have one of two values and D and E share the same parent and are also adjacent to each other. For example, when A is black, D and E will both need to be white however, D and E cannot be the same color.

3.) Adversarial search

i.) **Adversarial search description**

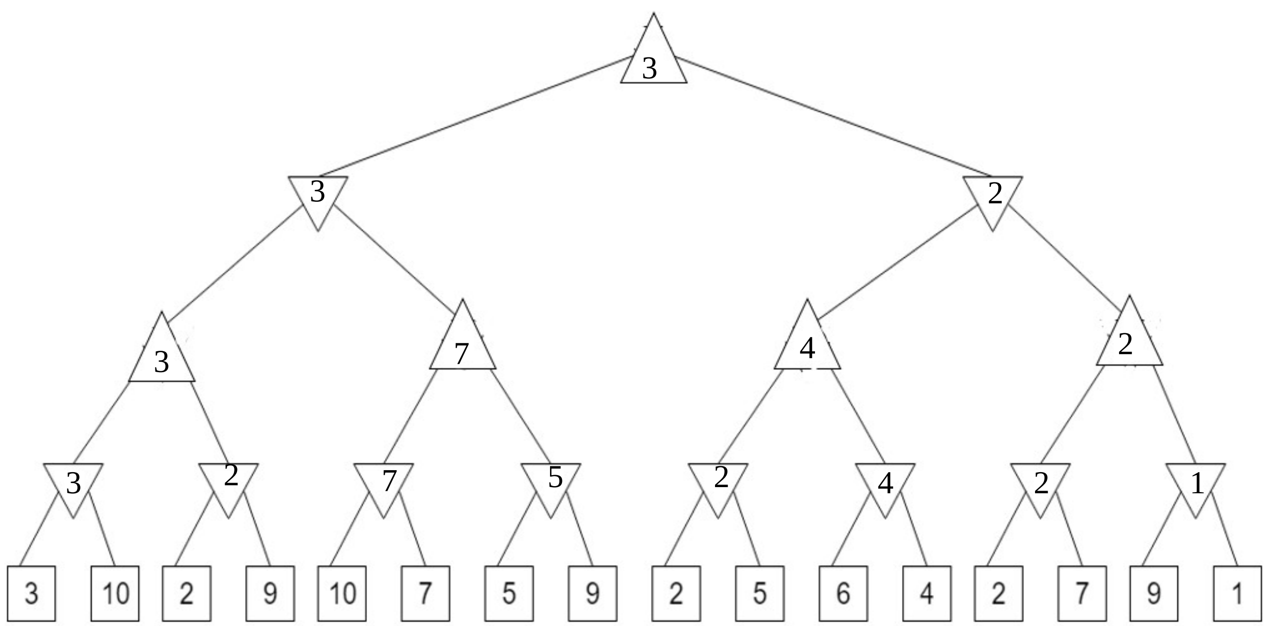
- First we pass the root of the tree to the value function.
- Then we get the type of the node that the root is based on its root Id.
- If the the node is a terminal node we return the value stored I that node.
- If the node is a max node we return the value returned by the max_value function done on the node
- If the node is a min node we return the value returned by the min_value function done on the node.
- In the max_value function
  - we set the initial state of the value to be returned to 0 (this is smaller than the smallest possible number in the tree.
  - Next we make a list of the children coming off of the node
  - We then loop through the the list of children and get the max value between them.
  - We get the value of the children through a recursive call to the value function with the child as the state
  - We then set the value of that node to that max value
  - then we return the value
- In the min_value function
  - we set the initial value to be returned to 100 (this is larger than any possible value in the tree)
  - next we make a list of the nodes children.
  - We loop through the children getting the min value between them.

- We get the value of the children by preforming a recursive call to the value function on each of the children
- we then set the value of that node to the the min value
- then we return the min value
- to get the path we simply make a call to the get_path function using the root of the tree as our starting state
- We append the node id to the path
- then we check if the node has any children
- if so we check if the left child's value is the same as our output value from the min-max algorithm
- if it is then we make a recursive call to the the get_path function using the left children
- other wise we make the call using the right children

ii.) **Adversarial search output**

- **Output Value: 3**
- **Output Path: ['l1_0', 'l2_1', 'l3_3', 'l4_7', 'l5_15']**
- *l1_0 represents a node at the first level of the tree and at index 0 in the tree

iii.) Output value: 3



Output path: [0, 1, 3, 7, 15]

iv.) Alpha beta pruning description:

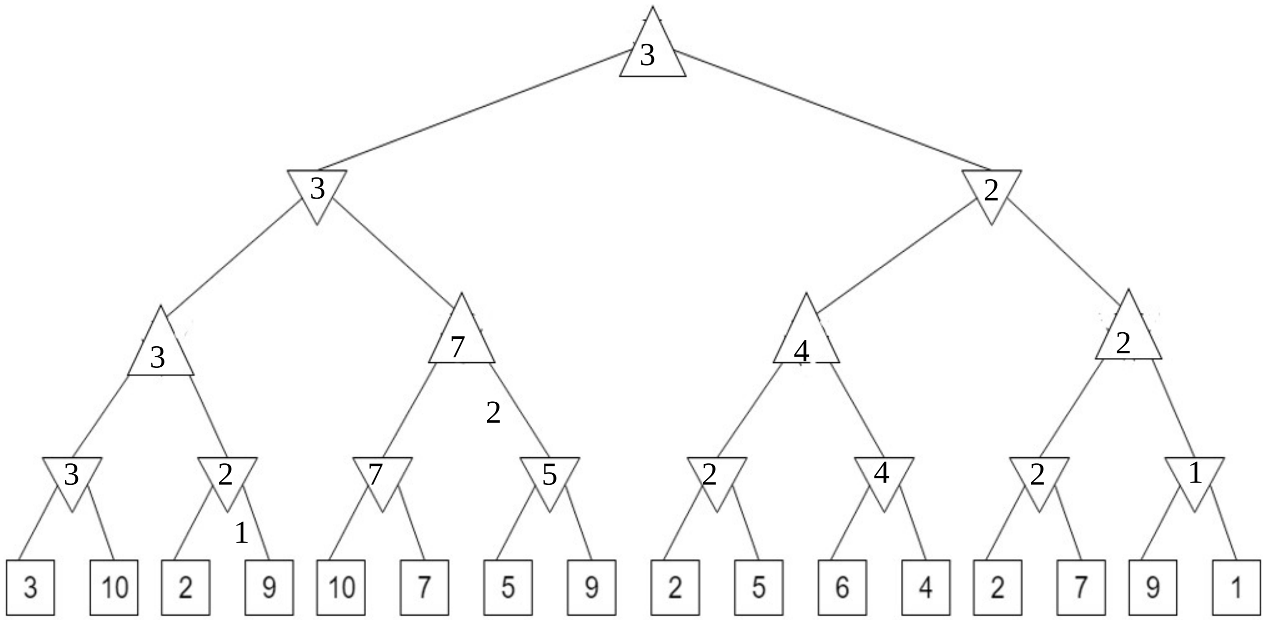
- First we call the value function with the root of the tree as the starting node along with our initial alpha and beta values. (a = 0 because that is smaller than the smallest possible value in the tree, and b = 11 because that is bigger than the largest possible value in the tree.

- In the value function we check the type of our current node and then call the respective function for a min node or max node, or we return the value if it is a terminal node.
- In the max_value function
  - we get out list of children and we loop through them getting the max value between the two
  - next we check if that value is larger than or equal to the current beta
  - if so we return that value
  - if not then we set the new alpha to the max between the current alpha and the new found value
  - we then return the found value
- In the min_value function
  - We get our list of children and then loop through them getting the smallest value between them
  - if that value is less than or equal to the current value we return that value
  - else we set the new beta to the min between the current beta and the found value
  - we then return the found value.
- To get the path we call the get_path function with the output from the search and the root of the tree
- we then append the nodes id to the path
- if the node has children we check if the left child has the same value as the output value.
- If so we call the get_path function with the left child as the staring point
- else we make the call on the right child

v.) Alpha beta pruning output

- **Output value: 3**
- **Output Path: ['l1_0', 'l2_1', 'l3_3', 'l4_7', 'l5_15']**
- *l1_0 represents a node at the first level of the tree and at index 0 in the tree

vi.)



1.) The terminal value 9 is the first value to be pruned. It is pruned because the fist value found was less than or equal to the current alpha which at this point is 3. The value for beta is +infinity

2.) The second spot of pruning is the sub tree with the root at index 10. The is because the fist value found from the sub tree at index 4 is larger than the current value for beta which at this point is 3. the value for alpha is -infinity.

No, the pruning does not get us the output path, because no pruning is done down the right side of the tree.