**CEN-435 Introduction to Artificial Intelligence**

**Homework 2 [30 pts total]**

**Greedy/A\* Search, CSP, MiniMax, AlphaBeta Pruning**

**Instructions**

In this homework you will use Python to implement heuristic graph search and A\* graph search we have covered in class, also the minimax tree search and alpha-beta pruning. Implementing these algorithms in program code is important, as these advanced search and planning methods are widely applied in various of AI applications and video games.

The homework should be done in Python (Python 3.X is recommended). Include a **documentation** (in .doc or .pdf format) that describes the complete steps of your solutions for each problem including: (1) The answer to any questions posed, (2) any results (include print outs or terminal outputs), and (3) the name of any Python module(s) and function(s) (Numpy, Matplotlib, etc.) you used. Discussion is allowed, but you must submit your own write-up and list your collaborators.

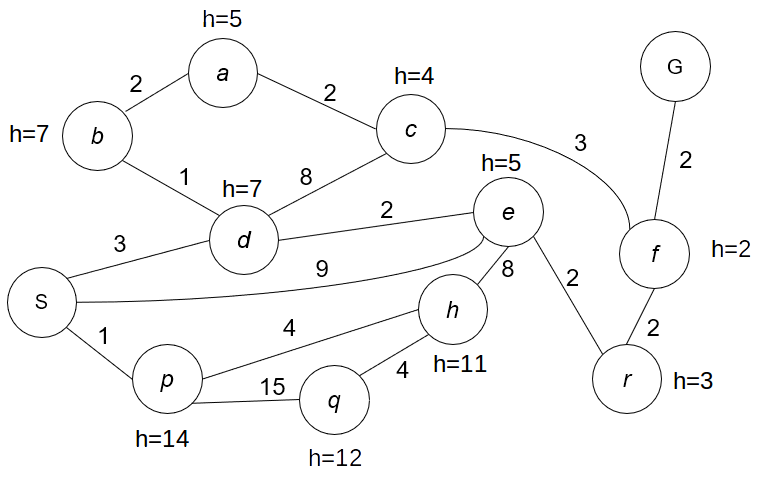
Zip all your documents with the **source code** (.py files, if there are self-implemented modules, organize them in different folders), **documentation**, and **data** (image files, text, etc.) into a file called **[Lastname\_FirstInitial\_Homework2.zip]** and upload it to the Blackboard prior to the due date.

Make sure your code works when you send it out. Please use as less third-party libraries as possible cause it is easy for instructors to reproduce your results. If you used the libraries that requires installation, please list the dependencies and its version in your ***README*** file or using pip to generate the dependency list like pip freeze > requirements.txt.

For late assignments, you will receive 10% off per day on any assignment handed in late up to a week. However, after a week on any given homework you will receive no credit for the assignment. So please start your assignment ASAP.

**Homework Description**

1. **Informed search (10pt)**



Refer to the weighted undirected graph and heuristic function h with values shown in the figure, where h indicates the heuristic estimation of how close the current state is to the goal. Use h(S) = 10. Answer following questions:

(1) Write code to find the shortest path from S to G using greedy search. Similar to HW1, your code should return **the search state that are visited and enqueued into the PQ** and the **path with lowest cost that the greedy search finds**. Manually exam the results to see if they are correct. (5pt)

(2) Write code to find the shortest path from S to G using A\* search. Similar to HW1, your code should return **the search state that are visited and enqueued into the PQ** and the **path with lowest cost that A\* finds**. Manually exam the results to see if they are correct. (5pt)

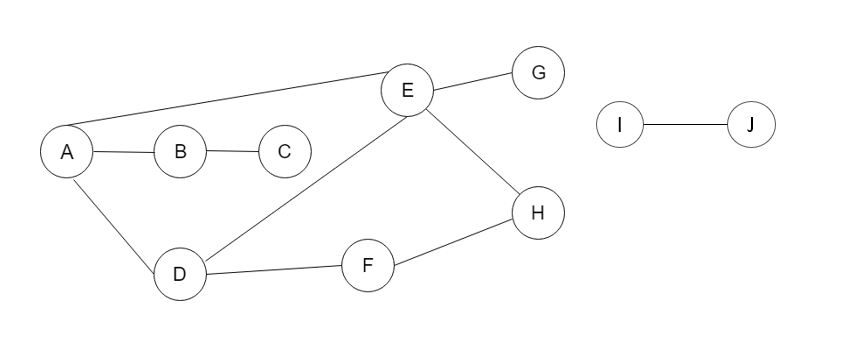
Note:

1. implement (1), (2) in Python. Put the source code in a folder named ‘*code*’ and record results in your documentation like ‘*hw2\_solution.doc’ or ‘.pdf*’.
2. You should use a while loop with a priority queue (PQ) implementation, and use either adjacency matrix or linked list implementation for graph representation.
3. **Constraint satisfaction problems (5pt)**

Consider a constraint satisfaction problem (CSP): coloring the given graph with only two colors {black, white} (i.e. binary constraint). Thus, adjacent nodes must have different colors. Initially, no states have been assigned.

For each of the following scenarios, list all variables that will be changed after the specified changes or conditions. For arc consistency check, refer to the pseudo code from Slide 42 of CSPs\_I lecture slides for arc consistency propagation. Here we follow alpha-numeric order when visiting nodes in the CSP graph. Also, we assume that the consistency propagation continues until the propagation queue is empty. In other words, here we continue the propagation even if we find out that when no solution to the CSP can be found.

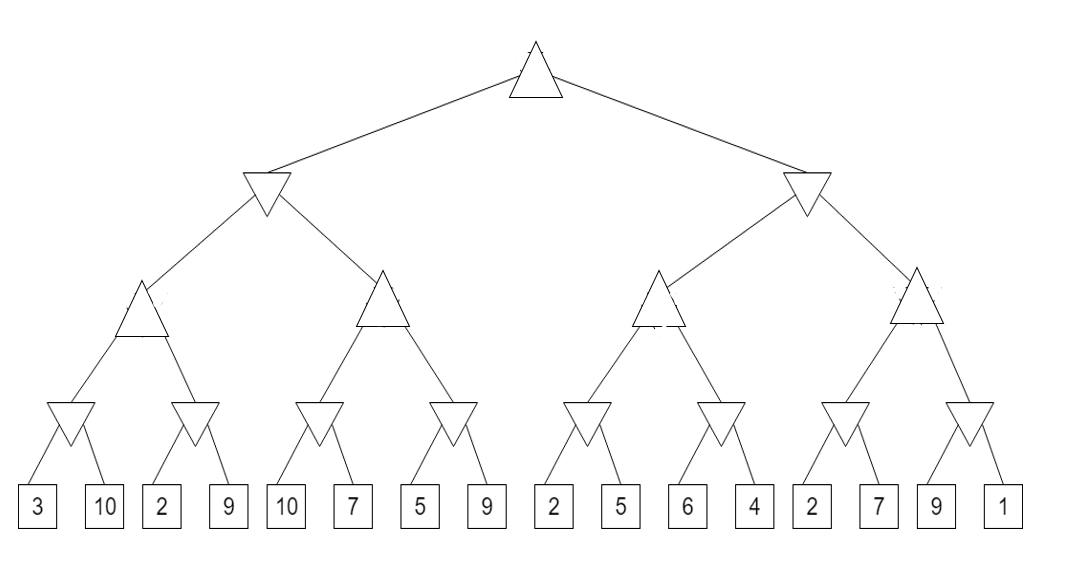
Explain how you obtain your answer in your documentation. This is a written assignment, so **no coding is involved in this problem**.



1. [1 pt] After a value is assigned to A, which domains might be changed after **forward checking** for A?
2. [1 pt] After a value is assigned to A, then **forward checking** is run for A. Then a value is assigned to D. Which domains might be changed as a result of running **forward checking** for D?
3. [1 pt] After a value is assigned to A, which domains might be changed as a result of enforcing **arc consistency** after this assignment?
4. [1 pt] After a value is assigned to A, and then **arc consistency** is enforced. Then a value is assigned to D. Which domains might be changed after enforcing **arc consistency** after the assignment to D?
5. [1 pt] Is there a valid solution for assigning colors to all graph nodes? Why?
6. **Adversarial search (15 pt)**

Below is a **minimax** game search tree, the leaf nodes contain terminal values.

represents the max node. represents the min node.



1. [5 pt] Implement the **minimax** search in Python. Use the above tree as input and return the chosen terminal state. Print the output value and output path.
2. [2 pt] Manually complete the minimax search tree by filling the non-leaf node. What is the output value and output path?
3. [5 pt] Implement **alpha-beta pruning** for the minimax tree search. Apply pruning from the leftmost to the rightmost. Return the chosen terminal state. Print the output value.
4. [3 pt] Show the terminal states of the minimax tree that has been pruned by alpha-beta pruning. List explicitly which leave node(s) or subtree(s) are pruned. Explain the pruning process and the value of alpha and beta there. Do we obtain the output path from the alpha-beta pruning?

Hint: This question involves both coding and hand-written answers. For the coding part, you should build a tree data structure to store the value and min-max operation flag for each node.

Sample code to implement binary tree with Python

class BinaryTree():

def \_\_init\_\_(self,rootid):

self.left = None

self.right = None

self.rootid = rootid

self.value = value

def getLeftChild(self):

return self.left

def getRightChild(self):

return self.right

def setNodeValue(self,value):

self.value = value

def getNodeValue(self):

return self.value

def insertRight(self,newNode):

if self.right == None:

self.right = BinaryTree(newNode)

else:

tree = BinaryTree(newNode)

tree.right = self.right

self.right = tree

def insertLeft(self,newNode):

if self.left == None:

self.left = BinaryTree(newNode)

else:

tree = BinaryTree(newNode)

tree.left = self.left

self.left = tree

def printTree(tree):

if tree != None:

printTree(tree.getLeftChild())

print(tree.getNodeValue())

printTree(tree.getRightChild())

def testTree():

myTree = BinaryTree("l\_0")

myTree.insertLeft("l\_10")

myTree.insertRight("l\_11")

l10 = myTree.getLeftChild()

l10.insertLeft("l\_20")

l10.insertRight("l\_21")

l11 = myTree.getRightChild()

l11.insertLeft("l\_22")

l11.insertRight("l\_23")

printTree(myTree)