Artificial Intelligence Lab - Spring 2021

All Implementations

Prolog
Example 01

```
female(mary).
male(adam).
male(john).
male(phillips).
male(jack).

lent(mary,john).
lent(mary,jack).
lent(mary,phillips).
lent(john,adam).
lent(john,phillips).
lent(phillips,adam).

/** mary lent money to a male */
mary_lent_male(X,Y) :- lent(X,Y), female(X), male(Y).

/** male lent money to anybody */
male_lent(X,Y) :- lent(X,Y), male(X).
```

Example 02

```
male(atif).
female(aliya).
female(momina).
female(farwa).
female(bushra).

parent(atif,momina).
parent(atif,farwa).
parent(atif,bushra).
parent(aliya,momina).
```

```
parent(aliya,farwa).
parent(aliya,bushra).
sibling(momina,farwa).
sibling(momina,bushra).
sibling(farwa,momina).
sibling(farwa,bushra).
sibling(bushra,momina).
sibling(bushra,farwa).

mother(X, Y) :- parent(X, Y), female(X).
father(X, Y) :- parent(X, Y), male(X).
daughter(X, Y) :- parent(Y, X), female(X).
sister(X, Y) :- parent(Z,X), parent(Z,Y), female(X), female(Y).
```

Example 03

```
food(eggs).
food(toast).
food(oatmeal).
food(beans).
food(pulses).
food(chapati).
food(chicken).
food(steak).
food(soup).
food(smoothie).

breakfast(eggs).
breakfast(toast).
breakfast(oatmeal).
breakfast(smoothie).

lunch(beans).
lunch(eggs).
lunch(pulses).
lunch(chapati).
lunch(smoothie).

dinner(chicken).
dinner(steak).
dinner(soup).
```

```
dinner(chapati).

/** checking if breakfast and lunch have an item in both */
breakfast_lunch(X) :- breakfast(X), lunch(X).

/** checking if dinner and lunch have an item in both */
lunch_dinner(X) :- lunch(X), dinner(X).

/** checking if breakfast and dinner have an item in both */
breakfast_dinner(X) :- breakfast(X), dinner(X).
```

Example 04

```
female(mary).
male(john).
male(adam).
likes(mary,pasta).
likes(mary,lasagna).
likes(mary,smoothies).
likes(adam,biryani).
likes(adam,pasta).
likes(john,apples).
likes(john,smoothies).

/** checking if mary and some other male has similar likes */
similar_with_mary(X,Y) :- likes(Y,Z),likes(X,Z),female(X),male(Y).

/** checking if adam and john or any two males have similar likes */
similar_adam_john(X,Y) :- likes(X,Z),likes(Y,Z),male(X),male(Y).
```

WEEK 01 & 02

BST Traversal and Deletion

```
class Dummy:
     f = False

class TreeNode:
```

```
     def __init__(self, val):
     self.val = val
     self.right = None
     self.left = None

class BST(TreeNode):
     def __init__(self, val, parent=None):
     super().__init__(val)
     self.parent = parent

def print_tree(tree, level=0, label='.'):
     print(' ' * (level*2) + label + ':' , tree.val)
     for child, lbl in zip([tree.left, tree.right], ['L', 'R']):  # do for
all children
     if child is not None:
          print_tree(child, level+1, lbl)



def insert(self, val):
     if val < self.val:
     if self.left is None:
          new_node = BST(val, parent = self)
          self.left = new_node
     else:
          self.left.insert(val)

     elif val > self.val:
     if self.right is None:
          new_node = BST(val, parent = self)
          self.right = new_node
     else:
          self.right.insert(val)

BST.insert = insert



def get_successor(self):
     self2 = self

     if self.right == None and self.left == None:
```

```
        return self2
        else:
        if self.left:
            self2 = self.left.get_successor()
            d.f = True
            return self2

        elif self.right:
            if d.f == True:
                return
            else:
                self2 = self.right.get_successor()
                return self2

BST.get_successor = get_successor


def delete(self, val):

        if self.val == val:


        # CASE 1 - only root node

        if self.parent is None and self.left is None and self.right is None:
            return None


        # CASE 2 - no child

        if self.left is None and self.right is None# if child node

            if self.parent.right:
                if self.parent.right.val == val:  # if child node is on
the right of parent
                self.parent.right = None

            if self.parent.left:
                if self.parent.left.val == val:   # if child node is on
the left of parent
                self.parent.left = None
```

```
    # CASE 3 - one child

    if self.left is None and self.right != None# if the node to be
deleted has only one child on its right
        self.parent.right = self.right

    if self.left != None and self.right is None# if the node to be
deleted has only one child on its left
        self.parent.left = self.left


    # CASE 4 - two children

    if self.right and self.left:

        successor = self.right.get_successor()# find successor -
left-most leaf from right subtree
        print(successor.val)
        self.delete(successor.val)
        self.val = successor.val

    else:
    if val < self.val:   # smaller value is on left side
        self.left.delete(val)

    if val > self.val:   # greater value is on right side
        self.right.delete(val)

    return

BST.delete = delete



def dfs_preorder(self):

    print(self.val)

    if self.left:
    self.left.dfs_preorder()
```

```
    if self.right:
    self.right.dfs_preorder()

    return

BST.dfs_preorder = dfs_preorder



def dfs_inorder(self):

    if self.left:
    self.left.dfs_inorder()

    print(self.val)

    if self.right:
    self.right.dfs_inorder()

    return

BST.dfs_inorder = dfs_inorder



def dfs_postorder(self):

    if self.left:
    self.left.dfs_postorder()

    if self.right:
    self.right.dfs_postorder()

    print(self.val)

    return

BST.dfs_postorder = dfs_postorder



def bfs(self):
```

```
      lst = [self]

      while lst:
      current = lst.pop(0)
      print(current.val)

      if current.left:
           lst.append(current.left)

      if current.right:
           lst.append(current.right)

BST.bfs = bfs



d = Dummy()

t = BST(12)
t.insert(8)
t.insert(14)
t.insert(6)
t.insert(9)
print_tree(t)

t.dfs_preorder()
t.dfs_inorder()
t.dfs_postorder()
t.bfs()


t2 = BST(5)
t2.insert(7)
t2.insert(2)
t2.insert(1)
t2.insert(10)
t2.insert(3)
t2.insert(9)
t2.insert(11)
t2.insert(6)
print_tree(t2)
```

```
t2.delete(5)

print_tree(t2)
```

Graphs

```
!pip install networkx

import networkx as nx
import matplotlib.pyplot as plt

%matplotlib inline
import warnings
warnings.filterwarnings("ignore")

def draw_graph_with_nx(G):
      pos = nx.spring_layout(G, iterations=200)
      options = {'node_color': 'white', 'alpha': 1, 'node_size': 2000,
'width': 0.002, 'font_color': 'darkred',
            'font_size': 25, 'arrows': True, 'edge_color': 'brown',
            'arrowstyle': 'Fancy, head_length=1, head_width=1,
tail_width=.4'
            }
      labels = nx.get_node_attributes(G, 'label')
      nx.draw(G, pos, labels=labels,  **options)
      plt.show()


class Digraph:
      def __init__(self):
      self.g = {}

      def add_node(self,node):
      if node in self.g:
            raise ValueError("Source already exists")

      self.g[node] = []

      def add_edge(self,src,dest):
      if src not in self.g and dest not in self.g:
            raise ValueError('Source/Destination not found')
```

```
    edges = self.g[src]

    if dest not in edges:
        edges.append(dest)

    else:
        raise ValueError("Destination already exists")

    def draw_graph(self):
    G = nx.DiGraph()
    for src in self.g:
        G.add_node(src, label=src)
        for dest in self.g[src]:
            G.add_edge(src, dest)

    draw_graph_with_nx(G)


g = Digraph()
g.add_node('Isd')
g.add_node('Pwr')
g.add_node('Grw')
g.add_node('Lhr')
g.add_node('Fsd')


g.add_edge('Isd','Pwr')
g.add_edge('Isd','Lhr')

g.draw_graph()
```

Linked List

```
class Node:
    def __init__(self, val=None):
    self.val = val
    self.next = None

class LinkedList:
    def __init__(self):
```

```
     self.head = None

def __str__(self):
     ret_str = '['

     temp = self.head
     while temp:
     ret_str += str(temp.val) + ', '
     temp = temp.next

     ret_str = ret_str.rstrip(', ')
     ret_str = ret_str + ']'
     return ret_str

LinkedList.__str__ = __str__


def push(self, val):
     new_node = Node(val)

     if self.head is None:
     self.head = new_node
     return

     last = self.head
     while last.next != None:
     last = last.next

     last.next = new_node

LinkedList.push = push


def removed(self):
     temp = self.head

     if temp is None:
     return 0

     if temp.next is None:
     self.head = None
     return
```

```
     while temp.next.next is not None:
     temp = temp.next

     temp.next = None
     temp = None

LinkedList.removed = removed


l = LinkedList()
l.push(4)
l.push(8)
l.push(12)
print(l)

l.removed()
print(l)
```

Stack

```
class Stack:
     def __init__(self):
     self.list = []

     def push(self, val):
     self.list.append(val)

     def pop(self):
     return self.list.pop()

     def display(self):
     for i in range(0, len(self.list)):
          print(self.list[i])

s = Stack()
s.push(4)
s.push(5)
s.push(8)
s.display()
```

Queue

```
class Queue:
      def __init__(self, size=5):
      self.queue = []
      self.size = size
      self._inc = 0
      self._dec = 0
      self.Empty = True
      self.Full = False

      for i in range(0, self.size):
            self.queue.append(0)

      def _in(self):
      if self._inc == self.size:
            self._inc = 0

      self._inc = self._inc + 1
      return

      def _out(self):
      if self._dec == self.size:
            self._dec = 0

      self._dec = self._dec + 1
      return


def enqueue(self, val):
      if self.Full:
      print("List is full")
      return



      self.queue[self._inc] = val
      self._in()

      if self._inc == self.size:
      self.Full = True
```

```
      self.Empty = False
      return

Queue.enqueue = enqueue


def dequeue(self):
      val = self.queue[self._dec]
      self.queue[self._dec] = 0
      self._out()

      if self._inc == self._dec:
      self.Empty = True

      self.Full = False

#     print("Dec 1:", self._dec)

      self.shifting()

      self._dec = 0

      return val

Queue.dequeue = dequeue


def shifting(self):

      var1 = self._dec

      for i in range(self.size):

      if self._dec != self.size:

            self.queue[self._dec-1] = self.queue[self._dec]
            self._out()

      else:
            self.queue[self._dec-1] = 0

      self._dec = var1
```

```
    return

Queue.shifting = shifting


q = Queue(6)
q.enqueue(5)
q.enqueue(6)
q.enqueue(2)
q.enqueue(45)
q.enqueue(9)

print(q.queue)

q.dequeue()

print(q.queue)
```

WEEK 03

Simple Reflex

```
import random

class Environment:

    def __init__(self):

    #instantiate locations and conditions
    # 0 indicates Clean and 1 indicated Dirty
    self.locationCondition = {'A':'0', 'B':'0'}

    #randomize conditions in location A and B
    self.locationCondition['A'] = random.randint(0,1)
    self.locationCondition['B'] = random.randint(0,1)
```

```
#we want to take minimum moves to clean rooms so when agent has to move
#to other room it's not good so we decrement score
#and increment score when we clean it to increase performance measurement

class SimpleVacAgent(Environment):

     def __init__(self, Environment):
     print(Environment.locationCondition)

     #Instantiate performance measurement
     score = 0

     #you can use alphabet A or B for vacuum position randomization
     vacuumLocation = random.randint(ord('A'),ord('B'))
     print('Location:',chr(vacuumLocation))

     #vacuum in room A
     if vacuumLocation == ord('A'):
          print('Vacuum is randomly placed in room A')

          #if room A is dirty
          if Environment.locationCondition['A'] == 1:
               print("Location A is Dirty")

               #suck dirt and mark it clean
               Environment.locationCondition['A'] = 0
               score += 1
               print('Location A has been cleaned')

               #move to B
               print('Moving to B...')
               score -= 1

               #if B is dirty
               if Environment.locationCondition['B'] == 1:
               print('Location B is dirty')

               #suck and mark clean
               Environment.locationCondition['B'] = 0
               score += 1
               print('Location B has been cleaned')
```

```
        else:
            #A is clean
            #move to B
            score -= 1
            print('Moving to B...')

            #if B is dirty
            if Environment.locationCondition['B'] == 1:
            print('Location B is dirty')

            #suck and mark clean
            Environment.locationCondition['B'] = 0
            score += 1
            print('Location B has been cleaned')

    elif vacuumLocation == ord('B'):
        print('Vacuum is placed in room B')

        #if room B is dirty
        if Environment.locationCondition['B'] == 1:
            print("Location B is Dirty")

            #suck dirt and mark it clean
            Environment.locationCondition['B'] = 0
            score += 1
            print('Location B has been cleaned')

            #move to A
            print('Moving to A...')
            score -= 1

            #if A is dirty
            if Environment.locationCondition['A'] == 1:
            print('Location A is dirty')

            #suck and mark clean
            Environment.locationCondition['A'] = 0
            score += 1
            print('Location A has been cleaned')

        else:
            #B is clean
```

```
                #move to A
                score -= 1
                print('Moving to A...')

                #if A is dirty
                if Environment.locationCondition['A'] == 1:
                print('Location A is dirty')

                #suck and mark clean
                Environment.locationCondition['A'] = 0
                score += 1
                print('Location A has been cleaned')

      #done cleaning
      print(Environment.locationCondition)
      print('Performance measurement: ',str(score))


if __name__ == '__main__':

    env = Environment()
    vac = SimpleVacAgent(env)
```

Table Driven

```
import random

class Environment:

      def __init__(self):

      #instantiate locations and conditions
      # 0 indicates Clean and 1 indicated Dirty
      self.locationCondition = {'A':'0', 'B':'0'}

      #randomize conditions in location A and B
      self.locationCondition['A'] = random.randint(0,1)
      self.locationCondition['B'] = random.randint(0,1)
```

```
class TableDrivenVacAgent(Environment):

      def __init__(self,Environment):

      print(Environment.locationCondition)

      #Implement lookup table
      dic = {('A',1):'Clean', ('A',0):'Right',
('B',1):'Clean',('B',0):'Left'}

      #Instantiate performance measurement
      score = 0

      #you can use alphabet A or B for vacuum position randomization
      vacuumLocation = random.randint(ord('A'),ord('B'))
      print('Location:',chr(vacuumLocation))

      #vacuum in room A
      if vacuumLocation == ord('A'):
            print('Vacuum is randomly placed in room A')

            #if room A is dirty
            if Environment.locationCondition['A'] == 1:
                  print("Location A is Dirty")

                  #suck dirt and mark it clean
                  ans = dic[('A',1)]
                  if ans == 'Clean':
                  Environment.locationCondition['A'] = 0
                  score += 1
                  print('Location A has been cleaned')

                  #move to B
                  ans = dic[('A',0)]
                  if ans == 'Right':
                        print('Moving to B...')
                        score -= 1

                        #if B is dirty
```

```
                if Environment.locationCondition['B'] == 1:
                    print('Location B is dirty')

                    #suck and mark clean
                    ans = dic[('B',1)]
                    if ans == 'Clean':
                    Environment.locationCondition['B'] = 0
                    score += 1
                    print('Location B has been cleaned')

                elif Environment.locationCondition['B'] == 0:
                    print('Room B is already clean')


        else:
            #if A is clean move to B
            print('Room A is already clean.')
            ans = dic[('A',0)]
            if ans == 'Right':
            score -= 1
            print('Moving to B...')

            #if B is dirty
            if Environment.locationCondition['B'] == 1:
            print('Location B is dirty')

            #suck and mark clean
            ans = dic[('B',1)]
            if ans == 'Clean':
                Environment.locationCondition['B'] = 0
                score += 1
                print('Location B has been cleaned')

            elif Environment.locationCondition['B'] == 0:
            print('Room B is already clean')

    elif vacuumLocation == ord('B'):
        print('Vacuum is placed randomly in room B')

        #if room B is dirty
        if Environment.locationCondition['B'] == 1:
            print("Location B is Dirty")
```

```
            #suck dirt and mark it clean
            ans = dic[('B',1)]
            if ans == 'Clean':
            Environment.locationCondition['B'] = 0
            score += 1
            print('Location B has been cleaned')

            #move to A
            ans = dic[('B',0)]
            if ans == 'Left':
                print('Moving to A...')
                score -= 1

                #if A is dirty
                if Environment.locationCondition['A'] == 1:
                    print('Location A is dirty')

                    #suck and mark clean
                    ans = dic[('A',1)]
                    if ans == 'Clean':
                    Environment.locationCondition['A'] = 0
                    score += 1
                    print('Location A has been cleaned')

                elif Environment.locationCondition['A'] == 0:
                    print('Room A is already clean')

        else:

            #B is clean so move to A
            print('Room B is already clean')
            ans = dic[('B',0)]
            if ans == 'Left':
            score -= 1
            print('Moving to A...')

            #if A is dirty
            if Environment.locationCondition['A'] == 1:
                print('Location A is dirty')

                #suck and mark clean
```

```
                      ans = dic[('A',1)]
                      if ans == 'Clean':
                           Environment.locationCondition['A'] = 0
                           score += 1
                           print('Location A has been cleaned')

                 elif Environment.locationCondition['A'] == 0:
                           print('Room A is already clean')

      #done cleaning
      print(Environment.locationCondition)
      print('Performance measurement: ',str(score))


if __name__ == '__main__':

    td = Environment()
    vac = TableDrivenVacAgent(td)
```

Special Case of Vacuum Cleaner

```
import random

class Environment:
      def __init__(self):
      #instantiate locations and conditions
      # 0 indicates Clean and 1 indicated Dirty
      self.locationCondition = {'A':'0', 'B':'0', 'C':'0', 'D':'0'}

      #randomize conditions in location A, B, C and D
      self.locationCondition['A'] = random.randint(0,1)
      self.locationCondition['B'] = random.randint(0,1)
      self.locationCondition['C'] = random.randint(0,1)
      self.locationCondition['D'] = random.randint(0,1)

##########################################################

def roomA(Environment, dic, score, count):

      # if A is dirty
```

```
    if Environment.locationCondition['A'] == 1:
    count += 1
    print('Location A is dirty')

    # suck dirt and mark it clean
    ans = dic[('A',1)]
    if ans == 'Clean':
        Environment.locationCondition['A'] = 0
        score += 1
        print('Location A has been cleaned')

        # move to B
        ans = dic[('A',0)]
        if ans == 'Right':
            score -= 1
            print('Moving to B...')

    # if A is clean
    elif Environment.locationCondition['B'] == 0:
    print('Location A is already clean')

    # move to B
    ans = dic[('A',0)]
    if ans == 'Right':
        score -= 1
        print('Moving to B...')

    return score, count

##########################################################

def roomB(Environment, dic, score, count):

    # if B is dirty
    if Environment.locationCondition['B'] == 1:
    count += 1
    print('Location B is dirty')

    # suck and mark clean
    ans = dic[('B',1)]
    if ans == 'Clean':
        Environment.locationCondition['B'] = 0
```

```
            score += 1
            print('Location B has been cleaned')

            ans1 = dic[('B',0)][0]
            ans2 = dic[('B',0)][1]

            # move to C
            if ans1 == 'rotate_right' and ans2 == 'Down':
                  score -= 1
                  print('Rotating')
                  print('Moving to C...')

      # if B is clean
      elif Environment.locationCondition['B'] == 0:
      print('Location B is already clean')

      ans1 = dic[('B',0)][0]
      ans2 = dic[('B',0)][1]

      # move to C
      if ans1 == 'rotate_right' and ans2 == 'Down':
            score -= 1
            print('Rotating')
            print('Moving to C...')


      return score, count

##########################################################

def roomC(Environment, dic, score, count):

      # if C is dirty
      if Environment.locationCondition['C'] == 1:
      if count != 2:
            count += 1
            print('Location C is dirty')

            # suck and mark clean
            ans = dic[('C',1)]
            if ans == 'Clean':
                  Environment.locationCondition['C'] = 0
```

```
            score += 1
            print('Location C has been cleaned')

            ans1 = dic[('C',0)][0]
            ans2 = dic[('C',0)][1]

            # move to D
            if ans1 == 'rotate_right' and ans2 == 'Left':
            score -= 1
            print('Rotating')
            print('Moving to D...')

    elif count == 2:
        ans1 = dic[('C',0)][0]
        ans2 = dic[('C',0)][1]

        # move to D
        if ans1 == 'rotate_right' and ans2 == 'Left':
            score -= 1
            print('Rotating')
            print('Moving to D...')

    elif Environment.locationCondition['C'] == 0:
    print('Location C is already clean')

    ans1 = dic[('C',0)][0]
    ans2 = dic[('C',0)][1]

    # move to D
    if ans1 == 'rotate_right' and ans2 == 'Left':
        score -= 1
        print('Rotating')
        print('Moving to D...')

    return score, count

##########################################################

def roomD(Environment, dic, score, count, home):

    # if D is dirty
    if Environment.locationCondition['D'] == 1:
```

```
    if count != 2:
        count += 1
        print('Location D is dirty')

        # suck and mark clean
        ans = dic[('D',1)]
        if ans == 'Clean':
            Environment.locationCondition['D'] = 0
            score += 1
            print('Location D has been cleaned')

            ans1 = dic[('D',0)][0]
            ans2 = dic[('D',0)][1]

            # move back to A
            if ans1 == 'rotate_right' and ans2 == 'Up':
            score -= 1
            print('Rotating')
            print('Moving to A...')

    elif count == 2:
        ans1 = dic[('D',0)][0]
        ans2 = dic[('D',0)][1]

        # move back to A
        if ans1 == 'rotate_right' and ans2 == 'Left':
            score -= 1
            print('Rotating')
            print('Moving up to A...')

    elif Environment.locationCondition['D'] == 0:
    print('Location D is already clean')

    ans1 = dic[('D',0)][0]
    ans2 = dic[('D',0)][1]

    # move back to A
    if ans1 == 'rotate_right' and ans2 == 'Up':
        score -= 1
        print('Rotating')
        print('Moving to A...')
```

```
    if home == True:
     print('Agent has reached home.')

    return score, count

##########################################################

class TableDriven(Environment):
    def __init__(self,Environment):
    print(Environment.locationCondition)

    #Implement lookup table
    dic =
{('A',1):'Clean',('A',0):'Right',('B',1):'Clean',('B',0):['rotate_right','D
own'],('C',1):'Clean',('C',0):['rotate_right','Left'],('D',1):'Clean',('D',
0):['rotate_right','Up']}

    #Instantiate performance measurement
    score = 0

    #count variable to maintain 2 dirty place
    count = 0

    # to check if it has returned home
    home = False

#    room A
    home = True
    score, count = roomA(Environment, dic, score, count)
    print('A count',count)
    print('A new condition',Environment.locationCondition['A'])
    print('----------------')

#    room B
    score, count = roomB(Environment, dic, score, count)
    print('B count',count)
    print('B new condition',Environment.locationCondition['B'])
    print('----------------')

#    room C
    score, count = roomC(Environment, dic, score, count)
    print('C count',count)
```

```
     print('C new condition',Environment.locationCondition['C'])
     print('----------------')

#    room D
     score, count = roomD(Environment, dic, score, count, home)
     print('D count',count)
     print('D new condition',Environment.locationCondition['D'])
     print('----------------')

     print('Updated LocCondition:',Environment.locationCondition)
     print('Performance measurement: ',str(score))



if __name__ == '__main__':

   td = Environment()
   vac = TableDriven(td)
```

Agent for irregularly-shaped room

```
import random

###################################################

#          A - B - C
#          D          L-shaped room

class Environment:
     def __init__(self):
     #instantiate locations and conditions

     # 0 is no obstacle, 1 is obstacle
     self.obstacle_ahead = random.randint(0,1)
     self.obstacle_right = random.randint(0,1)
     self.obstacle_left = random.randint(0,1)

     self.locationCondition = {'A':'0', 'B':'0', 'C':'0', 'D':'0'}

     #randomize conditions in location A, B, C and D
```

```
      # 2 indicates Clean and 3 indicated Dirty

      self.locationCondition['A'] = random.randint(2,3)
      self.locationCondition['B'] = random.randint(2,3)
      self.locationCondition['C'] = random.randint(2,3)
      self.locationCondition['D'] = random.randint(2,3)

###################################################


class TDLshaped(Environment):

      def __init__(self,Environment):

      print(Environment.locationCondition)

      # Implement lookup table
      dic =
{('A',3):'Clean',('A',2):{'B':'Right','D':'Down'},('B',3):'Clean',('B',2):{
'A':'Left','C':'Right'},('C',3):'Clean',('C',2):'Left',('D',3):'Clean',('D'
,2):['rotate_right','Up']}

      # Instantiate performance measurement
      score = 0

      # Whether room has been visited before
      roomA, roomB, roomC, roomD = False, False, False, False

      # always starting from A
      score, roomA, roomB, roomC, roomD = rA(Environment, dic, score,
roomA, roomB, roomC, roomD)
      print('score after A:',score)

#          # room B
      score, roomA, roomB, roomC, roomD = rB(Environment, dic, score,
roomA, roomB, roomC, roomD)
      print('score after B:',score)

      # room C
      rC(Environment, dic, score, roomA, roomB, roomC, roomD)

      # room D will be called from room A
```

```
    print('Updated LocCondition:',Environment.locationCondition)

###################################################

# ROOM A

def rA(Environment, dic, score, roomA, roomB, roomC, roomD):

    obstacle_ahead = random.randint(0,1)
    obstacle_right = random.randint(0,1)
    obstacle_left = random.randint(0,1)


    if roomA == True:
    print('Agent is coming from room B to A')

    print('Moving to D...')

    score -= 1

    score, roomA, roomB, roomC, roomD = rD(Environment, dic, score,
roomA, roomB, roomC, roomD)
    print('Performance measurement: ',str(score))

    else:

    # if A is dirty
    if Environment.locationCondition['A'] == 3:
        print('A is dirty')

        # suck dirt and mark clean
        ans = dic[('A',3)]
        if ans == 'Clean':
            Environment.locationCondition['A'] = 2
            print('A has been cleaned')
            score += 1
            roomA = True

            #move and check ahead
            if obstacle_ahead == 1:
            print('Obstacle ahead')
```

```
#check right
print("Checking right")
if obstacle_right == 0:
      print('No obstacle on right')
      print('Agent moving right')

      # move to B
      ans = dic[('A',2)]['B']
      if ans == 'Right':
            score -= 1
            print('Moving to B...')


elif obstacle_right == 1:
      print('Obstacle found on right')

      #check left
      print('Checking left')
      if obstacle_left == 0:
            print('Agent moving left')

            # move to B
            ans = dic[('A',2)]['B']
            if ans == 'Right':
            score -= 1
            print('Moving to B...')
      else:
            print('Obstacle found on left')
            print('Agent moving backward')

            # move to B
            ans = dic[('A',2)]['B']
            if ans == 'Right':
            score -= 1
            print('Moving to B...')

elif obstacle_ahead == 0:
print('No obstacle ahead')

# move to B
ans = dic[('A',2)]['B']
```

```
                if ans == 'Right':
                    score -= 1
                    print('Moving to B...')

    elif Environment.locationCondition['A'] == 2:
        print('A is already clean')
        roomA = True

        #move
        if obstacle_ahead == 1:
            print('Obstacle ahead')

            #check right
            print("Checking right")
            if obstacle_right == 0:
            print('No obstacle on right')
            print('Agent moving right')

            # move to B
            ans = dic[('A',2)]['B']
            if ans == 'Right':
                score -= 1
                print('Moving to B...')

            elif obstacle_right == 1:
            print('Obstacle found on right')
            #check left
            print('Checking left')

            if obstacle_left == 0:
                print('Agent moving left')

                # move to B
                ans = dic[('A',2)]['B']
                if ans == 'Right':
                    score -= 1
                    print('Moving to B...')
            else:
                print('Obstacle found on left')
                print('Agent moving backward')

                # move to B
```

```
                    ans = dic[('A',2)]['B']
                    if ans == 'Right':
                         score -= 1
                         print('Moving to B...')
          elif obstacle_ahead == 0:
               print('No obstacle ahead')

               # move to B
               ans = dic[('A',2)]['B']
               if ans == 'Right':
               score -= 1
               print('Moving to B...')

     return score, roomA, roomB, roomC, roomD

################################################################

# ROOM B

def rB(Environment, dic, score, roomA, roomB, roomC, roomD):

     obstacle_ahead = random.randint(0,1)
     obstacle_right = random.randint(0,1)
     obstacle_left = random.randint(0,1)


     if roomB == True:
     print('Agent is coming from C to B')
     score -= 1
     rA(Environment, dic, score, roomA, roomB, roomC, roomD)

     else:

     # if B is dirty
     if Environment.locationCondition['B'] == 3:
          print('B is dirty')

          # suck dirt and mark clean
          ans = dic[('B',3)]
          if ans == 'Clean':
               Environment.locationCondition['B'] = 2
               print('B has been cleaned')
```

```
        score += 1
        roomB = True

        #move and check ahead
        if obstacle_ahead == 1:
        print('Obstacle ahead')

        #check right
        print("Checking right")
        if obstacle_right == 0:
            print('No obstacle on right')
            print('Agent moving right')

            # move to C
            ans = dic[('B',2)]['C']
            if ans == 'Right':
                score -= 1
                print('Moving to C...')

        elif obstacle_right == 1:
            print('Obstacle found on right')

            #check left
            print('Checking left')
            if obstacle_left == 0:
                print('Agent moving left')

                # move to C
                ans = dic[('B',2)]['C']
                if ans == 'Right':
                score -= 1
                print('Moving to C...')
            else:
                print('Obstacle found on left')
                print('Agent moving backward')

                # move to C
                ans = dic[('B',2)]['C']
                if ans == 'Right':
                score -= 1
                print('Moving to C...')
```

```
                elif obstacle_ahead == 0:
                print('No obstacle ahead')

                # move to C
                ans = dic[('B',2)]['C']
                if ans == 'Right':
                     score -= 1
                     print('Moving to C...')

      elif Environment.locationCondition['B'] == 2:
           print('B is already clean')
           roomB = True

           #move and check ahead
           if obstacle_ahead == 1:
                print('Obstacle ahead')

                #check right
                print("Checking right")
                if obstacle_right == 0:
                print('No obstacle on right')
                print('Agent moving right')

                # move to C
                ans = dic[('B',2)]['C']
                if ans == 'Right':
                     score -= 1
                     print('Moving to C...')

                elif obstacle_right == 1:
                print('Obstacle found on right')

                #check left
                print('Checking left')

                if obstacle_left == 0:
                     print('Agent moving left')

                     # move to C
                     ans = dic[('B',2)]['C']
                     if ans == 'Right':
                          score -= 1
```

```
                              print('Moving to C...')
                 else:
                        print('Obstacle found on left')
                        print('Agent moving backward')

                        # move to C
                        ans = dic[('B',2)]['C']
                        if ans == 'Right':
                              score -= 1
                              print('Moving to C...')


           elif obstacle_ahead == 0:
                 print('No obstacle ahead')

                 # move to C
                 ans = dic[('B',2)]['C']
                 if ans == 'Right':
                 score -= 1
                 print('Moving to C...')

      return score, roomA, roomB, roomC, roomD

################################################################

# ROOM C

def rC(Environment, dic, score, roomA, roomB, roomC, roomD):

      obstacle_ahead = random.randint(0,1)
      obstacle_right = random.randint(0,1)
      obstacle_left = random.randint(0,1)

      # if C is dirty
      if Environment.locationCondition['C'] == 3:
      print('C is dirty')

      # suck dirt and mark clean
      ans = dic[('C',3)]
      if ans == 'Clean':
            Environment.locationCondition['C'] = 2
            print('C has been cleaned')
```

```
            score += 1
            roomC = True

            #move and check ahead
            if obstacle_ahead == 1:
                  print('Obstacle ahead')

                  #check right
                  print("Checking right")
                  if obstacle_right == 0:
                  print('No obstacle on right')
                  print('Agent moving right and rotating')

                  elif obstacle_right == 1:
                  print('Obstacle found on right')

                  #check left
                  print('Checking left')
                  if obstacle_left == 0:
                        print('Agent moving left and rotating')

                  else:
                        print('Obstacle found on left')
                        print('Agent moving backward and rotating')

            elif obstacle_ahead == 0:
                  print('No obstacle ahead')

      elif Environment.locationCondition['C'] == 2:
      print('C is already clean')
      roomC = True

      #move and check ahead
      if obstacle_ahead == 1:
            print('Obstacle ahead')

            #check right
            print("Checking right")
            if obstacle_right == 0:
                  print('No obstacle on right')
                  print('Agent moving right and rotating')
```

```
        elif obstacle_right == 1:
            print('Obstacle found on right')

            #check left
            print('Checking left')

            if obstacle_left == 0:
            print('Agent moving left and rotating')

            else:
            print('Obstacle found on left')
            print('Agent moving backward and rotating')


    elif obstacle_ahead == 0:
        print('No obstacle ahead')
        print('Agent is rotating')


    # going back to B
    print('Agent moving back to B')
    score -= 1

    rB(Environment, dic, score, roomA, roomB, roomC, roomD)


    return score, roomA, roomB, roomC, roomD

################################################################

# ROOM D

def rD(Environment, dic, score, roomA, roomB, roomC, roomD):

    obstacle_ahead = random.randint(0,1)
    obstacle_right = random.randint(0,1)
    obstacle_left = random.randint(0,1)

    # if D is dirty
    if Environment.locationCondition['D'] == 3:
    print('D is dirty')
```

```
    # suck dirt and mark clean
    ans = dic[('D',3)]
    if ans == 'Clean':
        Environment.locationCondition['D'] = 2
        print('D has been cleaned')
        score += 1
        roomD = True

        #move and check ahead
        if obstacle_ahead == 1:
            print('Obstacle ahead')

            #check right
            print("Checking right")
            if obstacle_right == 0:
            print('No obstacle on right')
            print('Agent moving right and rotating')

            elif obstacle_right == 1:
            print('Obstacle found on right')

            #check left
            print('Checking left')
            if obstacle_left == 0:
                print('Agent moving left and rotating')

            else:
                print('Obstacle found on left')
                print('Agent moving backward and rotating')

        elif obstacle_ahead == 0:
            print('No obstacle ahead')

    elif Environment.locationCondition['D'] == 2:
    print('D is already clean')
    roomD = True

    #move and check ahead
    if obstacle_ahead == 1:
        print('Obstacle ahead')

        #check right
```

```
            print("Checking right")
            if obstacle_right == 0:
                print('No obstacle on right')
                print('Agent moving right and rotating')

            elif obstacle_right == 1:
                print('Obstacle found on right')

                #check left
                print('Checking left')

                if obstacle_left == 0:
                print('Agent moving left and rotating')

                else:
                print('Obstacle found on left')
                print('Agent moving backward and rotating')


        elif obstacle_ahead == 0:
            print('No obstacle ahead')
            print('Agent is rotating')


        # going back to A
        print('Agent moving back to A')
        score -= 1

        return score, roomA, roomB, roomC, roomD

################################################################


if __name__ == '__main__':

    td = Environment()
    vac = TDLshaped(td)
```

WEEK 04

BFS & DFS

```
from collections import deque

def print_tree(tree, level=0, label='.'):
     print(' ' * (level*2) + label + ':' , tree.val)
     for child, lbl in zip([tree.left, tree.right], ['L', 'R']):  # do for
all children
     if child is not None:
          print_tree(child, level+1, lbl)


class TreeNode:
     def __init__(self, val):
     self.val = val
     self.right = None
     self.left = None

class BST(TreeNode):
     def __init__(self, val, parent=None):
     super().__init__(val)
     self.parent = parent


     def insert(self, val):
     if val < self.val:
          if self.left is None:
               new_node = BST(val, parent = self)
               self.left = new_node
          else:
               self.left.insert(val)

     elif val > self.val:
          if self.right is None:
               new_node = BST(val, parent = self)
               self.right = new_node
          else:
               self.right.insert(val)


     def bfs_search(root, search_key):
```

```
    count = 0
    found = False

    que = deque()
    que.append(root)

    while que:

        count += 1

        current = que.popleft()

        if current.val == search_key:
            found = True
            print('Found')
            print('Total nodes visited:',count)

            return

        else:
            if current.left:
            que.append(current.left)

            if current.right:
            que.append(current.right)

    if found != True:
        print('Keyword not found')

    print('Total nodes visited:',count)

    return



    def dfs_search(root, search_key):

    stk = []
    stk.append(root)

    count = 0
    found = False
```

```
    while stk:

        count += 1

        current = stk.pop(0)

        if search_key == current.val:

            found = True
            print('Found')
            print('Total nodes visited:',count)
            return

        else:

            if current.right:
            stk.append(current.right)

            if current.left:
            stk.append(current.left)


    if found != True:
        print('Keyword not found')

    print('Total nodes visited:',count)

    return



if __name__ == '__main__':

    b = BST(50)
    b.insert(30)
    b.insert(15)
    b.insert(35)
    b.insert(7)
    b.insert(22)
    b.insert(31)
```

```
    b.insert(40)
    b.insert(70)
    b.insert(62)
    b.insert(60)
    b.insert(65)
    b.insert(87)
    b.insert(85)
    b.insert(90)

    b.bfs_search(90)
    b.dfs_search(90)
```

Optimal actions for 3x3 room with dirt in the center

```
class Environment:
      def __init__(self):

      #instantiate locations and conditions
      # 0 indicates Clean and 1 indicated Dirty

      self.locationCondition =
{'A':'0','B':'0','C':'0','D':'0','E':'0','F':'0','G':'0','H':'0','I':'0'}

      #assuming all rooms are dirty
      self.locationCondition['A'] = 1
      self.locationCondition['B'] = 1
      self.locationCondition['C'] = 1
      self.locationCondition['D'] = 1
      self.locationCondition['E'] = 1
      self.locationCondition['F'] = 1
      self.locationCondition['G'] = 1
      self.locationCondition['H'] = 1
      self.locationCondition['I'] = 1

##############################################################

def move_clean(Environment):

      stk = []
```

```
    score = 0

    dic = {'A':['right','down'], 'B':['left','right','down'],
'C':['left','down'], 'D':['up','down','right'],
    'E':['up','down','left','right'], 'F':['up','down','left'],
'G':['up','right'], 'H':['up','left','right'],'I':['up','left'] }

    visited = []

    #assume agent is initially in the middle
    agentLoc = 'E'
    stk.append(agentLoc)

    print('Before:',Environment.locationCondition)

    # visiting all the 9 squares
    while stk:
    agentLoc = stk.pop(0)
    print('Agent is now in room ',agentLoc)

    if agentLoc not in visited:
        score = clean_room(Environment, dic, score, agentLoc)
        visited.append(agentLoc)


        if len(visited) != 9:
            ans = get_direction(dic, agentLoc)
            stk.append(ans)


    print('After:',Environment.locationCondition)
    print('Score:',score)

##################################################################

def clean_room(Environment, dic, score, room):

    # decrement for moving to the room except for E because agent is
starting from E
    if room != 'E':
    score -= 1
```

```
    #check if room dirty
    if Environment.locationCondition[room] == 1:
    #suck dirt and mark clean
    Environment.locationCondition[room] = 0
    score += 1

    return score

#################################################################

# assuming out agent will go from E -> B -> A -> D -> G -> H -> I -> F -> C

def get_direction(dic, agentLoc):
    if agentLoc == 'E':
    for i in dic.items():
        for j in i[1]:
            if j == 'up':
            return 'B'
    elif agentLoc == 'B':
    for i in dic.items():
        for j in i[1]:
            if j == 'left':
            return 'A'
    elif agentLoc == 'A':
    for i in dic.items():
        for j in i[1]:
            if j == 'down':
            return 'D'
    elif agentLoc == 'D':
    for i in dic.items():
        for j in i[1]:
            if j == 'down':
            return 'G'
    elif agentLoc == 'G':
    for i in dic.items():
        for j in i[1]:
            if j == 'right':
            return 'H'
    elif agentLoc == 'H':
    for i in dic.items():
        for j in i[1]:
            if j == 'right':
```

```
                return 'I'
     elif agentLoc == 'I':
     for i in dic.items():
          for j in i[1]:
               if j == 'up':
               return 'F'
     elif agentLoc == 'F':
     for i in dic.items():
          for j in i[1]:
               if j == 'up':
               return 'C'
     elif agentLoc == 'C':
     return True

####################################################################

if __name__ == '__main__':
    env = Environment()
    td = move_clean(env)
```

3x3 room with dirt probability 0.2

```
import random

class Environment:
     def __init__(self):
     #instantiate locations and conditions
     # 0.8 indicates Clean and 0.2 indicates Dirty

     self.locationCondition =
{'A':'0','B':'0','C':'0','D':'0','E':'0','F':'0','G':'0','H':'0','I':'0'}
     self.pathCost =
{'A':10,'B':8,'C':12,'D':7,'E':3,'F':2,'G':5,'H':1,'I':9}

     #randomize conditions in location A-I

     self.locationCondition['A'] = random.choice([0.2,0.8])
```

```
     self.locationCondition['B'] = random.choice([0.2,0.8])
     self.locationCondition['C'] = random.choice([0.2,0.8])
     self.locationCondition['D'] = random.choice([0.2,0.8])
     self.locationCondition['E'] = random.choice([0.2,0.8])
     self.locationCondition['F'] = random.choice([0.2,0.8])
     self.locationCondition['G'] = random.choice([0.2,0.8])
     self.locationCondition['H'] = random.choice([0.2,0.8])
     self.locationCondition['I'] = random.choice([0.2,0.8])

###############################################################

def move_clean(Environment):

     stk = []

     score = 0

     dic = {'A':['right','down'], 'B':['left','right','down'],
'C':['left','down'], 'D':['up','down','right'],
     'E':['up','down','left','right'], 'F':['up','down','left'],
'G':['up','right'], 'H':['up','left','right'],'I':['up','left'] }

     visited = []

     #assume agent is initially in the middle
     agentLoc = 'E'
     stk.append(agentLoc)

     print('Before:',Environment.locationCondition)

     while stk:
     agentLoc = stk.pop(0)
     print('Agent is now in room ',agentLoc)

     if agentLoc not in visited:
          score = clean_room(Environment, dic, score, agentLoc)
          visited.append(agentLoc)
#         print('score',score)

          # adding search cost i.e. node visited
          score += 1
```

```
        if len(visited) != 9:
            ans = get_direction(dic, agentLoc)
            stk.append(ans)


    print('After:',Environment.locationCondition)
    print('Score:',score)

########################################################

def clean_room(Environment, dic, score, room):

    # decrement for moving to the room except for E because agent is
starting from E
    if room != 'E':
    score -= 1

    # if room dirty
    if Environment.locationCondition[room] == 0.2:
    print(room,' is dirty')

    #suck dirt and mark clean
    Environment.locationCondition[room] = 0.8
    print(room,' has been cleaned')

    # adding path cost to performance measure(cleaning room)
    if room != 'E':
        score = score + 1 + Environment.pathCost[room]
    else:
        score = score + 1

    elif Environment.locationCondition[room] == 0.8:
    score = score + Environment.pathCost[room]
    print(room,' is already clean')

    return score

########################################################

# assuming our agent will go from E -> B -> A -> D -> G -> H -> I -> F -> C

def get_direction(dic, agentLoc):
```

```
    if agentLoc == 'E':
    for i in dic.items():
          for j in i[1]:
                if j == 'up':
                return 'B'
    elif agentLoc == 'B':
    for i in dic.items():
          for j in i[1]:
                if j == 'left':
                return 'A'
    elif agentLoc == 'A':
    for i in dic.items():
          for j in i[1]:
                if j == 'down':
                return 'D'
    elif agentLoc == 'D':
    for i in dic.items():
          for j in i[1]:
                if j == 'down':
                return 'G'
    elif agentLoc == 'G':
    for i in dic.items():
          for j in i[1]:
                if j == 'right':
                return 'H'
    elif agentLoc == 'H':
    for i in dic.items():
          for j in i[1]:
                if j == 'right':
                return 'I'
    elif agentLoc == 'I':
    for i in dic.items():
          for j in i[1]:
                if j == 'up':
                return 'F'
    elif agentLoc == 'F':
    for i in dic.items():
          for j in i[1]:
                if j == 'up':
                return 'C'
    elif agentLoc == 'C':
    return True
```

```
############################################################3

if __name__ == '__main__':

    env = Environment()
    td = move_clean(env)
```

Depth Limited Search & Iterative Deepening

```
def print_tree(tree, level=0, label='.'):
     print(' ' * (level*2) + label + ':' , tree.val)
     for child, lbl in zip([tree.left, tree.right], ['L', 'R']):  # do for
all children
     if child is not None:
          print_tree(child, level+1, lbl)

######################################################

class TreeNode:
     def __init__(self, val):
     self.val = val
     self.right = None
     self.left = None

######################################################

class BST(TreeNode):
     def __init__(self, val, parent=None):
     super().__init__(val)
     self.parent = parent


     def insert(self, val):
     if val < self.val:
          if self.left is None:
               new_node = BST(val, parent = self)
               self.left = new_node
          else:
               self.left.insert(val)
```

```
    elif val > self.val:
        if self.right is None:
            new_node = BST(val, parent = self)
            self.right = new_node
        else:
            self.right.insert(val)



def DLS(root, goal, limit):

    stk = []
    stk.append(root)
    found = False

    while stk:

        current = stk.pop()
        #print(current.val)

        if current:

            if current.val == goal:
            found = True
            break

            else:
            if current.right:
                depth = get_depth(current.right)

                if depth <= limit:
                    stk.append(current.right)

            if current.left:
                depth = get_depth(current.left)

                if depth <= limit:
                    stk.append(current.left)

    if found == True:
        depth = get_depth(current)
        print('Goal found at depth',depth)
```

```
            return True

      else:
            return False



      def IDS(root, goal):

      max_depth = 0

      # to get max depth
      while root:
            # because CBT so all leaf nodes will be at the same depth
            root = root.left
            if root:
                  max_depth += 1

      for depth in range(0, max_depth+1):
            flg = b.DLS(goal, depth)
            if flg == True:
                  return True

      if flg == False:
            return ('Goal not found')

#########################################################

def get_depth(root):
      level = 0

      while root:
      root = root.parent
      if root:
            level += 1

      return level

####################################

if __name__ == '__main__':
```

```python
b = BST('R')
b.insert('M')
b.insert('F')
b.insert('O')
b.insert('A')
b.insert('G')
b.insert('N')
b.insert('P')
b.insert('V')
b.insert('T')
b.insert('Y')
b.insert('S')
b.insert('U')
b.insert('X')
b.insert('Z')

print_tree(b)

print('Checking DLS for Z at depth 2:')
g = b.DLS('Z',2)
print(g)
print('-------')

print('Checking DLS for Z at depth 3:')
g = b.DLS('Z',3)
print(g)
print('-------')

print('Checking IDS for O:')
g = b.IDS('O')
print(g)
print('-------')

print('Checking IDS for X:')
g = b.IDS('X')
print(g)
print('-------')

print('Checking IDS for W:')
g = b.IDS('W')
print(g)
```

WEEK 05 & 06

Uniform Cost Search

```
from collections import deque

def print_tree(tree, level=0, label='.'):
      print(' ' * (level*2) + label + ':' , tree.val)
      for child, lbl in zip([tree.left, tree.right], ['L', 'R']):  # do for
all children
      if child is not None:
            print_tree(child, level+1, lbl)

class TreeNode:
      def __init__(self, val):
      self.val = val
      self.right = None
      self.left = None

class BST(TreeNode):
      def __init__(self, val, parent=None):
      super().__init__(val)
      self.parent = parent


      def insert(self, val):
      if val < self.val:
            if self.left is None:
                  new_node = BST(val, parent = self)
                  self.left = new_node
            else:
                  self.left.insert(val)

      elif val > self.val:
            if self.right is None:
                  new_node = BST(val, parent = self)
                  self.right = new_node
            else:
                  self.right.insert(val)
```

```
    # TREE

    def ucs_search(root, goal):

    dic = {'R':{'M':1, 'V':5}, 'M':{'F':3, 'O':6}, 'V':{'T':9, 'Y':2},
'F':{'A':3, 'G':4},'O':{'N':4, 'P':5}, 'T':{'S':8, 'U':6}, 'Y':{'X':7,
'Z':9}}

    paths_def = [['R','M','F','A'], ['R','M','F','G'], ['R','M','O','N'],
['R','M','O','P'], ['R','V','T','S'], ['R','V','T','U'], ['R','V','Y','X'],
['R','V','Y','Z']]

    # for path to goal state
    path = []

    # to keep track of visited nodes
    visited_nodes = []

    # to calculate path cost
    cost = 0

    que = deque()
    que.append(root)

    # if root is goal
    if goal == root.val:
        path.append(root.val)
        return path, cost

    # if root is not goal
    while que:
        current = que.popleft()

        # to traverse a node only once
        if current not in visited_nodes:

            visited_nodes.append(current.val)

            if current.val == goal:

            # loop to get to root from goal state - for path
            while current:
```

```
                    path.append(current.val)
                    current = current.parent

                else:
                if current.left:
                    que.append(current.left)

                if current.right:
                    que.append(current.right)

        path.reverse()

        for i in range(0,len(path)-1):
            first = path[i]
            second = path[i+1]
            cost = cost + dic[first][second]

        return path, cost

########################################################

if __name__ == '__main__':

    b = BST('R')
    b.insert('M')
    b.insert('F')
    b.insert('O')
    b.insert('A')
    b.insert('G')
    b.insert('N')
    b.insert('P')
    b.insert('V')
    b.insert('T')
    b.insert('Y')
    b.insert('S')
    b.insert('U')
    b.insert('X')
    b.insert('Z')

    print_tree(b)

    path, cost = b.ucs_search('V')
```

```
    print('Searching for',path[1])
    print(path, cost)
    print('----------')


    path, cost = b.ucs_search('Z')
    print('Searching for',path[1])
    print(path, cost)
```

WEEK 07 & 08

Greedy Best First Search and A* Search

```
import networkx as nx
import matplotlib.pyplot as plt
from collections import deque

%matplotlib inline
import warnings
warnings.filterwarnings("ignore")

import pprint
from queue import PriorityQueue

def draw_graph_with_nx(G):
    pos = nx.spring_layout(G, iterations=200)
    options = {'node_color': 'white', 'alpha': 1, 'node_size': 2000,
'width': 0.002, 'font_color': 'darkred',
           'font_size': 25, 'arrows': True, 'edge_color': 'brown',
           'arrowstyle': 'Fancy, head_length=1, head_width=1,
tail_width=.4'
           }
    labels = nx.get_node_attributes(G, 'label')
    nx.draw(G, pos, labels=labels,  **options)
    plt.show()

class Weighted:
    def __init__(self):
    self.g = {}
```

```
def add_node(self,node):
if node in self.g:
      raise ValueError("Already exists")

self.g[node] = []

def add_edge(self,src,dest,cost):
if src not in self.g or dest not in self.g:
      raise ValueError("Src/dest not found")

children = self.g[src]
if dest not in children:
      children.append((dest,cost))

def get_neighbours(self,src):
neigh = []

if src not in self.g:
      raise ValueError('Src not found')

for i in self.g[src]:
      neigh.append(i[0])
return neigh

def get_cost(self, src, dest):
cost = 0
if src not in self.g or dest not in self.g:
      raise ValueError('Src/Dest not found')

for i in range(0,len(self.g[src])):
      if self.g[src][i][0] == dest:
            cost = self.g[src][i][1]

return cost


def draw_graph(self):
G = nx.DiGraph()
for src in self.g:
      G.add_node(src, label=src)
      for dest in self.g[src]:
```

```
                G.add_edge(src, dest[0], weight=str(dest[1]))

    draw_graph_with_nx(G)

def greedyBFS(graph, src, dest):
    visited_nodes = []
    que = PriorityQueue()
    que.put((0, src))
    min_cost = float('inf')
    t_cost = 0

    while que:
    cost, node = que.get()
    print(cost,node)

    if node not in visited_nodes:
        visited_nodes.append(node)

        if node == dest:
            print(t_cost,node)
            return que

        for i in graph.get_neighbours(node):
            if i not in visited_nodes:
            cost = graph.get_cost(node,i)
            if cost < min_cost:
                min_cost = cost
                loc = i

        que.put((min_cost,loc))
        t_cost = t_cost + min_cost

    return False

# -----------------------------------------------------------------------

def heuristic(graph, node, parent):

    h_S = 8
    h_S1 = 7
    h_S2 = 3
    h_S3 = 3
```

```
    h_G = 0

    if node == 'S':
    g_S = 0
    f_n = g_S + h_S
    return f_n

    if node == 'S1':
    g_S1 = graph.get_cost('S','S1')
    f_n = g_S1 + h_S1
    return f_n

    if node == 'S2':
    g_S2 = graph.get_cost('S','S2')
    f_n = g_S2 + h_S2
    return f_n

    if node == 'S3':
    g_S3 = graph.get_cost('S','S3')
    f_n = g_S3 + h_S3
    return f_n

    if node == 'G':
    if parent == 'S1':
        g_G = graph.get_cost('S','S1') + graph.get_cost('S1','G')
        f_n = g_G + h_G
        return f_n

    if parent == 'S2':
        g_G = graph.get_cost('S','S2') + graph.get_cost('S2','G')
        f_n = g_G + h_G
        return f_n

    if parent == 'S3':
        g_G = graph.get_cost('S','S3') + graph.get_cost('S3','G')
        f_n = g_G + h_G
        return f_n


def Astar(graph, src, dest):

    que = deque()
```

```
    que.append(src)
    costs = []
    path = []
    parent = ''
    lst = []

    while que:

    min_cost = float('inf')

    for q in que:
          print('1 node,parent',q,parent)
          cost = heuristic(graph, q, parent)
          print('2 node,cost',q,cost)
          if cost < min_cost:
                min_cost = cost
                node = q

    if node == dest:
          return

    que.remove(node)

    print('3 Min:',node,min_cost)
    path.append(node)

    neigh = graph.get_neighbours(node)
    for n in neigh:
          print('4 n:',n)
          if n == 'G':
                if node == 'S1':
                parent = 'S1'
                if node == 'S2':
                parent = 'S2'
                if node == 'S3':
                parent = 'S3'

          que.append(n)

    print('-------',path)
```

```
g = Weighted()
g.add_node('A')
g.add_node('B')
g.add_node('C')
g.add_node('D')
g.add_node('E')
g.add_node('F')
g.add_node('G')

g.add_edge('A','B',12)
g.add_edge('A','C',2)
g.add_edge('B','D',7)
g.add_edge('C','D',4)
g.add_edge('C','G',1)
g.add_edge('D','G',2)
g.add_edge('F','A',1)
g.add_edge('F','G',5)

pprint.pprint(g.g)

greedyBFS(g,'A','G')


w = Weighted()
w.add_node('S')
w.add_node('S1')
w.add_node('S2')
w.add_node('S3')
w.add_node('G')

w.add_edge('S','S1',1)
w.add_edge('S','S2',5)
w.add_edge('S','S3',15)
w.add_edge('S1','G',10)
w.add_edge('S2','G',5)
w.add_edge('S3','G',5)

Astar(w,'S','G')
```

Recursive Best First Search

```
from math import sqrt
import numpy as np

class Grid_5x5:
      def __init__(self):
      self.grid = [
                        [3, 4, 1, 3, 1],
                        [3, 3, 3,'G',2],
                        [3, 1, 2, 2, 3],
                        [4, 2, 3, 3, 3],
                        [4, 1, 4, 3, 2]
                                        ]
      self.goal = 'G'
      self.goal_pos = {"row":1, "col":3}

      #
===========================================================================
==========================================
      # Prints 5x5 Grid and also can bold and underline Agents current
state while printing Grid
      def print_environment(self, current_state=None):

      for r in range(5):
            for c in range(5):

                  if current_state:
                  if r == current_state['row'] and c ==
current_state['col']:
                        # \033[1m is for bold, \033[4m is for underlined,
\033[0m is for finishing both bold and underlined (all)

print("\033[1m\033[4m{}\033[0m".format(self.grid[r][c]), end=' ')
                  else:
                        print(self.grid[r][c], end=' ')

                  else:
                  print(self.grid[r][c], end=' ')

            print()
      print()
      return
```

```
    #
===========================================================================
==========================================
    # Gives the new row after adding any num to it or Gives the new
column after adding any num to it,
    def _increment_pos(self, row_or_col, num_to_move):
    return (row_or_col+num_to_move)%5   # If adding num_to_move to
row_or_col exceeds 5 (given rows,cols of grid) so thats why using modulo to
move in circular


# THE NODES LOGIC HERE COULD BE IMPLEMENTED AS GENERAL TOO

#
===========================================================================
==========================================
# Would be used in rbfs searching tree we construct later, f(n) is
calculated here could also be used for other informed searches i.e A*,
greedy etc
class InformedNodeAlternative:
    def __init__(self, parent, state, parent_action, path_cost,
heuristic_score):
    self.parent = parent
    self.state = state
    self.parent_action = parent_action
    self.path_cost = path_cost   # g(n)
    self.heuristic_score = heuristic_score  # h(n)
    self.f = path_cost + heuristic_score  # f(n) = g(n) + h(n)
#
===========================================================================
==========================================
class InformedChildNodeAlternative(InformedNodeAlternative):
    def __init__(self, problem, parent, parent_action, heuristic_type):
    state = problem.transition_model(parent.state, parent_action)  # This
will give new state when a state applies an action
    path_cost = parent.path_cost + problem.step_cost(parent.state,
parent_action)  # This would sum of step costs of path at each individual
state
    heuristic_score = problem.calculate_heuristic(state, heuristic_type)
# calculating heuristic
#         print(parent.heuristic_score, heuristic_score)
```

```
     super().__init__(parent=parent,
                      state=state,
                      parent_action=parent_action,
                      path_cost=path_cost,
                      heuristic_score=heuristic_score
                                                       )# f would be
calculated in InformedNodeAlternative

#
================================================================================
==========================================
# According to Book a problem for searching has:
# 1. initial state
# 2. possible actions
# 3. transition model (A description what each action does)
# 4. goal test (which determines that has goal been reached at given state)
# 5. path cost (that assigns numeric cost to each path)

class Problem:
     def __init__(self, Environment, initial_state):
     self.initial_state = initial_state
     self.Environment = Environment
     self.possible_actions = ['horizontal', 'vertical']

     #
================================================================================
==========================================
     # Gives new state given current state and action applied at current
state
     def transition_model(self, current_state, action):
     state, new_state = current_state.copy(), current_state.copy()
     # Note: state/position in grid seemed better to represent as
dictionary for readibility
     row = state['row']
     col = state['col']
     num_to_move = self.Environment.grid[row][col]

     # if action is to move horizontal then increment the current col of
state according to current state's value
     if action == 'horizontal':
          new_state['col'] = self.Environment._increment_pos(col,
```

```
num_to_move)

        # if action is to move vertical then increment the current row of
state according to current state's value
        elif action == 'vertical':
            new_state['row'] = self.Environment._increment_pos(row,
num_to_move)

        return new_state

        #
============================================================================
==========================================
        # Tests that whether current node is goal state or not
        def goal_test(self, current_node):
        # print('CHECKING GOAL')
        state = current_node.state
        row = state['row']
        col = state['col']
        value_in_grid = self.Environment.grid[row][col]

#           print('{},{} -> {}'.format(row, col, value_in_grid))
        if value_in_grid == self.Environment.goal:
            return True

        return False

        #
============================================================================
==========================================
        # step cost of each individual step/state, as there are only two
actions horizontally and vertically so 1 as step cost for both seems better
        def step_cost(self, current_state, action):
        return 1  # In book assumption is that step costs are non negative

        #
============================================================================
==========================================
        # calculate euclidean heuristic
        def euclidean_heuristic(self, state):
        goal_pos = self.Environment.goal_pos
        return  sqrt( (state['row']-goal_pos['row'])**2
```

```
                         +
                  (state['col']-goal_pos['col'])**2
                                                   )
     #
===========================================================================
==========================================
     # calculate manhattan heuristic
     def manhattan_heuristic(self, state):
     goal_pos = self.Environment.goal_pos
     return  ( abs( state['row']-goal_pos['row'] )
                          +
               abs( state['col']-goal_pos['col'] )

     #
===========================================================================
==========================================
     # calculate euclidean heuristic or manhattan heuristic of a state
     def calculate_heuristic(self, state, heuristic_type):
     if heuristic_type == 'euclidean':
          return self.euclidean_heuristic(state)

     elif heuristic_type == 'manhattan':
          return self.manhattan_heuristic(state)

#
===========================================================================
==========================================
class GridSearchingAgent():
     def __init__(self, Problem):
#          Problem.Environment.print_environment()
#          Problem.Environment.print_environment(Problem.initial_state)

     self.Environment = Problem.Environment# seems better
     self.Problem = Problem

     #
===========================================================================
==========================================
     # Gives sequences of actions from from the branch where goal state
was passed on leaf starting from parent state to leaf node (goal state)
     def actions_to_take(self, current_node):
     if current_node.parent is None:   # base case for recursion
```

```
            return []

        return self.actions_to_take(current_node.parent) +
[current_node.parent_action]

        #
================================================================================
==========================================
        # recursive best first search algorithm, returns a sequence of
actions and performance measure
        def recursive_best_first_search_goal(self, heuristic_type):
        node = InformedNodeAlternative(parent=None,
                                       state=self.Problem.initial_state,
                                       parent_action=None,
                                       path_cost=0,

heuristic_score=problem.calculate_heuristic(self.Problem.initial_state,
heuristic_type))

        result, best, search_cost, path_cost = self.RBFS(node, np.inf,
heuristic_type, 0)  # np.inf is infinity provided in numpy  also passing
heuristic type so it would know which heuristic to compute, 0 is initial
search cost, not included in actual algorithm but I have included as it was
the trend above too
        # best is used in RBFS below but as its returned here it is
unnecessary here
        return result, search_cost, path_cost
        #
================================================================================
==========================================

        # actual rbfs
        def RBFS(self, node, f_limit, heuristic_type, search_cost):
        search_cost += 1

        # checking goal test on node
        if self.Problem.goal_test(node):
            return self.actions_to_take(node), 0, search_cost,
node.path_cost  # 0 is immaterial or unimportant as its only for logic to
work correctly

        # only creating child nodes of current node
```

```
    successors = []
    for action in self.Problem.possible_actions: #
possible_actions(node) but in this problem each state has two possible
actions so thats why
        child = InformedChildNodeAlternative(self.Problem, node,
action, heuristic_type)
        successors.append(child)

    if len(successors) == 0:
        return None, np.inf, search_cost, node.path_cost # None,
np.inf are used by algorithm

    # setting successor.f to parent's f if its greater
    for successor in successors:
        successor.f = max(successor.path_cost +
successor.heuristic_score, node.f)

    while True:
        successors.sort(key=lambda successor: successor.f) # For
priority queue so nodes would be in ascending order of f
        best = successors[0]   # Best node with least f value

        # This means that we need to unwind to alternative path from
any ancestor of current node
        if best.f > f_limit:
            return None, best.f, search_cost, node.path_cost # None
is cutoff

        alternative = successors[1].f   # As successors was in
ascending order of f so after best i.e 0 index, node on 1 index is best as
alternative

        result, best.f, search_cost, path_cost = self.RBFS(best,
min(f_limit, alternative), heuristic_type, search_cost)

        if result is not None:
            return result, best.f, search_cost, path_cost



    #
===========================================================================
==========================================
```

```
    # This helper method turns a state {row:x col:y} to (x,y) Note:
state/position in grid seemed better to represent as dictionary for
readibility but for displaying tuple seemed better
    def _state_to_tuple(self, state):
    x = state['row']
    y = state['col']
    return x,y

    #
===========================================================================
==========================================
    # This helper method gives new state (used for printing)
    def _change_state(self, state, action):
    return self.Problem.transition_model(state, action)

    #
===========================================================================
==========================================
    # This helper method displays
    def display_action(self, current_state, action):
    current_pos = self._state_to_tuple(current_state)
    new_state = self._change_state(current_state, action)
    new_pos = self._state_to_tuple(new_state)

    print('Agent moving {} from {} to {}'.format(action, current_pos,
new_pos))
    self.Environment.print_environment(new_state)

    return new_state

    #
===========================================================================
==========================================
    # This method will do searching and if solution exists it will also
display the actions
    def start(self, search_algo, heuristic_type=None):
    print("\n====== {} with h(n)={}========".format(search_algo.upper(),
heuristic_type))
    print("\n---Agent's initial state is {}---".format(
self._state_to_tuple(self.Problem.initial_state) ) )

self.Problem.Environment.print_environment(self.Problem.initial_state)
```

```
    current_state = self.Problem.initial_state

    # searching for solution
    if search_algo == 'bfs':
        solution = self.breadth_first_search_goal()
    elif search_algo == 'greedy':
        solution = self.greedy_best_first_search_goal(heuristic_type)
    elif search_algo == 'A*':
        solution = self.astar_search_goal(heuristic_type)
    elif search_algo == 'rbfs':
        solution =
self.recursive_best_first_search_goal(heuristic_type)

    actions_sequence, search_cost, path_cost = solution

    if actions_sequence:
        for action in actions_sequence:
            current_state = self.display_action(current_state,
action)

    print("---Agent has reached 'G' so stopping")
    self.Environment.print_environment(current_state)
    print("search cost:", search_cost)
    print("path cost:", path_cost)
    print("total cost:", search_cost+path_cost) # total cost combines
both search cost and path cost
    print('\n')

    return
#
============================================================================
==================================================================

environment = Grid_5x5()
row_input = int(input("Enter the ROW of initial state in 5x5 grid: "))
col_input = int(input("Enter the COL of initial state in 5x5 grid: "))
initial_state = {'row':row_input, 'col':col_input}

problem = Problem(environment, initial_state)
agent = GridSearchingAgent(problem)
```

```
search_algo = 'rbfs'
heuristic_type = input("Enter the heuristic (euclidean or manhattan)?: ")
agent.start(search_algo, heuristic_type)
```

WEEK 09

Hill Climbing

```
grid = [[10, 3, 4,  6,  23],
      [9,  32, 12, 2, 34],
      [7,  8,  0,  21, 11],
      [18, 67, 55, 89, 90],
      [22, 33, 14, 44, 50]]


def hill_climbing(grid):
      r, c = 2, 2
      current = grid[r][c]

      found = False
      max_val = []
      neighbor = 0

      while found is False:
      vals = []

      up = [r-1, c]
      dwn = [r+1, c]
      rgt = [r, c+1]
      lft = [r, c-1]

      if up[0] >= 0 and up[0] <= 4 and up[1] >= 0 and up[1] <= 4:
            print('up',up)
            vals.append(['up',up])

      if dwn[0] >= 0 and dwn[0] <= 4 and dwn[1] >= 0 and dwn[1] <= 4:
            print('dwn',dwn)
            vals.append(['dwn',dwn])
```

```
    if rgt[0] >= 0 and rgt[0] <= 4 and rgt[1] >= 0 and rgt[1] <= 4:
        print('rgt',rgt)
        vals.append(['rgt',rgt])

    if lft[0] >= 0 and lft[0] <= 4 and lft[1] >= 0 and lft[1] <= 4:
        print('lft',lft)
        vals.append(['lft',lft])


    for i in vals:
        a, b = i[1][0], i[1][1]
        if grid[a][b] > neighbor:
            neighbor = grid[a][b]
            r, c = a, b

    print('neighbor',neighbor,'r=',r,'c=',c)


    if neighbor <= current:
        return (r, c)

    current = grid[r][c]



if __name__ == '__main__':

    state = hill_climbing(grid)
    print('Now state is:',state)
```

Simulated Annealing

```
import itertools
import random
from math import exp



g = [[10, 3, 4,   6,  23],
    [9,  32, 12, 2,  34],
    [7,  8,  100, 21, 11],
    [18, 67, 55, 89, 90],
```

```
    [22, 33, 14, 44, 110]]


def schedule(t):
    return (pow(10,7)-t)


def simulated_annealing(grid, schedule):
    r, c = 2, 2
    current = [r,c]
    current_val = grid[current[0]][current[1]]


    found = False
    prob = 0

    # loop will run infinitely
    for t in itertools.count():
    if t == 0:
        continue

    probs = []
    max_prob = 0

    T = schedule(t)
#       print('Value of T:',T)

    if T == 0:
        return (r,c)


    neighbors = []

    up = [r-1, c]
    dwn = [r+1, c]
    rgt = [r, c+1]
    lft = [r, c-1]

    if up[0] >= 0 and up[0] <= 4 and up[1] >= 0 and up[1] <= 4:
#       print('up',up)
        neighbors.append(['up',up])
```

```
    if dwn[0] >= 0 and dwn[0] <= 4 and dwn[1] >= 0 and dwn[1] <= 4:
#           print('dwn',dwn)
            neighbors.append(['dwn',dwn])

    if rgt[0] >= 0 and rgt[0] <= 4 and rgt[1] >= 0 and rgt[1] <= 4:
#           print('rgt',rgt)
            neighbors.append(['rgt',rgt])

    if lft[0] >= 0 and lft[0] <= 4 and lft[1] >= 0 and lft[1] <= 4:
#           print('lft',lft)
            neighbors.append(['lft',lft])

    x = random.choice(neighbors)
#           print('Randomly chosen:',x)
    a, b = x[1][0], x[1][1]

    next_ = [a,b]
    next_val = grid[next_[0]][next_[1]]

    delta_E = next_val - current_val
#           print(next_val, current_val)
#           print('Delta E val:',delta_E)

    if delta_E > 0:
            current = next_
            r, c = next_[0], next_[1]
            current_val = grid[current[0]][current[1]]
#           print('New current when E > 0',current)

    else:
            #check for all neighbors
            for n in neighbors:
                    if n[1] == current:
                    continue
                    else:
#                           print('Neighbors:',n)
                    a, b = n[1][0], n[1][1]
                    delta_E = grid[a][b] - grid[current[0]][current[1]]
                    e = exp(delta_E/T)
#                           print('Probability (e)',e)

                    probs.append([[a,b], e])
```

```
            for p in probs:
                  if p[1] > max_prob:
                  max_prob = p[1]
                  r, c = p[0][0], p[0][1]

#           print('Maximum probability:',max_prob,'State:',r,',',c)
            current = [r, c]
            current_val = grid[current[0]][current[1]]

#           print('New current:',current)
#           print('-----------------')


if __name__ == '__main__':

      state = simulated_annealing(g, schedule)
      print(state)
```

WEEK 10

Local Beam Search

```
import random


g = [[10, 3, 4,   6,  23],
      [9,  32, 12, 2,  34],
      [7,  8,  100, 21, 11],
      [18, 67, 55, 89, 90],
      [22, 33, 14, 44, 110]]



def beam_search_best(grid, k):

      # loop to get k random states

      random_states = []
```

```
for i in range(0,k):
s = []

r = random.randint(0,4)
c = random.randint(0,4)

s.append(r)
s.append(c)

if s not in random_states:
      random_states.append(s)
else:
      s = []
      r = random.randint(0,4)
      c = random.randint(0,4)

      s.append(r)
      s.append(c)

      random_states.append(s)


print('Random states:',random_states)

# loop
while True:


# get all successors of all k states
successors = []
for i in random_states:
      r = i[0]
      c = i[1]

      up = [r-1, c]
      dwn = [r+1, c]
      rgt = [r, c+1]
      lft = [r, c-1]

      if up[0] >= 0 and up[0] <= 4 and up[1] >= 0 and up[1] <= 4:
            if up not in successors and up not in random_states:
            successors.append(up)
```

```
        if dwn[0] >= 0 and dwn[0] <= 4 and dwn[1] >= 0 and dwn[1] <= 4:
            if dwn not in successors and dwn not in random_states:
            successors.append(dwn)

        if rgt[0] >= 0 and rgt[0] <= 4 and rgt[1] >= 0 and rgt[1] <= 4:
            if rgt not in successors and rgt not in random_states:
            successors.append(rgt)

        if lft[0] >= 0 and lft[0] <= 4 and lft[1] >= 0 and lft[1] <= 4:
            if lft not in successors and lft not in random_states:
            successors.append(lft)

    print('successors:',successors)



    # finding sol: if any random state has greater value than all the
successors in the list

    for i in random_states:
        for j in successors:
            if grid[i[0]][i[1]] > grid[j[0]][j[1]]:
            flg = True
            continue
            else:
            flg = False
            break

        if flg == True:
            print('Solution:',i[0],i[1])
            return (i[0],i[1])



    # if solution not found
    # get k best successors

    succ_vals = []
    succ = successors
    for i in range(0,k):
```

```
            max_val = 0
            for j in succ:

                  val = grid[j[0]][j[1]]
                  print('Val:',val)

                  if val > max_val:
                  max_val = val
                  row = j[0]
                  col = j[1]

            succ_vals.append([row,col])
            print('Appending successor:',succ_vals)

            print('max',max_val)
            succ.remove([row,col])


      print('succ_vals:',succ_vals)

      random_states = succ_vals


#=============================================================


def beam_search_random(grid, k):

      # loop to get k random states

      random_states = []

      for i in range(0,k):
      s = []

      r = random.randint(0,4)
      c = random.randint(0,4)

      s.append(r)
      s.append(c)

      if s not in random_states:
```

```
            random_states.append(s)
        else:
            s = []
            r = random.randint(0,4)
            c = random.randint(0,4)

            s.append(r)
            s.append(c)

            random_states.append(s)


    print('Random states:',random_states)

    # loop
    while True:


    # get all successors of all k states
    successors = []
    for i in random_states:
            r = i[0]
            c = i[1]

            up = [r-1, c]
            dwn = [r+1, c]
            rgt = [r, c+1]
            lft = [r, c-1]

            if up[0] >= 0 and up[0] <= 4 and up[1] >= 0 and up[1] <= 4:
                if up not in successors and up not in random_states:
                successors.append(up)

            if dwn[0] >= 0 and dwn[0] <= 4 and dwn[1] >= 0 and dwn[1] <= 4:
                if dwn not in successors and dwn not in random_states:
                successors.append(dwn)

            if rgt[0] >= 0 and rgt[0] <= 4 and rgt[1] >= 0 and rgt[1] <= 4:
                if rgt not in successors and rgt not in random_states:
                successors.append(rgt)

            if lft[0] >= 0 and lft[0] <= 4 and lft[1] >= 0 and lft[1] <= 4:
```

```
                if lft not in successors and lft not in random_states:
                successors.append(lft)

    print('successors:',successors)



    # finding sol: if any random state has greater value than all the
successors in the list

    for i in random_states:
        for j in successors:
            if grid[i[0]][i[1]] > grid[j[0]][j[1]]:
            flg = True
            continue
            else:
            flg = False
            break

        if flg == True:
            print('Solution:',i[0],i[1])
            return (i[0],i[1])



    # if solution not found
    # get k random successors

    succ_vals = []
    for i in range(0,k):

        succ = random.choice(successors)
        succ_vals.append(succ)

    print('Randomly selected k successors:',succ_vals)
    random_states = succ_vals


#======================================================================

if __name__ == '__main__':

```

```
      result = beam_search_best(g, 4)
      print('Beam Search (best k successors)',result,'\n')

      print('-----------------\n')

      result2 = beam_search_random(g, 4)
      print('Beam Search (random k successors)',result2)
```

WEEK 11

Genetic Algorithm

```
import random

init_state = [[0, 0, 0, 0],
           [0, 0, 0, 0],
           [0, 0, 0, 0],
           [0, 0, 0, 0]]

initial_state = []
def get_1D(init_state):
      for i in init_state:
      for j in i:
           initial_state.append(j)

      return initial_state


def get_population(init_state):
      P = []
      for i in range(1,5):
      X = []

      for j in range(0,len(init_state)):
           chromo = random.randint(0,1)
           X.append(chromo)
      P.append(X)

      return P
```

```
def fitness_fn(population):
      fitness_vals = []
      for i in population:
      count = 0
      for j in i:
            if j == 1:
                  count += 1
      fitness_vals.append(count)

      return fitness_vals


def random_selection(population, fitness_fn, selected):
      fitness_vals = fitness_fn(population)
      probs = []
      max_prob = 0

      for i in fitness_vals:
      p = i/sum(fitness_vals)
      probs.append(p)


      for i in range(0,len(probs)):
      if probs[i] > max_prob:
            if i not in selected:
                  max_prob = probs[i]
                  index = i

      selected.append(index)

      return population[index], selected


def reproduce(x, y):
      n = len(x)

      # random crossover point
      c = random.randint(1,n)

      print('crossover point:',c)
```

```
    left_x = x[:c]

    right_y = y[c:]

    child = []

    print('x:',left_x)
    print('y:',right_y)

    for i in left_x:
    child.append(i)
    for i in right_y:
    child.append(i)

    print('child:',child)
    return child



def mutate(child):

    for i in range(0,len(child)):
    val = random.randint(0,100)

    # fixed 25 from 0-100, if 25 comes then mutation takes place
    if val == 25:
        pos = i
        child[i] = 1

    return child

def goal_test(population):
    fitness_vals = fitness_fn(population)
    print('Fitness vals:',fitness_vals)
    max_val = 0
    for i in range(0,len(fitness_vals)):
    if fitness_vals[i] > max_val:
        max_val = fitness_vals[i]
        index = i

    return population[index]
```

```
def genetic_algo(population, fitness_fn):

      leave = False

      small_random_probability = 0.01
      while True: #change it afterwards
      new_population = []

      # selection
      for i in range(0,len(population)):
            selected = []
            x, selected  = random_selection(population, fitness_fn,
selected)
            print('random X:',x)

            y, select  = random_selection(population, fitness_fn, selected)
            print('random Y:',y)



            # crossover
            child = reproduce(x,y)

            # mutation
            if small_random_probability == 0.01:
                  child = mutate(child)
                  print('Mutated child:',child)
                  print('----------------')

            new_population.append(child)

      population = new_population
      print('New population:',population)

      # to break while loop when we get goal state of child (all 1s)
      for i in population:
            count = 0
            for j in i:
                  if j == 1:
                  count += 1
```

```
            print('count:',count)
            if count == len(population[0]):
                leave = True
                break

        if leave == True:
            break

        # evaluation
        best_individual = goal_test(population)
        return best_individual

f __name__ == '__main__':

        init_state = get_1D(init_state)
        population = get_population(init_state)
        print('population:',population)
        result = genetic_algo(population, fitness_fn)
        print('Best individual:', result)
```

WEEK 12

K-Nearest Neighbors

```
import pandas as pd
from collections import Counter
Counter()


data = pd.read_csv('dataset.csv')


# isolating necessary columns
cols = [0,3,4,5,6]
data = data.iloc[:,cols]


# to fill up empty spaces with mean of the column
mass_mean = data['mass'].mean()
height_mean = data['height'].mean()
```

```
# to fill up empty spaces with data
data.loc[17] = pd.Series({'fruit_label':data.loc[17]['fruit_label'],
'mass':mass_mean, 'width':data.loc[17]['width'], 'height':height_mean,
'color_score':data.loc[17]['color_score']})
data.loc[18] = pd.Series({'fruit_label':data.loc[18]['fruit_label'],
'mass':mass_mean, 'width':data.loc[18]['width'], 'height':height_mean,
'color_score':data.loc[18]['color_score']})
data.loc[19] = pd.Series({'fruit_label':data.loc[19]['fruit_label'],
'mass':mass_mean, 'width':data.loc[19]['width'], 'height':height_mean,
'color_score':data.loc[19]['color_score']})
data.loc[20] = pd.Series({'fruit_label':data.loc[20]['fruit_label'],
'mass':mass_mean, 'width':data.loc[20]['width'], 'height':height_mean,
'color_score':data.loc[20]['color_score']})
data.loc[21] = pd.Series({'fruit_label':data.loc[21]['fruit_label'],
'mass':mass_mean, 'width':data.loc[21]['width'], 'height':height_mean,
'color_score':data.loc[21]['color_score']})
data.loc[22] = pd.Series({'fruit_label':data.loc[22]['fruit_label'],
'mass':mass_mean, 'width':data.loc[22]['width'], 'height':height_mean,
'color_score':data.loc[22]['color_score']})
data.loc[23] = pd.Series({'fruit_label':data.loc[23]['fruit_label'],
'mass':mass_mean, 'width':data.loc[23]['width'], 'height':height_mean,
'color_score':data.loc[23]['color_score']})
data.loc[24] = pd.Series({'fruit_label':data.loc[24]['fruit_label'],
'mass':mass_mean, 'width':data.loc[24]['width'], 'height':height_mean,
'color_score':data.loc[24]['color_score']})
data.loc[25] = pd.Series({'fruit_label':data.loc[25]['fruit_label'],
'mass':mass_mean, 'width':data.loc[25]['width'], 'height':height_mean,
'color_score':data.loc[25]['color_score']})


# to shuffle rows and to keep the indexes in order
data = data.sample(frac=1).reset_index(drop=True)




def KNN(data, k):

      distances = []
      predicted_labels = []

      for j in range(50,60):
      dist = {}
```

```
    for i in range(0,50):

        # calculating Euclidean distance
        ans = pow(data.loc[i]['mass']-data.loc[j]['mass'],2) +
pow(data.loc[i]['width']-data.loc[j]['width'],2) +
pow(data.loc[i]['height']-data.loc[j]['height'],2) +
pow(data.loc[i]['color_score']-data.loc[j]['color_score'],2)

        dist[i] = [ans, data.loc[i]['fruit_label']]

    # sorted dataset to get K elements
    sorted_dist = sorted(dist.items(), key=lambda x: x[1], reverse=False)
    distances.append(sorted_dist[0:k])


    print(*distances,sep='\n')


    # counting labels of test data
    for i in range(0,len(distances)):
    labels = []
    for j in range(0,k):
        labels.append(distances[i][j][1][1])

    lst = Counter(labels)
    predicted_labels.append(lst.most_common()[0][0])

    print('Predicted labels:',predicted_labels)


if __name__ == '__main__':

    KNN(data, 3)
```

WEEK 13
Perceptron

```
import pandas as pd
df = pd.read_csv('dataset.csv')
```

```
#missing values
mean_mass = df['mass'].mean()
df['mass'].fillna(mean_mass, inplace =True)

mean_height = df['height'].mean()
df['height'].fillna(mean_height, inplace=True)

#drop columns
df.drop(['fruit_name'], axis=1, inplace=True)
df.drop(['fruit_subtype'], axis=1, inplace=True)

# make labels binary because Perceptron is Binary Classification Model
for i in range(len(df)):
     if df['fruit_label'][i] == 2:
     df['fruit_label'][i] = 1
     elif df['fruit_label'][i] == 3 or df['fruit_label'][i] == 4:
     df['fruit_label'][i] = 0

import numpy as np

class Perceptron:

     def __init__(self, learning_rate=0.01, n_iters=1000):
     self.lr = learning_rate
     self.n_iters = n_iters
     self.activation_func = self._unit_step_func
     self.weights = None
     self.bias = None

     def fit(self, X, y):
     n_samples, n_features = X.shape

     # init parameters
#          self.weights = np.zeros(n_features)
     self.weights = [1,1,0,1]
     self.bias = 0

     y_ = np.array([1 if i > 0 else 0 for i in y])

     for _ in range(self.n_iters):

          for idx, x_i in enumerate(X):
```

```
                linear_output = np.dot(x_i, self.weights) + self.bias
                y_predicted = self.activation_func(linear_output)

                # Perceptron update rule
                update = self.lr * (y_[idx] - y_predicted)

                self.weights += update * x_i
                self.bias += update

     def predict(self, X):
     linear_output = np.dot(X, self.weights) + self.bias
     y_predicted = self.activation_func(linear_output)
     return y_predicted

     def _unit_step_func(self, x):
     return np.where(x>=0, 1, 0)


p = Perceptron()

# X = features/columns , y = labels
y = df['fruit_label']
df.drop(['fruit_label'], axis=1, inplace=True)
X = np.array(df)
p.fit(X, y)
p.predict(X)
```

WEEK 14
K-Means Clustering

```
import pandas as pd
import random
from math import sqrt
import numpy as np
from datetime import datetime, timedelta

corpus = pd.read_csv('dataset.csv', encoding='Latin-1')

corpus = corpus.iloc[:,[3,4,5,6]]
```

```
mass_mean = corpus['mass'].mean()
height_mean = corpus['height'].mean()

corpus.loc[17] = pd.Series({'mass':mass_mean,
'width':corpus.loc[17]['width'], 'height':height_mean,
'color_score':corpus.loc[17]['color_score']})
corpus.loc[18] = pd.Series({'mass':mass_mean,
'width':corpus.loc[18]['width'], 'height':height_mean,
'color_score':corpus.loc[18]['color_score']})
corpus.loc[19] = pd.Series({'mass':mass_mean,
'width':corpus.loc[19]['width'], 'height':height_mean,
'color_score':corpus.loc[19]['color_score']})
corpus.loc[20] = pd.Series({'mass':mass_mean,
'width':corpus.loc[20]['width'], 'height':height_mean,
'color_score':corpus.loc[20]['color_score']})
corpus.loc[21] = pd.Series({'mass':mass_mean,
'width':corpus.loc[21]['width'], 'height':height_mean,
'color_score':corpus.loc[21]['color_score']})
corpus.loc[22] = pd.Series({'mass':mass_mean,
'width':corpus.loc[22]['width'], 'height':height_mean,
'color_score':corpus.loc[22]['color_score']})
corpus.loc[23] = pd.Series({'mass':mass_mean,
'width':corpus.loc[23]['width'], 'height':height_mean,
'color_score':corpus.loc[23]['color_score']})
corpus.loc[24] = pd.Series({'mass':mass_mean,
'width':corpus.loc[24]['width'], 'height':height_mean,
'color_score':corpus.loc[24]['color_score']})
corpus.loc[25] = pd.Series({'mass':mass_mean,
'width':corpus.loc[25]['width'], 'height':height_mean,
'color_score':corpus.loc[25]['color_score']})

# Step 2 - Select centroids

def select_centroids(corpus, k, new_centroids):

      if new_centroids != []:
      return new_centroids

      sc = []
      selected_centroids = []
```

```
    for i in range(k):
    r = random.randint(0,len(corpus)-1)

    if r in sc:
        for i in range(10):
            r2 = random.randint(0, len(corpus)-1)
            if r2 not in sc:
            sc.append(r2)
            selected_centroids.append([corpus['mass'][r2],
corpus['width'][r2], corpus['height'][r2], corpus['color_score'][r2]])
            break

    else:
        sc.append(r)
        selected_centroids.append([corpus['mass'][r],
corpus['width'][r], corpus['height'][r], corpus['color_score'][r]])

    return selected_centroids

# Step 3 - Assign points to closest cluster centroid b calculating
Euclidean distance

def assign_points(corpus, selected_centroids):
    cluster1, cluster2, cluster3 = [], [], []
    for i in range(len(corpus)):

    row = [corpus['mass'][i], corpus['width'][i], corpus['height'][i],
corpus['color_score'][i]]

    if row not in selected_centroids:

        best_centroid = calc_distance(corpus, selected_centroids, row)

        if best_centroid == 0:
            cluster1.append(row)

        elif best_centroid == 1:
            cluster2.append(row)

        elif best_centroid == 2:
            cluster3.append(row)
```

```
    return cluster1, cluster2, cluster3


# calculate euclidean distance between centroids and all points

def calc_distance(corpus, selected_centroids, row):
    clusters = []
    d = float('inf')

    for i in range(len(selected_centroids)):

    dist = sqrt( pow(np.array(selected_centroids[i][0]-row[0]), 2) +
    pow(np.array(selected_centroids[i][1]-row[1]), 2) +
    pow(np.array(selected_centroids[i][2]-row[2]), 2) +
    pow(np.array(selected_centroids[i][3]-row[3]), 2) )

    if dist < d:
        d = dist
        b = i

    return b


# Step 4 - Recompute centroids

def recompute_centroids(selected_centroids, cluster1, cluster2, cluster3):

    avg_mass, avg_width, avg_height, avg_colorscore = 0, 0, 0, 0

    for i in cluster1:
    avg_mass += i[0]
    avg_width += i[1]
    avg_height += i[2]
    avg_colorscore += i[3]

    avg_mass = avg_mass/len(cluster1)
    avg_width = avg_width/len(cluster1)
    avg_height = avg_width/len(cluster1)
    avg_colorscore = avg_width/len(cluster1)
    selected_centroids[0] = [avg_mass, avg_width, avg_height,
avg_colorscore]
```

```
     avg_mass, avg_width, avg_height, avg_colorscore = 0, 0, 0, 0

     for i in cluster2:
     avg_mass += i[0]
     avg_width += i[1]
     avg_height += i[2]
     avg_colorscore += i[3]

     avg_mass = avg_mass/len(cluster2)
     avg_width = avg_width/len(cluster2)
     avg_height = avg_width/len(cluster2)
     avg_colorscore = avg_width/len(cluster2)
     selected_centroids[1] = [avg_mass, avg_width, avg_height,
avg_colorscore]


     avg_mass, avg_width, avg_height, avg_colorscore = 0, 0, 0, 0

     for i in cluster3:
     avg_mass += i[0]
     avg_width += i[1]
     avg_height += i[2]
     avg_colorscore += i[3]

     avg_mass = avg_mass/len(cluster3)
     avg_width = avg_width/len(cluster3)
     avg_height = avg_width/len(cluster3)
     avg_colorscore = avg_width/len(cluster3)
     selected_centroids[2] = [avg_mass, avg_width, avg_height,
avg_colorscore]

     return selected_centroids


def KMeans(corpus, k):

     print('Please wait, the algorithm is running for 3 seconds.')

     end_time = datetime.now() + timedelta(seconds=3)

     new_centroids = []
```

```
      while datetime.now() < end_time:

      selected_centroids = select_centroids(corpus, k, new_centroids)

      cluster1, cluster2, cluster3 = assign_points(corpus,
selected_centroids)

      new_centroids = recompute_centroids(selected_centroids, cluster1,
cluster2, cluster3)


      return cluster1, cluster2, cluster3

# -------------------------------------------------------------------

if __name__ == '__main__':

      k = 3
      c1, c2, c3 = KMeans(corpus,k)
      print('Cluster 1:',c1,sep='\n')
      print('---------------')
      print('Cluster 2:',c2,sep='\n')
      print('---------------')
      print('Cluster 3:',c3,sep='\n')
```