AI Lab Implementations

P180126 Ahmad Hasan Syed

Lab 1:#Data Structures

```
class stack:
    def __init__(self):
        self.l=[]
    def push(self,val):
        self.l.append(val)
    def pops(self):
        self.l.pop()
    def peek(self):
        return self.l[-1]
```

```
    #Queue
```

```
    class Queue:
    def __init__(self):
        self.size=8;
        self.q=[8]
        self.front=0
        self.rear=0
        self.len=0
    def isempty(self):
        if self.len==0:
            print("Queue is empty!")
            return True
        return false
    def enqueue(self,val):
        if self.len<8:
            self.q.append(val)
            self.rear+=1
            self.len+=1
    def pri(self):
        for i in range(0,self.len):
            print(self.q[i])
```

```
q=Queue()
q.pri()
q.isempty()
```

```python
q.enqueue(5)
q.enqueue(4)
q.pri()
```

```python
    #BSTrees
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.right = None
        self.left = None
```

```python
def dfs(self):
    print(self.val)
    if self.left:
        self.left.dfs()
    if self.right:
        self.right.dfs()
```

```python
TreeNode.dfs = dfs
```

```python
def dfs_inorder(self):
    if self.left:
        self.left.dfs()
    print(self.val)
    if self.right:
        self.right.dfs()
```

```python
TreeNode.dfs_inorder = dfs_inorder
```

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

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

```
def find_root(self):
    """Find the absolute root of BST to which self belongs.."""
    temp = self
    while temp.parent is not None:
        temp = temp.parent
    return temp
```

```
BST.find_root = find_root
```

```
def find_min(self):
    """Find the minimum value starting from self.
    IN BST, this is simple, keep going left until no left is 
left"""
    min_node = self
    if self.left is not None:
        min_node = self.left.find_min()
    return min_node
```

```
BST.find_min = find_min
```

```
def set_for_parent(self, new_ref):
    """Disconnect self from parent and attach new_ref to parent in 
self's place"""
    if self.parent is None: return

    if self.parent.right == self:
        self.parent.right = new_ref
    if self.parent.left == self:
        self.parent.left = new_ref
```

```
BST.set_for_parent = set_for_parent
```

```
def replace_with_node(self, node):
    """Replace self with node (Which is child). Make sure to fix 
the parent of the node and parent' pointing to node.
    Assume we have no children other than node"""
```

```
    self.set_for_parent(node)
    node.parent = self.parent
    self.parent = None
    return node.find_root()


BST.replace_with_node = replace_with_node


def delete(self, val):

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

    if self.val == val:

        if self.right is None and self.left is None:
            self.set_for_parent(None)
            return self.find_root()


        if self.right is None:
            return self.replace_with_node(self.left)


        if self.left is None:
            return self.replace_with_node(self.right)

        """(Our Successor is definitely in our right child and it
can't have two children
        because left child will always be smaller)"""
        successor = self.right.find_min()
        self.val = successor.val

        return self.right.delete(successor.val)
          if val < self.val:
        if self.left:
            return self.left.delete(val)
        else:
            return self.find_root()
    else:
        if self.right:
            return self.right.delete(val)
        else:
            return self.find_root()


BST.delete = delete

b = BST(20)
b.insert(13)
```

```python
b.insert(10)
b.insert(155)
b.insert(12)
print(":::::::: DFS in-order Traversal ::::::::::")

b = b.delete(20)
b = b.delete(155)
```

```python
#Graphs
class Digraph:
    def __init__(self):
        self.g = {}

    def addNode(self, node):
        if node in self.g:
            raise ValueError("Noe already in graph")

        self.g[node] = []

    def addEdge(self, src, dst):
        # Sanity Checks
        if src not in self.g or dst not in self.g:
            return
        else:
            nexts = self.g[src]
            if dst in nexts:
                return
            nexts.append(dst)
```

```python
g = Digraph()
nodes = ['a', 'b', 'c', 'd', 'e', 'f']
for n in nodes:
    g.addNode(n)

edges = [
    ('a', 'b'),
    ('a', 'c'),
    ('b', 'c'),
    ('b', 'd'),
    ('c', 'd'),
    ('d', 'c'),
    ('e', 'f'),
    ('f', 'c'),
]
for e in edges:
```

```
    g.addEdge(e[0], e[1])


print("printing graph in simple way")
print(g.g)
import pprint
print("\nPrinting Graph using PPRINT")
pprint.pprint(g.g)



def traverse_graph(self, start):
    q = [start]
    visited = []

    while q:
        current = q.pop(0)
        if current in visited:
            continue
        print(current)
        visited.append(current)
        next_nodes = self.g[current]
        for n in next_nodes:
            q.append(n)


Digraph.traverse_graph = traverse_graph

print("\nGraph Traversal")
g.traverse_graph('e')


def find_path(self, start, end, path=[]):
    """Find path (not shortest) from start to end"""
    if start not in self.g or end not in self.g:
        return ValueError("Source/End node not in graph")
    path = path + [start]
    if start == end:
        return path
    for node in self.g[start]:
        if node not in path:
            newpath = self.find_path(node, end, path)
            if newpath:
                return newpath
    return None


Digraph.find_path = find_path

print("\nPATH FINDING")
print(g.find_path('a', 'd'))
print(g.find_path('b', 'f'))
```

```python
print(g.find_path('b', 'd'))


def find_all_paths(self, start, end, path=[]):
    if start not in self.g or end not in self.g:
        return ValueError("Source/End node not in graph")
    path = path + [start]
    if start == end:
        return [path]
    all_paths = []

    for node in self.g[start]:
        if node not in path:
            all_newpaths = self.find_all_paths(node, end, path)
            for newpath in all_newpaths:
                all_paths.append(newpath)

    return all_paths


Digraph.find_all_paths = find_all_paths

print("\nFIND ALL PATHS")
print(g.find_all_paths('a', 'd'))


def find_shortest_path(self, start, end, path=[]):
    """Find path (not shortest) from start to end"""
    if start not in self.g or end not in self.g:
        return ValueError("Source/End node not in graph")
    path = path + [start]
    if start == end:
        return path

    shortest = None
    for node in self.g[start]:
        if node not in path:
            newpath = self.find_path(node, end, path)
            if newpath:
                if shortest is None or len(newpath) < 
len(shortest):  # Change
                    shortest = newpath

    return shortest


Digraph.find_shortest_path = find_shortest_path

print("\nFIND SHORTEST PATHS")
print(g.find_shortest_path('a', 'd'))
```

Lab 2:

```
a=[[1,3,5,6,8],
    [2,3,4,5,7],
    [1,2,3,4,5],
    [12,3,4,65,4]
]
n=4     #rows
m=5     #column     #static
l=[]
node=a[0][0]

class node:

    def hill():
        q,w=0
        if node==None:
            return
        for q in range (i):
            for w in range (j):
                if node[i][j]<node[i][j+1]:
                   # node[i][j]=node[i][j+1]   #right movement
                    l.append(node[i][j+1])

                if node[i][j]<node[i+1][j]:
                    #node[i][j]=node[i+1][j]   #down
                    l.append(node[i+1][j])

                if node[i][j]<node[i-1][j]:
                    #node[i][j]=node[i-1][j]
                    l.append(node[i-1][j])

                if node[i][j]<node[i][j-1]:
                    #node[i][j]=node[i][j-1]:
                    l.append(node[i][j-1])
            maximum(l)


    def maximum(l):
        x=max(a)
        print("Global max of 2d array is:",x)
        return


n=node()



###########Q2 Starts from here####################
Q2:
```

```python
from numpy import asarray
from numpy import exp
from numpy.random import randn
from numpy.random import rand
from numpy.random import seed



def objective(x):
    return x[0]**2.0

def simulated_annealing(objective, bounds, n_iterations, step_size, temp):

    best = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])

    best_eval = objective(best)

    curr, curr_eval = best, best_eval
    scores = list()

    for i in range(n_iterations):

        candidate = curr + randn(len(bounds)) * step_size

        candidate_eval = objective(candidate)

        if candidate_eval < best_eval:

            best, best_eval = candidate, candidate_eval

            scores.append(best_eval)

            print('>%d f(%s) = %.5f' % (i, best, best_eval))

        diff = candidate_eval - curr_eval

        t = temp / float(i + 1)

        metropolis = exp(-diff / t)

        if diff < 0 or rand() < metropolis:

            curr, curr_eval = candidate, candidate_eval
    return [best, best_eval, scores]

# seed
seed(1)
#  range for input
bounds = asarray([[-5.0, 5.0]])
# total iterations
```

```python
n_iterations = 1000
#  maximum step size
step_size = 0.1
# initial temp
temp = 10
#
best, score, scores = simulated_annealing(objective, bounds,
n_iterations, step_size, temp)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Lab 3:

```
class Binary_Search_Tree:
def __init__(self, data): self.data = data self.Left_child = None self.Right_child = None
def Add_Node(self, data): if data == self.data:
return # node already exist
if data < self.data: if self.Left_child:
self.Left_child.Add_Node(data) else:
self.Left_child = Binary_Search_Tree(data)
else:
if self.Right_child:
self.Right_child.Add_Node(data) else:
self.Right_child = Binary_Search_Tree(data)
def bfs(self): elements = [] path_cost=0
if self.Left_child:
elements += self.Left_child.bfs() path_cost=path_cost+1
elements.append(self.data) path_cost=path_cost+1
if self.Right_child:
elements += self.Right_child.bfs() path_cost=path_cost+1
print(path_cost) #each element traversed and added to queue will be counted as 1 cost of path
return elements
 P18-0126 Ahmad Hasan Syed
def dfs(self):
elements = [self.data] if self.Left_child:
elements += self.Left_child.dfs() if self.Right_child:
elements += self.Right_child.dfs() return elements
#vacum cleaner code
env_table={"Location A 0":"vacum at location A and location is dirty",
"Location A 1":"vacum at location A and location is clean", "Location B 0":"vacum at location B
and location is dirty", "Location B 1":"vacum at location B and location is clean", "Location C
0":"vacum at location C and location is dirty", "Location C 1":"vacum at location C and location is
clean", "Location D 0":"vacum at location D and location is dirty", "Location D 1":"vacum at
location D and location is clean", "Location E 0":"vacum at location E and location is dirty",
"Location E 1":"vacum at location E and location is clean", "Location F 0":"vacum at location F
and location is dirty", "Location F 1":"vacum at location F and location is clean", "Location G
0":"vacum at location G and location is dirty", "Location G 1":"vacum at location G and location is
clean", "Location H 0":"vacum at location H and location is dirty", "Location H 1":"vacum at
location H and location is clean", "Location I 0":"vacum at location I and location is dirty",
"Location I 1":"vacum at location I and location is clean", "Back":"All locations are clean"
}
def location_check():
n=0 #here we mean 0 is A b=Binary_Search_Tree(n) b.Add_Node(n)#setting initial state at A q=9
```

```
for i in range(q):
n = np.random.uniform(low=1.0, high=9.0) #starting from B b.insert(n)
c=b.bfs() #by this we will search the location using bfs in tree return c
def dirt_clean_cal(): num=1
for i in range(num): #for loop for flexibility so that if new state to be added num =
np.random.uniform(low=0.0, high=1.0)
if num > 0.2:
return 0 # prob of 0.2 of dirt if prob>0.2 return 0(location is dirty!) else:
return 1 #else location is clean
```

P18-0126 Ahmad Hasan Syed

```
def main_fun(): x=location_check() s=dirt_clean_cal() if x==0:
if s==0:
print(env_table["Location A 0"])
else:
print(env_table["Location A 1"])
if x==1:
if s==0:
print(env_table["Location B 0"]) else:
print(env_table["Location B 1"])
if x==2:
if s==0:
print(env_table["Location C 0"]) else:
print(env_table["Location C 1"]) if x==3:
if s==0:
print(env_table["Location D 0"])
else:
print(env_table["Location D 1"])
if x==4:
if s==0:
print(env_table["Location E 0"]) else:
print(env_table["Location E 1"]) if x==5:
if s==0:
print(env_table["Location F 0"])
else:
print(env_table["Location F 1"])
if x==6:
if s==0:
print(env_table["Location G 0"]) else:
print(env_table["Location G 1"]) if x==7:
if s==0:
print(env_table["Location H 0"])
else:
print(env_table["Location H 1"])
if x==8:
if s==0:
print(env_table["Location I 0"]) else:
print(env_table["Location I 1"]) else:
print(env_table["Back"]) return
```

Lab 4:

```
#modified bfs for greedy first search also
import queue as que
def bfs(self, matrix, root, goal):
cost = -1
visited, queue,opened = set(), collections.deque([root]),que opened.append(root)
while queue:
explore=True #set it to by-default True print("BFS QUEUE ")
for q in queue:
if opened!=None:
print(q.key, q.location) if opened==None:
return queue
#Dequeue a Node from queue\n"
node = queue.popleft()
visited.enqueue(node)
print(\"\\nExpanding Node: \"+ str(node.key) + \" \", end=\"\")
#check if current node is a goal if self.goalTest(node.key, goal):
print(\"\\n\\n===================\\nHurrah! Found Goal!\ \n====================\\n\\n\")
#call to a function: findGoalPath\
#calculating total cost and path from goal-node to the initial state
opened.dequeue(node) visited.enqueue(node) self.findGoalPath(node) break
return
#call to function: possibleActions\n",
ans = self.possibleActions(matrix, node.key, node.location) print("Locations of all possible actions =")
cost += 1
#this function returns an iterable list of 2D-Matrix-indices (i,j) where agent can move
next!\n",
for nextActionDirection, nextActionLoc in ans.items():
i, j = nextActionLoc[0], nextActionLoc[1]
newNodeVal = matrix[i][j] newNodeLoc = tuple((i,j)) newNodeLabel = nextActionDirection
#for first iteration, don't check parent of Root Node; No need\n", if cost<=0:
 newNode = root.insert(node.key, node.location, newNodeVal, newNodeLoc, newNodeLabel)
#print(\"New node = \", newNode.key, newNode.location)\n", #for the first level\n",
visited.add(newNode)
queue.append(newNode)
opened.dequeue(newNode)
#check not to add parent of a node under its child\n",
if cost > 0 and node.parent.location is not newNodeLoc:
newNode = root.insert(node.key, node.location, newNodeVal, newNodeLoc, newNodeLabel)
#check the neighbours of a node if they're already visited or not\n", #node's can have same value but Unique (row,col) location on matrix\n", for eachNode in visited:
if eachNode.location == newNodeLoc: explore=False
# If not visited, mark it as visited, and enqueue it\n", if explore:
visited.add(newNode) queue.append(newNode) opened.dequeue(newnNode)
return None
#==========================================\n", #End of Class\n",
#==========================================\n",
```

A* search general algorithm still working on its final form will submit it you allow me later couldn't complete due to some technical issues

```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.prepend(current)
    return total_path
// A* finds a path from start to goal.
// h is the heuristic function. h(n) estimates the cost to reach
```

```
goal from node n.
function A_Star(start, goal, h)
    // The set of discovered nodes that may need to be
(re-)expanded.
// Initially, only the start node is known.

// This is usually implemented as a min-heap or priority queue
rather than a hash-set.
    openSet := {start}
    // For node n, cameFrom[n] is the node immediately preceding
it on the cheapest path from start
    // to n currently known.
    cameFrom := an empty map
    // For node n, gScore[n] is the cost of the cheapest path from
start to n currently known.
    gScore := map with default value of Infinity
    gScore[start] := 0
    // For node n, fScore[n] := gScore[n] + h(n). fScore[n]
represents our current best guess as to
    // how short a path from start to finish can be if it goes
through n.
    fScore := map with default value of Infinity
    fScore[start] := h(start)
    while openSet is not empty
        // This operation can occur in O(1) time if openSet is a
min-heap or a priority queue
        current := the node in openSet having the lowest fScore[]
value
neighbor)
tentative_gScore := gScore[current] + d(current,
if tentative_gScore < gScore[neighbor]
    // This path to neighbor is better than any
if current = goal
    return reconstruct_path(cameFrom, current)
openSet.Remove(current)
for each neighbor of current
            // d(current,neighbor) is the weight of the edge from
current to neighbor
            // tentative_gScore is the distance from start to the
neighbor through current
previous one. Record it!
                cameFrom[neighbor] := current
                gScore[neighbor] := tentative_gScore
                fScore[neighbor] := gScore[neighbor] + h(neighbor)
                if neighbor not in openSet
                    openSet.add(neighbor)
    // Open set is empty but goal was never reached
    return failure
```

#5Hill Climbing:

```
from random import *
import random
import numpy
import copy

countCities = 20;
# 2D Array
cities = numpy.zeros(shape=(20,20))
# tour
hypothesis = [int]*countCities
visitedCities = []
saveState = []

threshold = 25
lastFitness = 0
trials = 0
cityIndex = 1

# calculates fitness based on the difference between the distances
def getFitness(fitness, hypothesis, saveState, cities):
    oldDistance = getDistance(cities, saveState)
    newDistance = getDistance(cities, hypothesis)
    print("Old Distance ",oldDistance,"km")
    print("New Distance ",newDistance,"km")

    if(oldDistance > newDistance):
        fitness += 1
    elif(oldDistance < newDistance):
        fitness -= 1

    return fitness

# choose random City at position cityIndex
def doRandomStep():
    global visitedCities
    global saveState
    global hypothesis
    if(len(visitedCities) >= countCities):
        visitedCities.clear()
        visitedCities.append(0)
    randomNumbers = list(set(saveState) - set(visitedCities))
    randomStep = random.choice(randomNumbers)
    visitedCities.append(randomStep)
    hypothesis.remove(randomStep)
    hypothesis.insert(cityIndex,randomStep)

# next city
def increment():
    global cityIndex
    global visitedCities
```

```
    if (cityIndex < countCities - 2):
        cityIndex += 1
    else:
        visitedCities.clear()
        cityIndex = 1

# calculates distance from tour
def getDistance(cities, hypothesis):
    distance = 0
    for i in range(countCities):
        if (i < countCities-1):
            distance += cities[hypothesis[i]][hypothesis[i+1]]
            print("[",hypothesis[i],"]",distance,"km ",end="")
        else:
            print("[",hypothesis[i],"]")

    return distance

if __name__ == '__main__':

    for i in range(countCities):
        hypothesis[i] = i
        for j in range(countCities):
            if (j > i):
                cities[i][j] = randint(1,100)
            elif(j < i):
                cities[i][j] = cities[j][i]

    print("=== START ===");
    while(lastFitness < threshold):


print("_________________________________________________________")
        saveState = copy.deepcopy(hypothesis)
        doRandomStep()
        currentFitness = getFitness(lastFitness, hypothesis, 
saveState, cities)
        print("Old fitness ",lastFitness)
        print("Current fitness ",currentFitness)

        if (currentFitness > lastFitness):
            lastFitness = currentFitness
        elif(currentFitness < lastFitness):
            hypothesis = copy.deepcopy(saveState)
            if(trials < 3):
                increment()
            else:
                trials = 0
            visitedCities.append(saveState[cityIndex])
```

#6(genetic algorithm):

```
import numpy as np
import pandas as pd


class CustomGeneticAlgorithm():

    def server_present(self, server, time):
        server_start_time = server[1]
        server_duration = server[2]
        server_end_time = server_start_time + server_duration
        if (time >= server_start_time) and (time < 
server_end_time):
            return True
        return False

    def deployed_to_hourlyplanning(self, 
deployed_hourly_cron_capacity):

        deployed_hourly_cron_capacity_week = []
        for day in deployed_hourly_cron_capacity:

            deployed_hourly_cron_capacity_day = []
            for server in day:

                server_present_hour = []
                for time in range(0, 24):

                    server_present_hour.append(
                        self.server_present(server, time))


deployed_hourly_cron_capacity_day.append(server_present_hour)

            deployed_hourly_cron_capacity_week.append(
                deployed_hourly_cron_capacity_day)

        deployed_hourly_cron_capacity_week = np.array(
            deployed_hourly_cron_capacity_week).sum(axis=1)
        return deployed_hourly_cron_capacity_week

    def generate_random_plan(self, n_days, n_racks):
        period_planning = []
        for _ in range(n_days):
            day_planning = []
            for server_id in range(n_racks):
                start_time = np.random.randint(0, 23)
```

```
                machines = np.random.randint(0, 12)
                server = [server_id, start_time, machines]
                day_planning.append(server)

            period_planning.append(day_planning)

        return period_planning

    def generate_initial_population(self, population_size,
n_days=7, n_racks=11):
        population = []
        for _ in range(population_size):
            member = self.generate_random_plan(
                n_days=n_days, n_racks=n_racks)
            population.append(member)
        return population

    def calculate_fitness(self, deployed_hourly_cron_capacity,
required_hourly_cron_capacity):
        deviation = deployed_hourly_cron_capacity -
required_hourly_cron_capacity
        overcapacity = abs(deviation[deviation > 0].sum())
        undercapacity = abs(deviation[deviation < 0].sum())

        overcapacity_cost = 0.5
        undercapacity_cost = 3

        fitness = overcapacity_cost * overcapacity +
undercapacity_cost * undercapacity
        return fitness

    def crossover(self, population, n_offspring):
        n_population = len(population)

        offspring = []

        for _ in range(n_offspring):
            random_one = population[np.random.randint(
                low=0, high=n_population - 1)]
            random_two = population[np.random.randint(
                low=0, high=n_population - 1)]

            dad_mask = np.random.randint(0, 2,
size=np.array(random_one).shape)
            mom_mask = np.logical_not(dad_mask)

            child = np.add(np.multiply(random_one, dad_mask),
                           np.multiply(random_two, mom_mask))

            offspring.append(child)
        return offspring
```

```
    def mutate_parent(self, parent, n_mutations):
        size1 = parent.shape[0]
        size2 = parent.shape[1]

        for _ in range(n_mutations):
            rand1 = np.random.randint(0, size1)
            rand2 = np.random.randint(0, size2)
            rand3 = np.random.randint(0, 2)
            parent[rand1, rand2, rand3] = np.random.randint(0, 12)
        return parent

    def mutate_gen(self, population, n_mutations):
        mutated_population = []
        for parent in population:
            mutated_population.append(self.mutate_parent(parent, 
n_mutations))
        return mutated_population

    def is_acceptable(self, parent):
        return np.logical_not((np.array(parent)[:, :, 2:] > 
12).any())

    def select_acceptable(self, population):
        population = [
            parent for parent in population if 
self.is_acceptable(parent)]
        return population

    def select_best(self, population, 
required_hourly_cron_capacity, n_best):
        fitness = []
        for idx, deployed_hourly_cron_capacity in 
enumerate(population):

            deployed_hourly_cron_capacity = 
self.deployed_to_hourlyplanning(
                deployed_hourly_cron_capacity)
            parent_fitness = 
self.calculate_fitness(deployed_hourly_cron_capacity,

required_hourly_cron_capacity)
            fitness.append([idx, parent_fitness])

        print('Current generation\'s optimal schedule has cost: 
{}'.format(
            pd.DataFrame(fitness)[1].min()))

        fitness_tmp = pd.DataFrame(fitness).sort_values(
            by=1, ascending=True).reset_index(drop=True)
        selected_parents_idx = list(fitness_tmp.iloc[:n_best, 0])
        selected_parents = [parent for idx, parent in enumerate(
            population) if idx in selected_parents_idx]
```

```
        return selected_parents

    def run(self, required_hourly_cron_capacity, n_iterations, 
n_population_size=500):

        population = self.generate_initial_population(
            population_size=n_population_size, n_days=5, 
n_racks=24)
        for _ in range(n_iterations):
            population = self.select_acceptable(population)
            population = self.select_best(
                population, required_hourly_cron_capacity, 
n_best=100)
            population = self.crossover(
                population, n_offspring=n_population_size)
            population = self.mutate_gen(population, 
n_mutations=1)

        best_child = self.select_best(
            population, required_hourly_cron_capacity, n_best=1)
        return best_child


def main():

    # Reading from the data file
    df = pd.read_csv("./data/cron_jobs_schedule.csv")

    dataset = df.astype(int).values.tolist()

    required_hourly_cron_capacity = [
        [0 for _ in range(24)] for _ in range(5)]

    for record in dataset:
        required_hourly_cron_capacity[record[1]][record[2]] += 
record[3]

    genetic_algorithm = CustomGeneticAlgorithm()
    optimal_schedule = genetic_algorithm.run(
        required_hourly_cron_capacity, n_iterations=100)
    print('\nOptimal Server Schedule: \n', 

genetic_algorithm.deployed_to_hourlyplanning(optimal_schedule[0]))


if __name__ == "__main__":
    main()
```

#7 KNN:


```
{
 "cells": [
  {
   "cell_type": "code",
   "execution_count": 85,
   "metadata": {},
   "outputs": [],
   "source": [
    "import math\n",
    "\n",
    "def KNN():\n",
    "    s =[\n",
    "        [7,7],\n",
    "        [7,4],\n",
    "        [3,4],\n",
    "        [1,4]\n",
    "\n",
    "        ]\n",
    "    K=3\n",
    "    x1 = 3\n",
    "    y1 = 7\n",
    "    a = len(s)\n",
    "    c=0\n",
    "    for i in range(a):\n",
    "        d=0\n",
    "        x2 = s[c][d]\n",
    "\n",
    "        d = d+1\n",
    "        y2 = s[c][d]\n",
    "        D1 = (x2-x1)**2\n",
    "\n",
    "        D2 = (y2-y1)**2\n",
    "\n",
    "        Distance = D1+D2\n",
    "        F = s[c]\n",
    "        F.append(Distance)\n",
    "\n",
    "        F.append(int(math.sqrt(Distance))- 1)\n",
    "\n",
    "        d=3\n",
    "        if K >= s[c][d]:\n",
    "            F.append(\"YEs\")\n",
    "        else:\n",
    "            F.append(\"No\")\n",
    "\n",
    "\n",
    "        if K == s[c][d]:\n",
    "            F.append(\"Bad\")\n",
    "        elif K > s[c][d]:\n",
    "            F.append(\"Good\")\n",
```

```
    "        else:\n",
    "            F.append(\"--\")\n",
    "        c = c + 1    \n",
    "    print(s)"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 82,
   "metadata": {},
   "outputs": [
    {
     "name": "stdout",
     "output_type": "stream",
     "text": [
      "[[7, 7, 16, 3, 'YEs', 'Bad'], [7, 4, 25, 4, 'No', '--'], [3, 4, 9, 2, 'YEs', 'Good'], [1, 4, 13, 2, 'YEs', 'Good']]\n"
     ]
    }
   ],
   "source": [
    "KNN()"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": []
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "Python 3",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 3
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython3",
   "version": "3.7.3"
  }
 },
 "nbformat": 4,
```

```
 "nbformat_minor": 2
}
```


#8 KMeans:

```
import numpy as np
import math
import pandas as pd




class K_Means:

    def __init__(self, k =3, tolerance = 0.0001, max_iterations = 
500):
        self.k = k
        self.tolerance = tolerance
        self.max_iterations = max_iterations

df = pd.read_csv(r"/Users/hasanaskary/Downloads/fruitsd.csv")

df = df[['one', 'two']]
dataset = df.astype(float).values.tolist()

X = df.values

def Euclidean_distance(feat_one, feat_two):

    squared_distance = 0



    for i in range(len(feat_one)):

            squared_distance += (feat_one[i] - feat_two[i])**2

    ed = sqrt(squared_distances)

    return ed;

for i in range(self.k):
    self.centroids[i] = data[i]

for i in range(self.max_iterations):
        self.classes = {}
        for i in range(self.k):
            self.classes[i] = []


        for features in data:
            distances = [np.linalg.norm(features - 
self.centroids[centroid]) for centroid in self.centroids]
```

```
            classification = distances.index(min(distances))
            self.classes[classification].append(features)
previous = dict(self.centroids)


for classification in self.classes:
    self.centroids[classification] = 
np.average(self.classes[classification], axis = 0)

isOptimal = True #to check the optimal no of centroids......

for centroid in self.centroids:

    original_centroid = previous[centroid]
    curr = self.centroids[centroid]

    if np.sum((curr - original_centroid)/original_centroid * 
100.0) > self.tolerance:
        isOptimal = False


if isOptimal:
        break
```

#9 Simulated Annealing:

```
# simulated annealing algorithm
def simulated_annealing(objective, bounds, n_iterations, 
step_size, temp):
    # generate an initial point
    best = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - 
bounds[:, 0])
    # evaluate the initial point
    best_eval = objective(best)
    # current working solution
    curr, curr_eval = best, best_eval
    # run the algorithm
    for i in range(n_iterations):
        # take a step
        candidate = curr + randn(len(bounds)) * step_size
        # evaluate candidate point
        candidate_eval = objective(candidate)
        # check for new best solution
        if candidate_eval < best_eval:
            # store new best point
            best, best_eval = candidate, candidate_eval
            # report progress
            print('>%d f(%s) = %.5f' % (i, best, best_eval))
        # difference between candidate and current point 
evaluation
        diff = candidate_eval - curr_eval
        # calculate temperature for current epoch
        t = temp / float(i + 1)
        # calculate metropolis acceptance criterion
        metropolis = exp(-diff / t)
        # check if we should keep the new point
        if diff < 0 or rand() < metropolis:
            # store the new current point
            curr, curr_eval = candidate, candidate_eval
    return [best, best_eval]
```