Water Jug

from collections import deque

def water\_jug\_BFS(x, y, z):

visited = set()

queue = deque([(0, 0)])

while queue:

jug\_a, jug\_b = queue.popleft()

if jug\_a == z or jug\_b == z or jug\_a + jug\_b == z:

return True

if (jug\_a, jug\_b) in visited:

continue

visited.add((jug\_a, jug\_b))

# Fill jug A

if jug\_a < x:

queue.append((x, jug\_b))

# Fill jug B

if jug\_b < y:

queue.append((jug\_a, y))

# Empty jug A

if jug\_a > 0:

queue.append((0, jug\_b))

# Empty jug B

if jug\_b > 0:

queue.append((jug\_a, 0))

# Pour from A to B

if jug\_a + jug\_b >= y:

queue.append((jug\_a - (y - jug\_b), y))

else:

queue.append((0, jug\_a + jug\_b))

# Pour from B to A

if jug\_a + jug\_b >= x:

queue.append((x, jug\_b - (x - jug\_a)))

else:

queue.append((jug\_a + jug\_b, 0))

return False

x = 2

y = 1

z = 4

if water\_jug\_BFS(x, y, z):

print(f'You can measure {z} liters of water using {x}-liter and {y}-liter jugs.')

else:

print(f'You cannot measure {z} liters of water using {x}-liter and {y}-liter jugs.')

=================================Travelling saleman

dst=[]

def travel(g, v, pos, n, count, cost):

if(count==n and g[pos][s]):

cost+=g[pos][s]

dst.append(cost)

return

for i in range(0,n):

if(v[i]==False and g[pos][i]):

v[i]=True

travel(g,v,i,n,count+1,cost+g[pos][i])

v[i]=False

n=4

g=[[0, 10, 15, 20],[10, 0, 35, 25],[15, 35, 0, 30],[20, 25, 30, 0]]

s=int(input("Enter a number between 1 and 4: "))

v=[False for i in range(0,n)]

s-=1

v[s]=True

travel(g,v,s,n,1,0)

print(dst)

print(min(dst))

==================================

BFS

import sys

import copy

q = []

visited = []

def compare(s,g):

if s==g:

return(1)

else:

return(0)

def find\_pos(s):

for i in range(3):

for j in range(3):

if s[i][j] == 0:

return([i,j])

def up(s,pos):

i = pos[0]

j = pos[1]

if i > 0:

temp = copy.deepcopy(s)

temp[i][j] = temp[i-1][j]

temp[i-1][j] = 0

return (temp)

else:

return (s)

def down(s,pos):

i = pos[0]

j = pos[1]

if i < 2:

temp = copy.deepcopy(s)

temp[i][j] = temp[i+1][j]

temp[i+1][j] = 0

return (temp)

else:

return (s)

def right(s,pos):

i = pos[0]

j = pos[1]

if j < 2:

temp = copy.deepcopy(s)

temp[i][j] = temp[i][j+1]

temp[i][j+1] = 0

return (temp)

else:

return (s)

def left(s,pos):

i = pos[0]

j = pos[1]

if j > 0:

temp = copy.deepcopy(s)

temp[i][j] = temp[i][j-1]

temp[i][j-1] = 0

return (temp)

else:

return (s)

def enqueue(s,val):

global q

q = q + [(val,s)]

def heuristic(s,g):

d = 0

for i in range(3):

for j in range(3):

if s[i][j] != g[i][j]:

d += 1

return d

def dequeue():

global q

global visited

q.sort()

visited = visited + [q[0][1]]

elem = q[0][1]

del q[0]

return (elem)

def search(s,g):

curr\_state = copy.deepcopy(s)

if s == g:

return

global visited

while(1):

pos = find\_pos(curr\_state)

new = up(curr\_state,pos)

if new != curr\_state:

if new == g:

print ("found!! The intermediate states are:")

print (visited + [g])

return

else:

if new not in visited:

enqueue(new,heuristic(new,g))

new = down(curr\_state,pos)

if new != curr\_state:

if new == g:

print ("found!! The intermediate states are:")

print (visited + [g])

return

else:

if new not in visited:

enqueue(new,heuristic(new,g))

new = right(curr\_state,pos)

if new != curr\_state:

if new == g:

print ("found!! The intermediate states are:")

print (visited + [g])

return

else:

if new not in visited:

enqueue(new,heuristic(new,g))

new = left(curr\_state,pos)

if new != curr\_state:

if new == g:

print ("found!! The intermediate states are:")

print (visited + [g])

return

else:

if new not in visited:

enqueue(new,heuristic(new,g))

if len(q) > 0:

curr\_state = dequeue()

else:

print ("not found")

return

def main():

s = [[2,0,3],[1,8,4],[7,6,5]]

g = [[1,2,3],[8,0,4],[7,6,5]]

global q

global visited

q = q

visited = visited + [s]

search(s,g)

if \_\_name\_\_ == "\_\_main\_\_":

main()

==================================

HILL CLIMB

import sys

import copy

curr\_min = sys.maxsize

q = []

visited = []

def compare(s,g):

if s==g:

return(1)

else:

return(0)

def find\_pos(s):

for i in range(3):

for j in range(3):

if s[i][j] == 0:

return([i,j])

def up(s,pos):

i = pos[0]

j = pos[1]

if i > 0:

temp = copy.deepcopy(s)

temp[i][j] = temp[i-1][j]

temp[i-1][j] = 0

return (temp)

else:

return (s)

def down(s,pos):

i = pos[0]

j = pos[1]

if i < 2:

temp = copy.deepcopy(s)

temp[i][j] = temp[i+1][j]

temp[i+1][j] = 0

return (temp)

else:

return (s)

def right(s,pos):

i = pos[0]

j = pos[1]

if j < 2:

temp = copy.deepcopy(s)

temp[i][j] = temp[i][j+1]

temp[i][j+1] = 0

return (temp)

else:

return (s)

def left(s,pos):

i = pos[0]

j = pos[1]

if j > 0:

temp = copy.deepcopy(s)

temp[i][j] = temp[i][j-1]

temp[i][j-1] = 0

return (temp)

else:

return (s)

def enqueue(s):

global q

q = q + [s]

def heuristic(s,g):

d = 0

for i in range(len(s)):

for j in range(len(s[0])):

if s[i][j] != g[i][j]:

d += 1

return d

def dequeue(g):

h = []

global q

global visited

global curr\_min

for i in range(len(q)):

h = h + [heuristic(q[i],g)]

if min(h) < curr\_min:

curr\_min = min(h)

index = h.index(min(h))

visited = visited + [q[index]]

elem = q[index]

q = []

return (elem)

else:

print ("optimal solution found !! The intermediate states are: ")

print (visited)

exit()

def search(s,g):

curr\_state = copy.deepcopy(s)

if s == g:

return

global visited

while(1):

pos = find\_pos(curr\_state)

new = up(curr\_state,pos)

if new != curr\_state:

if new == g:

print ("Goal State found !! The intermediate States are :")

print (visited + [g])

return

else:

if new not in visited:

enqueue(new)

new = down(curr\_state,pos)

if new != curr\_state:

if new == g:

print ("Goal State found !! The intermediate States are :")

print (visited + [g])

return

else:

if new not in visited:

enqueue(new)

new = right(curr\_state,pos)

if new != curr\_state:

if new == g:

print ("Goal State found !! The intermediate States are :")

print (visited + [g])

return

else:

if new not in visited:

enqueue(new)

new = left(curr\_state,pos)

if new != curr\_state:

if new == g:

print ("Goal State found !! The intermediate States are :")

print (visited + [g])

return

else:

if new not in visited:

enqueue(new)

if len(q) > 0:

curr\_state = dequeue(g)

else:

print ("not found")

return

def main():

s = [[2,8,3],[1,5,4],[7,6,0]]

g = [[1,2,7],[8,0,5],[3,4,6]]

global q

global visited

q = q + [s]

visited = visited + [s]

search(s,g)

if \_\_name\_\_ == "\_\_main\_\_":

main()

==================================

DECISSION TREE

import pandas as pd

import numpy as np

from sklearn.datasets import load\_iris

#convert to a dataframe

df = pd.DataFrame(data.data, columns = data.feature\_names)

df.head()

#create the species column

df['Species'] = data.target

df.head()

#replace this with the actual names

target = np.unique(data.target)

target\_names = np.unique(data.target\_names)

targets = dict(zip(target, target\_names))

df['Species'] = df['Species'].replace(targets)

x = df.drop(columns="Species")

y = df["Species"]

feature\_names = x.columns

labels = y.unique()

#split the dataset

from sklearn.model\_selection import train\_test\_split

X\_train, test\_x, y\_train, test\_lab = train\_test\_split(x,y, test\_size = 0.4,random\_state = 42)

from sklearn.tree import DecisionTreeClassifier

clf = DecisionTreeClassifier(criterion='entropy',max\_depth =3, random\_state = 42)

clf.fit(X\_train, y\_train)

from sklearn import tree

==

As a Tree Diagram

import matplotlib.pyplot as plt

plt.figure(figsize=(30,10), facecolor ='k')

a = tree.plot\_tree(clf,feature\_names = feature\_names, class\_names = labels, rounded = True, filled = True,fontsize=14)

plt.show()

#2. As a Text-Based Diagram

from sklearn.tree import export\_text

tree\_rules = export\_text(clf,feature\_names = list(feature\_names))

print(tree\_rules)

# Predict Class From Test Values

test\_pred\_decision\_tree = clf.predict(test\_x)

from sklearn import metrics

import seaborn as sns

import matplotlib.pyplot as plt

confusion\_matrix = metrics.confusion\_matrix(test\_lab, test\_pred\_decision\_tree)

matrix\_df = pd.DataFrame(confusion\_matrix)

ax = plt.axes()

sns.set(font\_scale=1.3)

plt.figure(figsize=(10,7))

sns.heatmap(matrix\_df, annot=True, fmt="g", ax=ax, cmap="magma")

ax.set\_title('Confusion Matrix - Decision Tree')

ax.set\_xlabel("Predicted label", fontsize =15)

ax.set\_xticklabels(['']+labels)

ax.set\_ylabel("True Label", fontsize=15)

ax.set\_yticklabels(list(labels), rotation = 0)

plt.show()

=====================================

KNN

Setup

import numpy as np

from sklearn import datasets

from sklearn import neighbors

import pylab as pl

import matplotlib.pyplot as plt

from matplotlib.colors import ListedColormap

iris = datasets.load\_iris()

print(iris.keys())

n\_samples, n\_features = iris.data.shape

print((n\_samples, n\_features))

print(iris.data[0])

print(iris.target.shape)

print(iris.target)

print(iris.target\_names)

x\_index = 0

y\_index = 1

# this formatter will label the colorbar with the correct target

formatter = plt.FuncFormatter(lambda i, \*args: iris.target\_names[int(i)])

plt.scatter(iris.data[:, x\_index], iris.data[:, y\_index],

c=iris.target, cmap=plt.cm.get\_cmap('RdYlBu', 3))

plt.colorbar(ticks=[0, 1, 2], format=formatter)

plt.clim(-0.5, 2.5)

plt.xlabel(iris.feature\_names[x\_index])

plt.ylabel(iris.feature\_names[y\_index]);

X, y = iris.data, iris.target

clf = neighbors.KNeighborsClassifier(n\_neighbors=5)

clf.fit(X, y)

result = clf.predict([[3, 5, 4, 2],])

print(iris.target\_names[result])

# Create color maps for 3-class classification problem, as with iris

cmap\_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])

cmap\_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

def plot\_iris\_knn():

iris = datasets.load\_iris()

X = iris.data[:, :2] # we only take the first two features.

y = iris.target

knn = neighbors.KNeighborsClassifier(n\_neighbors=3)

knn.fit(X, y)

x\_min, x\_max = X[:, 0].min() - .1, X[:, 0].max() + .1

y\_min, y\_max = X[:, 1].min() - .1, X[:, 1].max() + .1

xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, 100),

np.linspace(y\_min, y\_max, 100))

Z = knn.predict(np.c\_[xx.ravel(), yy.ravel()])

# Put the result into a color plot

Z = Z.reshape(xx.shape)

pl.figure()

pl.pcolormesh(xx, yy, Z, cmap=cmap\_light)

# Plot also the training point

pl.scatter(X[:, 0], X[:, 1], c=y,cmap=cmap\_bold)

pl.xlabel('sepal length (cm)')

pl.ylabel('sepal width (cm)')

pl.axis('tight')

plot\_iris\_knn()

=============================

NAÏVE BAYES

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.naive\_bayes import GaussianNB

X, y = load\_iris(return\_X\_y=True)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.5, random\_state=0)

gnb = GaussianNB()

y\_pred = gnb.fit(X\_train, y\_train).predict(X\_test)

print("Number of mislabeled points out of a total points :", ((y\_test != y\_pred).sum(), X\_test.shape[0]))

==================================

BEST F S

from queue import PriorityQueue

v = 14

graph = [[] for i in range(v)]

# Function For Implementing Best First Search

# Gives output path having lowest cost

def best\_first\_search(actual\_Src, target, n):

visited = [False] \* n

pq = PriorityQueue()

pq.put((0, actual\_Src))

visited[actual\_Src] = True

while pq.empty() == False:

u = pq.get()[1]

# Displaying the path having lowest cost

print(u, end=" ")

if u == target:

break

for v, c in graph[u]:

if visited[v] == False:

visited[v] = True

pq.put((c, v))

print()

# Function for adding edges to graph

def addedge(x, y, cost):

graph[x].append((y, cost))

graph[y].append((x, cost))

# The nodes shown in above example(by alphabets) are

# implemented using integers addedge(x,y,cost);

addedge(0, 1, 3)

addedge(0, 2, 6)

addedge(0, 3, 5)

addedge(1, 4, 9)

addedge(1, 5, 8)

addedge(2, 6, 12)

addedge(2, 7, 14)

addedge(3, 8, 7)

addedge(8, 9, 5)

addedge(8, 10, 6)

addedge(9, 11, 1)

addedge(9, 12, 10)

addedge(9, 13, 2)

source = 0

target = 9

best\_first\_search(source, target, v)

==================================

A\*

# Python program for A\* Search Algorithm

import math

import heapq

# Define the Cell class

class Cell:

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

# Parent cell's row index

self.parent\_i = 0

# Parent cell's column index

self.parent\_j = 0

# Total cost of the cell (g + h)

self.f = float('inf')

# Cost from start to this cell

self.g = float('inf')

# Heuristic cost from this cell to destination

self.h = 0

# Define the size of the grid

ROW = 9

COL = 10

# Check if a cell is valid (within the grid)

def is\_valid(row, col):

return (row >= 0) and (row < ROW) and (col >= 0) and (col < COL)

# Check if a cell is unblocked

def is\_unblocked(grid, row, col):

return grid[row][col] == 1

# Check if a cell is the destination

def is\_destination(row, col, dest):

return row == dest[0] and col == dest[1]

# Calculate the heuristic value of a cell (Euclidean distance to destination)

def calculate\_h\_value(row, col, dest):

return ((row - dest[0]) \*\* 2 + (col - dest[1]) \*\* 2) \*\* 0.5

# Trace the path from source to destination

def trace\_path(cell\_details, dest):

print("The Path is ")

path = []

row = dest[0]

col = dest[1]

# Trace the path from destination to source using parent cells

while not (cell\_details[row][col].parent\_i == row and cell\_details[row][col].parent\_j == col):

path.append((row, col))

temp\_row = cell\_details[row][col].parent\_i

temp\_col = cell\_details[row][col].parent\_j

row = temp\_row

col = temp\_col

# Add the source cell to the path

path.append((row, col))

# Reverse the path to get the path from source to destination

path.reverse()

# Print the path

for i in path:

print("->", i, end=" ")

print()

# Implement the A\* search algorithm

def a\_star\_search(grid, src, dest):

# Check if the source and destination are valid

if not is\_valid(src[0], src[1]) or not is\_valid(dest[0], dest[1]):

print("Source or destination is invalid")

return

# Check if the source and destination are unblocked

if not is\_unblocked(grid, src[0], src[1]) or not is\_unblocked(grid, dest[0], dest[1]):

print("Source or the destination is blocked")

return

# Check if we are already at the destination

if is\_destination(src[0], src[1], dest):

print("We are already at the destination")

return

# Initialize the closed list (visited cells)

closed\_list = [[False for \_ in range(COL)] for \_ in range(ROW)]

# Initialize the details of each cell

cell\_details = [[Cell() for \_ in range(COL)] for \_ in range(ROW)]

# Initialize the start cell details

i = src[0]

j = src[1]

cell\_details[i][j].f = 0

cell\_details[i][j].g = 0

cell\_details[i][j].h = 0

cell\_details[i][j].parent\_i = i

cell\_details[i][j].parent\_j = j

# Initialize the open list (cells to be visited) with the start cell

open\_list = []

heapq.heappush(open\_list, (0.0, i, j))

# Initialize the flag for whether destination is found

found\_dest = False

# Main loop of A\* search algorithm

while len(open\_list) > 0:

# Pop the cell with the smallest f value from the open list

p = heapq.heappop(open\_list)

# Mark the cell as visited

i = p[1]

j = p[2]

closed\_list[i][j] = True

# For each direction, check the successors

directions = [(0, 1), (0, -1), (1, 0), (-1, 0),

(1, 1), (1, -1), (-1, 1), (-1, -1)]

for dir in directions:

new\_i = i + dir[0]

new\_j = j + dir[1]

# If the successor is valid, unblocked, and not visited

if is\_valid(new\_i, new\_j) and is\_unblocked(grid, new\_i, new\_j) and not closed\_list[new\_i][new\_j]:

# If the successor is the destination

if is\_destination(new\_i, new\_j, dest):

# Set the parent of the destination cell

cell\_details[new\_i][new\_j].parent\_i = i

cell\_details[new\_i][new\_j].parent\_j = j

print("The destination cell is found")

# Trace and print the path from source to destination

trace\_path(cell\_details, dest)

found\_dest = True

return

else:

# Calculate the new f, g, and h values

g\_new = cell\_details[i][j].g + 1.0

h\_new = calculate\_h\_value(new\_i, new\_j, dest)

f\_new = g\_new + h\_new

# If the cell is not in the open list or the new f value is smaller

if cell\_details[new\_i][new\_j].f == float('inf') or cell\_details[new\_i][new\_j].f > f\_new:

# Add the cell to the open list

heapq.heappush(open\_list, (f\_new, new\_i, new\_j))

# Update the cell details

cell\_details[new\_i][new\_j].f = f\_new

cell\_details[new\_i][new\_j].g = g\_new

cell\_details[new\_i][new\_j].h = h\_new

cell\_details[new\_i][new\_j].parent\_i = i

cell\_details[new\_i][new\_j].parent\_j = j

# If the destination is not found after visiting all cells

if not found\_dest:

print("Failed to find the destination cell")

# Driver Code

def main():

# Define the grid (1 for unblocked, 0 for blocked)

grid = [

[1, 0, 1, 1, 1, 1, 0, 1, 1, 1],

[1, 1, 1, 0, 1, 1, 1, 0, 1, 1],

[1, 1, 1, 0, 1, 1, 0, 1, 0, 1],

[0, 0, 1, 0, 1, 0, 0, 0, 0, 1],

[1, 1, 1, 0, 1, 1, 1, 0, 1, 0],

[1, 0, 1, 1, 1, 1, 0, 1, 0, 0],

[1, 0, 0, 0, 0, 1, 0, 0, 0, 1],

[1, 0, 1, 1, 1, 1, 0, 1, 1, 1],

[1, 1, 1, 0, 0, 0, 1, 0, 0, 1]

]

# Define the source and destination

src = [8, 0]

dest = [0, 0]

# Run the A\* search algorithm

a\_star\_search(grid, src, dest)

if \_\_name\_\_ == "\_\_main\_\_":

main()

==================================

Ao \*

# Cost to find the AND and OR path

def Cost(H, condition, weight = 1):

cost = {}

if 'AND' in condition:

AND\_nodes = condition['AND']

Path\_A = ' AND '.join(AND\_nodes)

PathA = sum(H[node]+weight for node in AND\_nodes)

cost[Path\_A] = PathA

if 'OR' in condition:

OR\_nodes = condition['OR']

Path\_B =' OR '.join(OR\_nodes)

PathB = min(H[node]+weight for node in OR\_nodes)

cost[Path\_B] = PathB

return cost

# Update the cost

def update\_cost(H, Conditions, weight=1):

Main\_nodes = list(Conditions.keys())

Main\_nodes.reverse()

least\_cost= {}

for key in Main\_nodes:

condition = Conditions[key]

print(key,':', Conditions[key],'>>>', Cost(H, condition, weight))

c = Cost(H, condition, weight)

H[key] = min(c.values())

least\_cost[key] = Cost(H, condition, weight)

return least\_cost

# Print the shortest path

def shortest\_path(Start,Updated\_cost, H):

Path = Start

if Start in Updated\_cost.keys():

Min\_cost = min(Updated\_cost[Start].values())

key = list(Updated\_cost[Start].keys())

values = list(Updated\_cost[Start].values())

Index = values.index(Min\_cost)

# FIND MINIMIMUM PATH KEY

Next = key[Index].split()

# ADD TO PATH FOR OR PATH

if len(Next) == 1:

Start =Next[0]

Path += ' = ' +shortest\_path(Start, Updated\_cost, H)

# ADD TO PATH FOR AND PATH

else:

Path +='=('+key[Index]+') '

Start = Next[0]

Path += '[' +shortest\_path(Start, Updated\_cost, H) + ' + '

Start = Next[-1]

Path += shortest\_path(Start, Updated\_cost, H) + ']'

return Path

# Heuristic values of Nodes

H1 = {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T': 3}

Conditions = {

'A': {'OR': ['D'], 'AND': ['B', 'C']},

'B': {'OR': ['G', 'H']},

'C': {'OR': ['J']},

'D': {'AND': ['E', 'F']},

'G': {'OR': ['I']}

}

# weight

weight = 1

# Updated cost

print('Updated Cost :')

Updated\_cost = update\_cost(H1, Conditions, weight=1)

print('\*'\*75)

print('Shortest Path :\n',shortest\_path('A', Updated\_cost,H1))

==================================

UCS

# Python3 implementation of above approach

# returns the minimum cost in a vector( if

# there are multiple goal states)

def uniform\_cost\_search(goal, start):

# minimum cost upto

# goal state from starting

global graph,cost

answer = []

# create a priority queue

queue = []

# set the answer vector to max value

for i in range(len(goal)):

answer.append(10\*\*8)

# insert the starting index

queue.append([0, start])

# map to store visited node

visited = {}

# count

count = 0

# while the queue is not empty

while (len(queue) > 0):

# get the top element of the

queue = sorted(queue)

p = queue[-1]

# pop the element

del queue[-1]

# get the original value

p[0] \*= -1

# check if the element is part of

# the goal list

if (p[1] in goal):

# get the position

index = goal.index(p[1])

# if a new goal is reached

if (answer[index] == 10\*\*8):

count += 1

# if the cost is less

if (answer[index] > p[0]):

answer[index] = p[0]

# pop the element

del queue[-1]

queue = sorted(queue)

if (count == len(goal)):

return answer

# check for the non visited nodes

# which are adjacent to present node

if (p[1] not in visited):

for i in range(len(graph[p[1]])):

# value is multiplied by -1 so that

# least priority is at the top

queue.append( [(p[0] + cost[(p[1], graph[p[1]][i])])\* -1, graph[p[1]][i]])

# mark as visited

visited[p[1]] = 1

return answer

# main function

if \_\_name\_\_ == '\_\_main\_\_':

# create the graph

graph,cost = [[] for i in range(8)],{}

# add edge

graph[0].append(1)

graph[0].append(3)

graph[3].append(1)

graph[3].append(6)

graph[3].append(4)

graph[1].append(6)

graph[4].append(2)

graph[4].append(5)

graph[2].append(1)

graph[5].append(2)

graph[5].append(6)

graph[6].append(4)

# add the cost

cost[(0, 1)] = 2

cost[(0, 3)] = 5

cost[(1, 6)] = 1

cost[(3, 1)] = 5

cost[(3, 6)] = 6

cost[(3, 4)] = 2

cost[(2, 1)] = 4

cost[(4, 2)] = 4

cost[(4, 5)] = 3

cost[(5, 2)] = 6

cost[(5, 6)] = 3

cost[(6, 4)] = 7

# goal state

goal = []

# set the goal

# there can be multiple goal states

goal.append(6)

# get the answer

answer = uniform\_cost\_search(goal, 0)

# print the answer

print("Minimum cost from 0 to 6 is = ",answer[0])

==================================

Alpha beat pruning

class MiniMax:

# print utility value of root node (assuming it is max)

# print names of all nodes visited during search

def \_\_init\_\_(self, game\_tree):

self.game\_tree = game\_tree # GameTree

self.root = game\_tree.root # GameNode

self.currentNode = None # GameNode

self.successors = [] # List of GameNodes

return

def minimax(self, node):

# first, find the max value

best\_val = self.max\_value(node) # should be root node of tree

# second, find the node which HAS that max value

# –> means we need to propagate the values back up the

# tree as part of our minimax algorithm

successors = self.getSuccessors(node)

print "MiniMax: Utility Value of Root Node: = " + str(best\_val)

# find the node with our best move

best\_move = None

for elem in successors: # —> Need to propagate values up tree for this to work

if elem.value == best\_val:

best\_move = elem

break

# return that best value that we've found

return best\_move

def max\_value(self, node):

print "MiniMax–>MAX: Visited Node :: " + node.Name

if self.isTerminal(node):

return self.getUtility(node)

infinity = float('inf')

max\_value = -infinity

successors\_states = self.getSuccessors(node)

for state in successors\_states:

max\_value = max(max\_value, self.min\_value(state))

return max\_value

def min\_value(self, node):

print "MiniMax–>MIN: Visited Node :: " + node.Name

if self.isTerminal(node):

return self.getUtility(node)

infinity = float('inf')

min\_value = infinity

successor\_states = self.getSuccessors(node)

for state in successor\_states:

min\_value = min(min\_value, self.max\_value(state))

return min\_value

# UTILITY METHODS #

# successor states in a game tree are the child nodes…

def getSuccessors(self, node):

assert node is not None

return node.children

# return true if the node has NO children (successor states)

# return false if the node has children (successor states)

def isTerminal(self, node):

assert node is not None

return len(node.children) == 0

def getUtility(self, node):

assert node is not None

return node.value

class AlphaBeta:

# print utility value of root node (assuming it is max)

# print names of all nodes visited during search

def \_\_init\_\_(self, game\_tree):

self.game\_tree = game\_tree # GameTree

self.root = game\_tree.root # GameNode

return

def alpha\_beta\_search(self, node):

infinity = float('inf')

best\_val = -infinity

beta = infinity

successors = self.getSuccessors(node)

best\_state = None

for state in successors:

value = self.min\_value(state, best\_val, beta)

if value > best\_val:

best\_val = value

best\_state = state

print "AlphaBeta: Utility Value of Root Node: = " + str(best\_val)

print "AlphaBeta: Best State is: " + best\_state.Name

return best\_state

def max\_value(self, node, alpha, beta):

print "AlphaBeta–>MAX: Visited Node :: " + node.Name

if self.isTerminal(node):

return self.getUtility(node)

infinity = float('inf')

value = -infinity

successors = self.getSuccessors(node)

for state in successors:

value = max(value, self.min\_value(state, alpha, beta))

if value >= beta:

return value

alpha = max(alpha, value)

return value

def min\_value(self, node, alpha, beta):

print "AlphaBeta–>MIN: Visited Node :: " + node.Name

if self.isTerminal(node):

return self.getUtility(node)

infinity = float('inf')

value = infinity

successors = self.getSuccessors(node)

for state in successors:

value = min(value, self.max\_value(state, alpha, beta))

if value <= alpha:

return value

beta = min(beta, value)

return value

# #

# UTILITY METHODS #

# #

# successor states in a game tree are the child nodes…

def getSuccessors(self, node):

assert node is not None

return node.children

# return true if the node has NO children (successor states)

# return false if the node has children (successor states)

def isTerminal(self, node):

assert node is not None

return len(node.children) == 0

def getUtility(self, node):

assert node is not None

return node.value

==================================

DFS

# Using a Python dictionary to act as an adjacency list

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node): #function for dfs

if node not in visited:

print (node)

visited.add(node)

for neighbour in graph[node]:

dfs(visited, graph, neighbour)

# Driver Code

print("Following is the Depth-First Search")

dfs(visited, graph, '5')

================================

DFS-ID

# Python program to print DFS traversal from a given

# given graph

from collections import defaultdict

# This class represents a directed graph using adjacency

# list representation

class Graph:

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

# No. of vertices

self.V = vertices

# default dictionary to store graph

self.graph = defaultdict(list)

# function to add an edge to graph

def addEdge(self,u,v):

self.graph[u].append(v)

# A function to perform a Depth-Limited search

# from given source 'src'

def DLS(self,src,target,maxDepth):

if src == target : return True

# If reached the maximum depth, stop recursing.

if maxDepth <= 0 : return False

# Recur for all the vertices adjacent to this vertex

for i in self.graph[src]:

if(self.DLS(i,target,maxDepth-1)):

return True

return False

# IDDFS to search if target is reachable from v.

# It uses recursive DLS()

def IDDFS(self,src, target, maxDepth):

# Repeatedly depth-limit search till the

# maximum depth

for i in range(maxDepth):

if (self.DLS(src, target, i)):

return True

return False

# Create a graph given in the above diagram

g = Graph (7);

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 3)

g.addEdge(1, 4)

g.addEdge(2, 5)

g.addEdge(2, 6)

target = 6; maxDepth = 3; src = 0

if g.IDDFS(src, target, maxDepth) == True:

print ("Target is reachable from source " +

"within max depth")

else :

print ("Target is NOT reachable from source " +

"within max depth")

Steepest hill climbing

import random

# Define your objective function here, you can replace this with your own function

def objective\_function(x):

return -(x \*\* 2) # Example function to minimize x^2

# Steepest Hill Climb Algorithm

def steepest\_hill\_climb(max\_iterations, step\_size, initial\_solution):

current\_solution = initial\_solution

current\_value = objective\_function(current\_solution)

for \_ in range(max\_iterations):

neighbor\_solutions = [current\_solution + step\_size, current\_solution - step\_size]

neighbor\_values = [objective\_function(neighbor) for neighbor in neighbor\_solutions]

best\_neighbor\_value = max(neighbor\_values)

if best\_neighbor\_value <= current\_value:

break

best\_neighbor\_index = neighbor\_values.index(best\_neighbor\_value)

current\_solution = neighbor\_solutions[best\_neighbor\_index]

current\_value = best\_neighbor\_value

return current\_solution, current\_value

# Example usage

max\_iterations = 1000

step\_size = 0.1

initial\_solution = random.uniform(-10, 10)

best\_solution, best\_value = steepest\_hill\_climb(max\_iterations, step\_size, initial\_solution)

print("Best Solution:", best\_solution)

print("Objective Value at Best Solution:", best\_value)=========

Dfs id

# Depth-First Search (DFS) with Iterative Deepening (ID) in Python

# Graph represented as an adjacency list

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

# Depth-First Search (DFS) function

def dfs(graph, start, goal, max\_depth):

visited = set()

stack = [(start, 0)]

while stack:

node, depth = stack.pop()

visited.add(node)

if node == goal:

return True

if depth < max\_depth:

for neighbor in graph[node]:

if neighbor not in visited:

stack.append((neighbor, depth + 1))

return False

# Iterative Deepening (ID) function

def iterative\_deepening\_dfs(graph, start, goal):

max\_depth = 0

while True:

if dfs(graph, start, goal, max\_depth):

return True

max\_depth += 1

return False

# Example usage

start\_node = 'A'

goal\_node = 'F'

if iterative\_deepening\_dfs(graph, start\_node, goal\_node):

print("Goal node", goal\_node, "found from node", start\_node)

else:

print("Goal node", goal\_node, "not found from node", start\_node)