AI Assignment 4

-----------------------------

Name: Yash Oswal

Div: B Roll no.: 38

SRN: 201901226

-----------------------------

1. Best First Search

CODE:

import copy

from math import nextafter

from colorama import Fore,Back,Style,init

init(autoreset=True)

class Puzzle():

board = [

[1,2,3],

[0,4,6],

[7,5,8]

]

goal = [

[1,2,3],

[4,5,6],

[7,8,0]

]

startX = 0

startY = 0

queue = []

generatedBoards = []

def calcHeuristic(self,board):

h = 0

for i in range(3):

for j in range(3):

if board[i][j] != self.goal[i][j]:

h = h+1

return h-1

def getValidMoves(self,board):

for i in range(3):

for j in range(3):

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

self.startX = j

self.startY = i

position = [0]\*4

validMoves = []

position[0] = [self.startX+1,self.startY]

position[1] = [self.startX-1,self.startY]

position[2] = [self.startX,self.startY-1]

position[3] = [self.startX,self.startY+1]

for i in range(4):

if position[i][1]>-1 and position[i][1]<3 and position[i][0]>-1 and position[i][0]<3:

validMoves.append(position[i])

return validMoves

def playMove(self, move:list, board:list):

newBoard = copy.deepcopy(board)

temp = newBoard[move[1]][move[0]]

newBoard[move[1]][move[0]] = newBoard[self.startY][self.startX]

newBoard[self.startY][self.startX] = temp

return newBoard

def bestFirstSearch(self):

self.calcHeuristic(self.board)

self.queue.append((self.calcHeuristic(self.board), self.board))

self.generatedBoards.append(self.board)

i = 0

while(1<1000):

next = self.queue.pop()

moves = self.getValidMoves(next[1])

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

print(f" step {i}\n")

for j in range(3):

print(" ",next[1][j])

if next[1] == self.goal:

print(f"\nGoal state reached in {i} steps")

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

exit(1)

for move in moves:

newBoard = self.playMove(move,next[1])

if newBoard not in self.generatedBoards:

self.generatedBoards.append(newBoard)

self.queue.append((self.calcHeuristic(newBoard), newBoard))

self.queue.sort(reverse=True)

i+=1

return None

class Robot():

table = [

['-','-','-','-','-','-','-','-','-','-','-'],

['-',Fore.YELLOW+'#','-','-','-',Fore.YELLOW+'#',Fore.YELLOW+'#',Fore.YELLOW+'#',Fore.YELLOW+'#',Fore.YELLOW+'#','-'],

['-',Fore.YELLOW+'#',Fore.YELLOW+'#','-','-','-','-','-','-',Fore.YELLOW+'#','-'],

['-','-',Fore.YELLOW+'#',Fore.YELLOW+'#',Fore.YELLOW+'#',Fore.YELLOW+'#',Fore.YELLOW+'#',Fore.YELLOW+'#',Fore.YELLOW+'#',Fore.YELLOW+'#','-'],

['-','-','-','-','-','-','-','-','-','-','-']

]

goalX = 6

goalY = 2

startX = 0

startY = 3

newTable = copy.deepcopy(table)

queue = []

visited = []

def calcManhatten(self):

self.table[self.startY][self.startX] = Fore.BLUE+'S'

self.table[self.goalY][self.goalX] = Fore.RED+'G'

print("\n Manhatten Distance: \n")

for i in range(5):

for j in range(11):

if self.table[i][j]!=Fore.YELLOW+'#':

self.newTable[i][j] = abs(self.goalX - j) + abs(self.goalY-i)

print('\t',self.newTable[i][j], end='')

print('\n')

position = [self.startX,self.startY]

self.queue.append((self.newTable[self.startY][self.startX],position))

def getNeighbors(self):

position = [0]\*4

value = [0]\*4

position[0] = [self.startX+1,self.startY]

position[1] = [self.startX-1,self.startY]

position[2] = [self.startX,self.startY-1]

position[3] = [self.startX,self.startY+1]

for i in range(4):

if position[i][1]>-1 and position[i][1]<5 and position[i][0]>-1 and position[i][0]<11:

value[i] = self.newTable[position[i][1]][position[i][0]]

if value[i] != Fore.YELLOW+'#' and ((value[i], position[i]) not in self.visited) :

self.queue.append((value[i], position[i]))

self.queue.sort(reverse=True)

def bestFirstSearch(self):

steps = 0

while (self.queue) :

input()

print(f"Steps taken: {steps}")

print(f"Queue: {self.queue}")

next = self.queue.pop()

print(f"Selecting: {next}")

print(f"Current queue: {self.queue}")

if next[1][0] == self.goalX and next[1][1] == self.goalY :

print(f"Goal State reached in {steps} steps")

exit(1)

if next[1] == [self.startX,self.startY]:

self.table[next[1][1]][next[1][0]] = Fore.BLUE+'S'

else:

self.table[next[1][1]][next[1][0]] = Fore.GREEN+str(next[0])

self.visited.append(next)

self.startX = next[1][0]

self.startY = next[1][1]

self.getNeighbors()

print(f"Adding neighbours of {next} to queue\nCurrent queue: {self.queue}\n")

self.printTable(self.table)

print('\t---------------------------------------------------------------------------------')

steps+=1

# print(self.queue)

def printTable(self,table):

for i in range(5):

for j in range(11):

print("\t"+Fore.WHITE+table[i][j], end='')

# print(self.table[i][j], end='|')

print('\n')

class cityDistance():

cityMap = {

'Delhi' : [(800, 'Indore'),(1300, 'Kolkata')],

'Indore': [(600, 'Mumbai'),(500, 'Nagpur'),(800,'Delhi')],

'Kolkata': [(1200,'Nagpur'),(1500,'Hyderabad'),(1300,'Delhi')],

'Mumbai': [(800,'Hyderabad'),(1000,'Bangalore'),(600,'Indore')],

'Nagpur':[(500,'Indore'),(1200,'Kolkata'),(500,'Hyderabad')],

'Hyderabad':[(800,'Mumbai'),(500,'Nagpur'),(1500,'Kolkata'),(500,'Bangalore')],

'Bangalore':[(1000,'Mumbai'),(500,'Hyderabad')]

} #based on ppt bfs cities distance problem

hSLD = {

'Delhi':0,

'Indore':800,

'Mumbai':1300,

'Hyderabad':1500,

'Bangalore':1800,

'Nagpur':1000,

'Kolkata':1300

}

queue = []

open = []

closed = []

start = "Hyderabad"

end = "Delhi"

totalDistance = 0

def expand(self,s:str):

near\_cities:list = self.cityMap.get(s)

near\_cities.sort(reverse=True)

return near\_cities

def validMove(self,near\_cities:list):

distance = 0

for city in near\_cities:

self.queue.append((self.hSLD.get(city[1]),city[1],city[0]))

if city[1] not in self.closed:

self.open.append(city[1])

if self.open.count(city[1])>1:

self.open.remove(city[1])

self.queue.sort(reverse=True)

def bestFirstSearch(self):

self.queue.append((self.hSLD.get(self.start),self.start,0))

self.open.append(self.start)

i = 0

while(1):

next:str = self.queue.pop()

near\_cities = self.expand(next[1])

self.closed.append(next[1])

self.totalDistance = self.totalDistance + int(next[2])

self.validMove(near\_cities)

self.open.remove(next[1])

print(f"\nOpen List: {self.open}\nClosed List: {self.closed}")

if next[1] == self.end:

print("Path Reached")

print(f"Total Distance from {self.start} to {self.end}: {self.totalDistance} km")

exit(1)

i+=1

print("Select a problem(Best First Search): ")

print("1. 8-Puzzle")

print("2. Robot Navigation")

print("3. City Distance Problem")

ch = int(input("Enter your choice(1-3):"))

if ch == 1:

print("[+]8 Puzzle")

s = Puzzle()

s.bestFirstSearch()

elif ch == 2:

print('[+]Robot Navigation')

s = Robot()

s.calcManhatten()

print('\n Current State:')

s.printTable(s.table)

s.bestFirstSearch()

elif ch == 3:

print('[+]City Distance')

s = cityDistance()

s.bestFirstSearch()

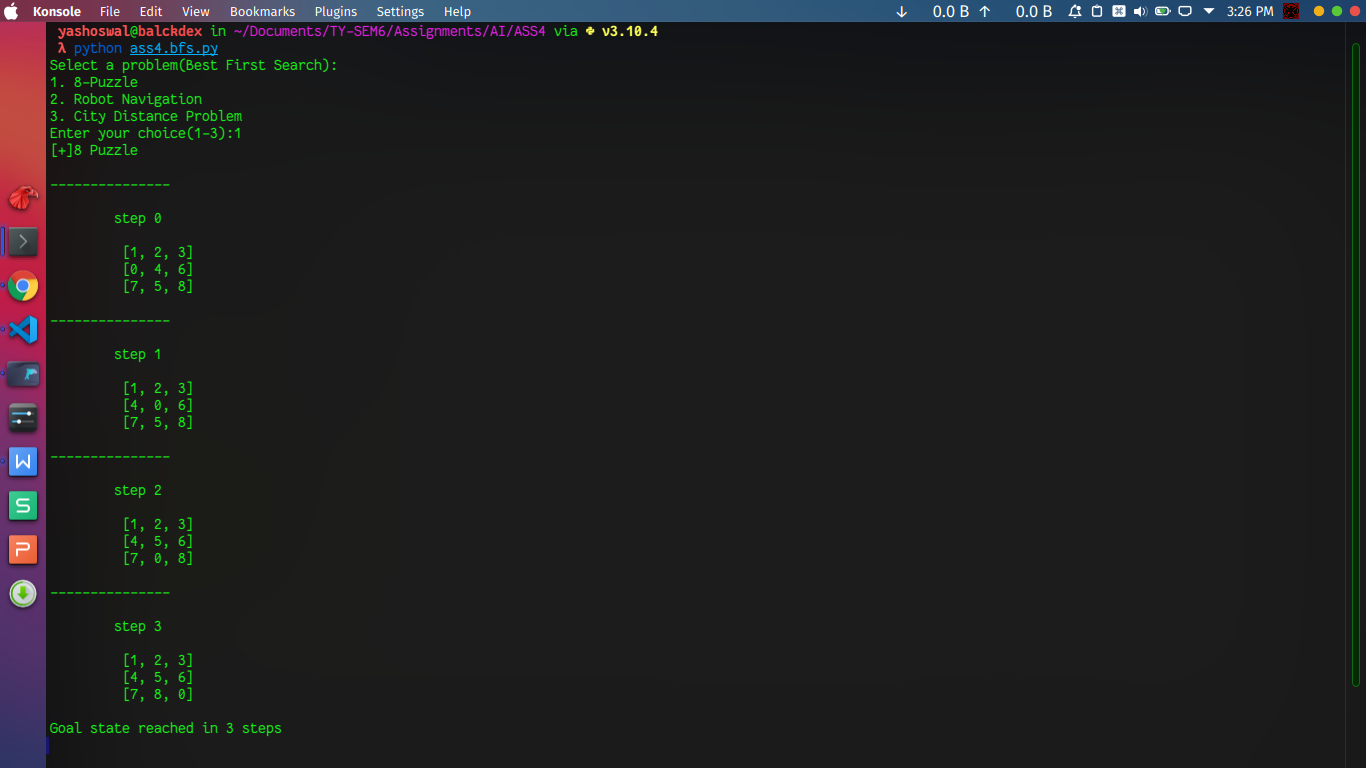
else:

print("[-]Run again and enter valid choice")

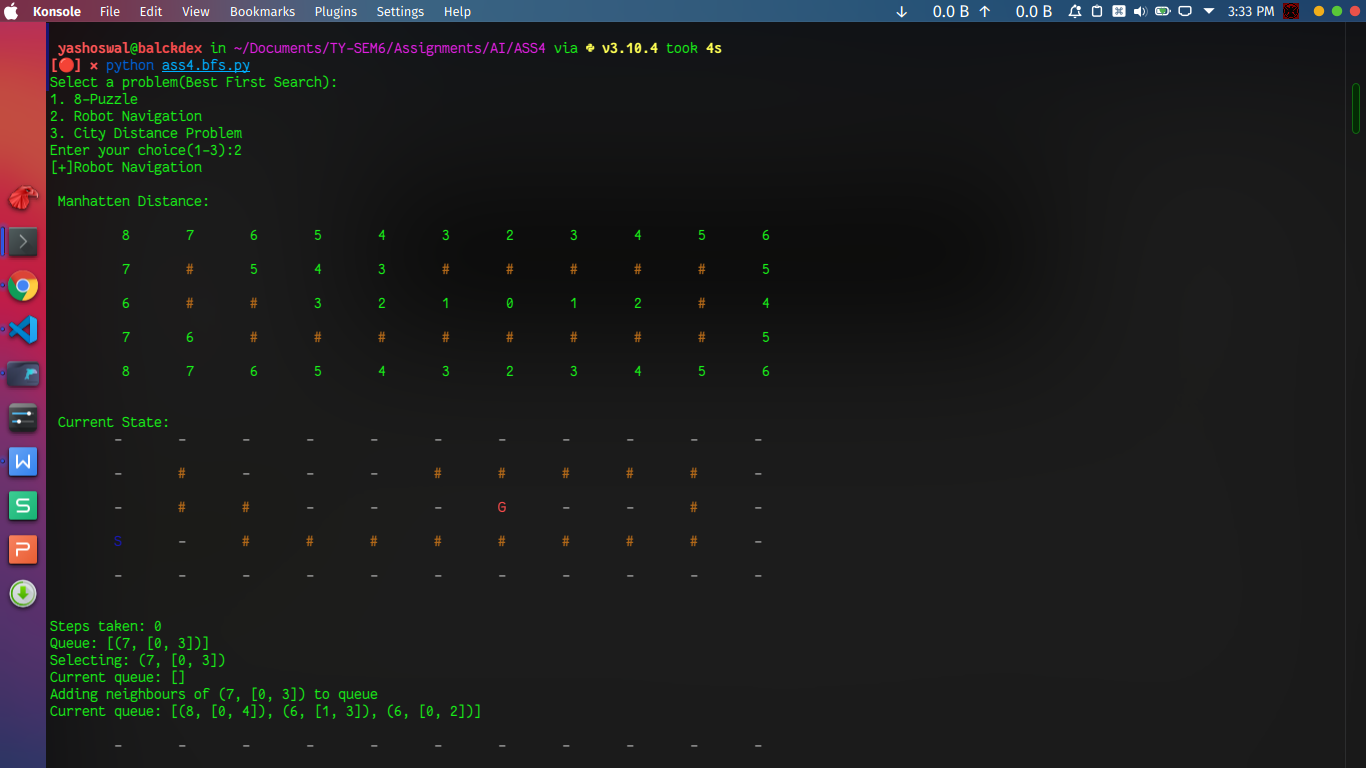
exit(1)

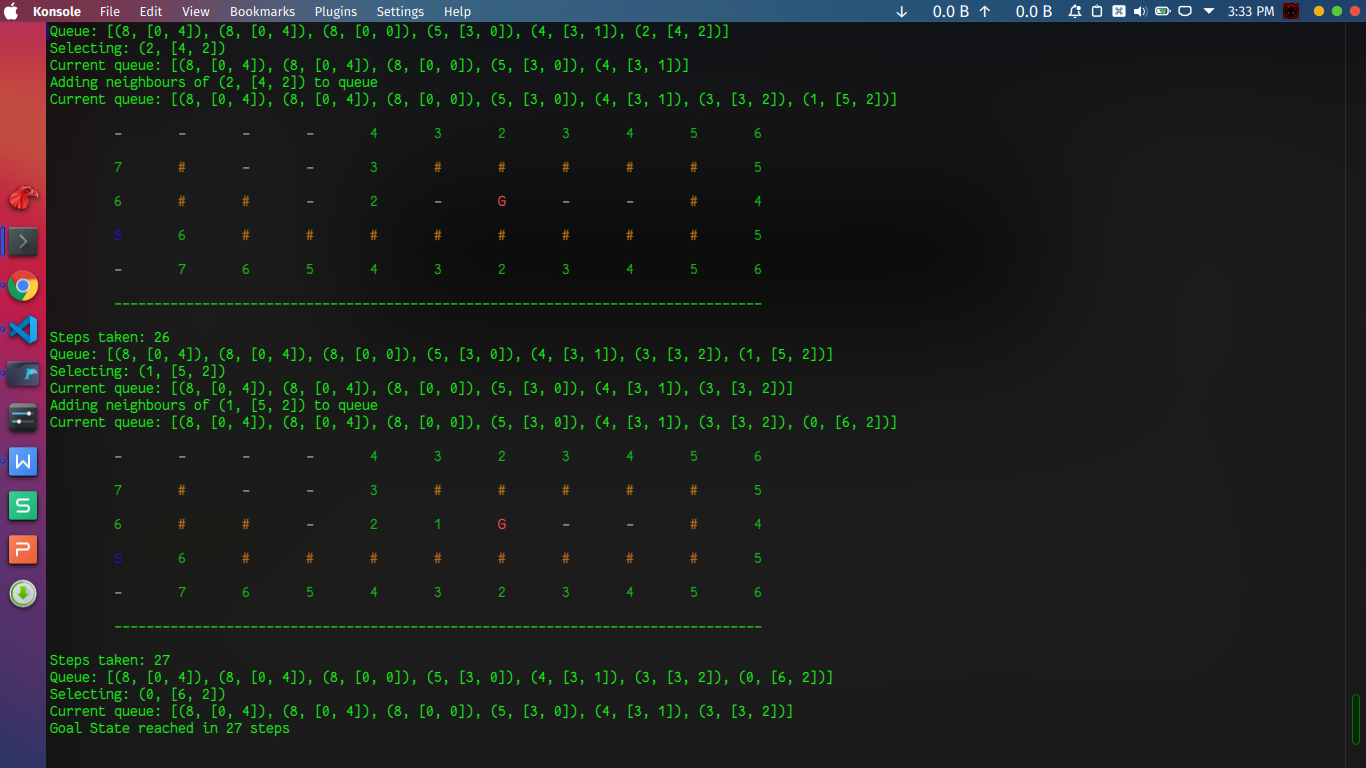
OUTPUT:

1. 8-Puzzle:

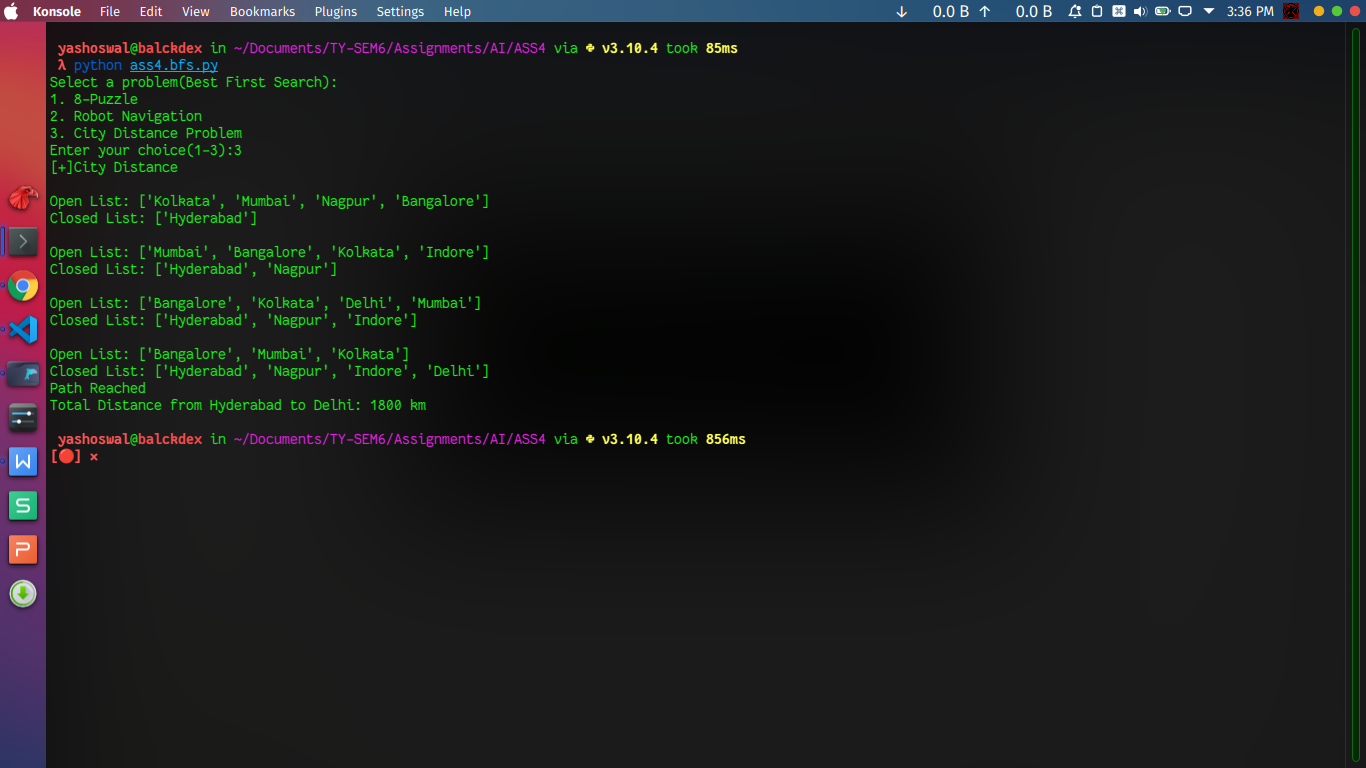


1. Robot Navigation:





1. City Distance:



1. A star:

CODE:

from colorama import Fore,Back,Style,init

init(autoreset=True)

import copy

from copy import deepcopy

class Puzzle():

goal = [[1, 2, 3],

[8, 0, 4],

[7, 6, 5]]

board\_config = [[2, 3, 4],

[1, 8, 0],

[7, 6, 5]]

steps = 0

def calculate\_fOfn(self, cal\_config):

h = 0

for i in range(0, 3):

for j in range(0, 3):

if cal\_config[i][j] != self.goal[i][j]:

h += 1

return h

def isSafe(self, x, y):

return x >= 0 and x < 3 and y >= 0 and y < 3

def print\_board(self, print\_config):

for i in range(0, 3):

for j in range(0, 3):

print(" " + str(print\_config[i][j]) + " ", end="")

print()

def find\_all\_configs(self, all\_config):

config\_boards = []

config1 = deepcopy(all\_config)

config2 = deepcopy(all\_config)

config3 = deepcopy(all\_config)

config4 = deepcopy(all\_config)

for i in range(0, 3):

for j in range(0, 3):

if all\_config[i][j] != 0:

if self.isSafe(i - 1, j):

if all\_config[i - 1][j] == 0:

config1[i - 1][j] = config1[i][j]

config1[i][j] = 0

config\_boards.append(config1)

if self.isSafe(i + 1, j):

if all\_config[i + 1][j] == 0:

config2[i + 1][j] = config2[i][j]

config2[i][j] = 0

config\_boards.append(config2)

if self.isSafe(i, j + 1):

if all\_config[i][j + 1] == 0:

config3[i][j + 1] = config3[i][j]

config3[i][j] = 0

config\_boards.append(config3)

if self.isSafe(i, j - 1):

if all\_config[i][j - 1] == 0:

config4[i][j - 1] = config4[i][j]

config4[i][j] = 0

config\_boards.append(config4)

return config\_boards

def puzzle\_start(self, config, goal\_heuristic):

objective\_values = []

new\_config = deepcopy(config)

boards\_configs = []

open\_list = []

closed\_list = []

visited = []

open\_list.append(new\_config)

visited.append(new\_config)

print(Fore.RED+"\t\t\tLIST IS DISPLAYED IN ROW MAJOR ORDER\n\n")

print("Initially - ")

print("Open List - ")

print(open\_list)

print("Closed List - ")

print(closed\_list)

print("\n\n")

while True:

self.steps += 1

boards\_configs.clear()

open\_list.remove(new\_config)

closed\_list.append(new\_config)

heuristic\_value = self.calculate\_fOfn(new\_config)

if heuristic\_value == goal\_heuristic:

print("Solution Reached !!")

print(f"\nIn {self.steps} Steps\n")

break

boards\_configs = self.find\_all\_configs(new\_config)

for i in boards\_configs:

visited.append(i)

open\_list.append(i)

h = self.calculate\_fOfn(i)

objective\_values.append(h)

print("Open List - ")

print(open\_list)

print("Closed List - ")

print(closed\_list)

print("\n\n")

min\_value = min(objective\_values)

min\_value\_index = objective\_values.index(min\_value)

new\_config = boards\_configs[min\_value\_index]

print(f"Board Configuration Selected With Heuristic Value - {str(self.steps)} + {str(min\_value)}")

self.print\_board(new\_config)

print("\n")

objective\_values.clear()

def Start\_Puzzle(self):

print("8-Puzzle Problem Using Best First Search\n")

goal\_heuristic = self.calculate\_fOfn(self.goal)

self.puzzle\_start(self.board\_config, goal\_heuristic)

class Robot():

table = [

['-','-','-','-','-','-','-','-','-','-','-'],

['-',Fore.YELLOW+'#','-','-','-',Fore.YELLOW+'#',Fore.YELLOW+'#',Fore.YELLOW+'#',Fore.YELLOW+'#',Fore.YELLOW+'#','-'],

['-',Fore.YELLOW+'#',Fore.YELLOW+'#','-','-','-','-','-','-',Fore.YELLOW+'#','-'],

['-','-',Fore.YELLOW+'#',Fore.YELLOW+'#',Fore.YELLOW+'#',Fore.YELLOW+'#',Fore.YELLOW+'#',Fore.YELLOW+'#',Fore.YELLOW+'#',Fore.YELLOW+'#','-'],

['-','-','-','-','-','-','-','-','-','-','-']

]

goalX = 6

goalY = 2

startX = 0

startY = 3

newTable = copy.deepcopy(table)

newTable\_2 = copy.deepcopy(table)

newTable\_3 = copy.deepcopy(table)

queue = []

visited = []

def calcManhatten(self):

self.table[self.startY][self.startX] = Fore.BLUE+'S'

self.table[self.goalY][self.goalX] = Fore.RED+'G'

print("\n Manhatten Distance: \n")

for i in range(5):

for j in range(11):

if self.table[i][j]!=Fore.YELLOW+'#':

self.newTable\_2[i][j] = abs(self.goalX - j) + abs(self.goalY-i)

self.newTable\_3[i][j] = abs(self.startX - j) + abs(self.startY-i)

self.newTable[i][j] = self.newTable\_2[i][j] + self.newTable\_3[i][j]

print(f'\t{self.newTable\_2[i][j]}+{self.newTable\_3[i][j]}', end='')

print('\n')

position = [self.startX,self.startY]

self.queue.append((self.newTable[self.startY][self.startX],position))

def getNeighbors(self):

position = [0]\*4

value = [0]\*4

position[0] = [self.startX+1,self.startY]

position[1] = [self.startX-1,self.startY]

position[2] = [self.startX,self.startY-1]

position[3] = [self.startX,self.startY+1]

for i in range(4):

if position[i][1]>-1 and position[i][1]<5 and position[i][0]>-1 and position[i][0]<11:

value[i] = self.newTable[position[i][1]][position[i][0]]

if value[i] != Fore.YELLOW+'#' and ((value[i], position[i]) not in self.visited) and ((value[i], position[i]) not in self.queue) :

self.queue.append((value[i], position[i]))

self.queue.sort(reverse=True)

def bestFirstSearch(self):

steps = 0

while (self.queue) :

input()

print(f"Steps taken: {steps}")

print(f"Queue: {self.queue}")

next = self.queue.pop()

print(f"Selecting: {next}")

print(f"Current queue: {self.queue}")

if next[1][0] == self.goalX and next[1][1] == self.goalY :

print(f"Goal State reached in {steps} steps")

exit(1)

if next[1] == [self.startX,self.startY]:

self.table[next[1][1]][next[1][0]] = Fore.BLUE+'S'

else:

self.table[next[1][1]][next[1][0]] = f"{Fore.GREEN+str(self.newTable\_2[next[1][1]][next[1][0]])}+{str(self.newTable\_3[next[1][1]][next[1][0]])}"

self.visited.append(next)

self.startX = next[1][0]

self.startY = next[1][1]

self.getNeighbors()

print(f"Adding neighbours of {next} to queue\nCurrent queue: {self.queue}\n")

self.printTable(self.table)

print('\t---------------------------------------------------------------------------------')

steps+=1

def printTable(self,table):

for i in range(5):

for j in range(11):

print("\t"+Fore.WHITE+str(table[i][j]), end='')

print('\n')

class City\_Distance():

class Graph:

def \_\_init\_\_(self, graph\_dict=None, directed=True):

self.graph\_dict = graph\_dict or {}

self.directed = directed

if not directed:

self.make\_undirected()

def make\_undirected(self):

for a in list(self.graph\_dict.keys()):

for (b, dist) in self.graph\_dict[a].items():

self.graph\_dict.setdefault(b, {})[a] = dist

def connect(self, A, B, distance=1):

self.graph\_dict.setdefault(A, {})[B] = distance

if not self.directed:

self.graph\_dict.setdefault(B, {})[A] = distance

def get(self, a, b=None):

links = self.graph\_dict.setdefault(a, {})

if b is None:

return links

else:

return links.get(b)

def nodes(self):

s1 = set([k for k in self.graph\_dict.keys()])

s2 = set([k2 for v in self.graph\_dict.values() for k2, v2 in v.items()])

nodes = s1.union(s2)

return list(nodes)

def display\_graph(self):

print(Fore.YELLOW+"\n\t\t\tTHE GRAPH IS - \n")

for key in self.graph\_dict:

print(Fore.CYAN+key, Fore.WHITE+' -> ', self.graph\_dict[key])

class Node:

def \_\_init\_\_(self, name: str, parent: str):

self.name = name

self.parent = parent

self.g = 0

self.h = 0

self.f = 0

def \_\_eq\_\_(self, other):

return self.name == other.name

def \_\_lt\_\_(self, other):

return self.f < other.f

def \_\_repr\_\_(self):

return ('({0},{1})'.format(self.position, self.f))

def best\_first\_search(self, graph, heuristics, start, end):

open = []

closed = []

start\_node = self.Node(start, None)

goal\_node = self.Node(end, None)

open.append(start\_node)

while len(open) > 0:

print(Fore.BLUE+"\n\nOpen List - ")

for i in open:

print(i.name, end=" | ")

print()

print(Fore.BLUE+"Closed List - ")

for i in closed:

print(i.name, end=" | ")

open.sort()

current\_node = open.pop(0)

closed.append(current\_node)

if current\_node == goal\_node:

path = []

while current\_node != start\_node:

path.append(current\_node.name)

current\_node = current\_node.parent

path.append(start\_node.name)

return path[::-1]

neighbors = graph.get(current\_node.name)

for key, value in neighbors.items():

neighbor = self.Node(key, current\_node)

if (neighbor in closed):

continue

neighbor.g = current\_node.g + graph.get(current\_node.name, neighbor.name)

neighbor.h = heuristics.get(neighbor.name)

neighbor.f = neighbor.g + neighbor.h

if (self.add\_to\_open(open, neighbor) == True):

open.append(neighbor)

return None

def add\_to\_open(self, open, neighbor):

for node in open:

if (neighbor == node and neighbor.f >= node.f):

return False

return True

def start(self):

graph = self.Graph()

graph.connect('Oradea', 'Zerind', 71)

graph.connect('Oradea', 'Sibiu', 151)

graph.connect('Zerind', 'Arad', 75)

graph.connect('Arad', 'Sibiu', 140)

graph.connect('Arad', 'Timisoara', 118)

graph.connect('Timisoara', 'Lugoj', 111)

graph.connect('Lugoj', 'Mehadia', 70)

graph.connect('Mehadia', 'Drobeta', 75)

graph.connect('Drobeta', 'Craiova', 120)

graph.connect('Craiova', 'Pitesti', 138)

graph.connect('Craiova', 'Rimnicu Vilcea', 146)

graph.connect('Sibiu', 'Fagaras', 99)

graph.connect('Fagaras', 'Bucharest', 211)

graph.connect('Sibiu', 'Rimnicu Vilcea', 80)

graph.connect('Rimnicu Vilcea', 'Pitesti', 97)

graph.connect('Pitesti', 'Bucharest', 101)

graph.connect('Bucharest', 'Giurgui', 90)

graph.make\_undirected()

graph.display\_graph()

heuristics = {}

heuristics['Arad'] = 366

heuristics['Bucharest'] = 0

heuristics['Craiova'] = 160

heuristics['Drobeta'] = 242

heuristics['Fagaras'] = 176

heuristics['Guirgiu'] = 77

heuristics['Lugoj'] = 244

heuristics['Mehadia'] = 241

heuristics['Oradea'] = 380

heuristics['Pitesti'] = 100

heuristics['Rimnicu Vilcea'] = 193

heuristics['Sibiu'] = 253

heuristics['Timisoara'] = 329

heuristics['Zerind'] = 800

path = self.best\_first\_search(graph, heuristics, 'Arad', 'Bucharest')

print(Fore.GREEN+"\n\nThe Path Is - ")

if path is not None:

for i in range(len(path)):

print(path[i])

print(Fore.BLUE+"\t\t\t\t\t\tA\* Search\n")

choice = int(input("Enter Your Choice - \n1. 8-Puzzle Problem\n2. Robot Navigation\n3. City-Distance Problem\nChoice - "))

if(choice == 1):

temp = Puzzle()

temp.Start\_Puzzle()

elif(choice == 2):

temp = Robot()

temp.calcManhatten()

print('\n Current State:')

temp.printTable(temp.table)

temp.bestFirstSearch()

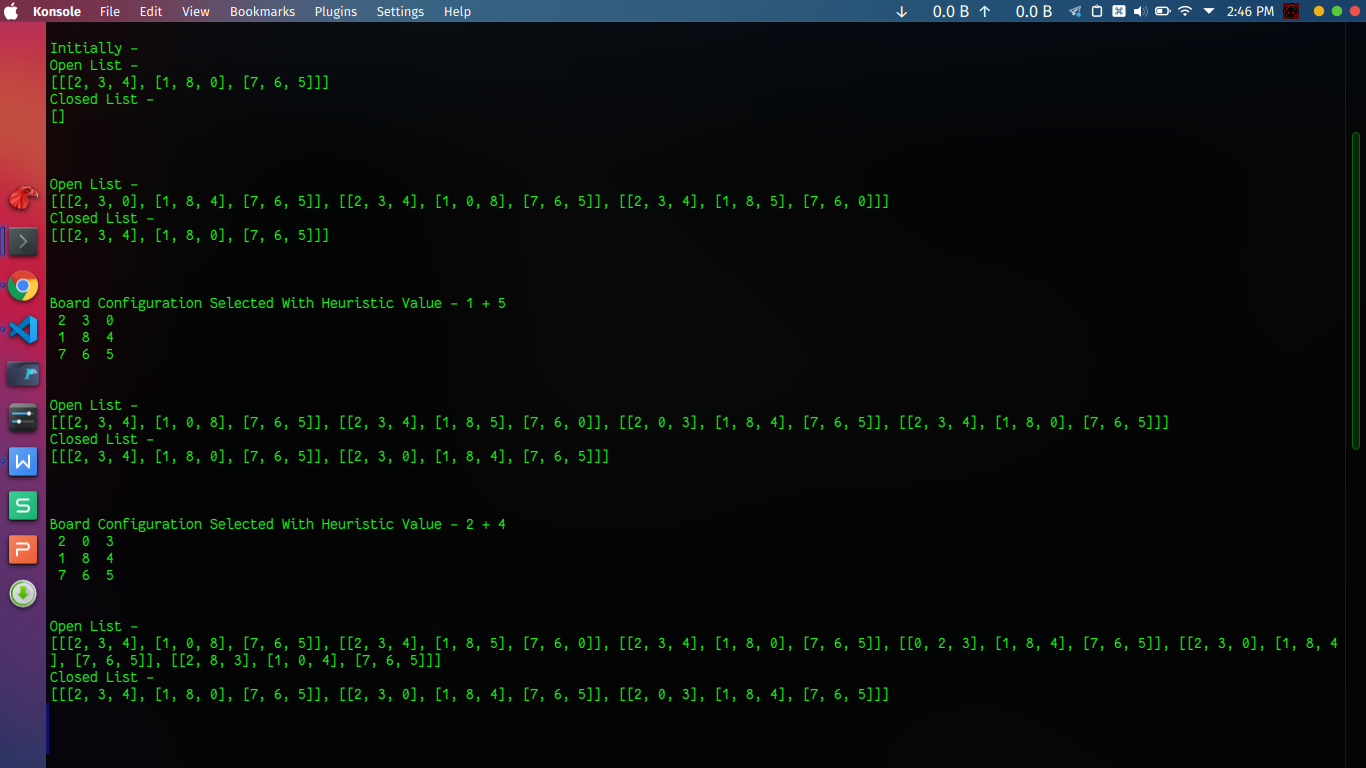
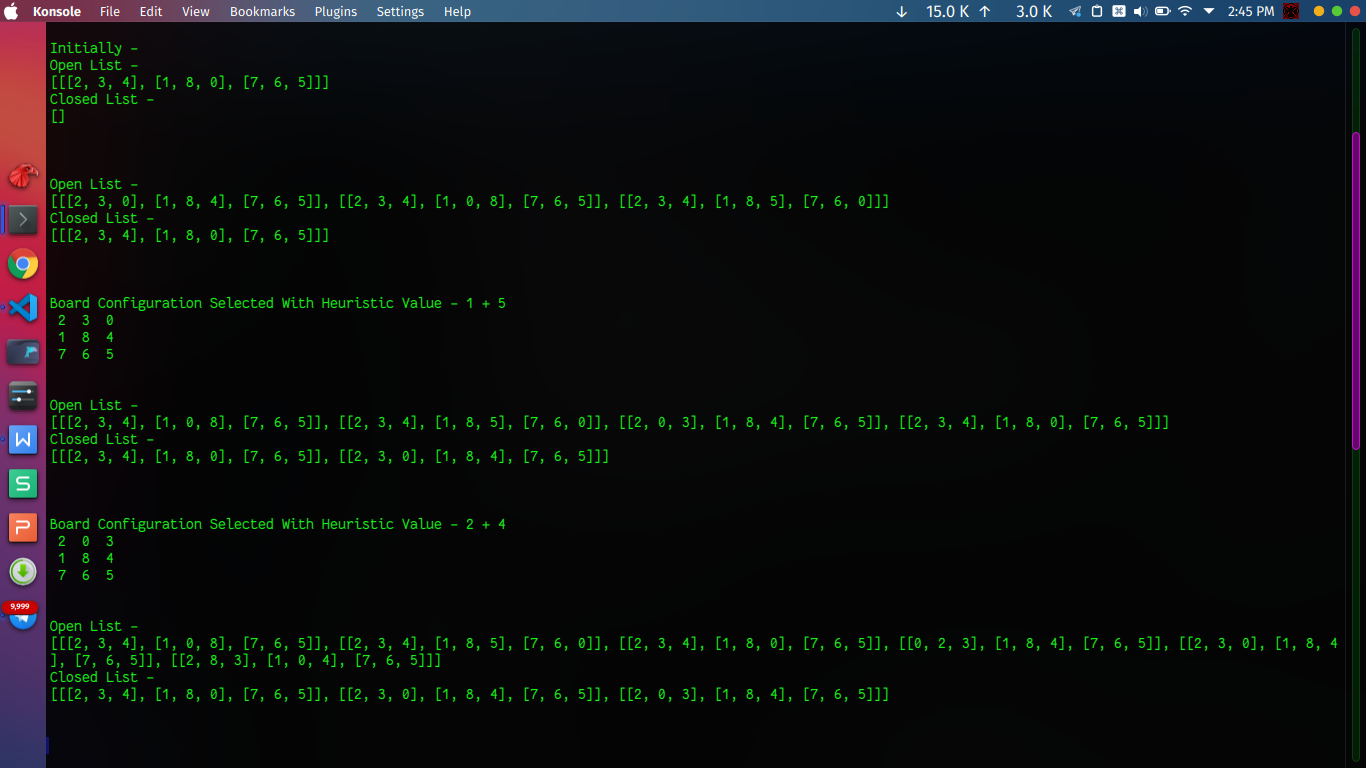
elif(choice == 3):

temp = City\_Distance()

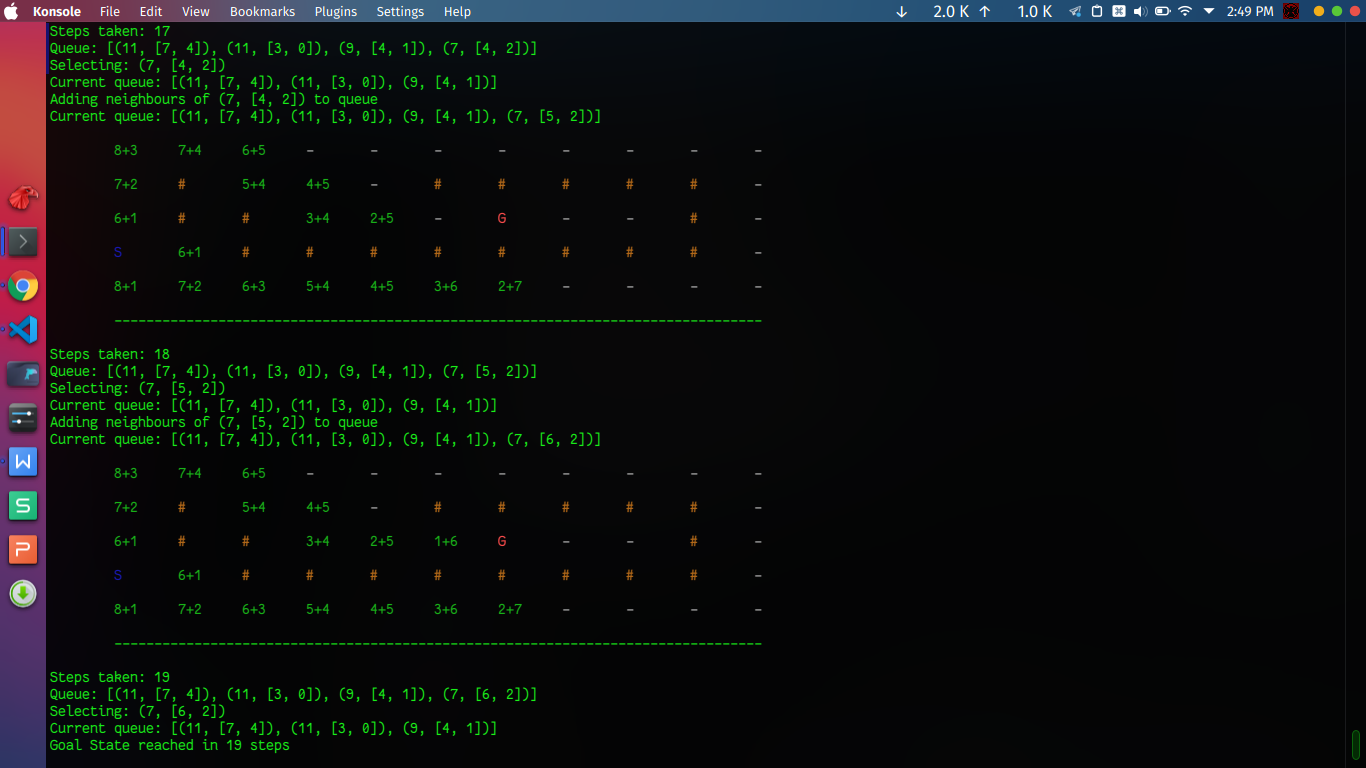
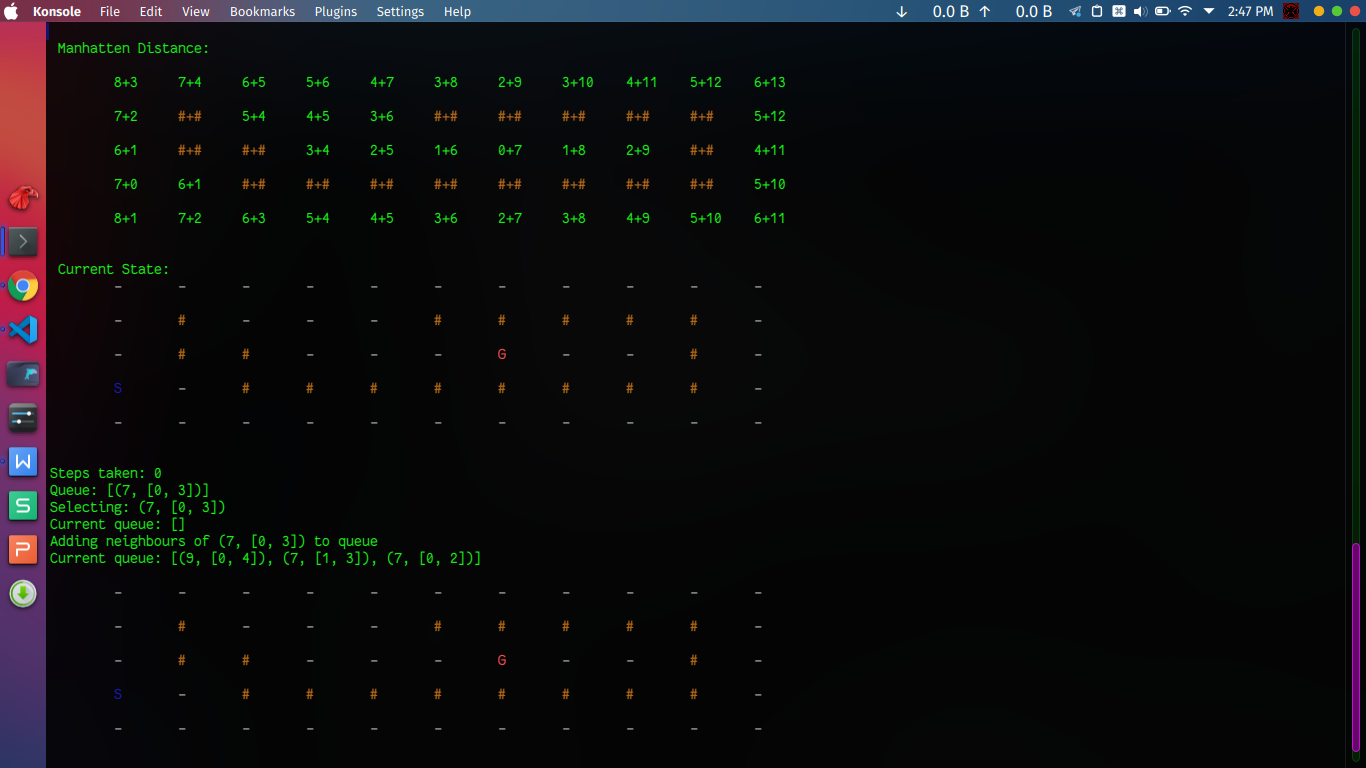
temp.start()

OUTPUT:

1. 8 Puzzle



1. Robot Navigation



1. City Distance Problem

