Q1. Prove that a monotonic heuristic is admissible.

A heuristic function ***h(n)*** is:

* Monotonic/consistent if the estimated cost from node ***n*** to the goal is no greater than the step cost to its successor ***n′*** plus the estimated cost from the successor to the goal.
* Admissible if ***h(n)*** never overestimates the true cost to the goal state.

We will prove that monotonicity/consistency implies admissibility using induction.

If ***t*** is the goal node, then ***h(t) = 0*** if the heuristic function ***h*** is admissible since edge costs are assumed to be non-negative and thus, the optimal cost from one node to itself is necessarily 0. This certainly holds in case the heuristic function is admissible. But we assume further that ***h(t)=0*** for any goal, to proof that *consistency necessarily implies admissibility*.

**Base case:**

Take any predecessor of the goal node ***t***. Let ***n*** denote it, so that ***t*** is a successor of ***n***. If the heuristic function is consistent, then ***h(n) ≤* *c(n, t) + h(t) = c(n, t) + 0 = c(n, t)*** and hence, ***h*** behaves admissibly in this case.

Note that the base case does not assume that the edge **⟨*n,t*⟩** is necessarily the optimal solution from n to t and, indeed, there might be a different path from n to t with a lower cost. The importance of the base case, is that h(n)≤c(n,t) for all ancestors of node t! This result will be reused in the induction step.

**Induction:**

Consider a node ***n***. The optimal cost to reach the goal ***t*** from ***n***, ***h*∗*(n)***, is computed as:

**minm ∈ SCS(n) {c(n, m) + h\*(m)}**, where **SCS(n)** is the set of successors of node ***n***.

As consistency is assumed by hypothesis, then **h(n) ≤ c(n, n′) + h(n′)**.

Furthermore, as **h(n′) ≤ h\*(n′)** is assumed by the induction step then **h(n) ≤ c(n, n′) + h∗(n′)** and this is true for all successors **n′** of node **n**.

In other words: **h(n) ≤ minm ∈ SCS(n){c(n, m) + h∗(m)} = h∗(n)**, so that **h(n) ≤ h∗(n)**.

Q2 (A). 8 Piece Puzzle using Iterative Deepening Depth First Search.

import copy

init\_config = [

['8', '4', '7'],

['1', '6', '5'],

['2', '3', ' ']

]

goal = [

['1', '8', '7'],

['2', ' ', '6'],

['3', '4', '5']

]

def main():

depth = 1

status = DFS(init\_config, depth)

while status != True and depth <= 20:

depth += 1

status = DFS(init\_config, depth)

def DFS(config, depth):

stack = []

stack.append([config])

visited = []

visited.append(config)

while (len(stack) != 0):

path = stack.pop()

# depth check

if len(path) > depth:

continue

# prepare to extend

neighbours = get\_neighours(path[len(path) - 1])

# check if any neighbour is the goal, if not then extend

for neighbour in neighbours:

if neighbour == goal:

print("Goal found at depth: " + str(depth))

print(path + goal)

return True

elif neighbour not in visited:

stack.append(path + [neighbour])

visited.append(neighbour)

# couldnt find goal

return False

# computes all possible mutations of given config

def get\_neighours(config):

mutations = []

for i in range(3):

if ' ' in config[i]:

j = config[i].index(' ')

new\_moves = [

(i - 1, j), (i + 1, j),

(i, j + 1), (i, j - 1)

]

for a, b in new\_moves:

if (a >= 0 and b >= 0) and (a < 3 and b < 3):

temp = copy.deepcopy(config)

temp[i][j], temp[a][b] = temp[a][b], temp[i][j]

mutations.append(temp)

break

return mutations

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

main()

Q2 (B). 8 Piece Puzzle using Best First Search.

import copy

from collections import deque

init\_config = [

['8', '4', '7'],

['1', '6', '5'],

['2', '3', ' ']

]

goal = [

['1', '8', '7'],

['2', ' ', '6'],

['3', '4', '5']

]

def main():

BestFS(init\_config)

def BestFS(config):

queue = deque()

queue.append([config])

visited = []

visited.append(config)

while (len(queue) != 0):

path = queue.popleft()

parent = copy.deepcopy(path[len(path) - 1])

# prepare to extend

neighbours = get\_neighours(parent)

scores = []

for neighbour in neighbours:

scores.append(calc\_score(neighbour))

# sort according to heuristic scores

sorted\_neighbours = [x for \_,x in sorted(zip(scores, neighbours))]

# check if any neighbour is the goal, if not then extend

for neighbour in sorted\_neighbours:

if neighbour == goal:

print("Goal found at depth: " + str(len(path)))

print(path + goal)

return len(path)

elif neighbour not in visited:

queue.append(path + [neighbour])

visited.append(neighbour)

# couldnt find goal

print("Could not find goal.")

return -1

# computes all possible mutations of given config

def get\_neighours(config):

mutations = []

for i in range(3):

if ' ' in config[i]:

j = config[i].index(' ')

new\_moves = [

(i - 1, j), (i + 1, j),

(i, j + 1), (i, j - 1)

]

for a, b in new\_moves:

if (a >= 0 and b >= 0) and (a < 3 and b < 3):

temp = copy.deepcopy(config)

temp[i][j], temp[a][b] = temp[a][b], temp[i][j]

mutations.append(temp)

break

return mutations

# heuristic function

def calc\_score(config):

return tiles\_in\_place(config)

#return manhattan\_distance(config)

def manhattan\_distance(config):

sum = 0

for i in range(3):

for j in range(3):

for k in range(3):

for l in range(3):

if goal[i][j] != ' ' and goal[i][j] == config[k][l]:

sum += (abs(i - k) + abs(j - l))

return sum

def tiles\_in\_place(config):

score = 0

for i in range(3):

for j in range(3):

if config[i][j] == goal[i][j]:

score += 1

return score

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

main()