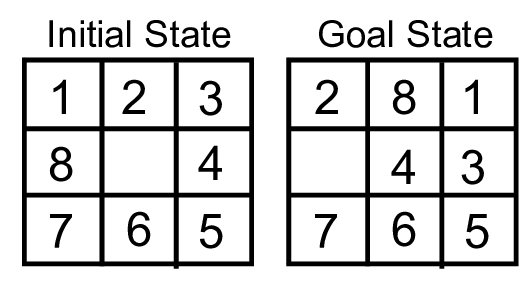
8-Puzzle Problem

# What is the “8-Puzzle” problem?

N-Puzzle or sliding puzzle is a popular puzzle that consists of N tiles where N can be 8, 15, 24, and so on. In our example N=8. The puzzle is divided into sqrt(N+1) rows and sqrt(N+1) columns. Eg. a 15-Puzzle will have 4 rows and 4 columns and an 8-Puzzle will have 3 rows and 3 columns. The puzzle consists of N tiles and one empty space where the tiles can be moved. The puzzle's start and Goal configurations (also called state) are provided. The puzzle can be solved by moving the tiles one by one in a single empty space, thus achieving the Goal state.



**Fig1. Start and Goal State of an 8-puzzle**

The tiles in the start (initial) state can be moved in the empty space in a particular order and thus achieve the goal state.

# Rules for solving the puzzle

Instead of moving the tiles in the empty space, we can visualize moving the empty space in place of the tile, basically swapping the tile with the empty space. The empty space can only move in four directions:

1. Up
2. Down
3. Left
4. Right,

The empty space can’t move diagonally and can take only one step at a time (i.e. move the empty space one position at a time).

# Searching Techniques

Basically, there are two types of searching techniques: ***Uninformed Search*** and ***Informed Search***.

## Uninformed Search Algorithm

Uninformed search is a class of general-purpose search algorithms that operates in a brute-force way. Uninformed search algorithms don’t have additional information about the state or search space other than how to traverse the tree, so it is called blind search. Following are the various types of uninformed search algorithms: ***Breadth-first search, Depth-first search, Depth limited search, Iterative deepening depth-first search, Uniform cost search, and Bidirectional search***.

## Informed Search Algorithm

An informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach the goal state, etc… This knowledge help agents explore less the search space and find the goal state.

The informed search algorithm is more useful for large search spaces. An informed search algorithm uses the idea of heuristic, so it is also called ***Heuristic search***.

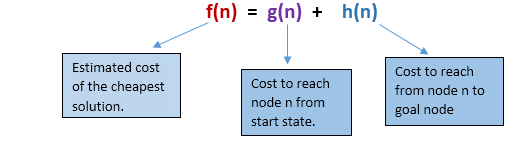
**Heuristic function:** Heuristic is a function that can be used in informed search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close the agent is to the goal. The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in a reasonable time. The ***heuristic function*** estimates how close a state is to the goal. It is represented by ***h(n)***, and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

Pure heuristic search: It is the simplest form of the heuristic search algorithm. It expands states based on their heuristic value h(n). It maintains two lists, OPEN and CLOSED list. In the CLOSED list, it places those states which have already expanded, and in the OPEN list, it places states which have yet not been expanded. On each iteration, each state n with the lowest heuristic value is expanded and generates all its successors, and n is placed in the closed list. The algorithm continues unit a goal state is found. Following are the various types of informed search algorithms: ***Best-first search algorithm (Greedy search)* & *A\* search algorithm.*** We have used the A\* algorithm to implement the 8-Puzzle problem.

# A\* search algorithm

A\* search algorithm is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). It has combined features of Uninform-cost search (UCS) and greedy best-first search, by which it solves the problem efficiently. A\* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands fewer search trees and provides optimal results faster. It is similar to UCS except that it uses f(n) = g(n) + h(n) instead of g(n).

In the A\* algorithm, we use heuristic as well as the cost to reach the goal state. Hence we can combine both costs as follows, and this sum is called a ***fitness number.***



**Fig2. Fitness number**

The key feature of the A\* algorithm is that it keeps a track of each visited state which helps in ignoring the states that are already visited, saving a huge amount of time. It also has a list that holds all the states that are left to be explored and it chooses the most optimal state from this list, thus saving time not exploring unnecessary or less optimal states.

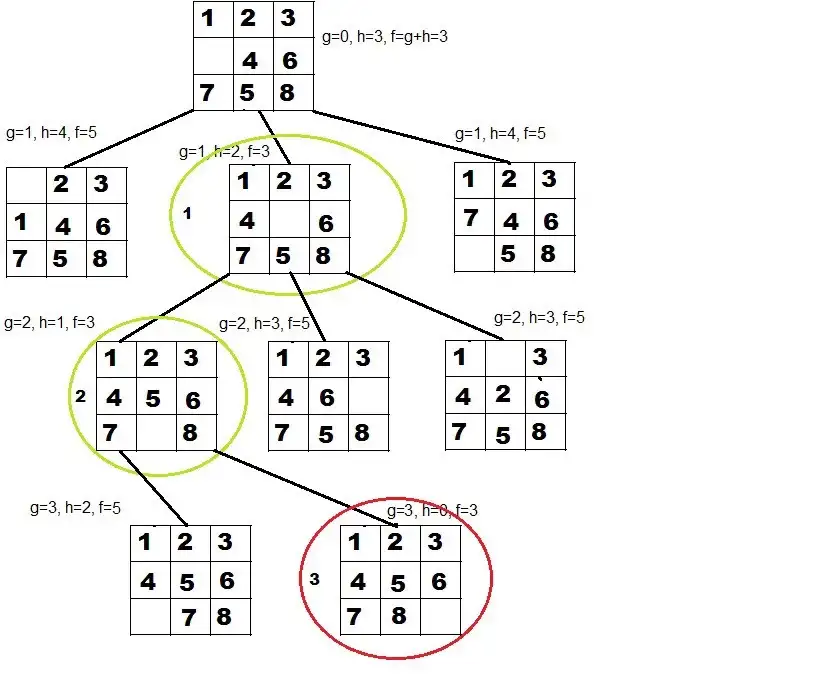
So we use two lists namely ‘open’ and ‘closed’ lists. The open list contains all the nodes that are being generated and are not existing in the closed list and each node explored after its neighboring nodes are discovered is put in the closed list and the neighbors are put in the open list this is how the nodes expand. Initially, the open list holds the start (initial) node. The next node chosen from the open list is based on its fitness number, the node with the least fitness number is picked up and explored.

In our 8-Puzzle problem, we have defined the h(n) as the number of misplaced tiles by comparing the current state and the goal state or summation of the Manhattan distance between misplaced nodes. g(n) will remain as the number of nodes traversed from a start node to get to the current node. From Fig1, we can calculate the h(n) by comparing the initial state and goal state and counting the number of misplaced tiles. Thus, h(n) = 5 and g(n) = 0 as the number of nodes traversed from the start node to the current node is 0.

# How does A\* solve the 8-Puzzle problem?

We first move the empty space in all the possible directions in the start state and calculate the fitness number for each state. This is called expanding the current state.

After expanding the current state, it is pushed into the closed list and the newly generated states are pushed into the open list. A state with the least fitness number is selected and expanded again. This process continues until the goal state occurs as the current state. Basically, here we are providing the algorithm a measure to choose its actions. The algorithm chooses the best possible action and proceeds in that path. This solves the issue of generating redundant child states, as the algorithm will expand the node with the least fitness number.



**Fig3. A\* algorithm solves 8-puzzle**

# Implementation of N-Puzzle

class State:

def \_\_init\_\_(self, data, level, fval):

# initialize the node with data, level of the node and calculated fval

self.data = data

self.level = level

self.fval = fval

def generate\_child(self):

# generate child nodes from the given node by moving the blank space in

# either of the 4 directions [up, down, left, right]

x, y = self.find(self.data, '\_')

# contains position values for moving the blank space in either of the 4

# directions [up, down, left, right]

moves = [[x, y-1], [x, y+1], [x-1, y], [x+1, y]]

children = list()

for move in moves:

child = self.shuffle(self.data, x, y, move[0], move[1])

if child is not None:

child\_state = State(child, self.level+1, 0)

children.append(child\_state)

return children

def shuffle(self, state, x1, y1, x2, y2):

# move the blank space in the given direction

# if position value are out of the limits return None

if 0 <= x2 < len(self.data) and 0 <= y2 < len(self.data):

new\_state = list()

new\_state = self.copy(state)

new\_state[x1][y1], new\_state[x2][y2] = new\_state[x2][y2], new\_state[x1][y1]

return new\_state

else:

return None

def copy(self, root):

# copy function to create a similar matrix of the given state

new = list()

for i in root:

temp = list()

for j in i:

temp.append(j)

new.append(temp)

return new

def find(self, state, x):

# find position of blank space

n = len(self.data)

for i in range(n):

for j in range(n):

if state[i][j] == x:

return i, j

class Puzzle:

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

# Initialize the puzzle size and open & closed lists

self.n = n

self.open = list()

self.closed = list()

def state(self):

state = list()

for i in range(self.n):

row = input().split(" ")

state.append(row)

print()

return state

# Heuristic value f(x) = h(x) + g(x)

def fvalue(self, start, goal):

return self.hvalue(start.data, goal) + start.level

# misplaced valued

def hvalue(self, start, goal):

# calculate the different between the given puzzles

diff = 0

for i in range(self.n):

for j in range(self.n):

if start[i][j] != goal[i][j] and start[i][j] != '\_':

diff += 1

return diff

def solve(self):

# initialize Start and Goal puzzle state

print("Enter the goal state matrix:-")

goal = self.state()

print("Enter the start state matrix:-")

start = self.state()

start = State(start, 0, 0)

start.fval = self.fvalue(start, goal)

# put the start node in open list

print("Solution:-\n")

self.open.append(start)

step = 0

while True:

curr = self.open[0]

print(f' STEP {step}')

print(f'f(n)={curr.fval}, h(n)={curr.fval - curr.level}, g(n)={curr.level}')

for i in curr.data:

print(" ", end=' ')

for j in i:

print(j, end=' ')

print()

# if the difference between current and goal node is 0 we have reached goal node

if self.hvalue(curr.data, goal) == 0:

break

self.closed.append(curr.data)

for i in curr.generate\_child():

i.fval = self.fvalue(i, goal)

if i.data not in self.closed:

self.open.append(i)

del self.open[0]

# sort the open list based on f value

self.open.sort(key=lambda x: x.fval, reverse=False)

print()

print(" | ")

print(" | ")

print(" \\\'/ \n")

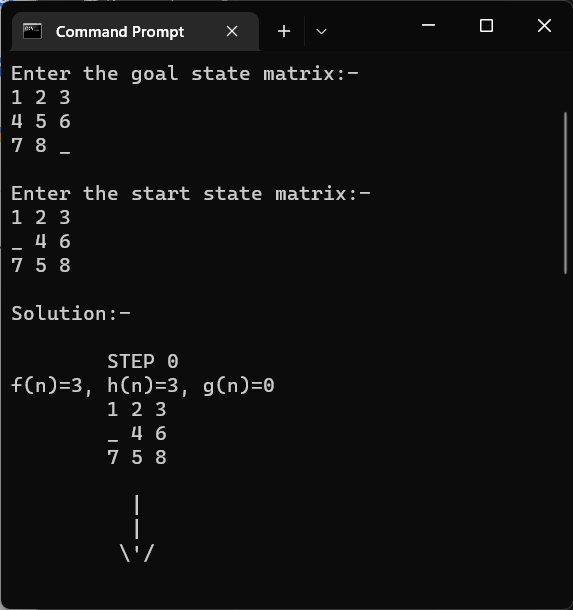
step += 1

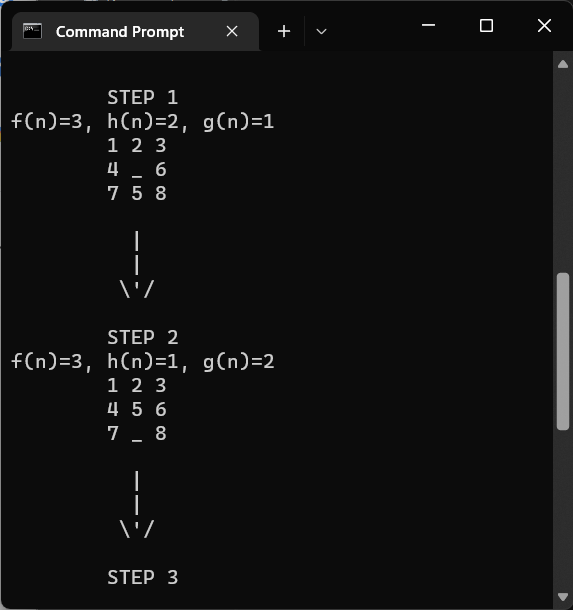
if \_\_name\_\_ == "\_\_main\_\_":

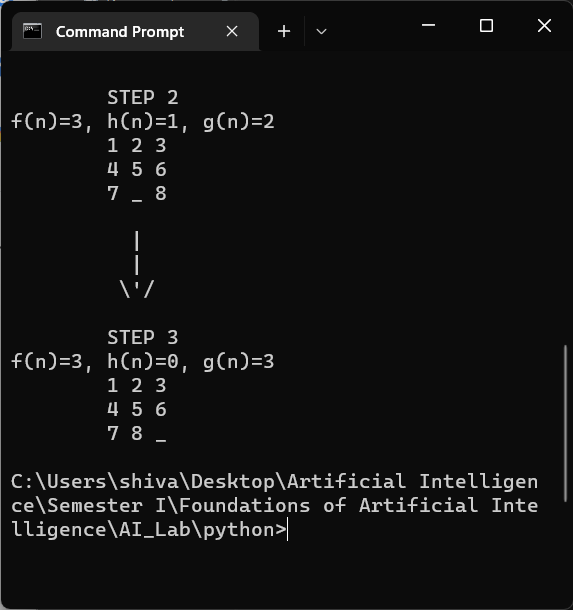
puzzle = Puzzle(3)

puzzle.solve()

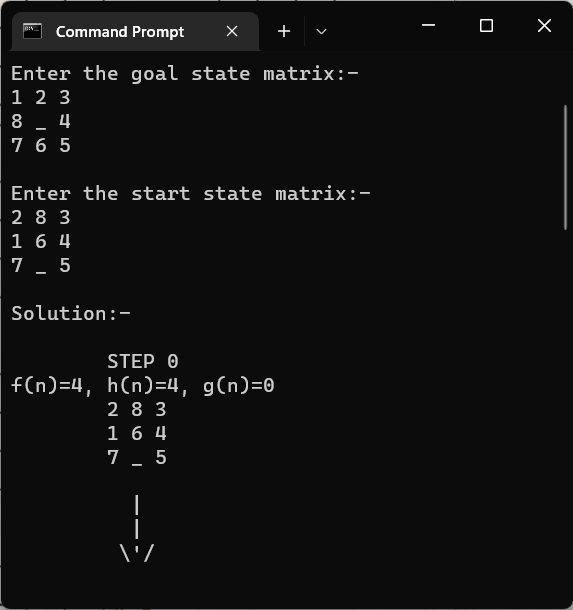
**Output 1: -**

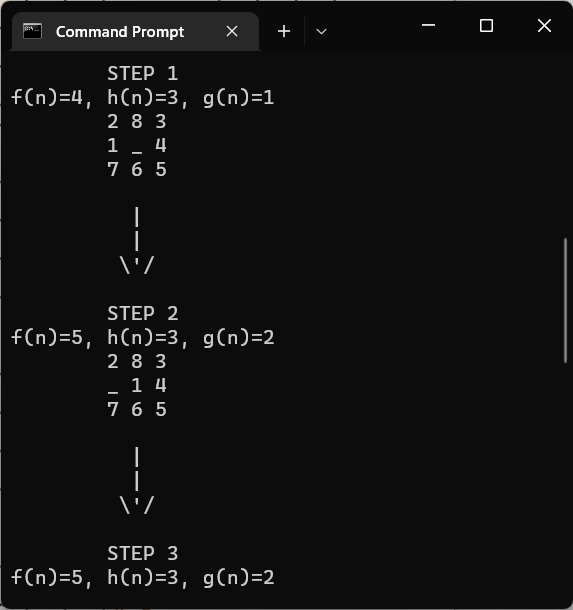
****

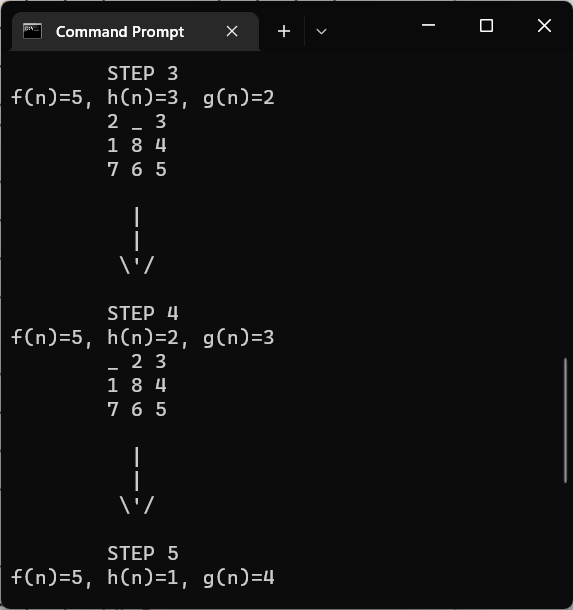
****

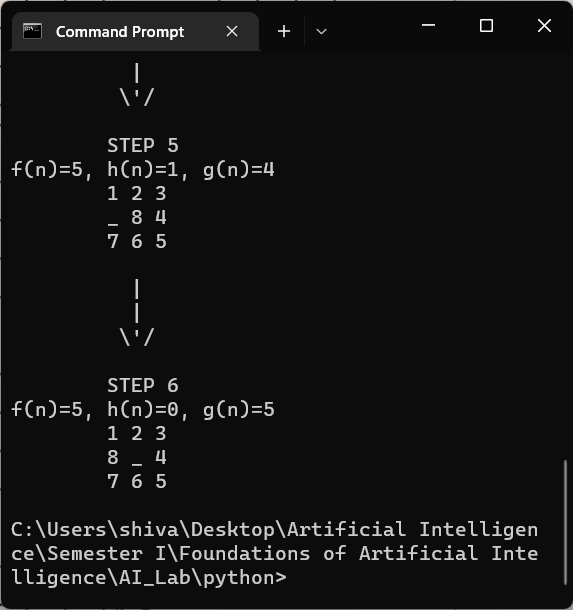
****

**Output 2: -**

****

****

****

****