

**ITCS 6150-Intelligent Systems**

**Project 1**

**Solving the 8-puzzle problem using A\* search algorithm**

**Submitted By Instructor**

Srushti Khot Prof. Dewan T. Ahmed, Ph.D.

Sanket Revadigar

**Aim:** To solve 8 puzzle problem using A\* algorithm

**Problem Statement:**

Implement A\* search algorithm and apply it to 8-puzzle problem and provide state space representation, operators, g (cost) and two heuristic functions of the 8-puzzle problem.

**Project Implementation:**

The 8-puzzle problem using A\* search algorithm is implemented using Python language.

**A\* Algorithm**

A\* is an algorithm that is used in pathfinding and graph traversal. It is used to find a best path between initial state and goal state(nodes). It enjoys widespread use because of its performance and accuracy. It is a combination of uniform cost search and best first search, which avoids expanding expensive paths. A\* star uses admissible heuristics which is optimal as it never over-estimates the path to goal.

The evaluation function f(n), A\* star uses for calculating distance is

**F(n)=g(n)+h(n)**

**g(n) =** cost so far to reach n

**h(n) =** estimated cost from n to goal

**f(n) =** estimated total cost of path through n to goal

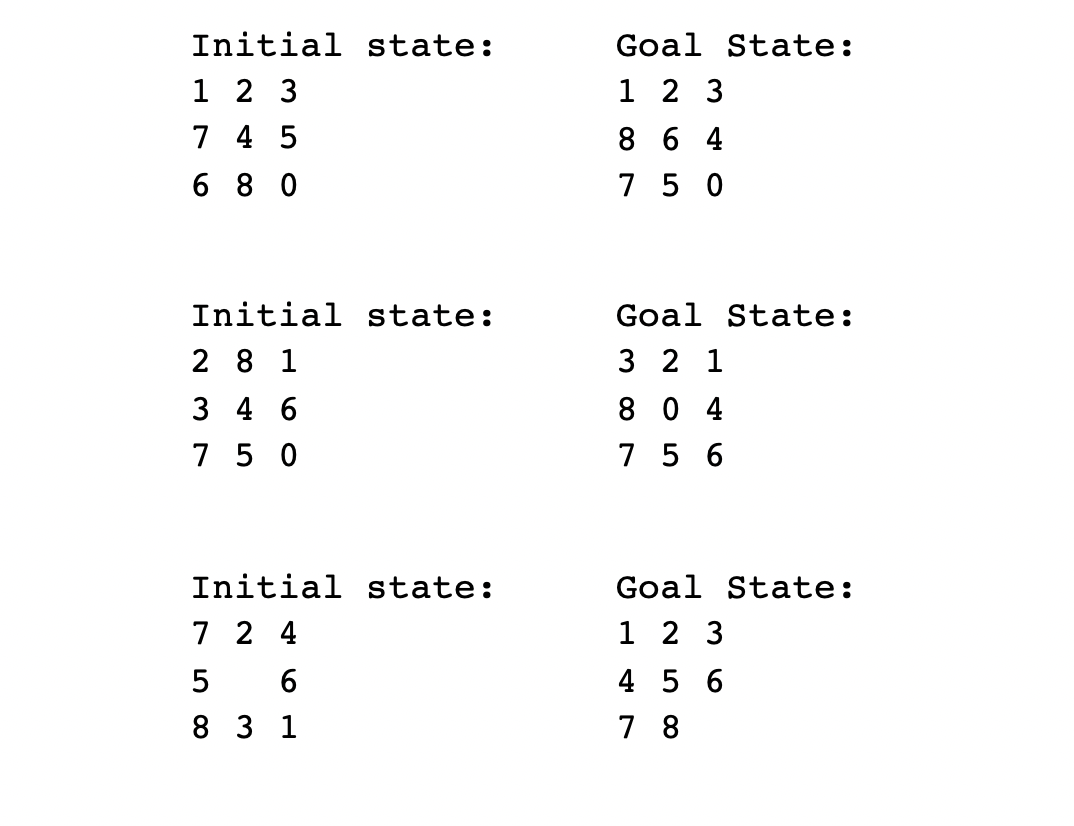
The Properties of A\* algorithm are as follows:

|  |  |
| --- | --- |
| Completeness | Yes (Till there are infinite nodes) |
| Time | Exponential |
| Space | Keeps all nodes in memory |
| Optimality | Yes |

**Problem formulation**

Problem formulation is a step where description of a problem is done. Understanding of the problem and important factors like variables, functions and relationship between them is obtained.

The 8-puzzle problem contains 3\*3 matrix. Each position is a tile and that makes a total of 9 tiles. Each tile can contain numbers from 1 to 8 and a blank space. Objective is to obtain goal state from initial state by exploring nodes. A\* algorithm is used to explore nodes. Tiles can move one step up, down, left or right.



To solve 8 puzzle problem using A\* algorithm, heuristic function is calculated using two techniques mentioned below.

**1.Heuristic 1(h1): Misplaced tiles**- Number of tiles misplaced in initial state in accordance with the goal state.

**2. Heuristic 2(h2): Manhattan distance**- This calculates total number of moves required for the misplaced tiles to get into their respective place in accordance with the goal state.

**Project Details:**

**Usage Overview:**

1. User is asked to input array of numbers from 1 to 8 separated by space to describe the initial state of the puzzle. Add 0 for blank square.
2. User is asked to input array of numbers from 1 to 8 separated by space to describe the goal state of the puzzle. Add 0 for blank square.

User is asked to input their choice.

1-Misplaced tiles

2-Manhattan distance

1. Solution is determined in every move by comparing the current state with the goal state.
2. For every ideal move at every level of the expansion following variables are:
   * + g(n): Distance calculated based on the levels (at the time of expansion) completed by the puzzle state so far.
     + h(n): estimated cost from n to goal.
3. Once the puzzle has been solved, the application prints
   * The total number of nodes that have been expanded.
   * The total number of nodes that have been generated.

**Program structure**

A\* algorithm for 8 puzzle problem is implemented using Python language. The code contains one class named Node. Node class contains functions needed to implement 8 puzzle using A\* algorithm.

**Global variables**

visited\_nodes and opened\_nodes are used to calculate and keep count of number of nodes already visited and expanded nodes respectively.

**Class used**:Node

**1)\_\_init\_\_** function is a generator function. It acts like a constructor and gets called every time an object of type Node is created. It contains the following variables:

* state\_of\_node: It is an array of length 9.
* move\_direction:It specifies which move(left,right,up,down)led to the current state.
* gn: Cost to reach n from initial state
* hn: Heuristic cost
* parent: parent node

**2)Move functions**: There are 4 move functions(Up,Down,Left,Right)

**up\_check**:This method is used to check if upward movement leads to a valid move.If so returns the new state otherwise returns None.

**left\_check**: This method is used to check if left movement leads to a valid move.If so returns the new state otherwise returns None.

**right\_check**: This method is used to check if right movement leads to a valid move.If so returns the new state otherwise returns None.

**down\_check**: This method is used to check if downward movement leads to a valid move.If so returns the new state otherwise returns None.

**3)Heuristic Functions**

**misplaced\_heuristic**-This function calculates the total number of misplaced tiles in the initial state in accordance with the goal state.

**manhattan\_distance**-This function calculates the total number of valid tile moves required to attain the goal state.

**heuristic\_function**: This method takes the user interest heuristic choice as input and returns the desired output according to users choice.

**4)A \* algorithm function**

**Puzzle:**

* Takes goal, heuristic\_function as parameters.
* This method maintains priority queues for step cost and heuristic cost.
* It maintains explored, expanded and visited nodes queues.
* If the step\_costs queue is not empty pop current node from queue and add current node to visited\_info\_set node set.Increment visited\_nodes count.
* If goal state\_of\_node has been reached, path taken to reach goal state\_of\_node is printed.
* If the priority queue has elements, and if the element is not in the visited nodes, the algorithm calls the move functions(up,down,left,right).
* After a new state is generated from the move functions, again the algorithm checks its in the visited node.
* If not then it adds to the priority queue, along with its step cost and heuristic cost.

**5)Print\_path:** This method prints the final path to reach goal state, steps taken, action of the movement, path cost(gn), heuristic cost(hn), total explored nodes and expanded nodes.

**6)Main function:** It takes the input from the user of the initial state, goal state, and the heuristic function of choice, and also creates a temporary node to initialize the algorithm function with desired choice of heuristic function. The initial state is given as the root of the tree.

**Code of the problem**

import numpy as np

visited\_nodes = 0 # explored nodes

opened\_nodes = 0 # expanded nodes

class Node():

# Constructor for node

def \_\_init\_\_(self, state\_of\_node, move\_direction, gn, hn, parent):

self.state\_of\_node = state\_of\_node

self.move\_direction = move\_direction

self.gn = gn

self.hn = hn

self.parent = parent

self.successor\_directions = {"up": None,"left": None, "right": None, "down": None}

#Function to check if the tile can move left

def left\_check(self):

w=np.where(self.state\_of\_node == 0)

zero = w[0][0]

# Boundary for left movement in grid - (0, 0), (1, 0), (2, 0)

left\_boundary=np.arange(0,7,3)

if zero in left\_boundary:

return None

else:

# If not boundary, swap blank tile with right tile

changed\_index=zero - 1

interchanging\_value = self.state\_of\_node[changed\_index]

next\_state = self.state\_of\_node.copy()

next\_state[changed\_index] = 0

next\_state[zero] = interchanging\_value

return next\_state

def right\_check(self):

w=np.where(self.state\_of\_node == 0)

zero = w[0][0]

# Boundary for right movement in grid - (0, 2), (1, 2), (2, 2)

right\_boundary=np.arange(2,9,3)

if zero in right\_boundary:

return None

else:

# If not boundary, swap blank tile with left tile

changed\_index=zero + 1

interchanging\_value = self.state\_of\_node[changed\_index]

next\_state = self.state\_of\_node.copy()

next\_state[zero + 1] = 0

next\_state[zero] = interchanging\_value

return next\_state

def up\_check(self):

w=np.where(self.state\_of\_node == 0)

zero = w[0][0]

# Boundary for up movement in grid - (0, 0), (0, 1), (0, 2l=[0,1,2]

up\_boundary=np.arange(3)

if zero in up\_boundary:

return None

else:

# If not boundary, swap blank tile with down tile

changed\_index=zero-3

interchanging\_value = self.state\_of\_node[changed\_index]

next\_state = self.state\_of\_node.copy()

next\_state[changed\_index] = 0

next\_state[zero] = interchanging\_value

return next\_state

#Function to check if the tile can move right

#Function to check if the tile can move down

def down\_check(self):

w=np.where(self.state\_of\_node == 0)

zero = w[0][0]

# Boundary for down movement in grid - (2, 0), (2, 1), (2, 2)

down\_boundary=np.arange(6,9,1)

if zero in down\_boundary:

return None

else:

# If not boundary, swap blank tile with up tile

changed\_index=zero + 3

interchanging\_value = self.state\_of\_node[changed\_index]

next\_state = self.state\_of\_node.copy()

next\_state[changed\_index] = 0

next\_state[zero] = interchanging\_value

return next\_state

#Function to calculate heuristic using number of misplaced tiles

def misplaced\_heuristic(self, next\_state, goal):

''' Function for counting misplaced tiles '''

misplaced\_tiles = 0

for i in range(9):

# if digit not in the grid position as that in the goal state\_of\_node, increase cost

if next\_state[i] != goal[i]:

misplaced\_tiles =misplaced\_tiles+ 1

return misplaced\_tiles

#Function to calculate heuristic using manhattan distance

def manhattan\_distance(self, next\_state, goal):

''' Function for calculating Manhattan distance '''

current\_node = next\_state

distance = 0

goal\_dict = {}

for i in range(9):

element=goal[i]

goal\_dict[element] = i

for i in range(9):

# count digit's distance from goal grid position

distance =distance+ abs(int(i/3) - int(goal\_dict[current\_node[i]]/3)) + abs(i % 3 - goal\_dict[current\_node[i]] % 3)

return distance

#Function to select which the user chooses and perform actions possible

def heuristic\_function(self, next\_state, goal, heuristic\_function):

if heuristic\_function == "1":

return self.misplaced\_heuristic(next\_state, goal)

if heuristic\_function == "2":

return self.manhattan\_distance(next\_state, goal)

def puzzle(self, goal, heuristic\_function):

''' A\* algorithm node explorer function '''

# Priority queues for storing step costs

step\_costs = [(self, 0)]

gn\_queue = [(0, 0)]

# Set that maintains visited\_info\_set node information

visited\_info\_set = set([])

# Variables for counting explored nodes and explanded nodes

global visited\_nodes

global opened\_nodes

visited\_nodes = 0

opened\_nodes = 0

# Until queue is empty

while step\_costs:

# maintain priority queue

step\_costs = sorted(step\_costs, key=lambda z: z[1])

gn\_queue = sorted(gn\_queue, key=lambda z: z[1])

# pop current node from queue

current\_node = step\_costs.pop(0)[0]

current\_gn = gn\_queue.pop(0)[0]

# add current node to visited\_info\_set node set

tup=tuple(current\_node.state\_of\_node.reshape(1, 9)[0])

visited\_info\_set.add(tup)

# increment explored node count

visited\_nodes =visited\_nodes+ 1

# If goal state\_of\_node has been reached, print the path taken to reach goal state\_of\_node

if np.array\_equal(current\_node.state\_of\_node, goal):

current\_node.print\_path()

return True

else:

# If goal state\_of\_node has not been reached expand nodes

if current\_node.up\_check() is not None:

next\_state = current\_node.up\_check()

if tuple(next\_state.reshape(1, 9)[0]) not in visited\_info\_set:

gn = current\_gn + 1

hn = self.heuristic\_function(next\_state, goal, heuristic\_function)

fn = gn + hn

current\_node.successor\_directions["up"] = Node(

next\_state, "up", gn, hn, current\_node)

step\_costs.append((current\_node.successor\_directions["up"], fn))

opened\_nodes += 1

gn\_queue.append((gn, fn))

if current\_node.left\_check() is not None:

next\_state = current\_node.left\_check()

if tuple(next\_state.reshape(1, 9)[0]) not in visited\_info\_set:

gn = current\_gn + 1

hn = self.heuristic\_function(next\_state, goal, heuristic\_function)

fn = gn + hn

current\_node.successor\_directions["left"] = Node(next\_state, "left", gn, hn, current\_node)

step\_costs.append((current\_node.successor\_directions["left"], fn))

opened\_nodes = opened\_nodes+1

gn\_queue.append((gn, fn))

if current\_node.right\_check() is not None:

next\_state = current\_node.right\_check()

if tuple(next\_state.reshape(1, 9)[0]) not in visited\_info\_set:

gn = current\_gn + 1

hn = self.heuristic\_function(next\_state, goal, heuristic\_function)

fn = gn + hn

current\_node.successor\_directions["right"] = Node(next\_state, "right", gn, hn, current\_node)

step\_costs.append((current\_node.successor\_directions["right"], fn))

opened\_nodes =opened\_nodes+ 1

gn\_queue.append((gn, fn))

if current\_node.down\_check() is not None:

next\_state = current\_node.down\_check()

if tuple(next\_state.reshape(1, 9)[0]) not in visited\_info\_set:

gn = current\_gn + 1

hn = self.heuristic\_function(next\_state, goal, heuristic\_function)

fn = gn + hn

current\_node.successor\_directions["down"] = Node(next\_state, "down", gn, hn, current\_node)

step\_costs.append((current\_node.successor\_directions["down"], fn))

opened\_nodes =opened\_nodes+1

gn\_queue.append((gn, fn))

def print\_path(self):

path = {"state\_of\_node": [self.state\_of\_node], "move\_direction": [ self.move\_direction], "gn": [self.gn], "hn": [self.hn]}

while self.parent:

self = self.parent

path["state\_of\_node"].append(self.state\_of\_node)

path["move\_direction"].append(self.move\_direction)

path["gn"].append(self.gn)

path["hn"].append(self.hn)

step\_count = 0

while path["state\_of\_node"]:

print("Step: ", step\_count)

print(path["state\_of\_node"].pop())

print("Action: ", path["move\_direction"].pop())

print("Path cost(gn): ", path["gn"].pop())

print("Heuristic cost(hn): ", path["hn"].pop())

print()

step\_count =step\_count+ 1

print("Total Explored nodes: ", visited\_nodes)

print("Total Expanded nodes: ", opened\_nodes)

def main():

#Take input from user for initial and goal node\_states

initial = input("Enter initial state separated by space:\n").split(' ')

initial = np.asarray(initial, dtype=int)

goal = input("Enter goal state separated by space:\n").split(' ')

goal = np.asarray(goal, dtype=int)

# Take input for which heuristic to use

heuristic\_function = input("Press '1' for Misplaced Tiles AND '2' for Manhattan Distance.\n")

# Temp node for calling function from Node class for initializing heuristic

temp\_node = Node(goal, None, 0, 0, None)

heuristic\_initialize = temp\_node.heuristic\_function(initial, goal, heuristic\_function)

# Generating root node for tree initialization

root = Node(initial, None, 0, heuristic\_initialize, None)

root.puzzle(goal, heuristic\_function)

main()

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Initial Node** | **Heuristic Manhattan Distance** | | **Heuristic misplaced tiles** | | **Goal Node** |
| **Expanded**  **nodes** | **Explored**  **nodes** | **Expanded**  **nodes** | **Explored**  **nodes** |
| 1 2 3 0 4 6 7 5 8 | 8 | 4 | 8 | 4 | 1 2 3 4 5 6 7 8 0 |
| 2 8 1 3 4 6 7 5 0 | 12 | 7 | 14 | 8 | 3 2 1 8 0 4 7 5 6 |
| 1 3 2 4 5 6 0 8 7 | 2360 | 1459 | 7112 | 4488 | 1 2 3 4 5 6 7 8 0 |
| 3 5 1 4 2 6 7 8 0 | 625 | 381 | 1126 | 695 | 1 3 5 4 2 6 7 8 0 |
| 1 2 3 8 0 4 7 6 5 | 55 | 32 | 70 | 40 | 2 8 1 0 4 3 7 6 5 |
| 1 2 3 7 4 5 6 8 0 | 22 | 12 | 43 | 24 | 1 2 3 8 6 4 7 5 0 |

**Testcases summary**