UNC CHARLOTTE
College of Computing and Informatics

PROJECT –1

8 – PUZZLE USING A* ALGORITHM

DEPARTMENT OF COMPUTER SCIENCE
ITCS 6150 - Intelligent Systems

Submitted To

Dewan T. Ahmed, Ph.D.

Submitted By

Gowtham Bharadwaj	**Medha Nagaraj**
801101552	**801101751**
gbharadw@uncc.edu	**mnagara1@uncc.edu**

8 – PUZZLE USING A* ALGORITHM

1. Table of Contents

PROBLEM FORMULATION

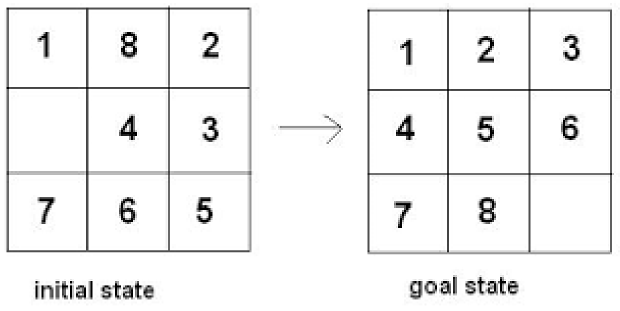
1.1 INTRODUCTION

What is 8 – Puzzle Problem?

The puzzle consists of an area divided into a grid (for example 3 by 3 for the 8-puzzle). Each grid on the square is a tile and one of the squares always remain empty. Hence, there are 8 tiles for the 8-puzzle problem. The tiles are numbered from 1 to 8 for the puzzle so that each tile can be uniquely identified.

The tile which is next to the empty grid square can be moved into the empty space, leaving its previous position empty in turn. The tiles can move in either horizontally or vertically. The aim of the puzzle is to achieve a given goal state from a given initial configuration by sliding the individual tiles around the grid in minimum number of moves possible. The optimal solution for the 8-puzzle problem is **NP Hard**.

Below is the pictorial description for the 8-puzzle problem wherein the puzzle computes from the initial state to the goal state. Here in our project, both the initial and the final configuration states are user defined and the algorithm computes the solution for the same.



A* Search technique:

The A* Search Algorithm is the best-known form of Best First search. It is widely used for path findings and graph traversals. A priority queue is used wherein the elements are stored based on their priority. It expands most promising paths first and avoids the ones which are expensive by using recursion method.

There are basically two heuristics approaches used in the algorithm:

- **Heuristic 1: Misplaced tiles**
  It considers whether a tile has been misplaced from its original location.

- **Heuristic 2: Manhattan Distance**
  The mathematical formulation is shown below:
  f(n) = g(n) + h(n), where
  - g(n) the cost (so far) to reach the node
  - h(n) estimated cost to get from the node to the goal
  - f(n) estimated total cost of path through n to goal.

1.2 ALGORITHM PSEUDOCODE

```
Boolean
Solution::search(PriorityQueue Q)
{
    if Q.isEmpty( ) then
        return false
    puzzle = Q.extract( )
    if puzzle.isGoal( )
        return true
    successors = puzzle.expand( ) // all possible successors to puzzle
    for each suc in successors do
    Q.insert(suc)
    if search(Q)
        return true
    else
        return false
}
```

PROGRAM STRUCTURE

2.1 FUNCTIONS/PROCEDURES

The code implements A* Search algorithm using Manhattan Distance Heuristic and Misplaced Tiles Heuristic. Codes have separate implementation.

- Language Used – Python
- Data Structure Used – Priority queues.

The following functions are used in the code:

- misplacedCost() – For calculating misplaced tiles heuristic (Misplaced Tiles function)
- manhDist() – For calculation of Manhattan Distance (Manhattan distance function)
- coordinate() – assign each digit of the state, the coordinate to calculate Manhattan distance
- all() – to set goal elements
- optimalSteps() – to calculate best optimal steps to goal
- calculate () – to compare initial state with goal state and make moves towards the goal
- body() – to read input/output states

2.2 GOLBAL/LOCAL VARIABLES

The local variables within function are:

➔ **misplacedCost: cost –** Calculates misplaced heuristic

➔ **manhDist**: **mdist –** Calculates Manhattan distance

➔ **coordinate:**
  - **cord** – for the array
  - **x, y** – for the range

➔ **optimalSteps:**
  - **optimal** – storing optimal states
  - **length**- length of the state

- ➔ **calculate –**
  - **moves**: up, dn, rt, lt for moving up, down, right and left positions respectively
  - **dtype**: distance type
  - **diststate**:
  - **goalcord**: coordinate of goal state
  - **parent**: parent state initialization
  - **g_n**: cost
  - **h_n**: heuristic calculation
  - **f_n**: function
  - **dtpri**: distance priority
  - **dgpri**: priority of digits
  - **pos**: position
  - **loc**: location
  - **succ**: successor state
  - **q**: priority queue

- ➔ **body:** Call to the body function internally has calls to the above-mentioned functions

- ➔ **The global variables are:**
  - **goal**: Store or pass the goal board configuration
  - **board**: Store or pass the initial board configuration
  - **state**: Holds the current state of the board

2.3 LOGIC

In the code implemented, we have used the priority queues as the data structures. The indexes calculate coordinates of each digit of the input and output and compare the difference and sort them accordingly. We repeat this until our goal is reached by generating the successor nodes and comparing them further. We calculate the heuristics based on these coordinates and the following information are used:

- Goal achieved status
- Total number of nodes generated
- Total number of nodes explored
- Total number of optimized steps
- Total Time Taken

The code works perfectly for states which can be solved, but it goes into a loop for the states which cannot be solved.

CODE FOR EACH HEURISTIC

3.1 Manhattan Distance:

```
import numpy as np

from copy import deepcopy

import time


# This function is used to calculate the Manhattan distance for each digit with reference to the provided goal state

def manhDist(s, g):

   mdist = abs(s // 3 - g // 3) + abs(s % 3 - g % 3)

   return sum(mdist[1:])


# Assign each digit the coordinate that it needs to calculate the Manhattan distance

def coordinate(s):

   cord = np.array(range(9))

   for x, y in enumerate(s):

      cord[y] = x

   return cord


def all(s):

   set=string
```

```
    return 0 not in [c in s for c in set]


#Generate the board list as the sequence in optimized steps
def optimalSteps(state):
    optimal = np.array([], int).reshape(-1, 9)
    length = len(state) - 1
    while length != -1:
        optimal = np.insert(optimal, 0, state[length]['board'], 0)
        length = int(state[length]['parent'])
    return optimal.reshape(-1, 3, 3)
# Calculate the 8-puzzle board using Manhattan distance
def calculate(board, goal):
    moves = np.array(   [   ('up', [0, 1, 2], -3),
                            ('dn', [6, 7, 8],  3),
                            ('lt', [0, 3, 6], -1),
                            ('rt', [2, 5, 8],  1)
                            ],
                dtype=  [  ('move',  str, 1),
                           ('pos',   list),
                           ('delta', int)
                           ]
                        )
diststate = [ ('board',  list),
                ('parent', int),
                ('g_n',    int),
                ('h_n',    int)  ]

    goalcord = coordinate(goal)
    # Initial state values
    parent = -1 #Initial parent state
    g_n    = 0
    h_n    = manhDist(coordinate(board), goalcord) # Calculate Manhattan
distance between the initial state and the goal state
    state = np.array([(board, parent, g_n, h_n)], diststate) # Initializing state

    #Initialize the priority queue
    dtpri = [  ('pos', int),
                ('f_n', int)
                ]

    dgpri = np.array( [(0, h_n)], dtpri)
    while True:
        dgpri = np.sort(dgpri, kind='mergesort', order=['f_n', 'pos'])     #Sort the
priority queue
        pos, f_n = dgpri[0]    #Choose the first element from the sorted list to
explore
```

```
    dgpri = np.delete(dgpri, 0, 0)      #Remove the element that we have explored
    board, parent, g_n, h_n = state[pos]
    board = np.array(board)
    loc = int(np.where(board == 0)[0])      #locate '0' (blank)
    g_n = g_n + 1      #Increase the cost of g(n) by 1
    for m in moves:
        if loc not in m['pos']:
            succ = deepcopy(board)      # Generate the new state as a copy of the current state
            succ[loc], succ[loc + m['delta']] = succ[loc + m['delta']], succ[loc]      #Move the digit

            if ~(np.all(list(state['board']) == succ, 1)).any():    #Check if the new configuration is not generated before

                h_n = manhDist(coordinate(succ), goalcord)      #Calculate the Manhattan distance
                q = np.array(   [(succ, pos, g_n, h_n)], diststate)     # Generate and add the new state to the list
                state = np.append(state, q, 0)
                f_n = g_n + h_n      #Calculate the value of f(n)
                q = np.array([(len(state) - 1, f_n)], dtpri)    #Add to the priority queue
                dgpri = np.append(dgpri, q, 0)
            if np.array_equal(succ, goal):     #Check if this is the desired goal
state
              print("\n Goal reached")
              return state, len(dgpri)
return state, len(dgpri)
#########################################################
def body():
  print()
  print ("Solution to 8-Puzzle problem using Manhattan distance
approach\n")
  print("Feed the initial board configuration: (Leave a space between each
input value)")   #Get the initial state
  string = [int(x) for x in input().split()]     # Enter the desired input state.
Leave a blank space in between the input characters. Add '0' for the blank
space
  print("\n Feed the goal board configuration: (Leave a space between each
input value)")
  goal = [int(x) for x in input().split()]      # Enter the desired input state.
Leave a blank space in between the input characters. Add '0' for the blank
space
  if len(string) != 9:
    print('Enter only 9 values')
    return
  board = np.array(list(map(int, string)))
```

```
    print (board)
 tstart=time.time()
    state, explored = calculate(board, goal)
    tend=time.time()
    print()
    print('Total number of nodes generated:', len(state))
    print('Total number of nodes explored: ', len(state) - explored)
    print()
    # Generate the optimized steps and display those steps
    optimal = optimalSteps(state)
    print('Total optimized steps:', len(optimal) - 1)
    print()
    print(optimal)
    print()
    print ("The algorithm took " + str((tend-tstart) * 1000)  + " ms of time. to 
compute")
#############################################################
# Main portion of the code
body()
```

3.2 Misplaced Tiles:

```
import numpy as np

from copy import deepcopy

import time


def misplacedCost(s, g): #Function to calculate the misplaced tiles

        cost = np.sum(s != g)-1

        if cost > 0:

            return cost

        else:

            return 0


def all(s):

    set=string


    return 0 not in [c in s for c in set]


#Generate the board list as the sequence in optimized steps

def optimalSteps(state):

    optimal = np.array([], int).reshape(-1, 9)

    length = len(state) - 1

    while length != -1:

        optimal = np.insert(optimal, 0, state[length]['board'], 0)
        length = int(state[length]['parent'])
    return optimal.reshape(-1, 3, 3)


# solve the board
def calculate(board, goal):
    moves = np.array(   [   ('up', [0, 1, 2], -3),
                            ('dn', [6, 7, 8],  3),
                            ('lt', [0, 3, 6], -1),
                            ('rt', [2, 5, 8],  1)
                            ],
                dtype=  [  ('move',  str, 1),
                           ('pos',   list),
                           ('delta', int)
                           ]
                        )

    diststate = [ ('board',  list),
                ('parent', int),
                ('g_n',     int),
                ('h_n',     int)
                ]

    # Initial state values
    parent = -1     #Initial parent state

    g_n     = 0

    h_n     = misplacedCost(board, goal)    #Calculate misplaced tiles between the initial state and the goal state

    state = np.array([(board, parent, g_n, h_n)], diststate)    #Initializing state


    #Initialize the priority queue

    dtpri = [  ('pos', int),

                ('f_n', int)

                ]


    dgpri = np.array( [(0, h_n)], dtpri)

    while True:

        dgpri = np.sort(dgpri, kind='mergesort', order=['f_n', 'pos'])      #Sort the priority queue

        pos, f_n = dgpri[0]     #Choose the first element from the sorted list to explore

        dgpri = np.delete(dgpri, 0, 0)      #Remove the element that we have explored

        board, parent, g_n, h_n = state[pos]

        board = np.array(board)

        loc = int(np.where(board == 0)[0])      #locate '0' (blank)

        g_n = g_n + 1      #Increase the cost of g(n) by 1

        for m in moves:
          if loc not in m['pos']:
             succ = deepcopy(board)    #Generate the new state as a copy of the
 current state
             succ[loc], succ[loc + m['delta']] = succ[loc + m['delta']], succ[loc]
 #Move the digit

             if ~(np.all(list(state['board']) == succ, 1)).any():    #Check if the new
 configuration is not generated before
                h_n = misplacedCost(succ, goal)    #Calculate the Misplaced tiles
                q = np.array(   [(succ, pos, g_n, h_n)], diststate)    #Generate and add
 the new state to the list
                state = np.append(state, q, 0)
                f_n = g_n + h_n                               #Calculate the value of f(n)
                q = np.array([(len(state) - 1, f_n)], dtpri)    #Add to the priority
 queue
                dgpri = np.append(dgpri, q, 0)

                if np.array_equal(succ, goal):    #Check if this is the desired goal
 state
                   print('Goal reached')
                   return state, len(dgpri)
    return state, len(dgpri)
 #########################################################
 def body():
    print()
    alist = []
    print ("Using Misplaced Tiles, solving the 8 puzzle:")
    print("Feed the initial board configuration: (Leave a space between each input value)")   #Get the initial state
    string = [int(x) for x in input().split()]  # Enter the desired input state. Leave a blank space in between the input characters. Add '0' for the blank space
    print("\n Feed the goal board configuration: (Leave a space between each input value)")
    alist = [int(x) for x in input().split()]   # Enter the desired input state. Leave a blank space in between the input characters. Add '0' for the blank space
    goal=alist

    if len(string) != 9:
        print('Enter only 9 values')
        return

    board = np.array(list(map(int, string)))
    print (board)

    tstart=time.time()
    state, explored = calculate(board, goal)
    tend=time.time()
    print()
    print('Total number of nodes generated:', len(state))

    print('Total number of nodes explored: ', len(state) - explored)

    print()

    # Generate the optimized steps and display those steps

    optimal = optimalSteps(state)

    print('Total optimized steps:', len(optimal) - 1)

    print()

    print(optimal)

    print()

    print ("The algorithm took " + str((tend-tstart) * 1000)  + " ms of time. to 
compute")


################################################################

# Main portion of the code


body()
```

SAMPLE OUTPUT

4.1 Manhattan Distance

1. Solution to 8-Puzzle problem using Manhattan distance approach

Feed the initial board configuration: (Leave a space between each input value)

1 2 3 7 4 5 6 8 0

Feed the goal board configuration: (Leave a space between each input value)

1 2 3 8 6 4 7 5 0

[1 2 3 7 4 5 6 8 0]

Goal reached

Total number of nodes generated: 19

Total number of nodes explored: 9

Total optimized steps: 8

[1 2 3] [7 4 5] → [6 8 0]	[1 2 3] [7 4 0] → [6 8 5]	[1 2 3] [7 0 4] → [6 8 5]	[1 2 3] [7 8 4] → [6 0 5]	[1 2 3] [7 8 4] → [0 6 5]	[1 2 3] [0 8 4] → [7 6 5]	[1 2 3] [8 0 4] → [7 6 5]
[1 2 3] [8 6 4] → [7 0 5]	[1 2 3] [8 6 4] [7 5 0]					

The algorithm took 2.961874008178711 MS of time. to compute

2. Solution to 8-Puzzle problem using Manhattan distance approach

Feed the initial board configuration: (Leave a space between each input value)

2 8 1 3 4 6 7 5 0

Feed the goal board configuration: (Leave a space between each input value)

3 2 1 8 0 4 7 5 6

[2 8 1 3 4 6 7 5 0]

Goal reached

Total number of nodes generated: 13

Total number of nodes explored: 6

Total optimized steps: 6

[2 8 1] [3 4 6] → [7 5 0]	[2 8 1] [3 4 0] → [7 5 6]	[2 8 1] [3 0 4] → [7 5 6]	[2 0 1] [3 8 4] → [7 5 6]	[0 2 1] [3 8 4] → [7 5 6]	[3 2 1] [0 8 4] → [7 5 6]	[3 2 1] [8 0 4] [7 5 6]

The algorithm took 2.899169921875 ms of time. to compute

3. Solution to 8-Puzzle problem using Manhattan distance approach

Feed the initial board configuration: (Leave a space between each input value)

1 2 3 4 5 6 7 8 0

Feed the goal board configuration: (Leave a space between each input value)

1 2 3 0 6 8 4 7 5

[1 2 3 4 5 6 7 8 0]

Goal reached

Total number of nodes generated: 17

Total number of nodes explored: 9

Total optimized steps: 7

[1 2 3] [4 5 6] → [7 8 0]	[1 2 3] [4 5 6] → [7 0 8]	[1 2 3] [4 0 6] → [7 5 8]	[1 2 3] [4 6 0] → [7 5 8]	[1 2 3] [4 6 8] → [7 5 0]	[1 2 3] [4 6 8] → [7 0 5]	[1 2 3] [4 6 8] → [0 7 5]
[1 2 3] [0 6 8] [4 7 5]						

The algorithm took 3.902912139892578 ms of time. to compute.

4. Solution to 8-Puzzle problem using Manhattan distance approach

Feed the initial board configuration: (Leave a space between each input value)

4 5 1 2 0 8 7 6 3

Feed the goal board configuration: (Leave a space between each input value)

4 0 1 2 5 3 7 8 6

[4 5 1 2 0 8 7 6 3]

Goal reached

Total number of nodes generated: 13

Total number of nodes explored: 6

Total optimized steps: 5

[4 5 1] [2 0 8] → [7 6 3]	[4 5 1] [2 8 0] → [7 6 3]	[4 5 1] [2 8 3] → [7 6 0]	[4 5 1] [2 8 3] → [7 0 6]	[4 5 1] [2 0 3] → [7 8 6]	[4 0 1] [2 5 3] [7 8 6]

The algorithm took 1.9671916961669922 ms of time. to compute

4.2 Misplaced Tiles:

1. Using Misplaced Tiles, solving the 8 puzzles:

Feed the initial board configuration: (Leave a space between each input value)

2 8 1 3 4 6 7 5 0

Feed the goal board configuration: (Leave a space between each input value)

3 2 1 8 0 4 7 5 6

[2 8 1 3 4 6 7 5 0]

Goal reached

Total number of nodes generated: 15

Total number of nodes explored: 7

Total optimized steps: 6

[2 8 1] [3 4 6] → [7 5 0]	[2 8 1] [3 4 0] → [7 5 6]	[2 8 1] [3 0 4] → [7 5 6]	[2 0 1] [3 8 4] → [7 5 6]	[0 2 1] [3 8 4] → [7 5 6]	[3 2 1] [0 8 4] → [7 5 6]	[3 2 1] [8 0 4] [7 5 6]

The algorithm took 2.9914379119873047 ms of time to compute.

2. Using Misplaced Tiles, solving the 8 puzzles:

Feed the initial board configuration: (Leave a space between each input value)

1 2 3 7 4 5 6 8 0

Feed the goal board configuration: (Leave a space between each input value)

1 2 3 8 6 4 7 5 0

[1 2 3 7 4 5 6 8 0]

Goal reached

Total number of nodes generated: 44

Total number of nodes explored: 23

Total optimized steps: 8

[1 2 3] [7 4 5] → [6 8 0]	[1 2 3] [7 4 0] → [6 8 5]	[1 2 3] [7 0 4] → [6 8 5]	[1 2 3] [7 8 4] → [6 0 5]	[1 2 3] [7 8 4] → [0 6 5]	[1 2 3] [0 8 4] → [7 6 5]	[1 2 3] [8 0 4] → [7 6 5]
[1 2 3] [8 6 4] → [7 0 5]	[1 2 3] [8 6 4] [7 5 0]					

The algorithm took 15.610456466674805 ms of time. to compute.

3. Using Misplaced Tiles, solving the 8 puzzles:

Feed the initial board configuration: (Leave a space between each input value)

1 2 3 4 5 6 7 8 0

Feed the goal board configuration: (Leave a space between each input value)

1 2 3 0 6 8 4 7 5

[1 2 3 4 5 6 7 8 0]

Goal reached

Total number of nodes generated: 32

Total number of nodes explored: 16

Total optimized steps: 7

[1 2 3] [4 5 6] → [7 8 0]	[1 2 3] [4 5 6] → [7 0 8]	[1 2 3] [4 0 6] → [7 5 8]	[1 2 3] [4 6 0] → [7 5 8]	[1 2 3] [4 6 8] → [7 5 0]	[1 2 3] [4 6 8] → [7 0 5]	[1 2 3] [4 6 8] → [0 7 5]
[1 2 3] [0 6 8] [4 7 5]						

The algorithm took 15.610456466674805 ms of time to compute.

4. Using Misplaced Tiles, solving the 8 puzzles:

Feed the initial board configuration: (Leave a space between each input value)

4 5 1 2 0 8 7 6 3

Feed the goal board configuration: (Leave a space between each input value)

4 0 1 2 5 3 7 8 6

[4 5 1 2 0 8 7 6 3]

Goal reached

Total number of nodes generated: 15

Total number of nodes explored: 7

Total optimized steps: 5

[4 5 1] [2 0 8] → [7 6 3]	[4 5 1] [2 8 0] → [7 6 3]	[4 5 1] [2 8 3] → [7 6 0]	[4 5 1] [2 8 3] → [7 0 6]	[4 5 1] [2 0 3] → [7 8 6]	[4 0 1] [2 5 3] [7 8 6]

The algorithm took 15.607357025146484 ms of time. to compute.

CONCLUSION

From the above experiment in the project we found out that the Informed Search Strategies perform better in comparison to the Uninformed Search Strategies. The heuristic information gained during the search helps the program to perform better. Both the heuristics are admissible because they never overestimate the path costs which is the estimated and the total. This happens by estimating the solution using a relaxed version of the puzzle. Though the algorithm is complicated to implement, the solution it provides is reliable, efficient and usable for very hard puzzles.

On running the program for the above input/output configuration for Manhattan distance and the misplaced tiles approaches, we could infer which is the better algorithm between the two. Manhattan distance method clearly generates lesser number of nodes to reach the specified goal state, whereas misplaced tiles method generates greater number of nodes and takes greater time than the Manhattan tiles method. Thus, we can conclude that the Manhattan Distance Method is a better approach to solve the 8 – Puzzle Problem.

REFERENCES

[1] 8-Puzzle Problem, Heuristics, Wikipedia.

[2] 8 puzzle Problem using Branch and Bound, Geeks for Geeks