

8 – Puzzle using A* Algorithm

PROJECT DOCUMENTATION REPORT

PROGRAMMING PROJECT 1
ITCS 6150 - Intelligent Systems

DEPARTMENT OF COMPUTER SCIENCE

SUBMITTED TO
Dewan T. Ahmed, Ph.D.

SUBMITTED BY:

Urma Haldar
801074045

Sahil Sood
801082267



Table of Contents

1 PROBLEM FORMULATION	2
1.1 Introduction	2
1.2 Algorithm Pseudocode	3
2 PROGRAM STRUCTURE	4
2.1 Global/ Local Variables	4
2.2 Functions/Procedures	4
2.3 Logic	5
3 CODE FOR EACH HEURISTIC	6
3.1 Manhattan Distance	6
3.2 Misplaced Tiles	12
4 SAMPLE OUTPUTS	17
4.1 Manhattan Distance	17
4.2 Misplaced Tiles	18
CONCLUSIONS	19
REFERENCES	20

PROBLEM FORMULATION

1.1 INTRODUCTION

What is 8 – Puzzle Problem?

The 8-puzzle is a sliding puzzle that consists of a frame of numbered (1-8) square tiles in random order with one tile missing. The goal is usually to place those numbers in correct numerical order. *We can slide four adjacent (left, right, above and below) tiles into the empty space.* The more general n-puzzle is a classical problem which can be solved using graph search techniques. The problem of finding the optimal solution is NP-hard.

8		6
5	4	7
2	3	1

	1	2
3	4	5
6	7	8

In our project we have user defined initial as well as output state.

For example,

Initial configuration	Final configuration																		
<table><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>5</td><td>6</td><td></td></tr><tr><td>7</td><td>8</td><td>4</td></tr></table>	1	2	3	5	6		7	8	4	<table><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>5</td><td>8</td><td>6</td></tr><tr><td></td><td>7</td><td>4</td></tr></table>	1	2	3	5	8	6		7	4
1	2	3																	
5	6																		
7	8	4																	
1	2	3																	
5	8	6																	
	7	4																	

A* Search technique:

The A* searching algorithm is a recursive algorithm that continuously calls itself until a winning state is found. It uses the sum of moves to current step and Manhattan priority function as cost function. A priority queue of search node containing number of moves, current board and previous search node is created. For each move, the search node with minimum cost is dequeued and neighboring nodes of this search node are then inserted into the priority queue. The sequence of moves using fewest number of moves to solve the puzzle is obtained when the goal board is ultimately dequeued (total number of moves is always at least its priority).

1.2 ALGORITHM PSEUDOCODE

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

PROGRAM STRUCTURE

2.1 FUNCTIONS/PROCEDURES

Our code implements A* Search algorithm using Manhattan Distance Heuristic Calculation and also Misplaced Tiles Heuristic Calculation (separate codes) implementation using Python and priority queues.

Our codes handles just the heuristic calculation differently. The rest structure remains same. We use the following functions within the code:

h_misplaced_cost() – For calculating misplaced tiles heuristic (used in misplaced_tiles code)
mhd() – For calculation of Manhattan Distance (used in the one using Manhattan distance)
coor() – assign each digit of the state, the coordinate to calculate Manhattan distance
all() – to set goal elements
stepsoptimal() – to calculate best optimal steps to goal
solve() – to compare initial state with goal state and make moves towards the goal
main() – to read input/output states

2.2 GLOBAL/LOCAL VARIABLES

We use various global/local variables in our program:

The local variables within function are:

h_misplaced_cost – cost – Calculates misplaced heuristic

mhd – m – calculated manhattan distance

coor –

- c – for the array
- x, y – for the range

all -

stepsoptimal –

- optimal – storing optimal states
- last- length of state

solve –

- moves- up,down,right,left moves
- dtype- distance type
- dtstate -
- goalc – coordinate of goal state
- parent – parent state initialization

8 – PUZZLE USING A* ALGORITHM

- gn - cost
- hn – heuristic calculation
- dtpriority – distance priority
- priority – priority of digits
- pos - position
- fn - function
- loc - location
- succ – successor state
- q – priority queue

main-

The global variables are:

- goal - storing/passing goal state
- board –storing/passing initial state
- state – current state

2.3 LOGIC

We use priority queues as data structure. We use the indexes to calculate coordinates of each digit of the input and output and compare the difference and sort accordingly. We repeat this until our goal is reached by generating successor nodes and further comparing them. We calculate the heuristic based on these coordinates. It calculates the following information using both the heuristics:

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

The code works perfectly for solvable states, but for unsolvable ones, it goes on in a loop.

CODE FOR EACH HEURISTIC

3.1 Manhattan Distance

```
import numpy as np
from copy import deepcopy
import time

# calculate Manhattan distance for each digit as per goal
def mhd(s, g):
    m = abs(s // 3 - g // 3) + abs(s % 3 - g % 3)
    return sum(m[1:])

# assign each digit the coordinate to calculate Manhattan distance
def coor(s):
    c = np.array(range(9))
    for x, y in enumerate(s):
        c[y] = x
    return c

def all(s):
    #set = '012345678'
    set=string

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

# generate board list as per optimized steps in sequence
```

```

def stepsoptimal(state):
    optimal = np.array([], int).reshape(-1, 9)
    last = len(state) - 1
    while last != -1:
        optimal = np.insert(optimal, 0, state[last]['board'], 0)
        last = int(state[last]['parent'])
    return optimal.reshape(-1, 3, 3)

# solve the board
def solve(board, goal):
    #
    moves = np.array( [ ('u', [0, 1, 2], -3),
                        ('d', [6, 7, 8], 3),
                        ('l', [0, 3, 6], -1),
                        ('r', [2, 5, 8], 1)
                      ],
                      dtype= [ ('move', str, 1),
                                ('pos', list),
                                ('delta', int)
                              ]
    )

    dtstate = [ ('board', list),
                ('parent', int),
                ('gn', int),
                ('hn', int)
              ]

    goalc = coor(goal)

```


8 – PUZZLE USING A* ALGORITHM

```
# initial state values
parent = -1 #initial parent state
gn = 0
hn = mhd(coor(board), goalc) #calculating manhattan distance
between initial and goal state
state = np.array([(board, parent, gn, hn)], dtype) #initializing state

#priority queue initialization
dtpriority = [ ('pos', int),
               ('fn', int)
               ]

priority = np.array( [(0, hn)], dtpriority)
#
while True:
    priority = np.sort(priority, kind='mergesort', order=['fn', 'pos']) #
sort priority queue
    pos, fn = priority[0] # pick out first from sorted to
explore
    priority = np.delete(priority, 0, 0) # remove from queue what we
are exploring
    board, parent, gn, hn = state[pos]
    board = np.array(board)
    loc = int(np.where(board == 0)[0]) # locate 'o' (blank)
    gn = gn + 1 # increase cost g(n) by 1
    for m in moves:
        if loc not in m['pos']:
            succ = deepcopy(board) # generate new state as copy of
current
```

```

succ[loc], succ[loc + m['delta']] = succ[loc + m['delta']],
succ[loc]# do the move

if ~(np.all(list(state['board']) == succ, 1)).any():#check if new
(not repeat)

hn = mhd(coor(succ), goalc)                # calculate
Manhattan distance
q = np.array( [(succ, pos, gn, hn)], dtstate)  # generate
and add new state in the list
state = np.append(state, q, 0)
fn = gn + hn                                # calculate f(n)
q = np.array([(len(state) - 1, fn)], dtpriority) # add to
priority queue
priority = np.append(priority, q, 0)

if np.array_equal(succ, goal):                # is this goal
state?
    print('Goal achieved!')
    return state, len(priority)

return state, len(priority)

def main():
    print()
    print ("Using Manhattan Distance, solving the 8 puzzle:")

```

```
print("Please enter the goal state: (Please enter a space inbetween
numbers)")

goal = [int(x) for x in input().split()]# read goal state

print('Enter initial board: (Please enter a space inbetween numbers)
')# read initial state

string = [int(x) for x in input().split()]# read goal state


if len(string) != 9:
    print('incorrect input')
    return


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


t1=time.time()
state, explored = solve(board, goal)
t2=time.time()
print()
print('Total generated:', len(state))
print('Total explored: ', len(state) - explored)
print()
# generate and show optimized steps
optimal = stepsoptimal(state)
print('Total optimized steps:', len(optimal) - 1)
print()
print(optimal)
```

8 – PUZZLE USING A* ALGORITHM

```
print()
```

```
print ("The algorithm took " + str((t2-t1) * 1000) + " ms of time.")
```

```
# Main Program
```

```
if __name__ == '__main__':
```

```
    main()
```

3.1 Misplaced Tiles

```
import numpy as np
from copy import deepcopy
import time

def h_misplaced_cost(s, g): # calculating misplaced tiles
    cost = np.sum(s != g)-1
    #print (cost)# minus 1 to exclude the empty tile
    if cost > 0:
        return cost
    else:
        return 0

def all(s):
    #set = '012345678'
    set=string

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

# generate board list as per optimized steps in sequence
def genoptimal(state):
    optimal = np.array([], int).reshape(-1, 9)
    last = len(state) - 1
    while last != -1:
        optimal = np.insert(optimal, 0, state[last]['board'], 0)
        last = int(state[last]['parent'])
    return optimal.reshape(-1, 3, 3)
```

solve the board

def solve(board, goal):

#

moves = np.array([('u', [0, 1, 2], -3),

('d', [6, 7, 8], 3),

('l', [0, 3, 6], -1),

('r', [2, 5, 8], 1)

],

dtype= [('move', str, 1),

('pos', list),

('delta', int)

]

)

dtstate = [('board', list),

('parent', int),

('gn', int),

('hn', int)

]

initial state values

parent = -1 #initial parent state

gn = 0

hn = h_misplaced_cost(board, goal) #calculating misplaced tiles

between initial and goal state

state = np.array([(board, parent, gn, hn)], dtstate) #initializing state

#priority queue initialization

```

dtpriority = [ ('pos', int),
                     ('fn', int)
               ]

priority = np.array( [(0, hn)], dtpriority)
#
while True:
    priority = np.sort(priority, kind='mergesort', order=['fn', 'pos']) #
sort priority queue
    pos, fn = priority[0] # pick out first from sorted to
explore
    priority = np.delete(priority, 0, 0) # remove from queue what we
are exploring
    board, parent, gn, hn = state[pos]
    board = np.array(board)
    loc = int(np.where(board == 0)[0]) # locate '0' (blank)
    gn = gn + 1 # increase cost g(n) by 1
    for m in moves:
        if loc not in m['pos']:
            succ = deepcopy(board) # generate new state as copy of
current
            succ[loc], succ[loc + m['delta']] = succ[loc + m['delta']],
succ[loc]# do the move

            if ~(np.all(list(state['board']) == succ, 1)).any():# check if new
(not repeat)
                hn = h_misplaced_cost(succ, goal) # calculate
Misplaced tiles

```

```

        q = np.array( [(succ, pos, gn, hn)], dtstate)    # generate
and add new state in the list

```

```

        state = np.append(state, q, 0)

```

```

        fn = gn + hn                                # calculate f(n)

```

```

        q = np.array([(len(state) - 1, fn)], dtpriority)    # add to
priority queue

```

```

        priority = np.append(priority, q, 0)

```

```

        if np.array_equal(succ, goal):                # is this goal
state?

```

```

            print('Goal achieved!')

```

```

            return state, len(priority)

```

```

    return state, len(priority)

```

```

def main():

```

```

    print()

```

```

    alist = []

```

```

    print ("Using Misplaced Tiles, solving the 8 puzzle:")

```

```

    print("Please enter the goal state: (please enter a space inbetween
numbers)")

```

```

    alist = [int(x) for x in input().split()]# read goal state

```

```

    goal=alist

```

```

    print('Enter initial board (please enter a space inbetween numbers):
')# read initial state

```

```

    string = [int(x) for x in input().split()]# read goal state

```

```

    if len(string) != 9:

```



```
    print('incorrect input')
    return

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


t1=time.time()
state, explored = solve(board, goal)
t2=time.time()
print()
print('Total generated:', len(state))
print('Total explored: ', len(state) - explored)
print()
# generate and show optimized steps
optimal = genoptimal(state)
print('Total optimized steps:', len(optimal) - 1)
print()
print(optimal)
print()
print ("The algorithm took " + str((t2-t1) * 1000) + " ms of time.")


# Main Program

if __name__ == '__main__':
    main()
```

SAMPLE OUTPUT

4.1 Manhattan Distance

1. Please enter the goal state:

→ 3 2 1 8 0 4 7 5 6

Enter initial board:

→ 2 8 1 3 4 6 7 5 0

Goal achieved!

Total generated: 13

Total 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 1.979827880859375 ms of time.

2. Please enter the goal state:

→ 1 2 3 8 6 4 7 5 0

Enter initial board:

→ 1 2 3 7 4 5 6 8 0

Goal achieved!

Total generated: 19

Total 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 3.980875015258789 ms of time.

4.2 Misplaced Tiles

1. Please enter the goal state:

→ 1 2 3 8 6 4 7 5 0

Enter initial board:

→ 1 2 3 7 4 5 6 8 0

Goal achieved!

Total generated: 44

Total 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 6.996631622314453 ms of time.

2. Please enter the goal state:

→ 3 2 1 8 0 4 7 5 6

Enter initial board:

→ 2 8 1 3 4 6 7 5 0

Goal achieved!

Total generated: 15

Total 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 2.9981136322021484 ms of time.

CONCLUSION

The informed searches performs better as compared to uninformed search due to the information gained about the search in their heuristics. Both heuristics were admissible because they never overestimated the path costs (estimated and total). They did this by estimating the solution using a relaxed version of the puzzle. The algorithms were slightly more complex to implement, but reliably provided efficient and usable solutions for difficult puzzles.

Finally, from the sample outputs of both the informed techniques, we can infer which A* method is better for optimal use. Manhattan clearly generates less nodes to give output whereas misplaced tiles generates more number of nodes and also, it takes lesser time than misplaced tiles. Hence, Manhattan Distance is a better approach.

REFERENCES

- [1] 8-Puzzle Problem, Heuristics, Wikipedia. Accessed: October 2018.
- [2] 8 puzzle Problem using Branch And Bound, Geeks for Geeks Accessed: October 2018.
- [3] hrishikeshparanjape/8-puzzle, Github. Accessed: October 2018.