CS170: Introduction to Artificial Intelligence, Dr. Eamonn Keogh
Project 1 Eight Puzzle
Tongyuan He
SID: 862217840
Email: The033@ucr.edu
Date: Nov 8,2021

Consult References

1. Heuristic search Lecture slides from Dr. Eamonn Keogh
   "3_Heuristic_search.pptx". pg 20 - 42
2. Project formatting example from Dr. Eamonn Keogh
   Project_1_The_Eight_Puzzle_CS_170_2021.pdf

3. N-Puzzle node expansion visualizer for testing correctness of depth, node, queue.
   https://tristanpenman.com/demos/n-puzzle/
4. Solvable Eight Puzzle generator
   https://deniz.co/8-puzzle-solver/

Python Libraries

1. **Copy** for appending list by making a copy of data without changing original elements.
   https://docs.python.org/3/library/copy.html
2. **sorted** using key function for sorting node objects in the queue by its attribute of cost (h(n),g(n))
   https://docs.python.org/3/howto/sorting.html section: "Key Function"

Only helper libraries and references above are not my original knowledge, all other important codes are my original work.

Report Table of Content

CS 170 Project 1 Eight Puzzle Report

Tongyuan He, SID 862217840, Nov 8, 2021

Introduction

A sliding puzzle, sliding block puzzle, or sliding tile puzzle is a combination puzzle that challenges a player to slide (frequently flat) pieces along certain routes (usually on a board) to establish a certain end configuration.[1] An Eight puzzle is a variation of a sliding puzzle and the number eight is determined by the number of tiles in a 3x3 puzzle grid. In this project, we use three different searching algorithms we learned in this course to solve this puzzle. One blind search: *uniform cost search*[2], and two heuristic searches: *A* Misplaced Tile, A* Manhattan Distance heuristics*[3]. In this project, I implemented these three search algorithms using python and observed the difference of pro/con of each algorithm by collecting data of time by counting the number of nodes expanded and memory space by its maximum queue size.

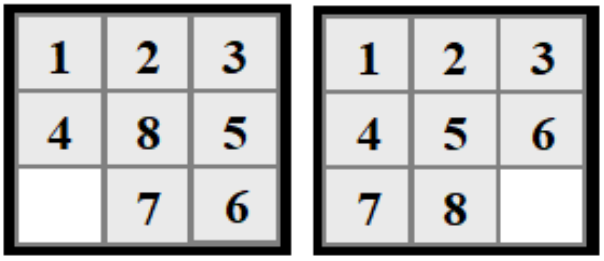


Figure1. An example of eight puzzles from Dr. Eamonn Keogh's lecture slides. The left puzzle is a given random state, Right puzzle is the goal state.

Players can slide tiles up, down, left, or right into the empty slot. repeating the process until reaches the goal state puzzle. In this project, each algorithm will do this by itself by prioritizing lower cost moves.

Comparison of Algorithms:

The three algorithms for this project are implemented similarly, the only difference is the calculation of the cost. In this project, I used a queue to store nodes(puzzle with its cost) and sorted them by cost in ascending order.
In all three search algorithms, the cost calculated to reach reach goal is cost = depth + heuristics. f(n) = g(n) + h(n). `

Uniform Cost Search

This algorithm only uses depth(g(n)) as the cost to solve the puzzle. Since it's an uninformed search, the heuristics value is hardcoded to zero (h(n) = 0). When solving this puzzle, the algorithm will always choose the nodes to expand in the order of enqueueing, each layer of expansion will increment the cost (g(n)) by one the search traversal process is the same as breadth first search (BFS).

$$f(n) = g(n)$$

[1] Sliding puzzle Wikipedia https://en.wikipedia.org/wiki/Sliding_puzzle
[2] Lecture sides from Dr. Eamonn Keogh "3_Heuristic_search.pptx"
[3] Lecture sides from Dr. Eamonn Keogh "3_Heuristic_search.pptx"

Misplaced Tile Heuristic

This algorithm uses the sum(f(n)) of depth(g(n)) and the number of misplaced tiles(h(n)) for the current puzzle as the cost to solve the puzzle.

$$f(n) = g(n)+h(n)$$

To determine whether the tile is misplaced, the algorithm will compare the index/position of the tile from the current node's puzzle with the goal state, if they are not the same, then it's misplaced. For the eight puzzle, we will need to compare 8 tiles and the number of misplaced tiles is between 0-8. The cost is calculated before enqueueing the nodes into the queue and then sorted to find the cheapest cost move to expand.

Example: In Figure1, assume the left puzzle generated by the third expansion from the root state, the depth is 3, g(n) = 3. Total of 2 tiles was misplaced, #5,#6,#7, and #8. Therefore the cost to expand this puzzle is 3+4 = 7.

Manhattan Distance Heuristic

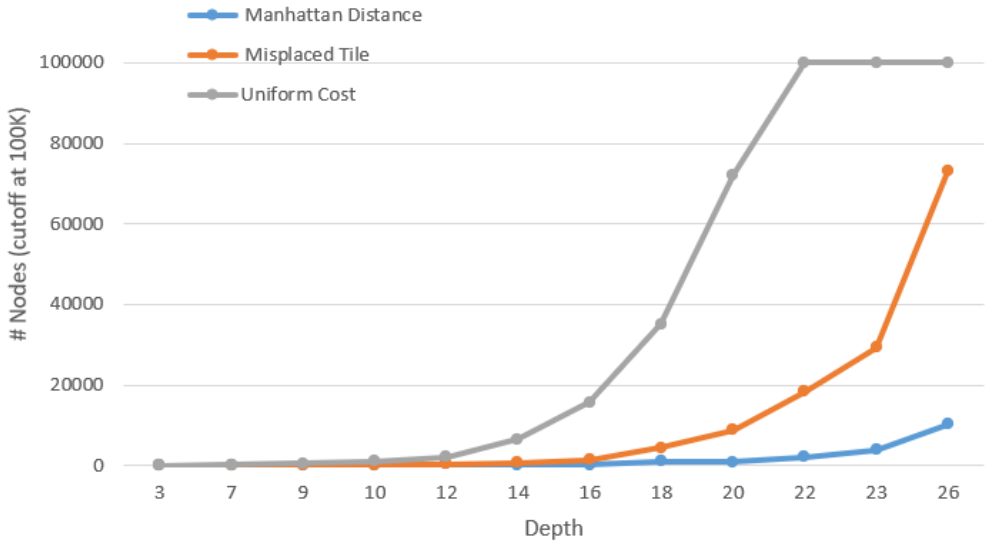
This algorithm uses the sum(f(n)) of depth(g(n)) and manhattan distance of tile from goal state(h(n)) for the current puzzle as the cost to solve the puzzle.

$$f(n) = g(n)+h(n)$$

To calculate the manhattan distance heuristic value of a tile, we ignore other obstacle tiles on the puzzle, and only calculate the displacements of the target tile from the goal state location with valid moves of up, down, left, and right.

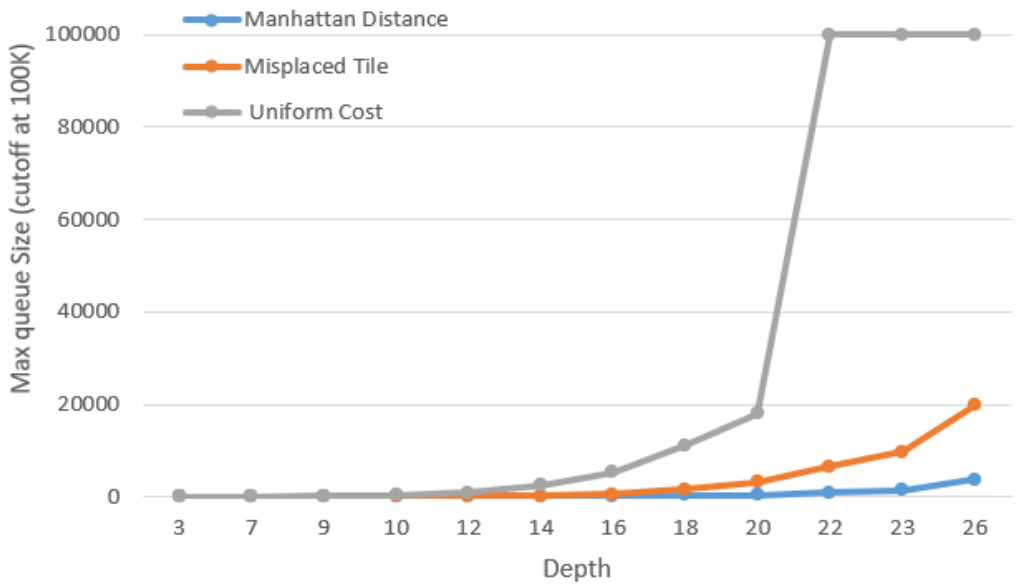
Example: In Figure1, assume the left puzzle generated by the third expansion from the root state again, the depth is 3, g(n) = 3. Tile #5 needs to move left by one step, #6 needs to move up by one step, #7 needs to move left by one step, and #8 needs to move down by one step, the manhattan distance is 4. Therefore the cost to expand this puzzle is 3+4 = 7.

Comparison of Algorithms



Graph 1. The number of nodes expanded v.s Search depth of three algorithms. The program limits node count to 100K for unpractical search cases.

* Uniform Cost Search fails to find a solution within 100K node range at depth = 22, graph value reflected as 100K.



Graph 2. The Maximum Queue Size v.s Search depth of three algorithms. The program limits node count to 100K for unpractical search cases.
* Uniform Cost Search fails to find a solution within 100K node range at depth = 22, graph value reflected as 100K.

Depth	Number of Nodes			Max Queue Size		
	Manhattan Distance	Misplaced Tile	Uniform Cost	Manhattan Distance	Misplaced Tile	Uniform Cost
3	18	18	32	10	10	14
7	36	46	241	18	21	103
9	48	80	553	22	34	220
10	73	138	1089	33	57	395
12	272	284	2084	118	117	815
14	115	652	6567	44	254	2498
16	168	1349	15748	68	522	5315
18	1061	4654	35173	412	1652	11191
20	829	8823	72133	327	3265	18045
22	2062	18282	>100K	837	6500	N/A
23	3947	29340	>100K	1488	9688	N/A
26	10213	73055	>100K	3690	19703	N/A

Table 1. Test Cases for randomly generated solvable puzzles in Graph 1 and Graph 2.
* For test cases with the same depth, the average value is recorded.

Test Data Analysis

In the graphs above, the number of nodes expanded is used for representing the time complexity of an algorithm by counting the number of loops in the search process, and Max queue size reflects the space complexity as the maximum memory allocated at any moment by the program.

By observing the graphs, I found two graphs are in similar shapes. For easy eight puzzles like depth < 9 three search algorithms provide similar results, with very minimum time and space used. By observing the data, puzzles with depth < 3, Manhattan Distance and Misplaced Tile provides an identical number of nodes and max queue size.

As the puzzle gets more complicated, depth > 14, Uniform Cost Search becomes extremely time and space consuming, and it's almost impossible to puzzles with depth > 20 in reasonable time period. On the other hand, Manhattan Distance and Misplaced Tile Heuristic are both efficient until depth reaches 23, Misplaced Tile Heuristic slows down significantly.

Lastly, Manhattan Distance Heuristic is currently the best of three algorithms and can solve very complex puzzles in a very short time.

Conclusion

By comparing three algorithms, we can conclude that The Uniform Cost Search is the slowest due to its property of blind search. The cost only directs the search algorithm to a breadth first search (BFS), which is complete and eventually will find the solution, however, due to its time and space complexity of O(b^d) where b= (valid tile moves for the current state) and d=(depth), it's not practical for solving real-world problems.

Both heuristics/informed searches are significantly better than blind searches, the only difference is how the cost is calculated. By comparing the Manhattan Distance heuristic and Misplaced Tile heuristic, both heuristic search algorithms are significantly faster and more space-efficient than blind search, and to improve A* heuristic search, we can use a better way to calculate the heuristic value to estimate more accurate costs.

Sample output of easy puzzle (depth 3):

```
1. use default puzzle
2. create a puzzle
2
Enter 9 tiles, EX: 1 2 3 4 5 6 7 8 0
1 2 3 0 5 6 4 7 8
your puzzle:
--------------
| 1  | 2  | 3 |
--------------
| 0  | 5  | 6 |
--------------
| 4  | 7  | 8 |
--------------
1. Uniform Cost Search.
2. A* with the Misplaced Tile heuristic.
3. A* with the Manhattan Distance heuristic.
4. Rum all 3 Algorithm without print intermediate states
3
Expanding node with g(n) = 0 h(n) = 0 Puzzle:
--------------
| 1  | 2  | 3 |
--------------
| 0  | 5  | 6 |
--------------
| 4  | 7  | 8 |
--------------
Expanding node with g(n) = 1 h(n) = 3 Puzzle:
--------------
| 0  | 2  | 3 |
--------------
| 1  | 5  | 6 |
--------------
| 4  | 7  | 8 |
--------------
Expanding node with g(n) = 1 h(n) = 3 Puzzle:
--------------
| 1  | 2  | 3 |
--------------
| 5  | 0  | 6 |
--------------
| 4  | 7  | 8 |
--------------
Expanding node with g(n) = 1 h(n) = 3 Puzzle:
--------------
| 1  | 2  | 3 |
--------------
| 4  | 5  | 6 |
--------------
| 0  | 7  | 8 |
--------------
Expanding node with g(n) = 1 h(n) = 2 Puzzle:
--------------
| 1  | 2  | 3 |
--------------
| 4  | 5  | 6 |
--------------
| 0  | 7  | 8 |
--------------
Expanding node with g(n) = 1 h(n) = 4 Puzzle:
--------------
| 0  | 2  | 3 |
--------------
| 1  | 5  | 6 |
--------------
| 4  | 7  | 8 |
--------------
Expanding node with g(n) = 2 h(n) = 1 Puzzle:
--------------
| 1  | 2  | 3 |
--------------
| 4  | 5  | 6 |
--------------
| 7  | 0  | 8 |
--------------
Expanding node with g(n) = 2 h(n) = 5 Puzzle:
--------------
| 1  | 2  | 3 |
--------------
| 5  | 6  | 0 |
--------------
| 4  | 7  | 8 |
--------------
Expanding node with g(n) = 3 h(n) = 0 Puzzle:
--------------
| 1  | 2  | 3 |
--------------
| 4  | 5  | 6 |
--------------
| 7  | 8  | 0 |
--------------
==== Goal State ! ====
Solution Depth: 3
Number of nodes expanded: 12
Max queue size: 7
```

- Sample output of hard puzzle (depth 23)

```
1. use default puzzle
2. create a puzzle
2
Enter 9 tiles, EX: 1 2 3 4 5 6 7 8 0
8 6 4 0 7 2 5 1 3
your puzzle:
| 8 | 6 | 4 |
| 0 | 7 | 2 |
| 5 | 1 | 3 |
1. Uniform Cost Search.
2. A* with the Misplaced Tile heuristic.
3. A* with the Manhattan Distance heuristic.
4. Rum all 3 Algorithm without print intermediate states
3
Expanding node with g(n) = 0 h(n) = 0 Puzzle:
| 8 | 6 | 4 |
| 0 | 7 | 2 |
| 5 | 1 | 3 |
Expanding node with g(n) = 1 h(n) = 18 Puzzle:
| 8 | 6 | 4 |
| 7 | 0 | 2 |
| 5 | 1 | 3 |
Expanding node with g(n) = 1 h(n) = 18 Puzzle:
| 0 | 6 | 4 |
| 8 | 7 | 2 |
| 5 | 1 | 3 |
Expanding node with g(n) = 1 h(n) = 18 Puzzle:
```

```
Expanding node with g(n) = 16 h(n) = 7 Puzzle:
| 1 | 4 | 2 |
| 7 | 5 | 6 |
| 8 | 0 | 3 |
Expanding node with g(n) = 22 h(n) = 1 Puzzle:
| 1 | 2 | 3 |
| 4 | 5 | 0 |
| 7 | 8 | 6 |
Expanding node with g(n) = 17 h(n) = 6 Puzzle:
| 1 | 2 | 0 |
| 7 | 4 | 6 |
| 8 | 5 | 3 |
Expanding node with g(n) = 17 h(n) = 6 Puzzle:
| 1 | 2 | 6 |
| 7 | 4 | 3 |
| 8 | 5 | 0 |
Expanding node with g(n) = 23 h(n) = 0 Puzzle:
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 0 |
=== Goal State ! ===
Solution Depth: 23
Number of nodes expanded: 1438
Max queue size: 525
```

// Aprox 2k nodes omited

The method used for collecting data for Graph1, Graph2.

```
1. use default puzzle
2. create a puzzle
2
Enter 9 tiles, EX: 1 2 3 4 5 6 7 8 0
1 2 3 7 4 0 8 6 5
your puzzle:
---------------
| 1  | 2  | 3  |
---------------
| 7  | 4  | 0  |
---------------
| 8  | 6  | 5  |
---------------
1. Uniform Cost Search.
2. A* with the Misplaced Tile heuristic.
3. A* with the Manhattan Distance heuristic.
4. Rum all 3 Algorithm without print intermediate states
4
=== Search with Manhattan Distance ===
===== Goal State ! ======
Solution Depth: 7
Number of nodes expanded: 36
Max queue size: 18
=== Search with Misplaced Tile ===
===== Goal State ! ======
Solution Depth: 7
Number of nodes expanded: 46
Max queue size: 21
=== Search with Uniform Cost ===
===== Goal State ! ======
Solution Depth: 7
Number of nodes expanded: 241
Max queue size: 103
```

In [ ]:

```
Github Link: https://github.com/the1323/CS170_Project_1
from copy import copy

goal = [1, 2, 3, 4, 5, 6, 7, 8, 0]
goalnum = 1234567890
printpuzzle = True

class node:
    def __init__(self, istate):
        self.puzzle = istate
        self.fval = 0
        self.gval = 0
        self.hval = 0
        self.children = []
        self.num_children = 0
# print puzzle in nice looking format
def print_puzzle(puzzle):
    # display puzzle
    print('---------------')
    print(f'| {puzzle[0]} | {puzzle[1]} | {puzzle[2]} |')
    print('---------------')
    print(f'| {puzzle[3]} | {puzzle[4]} | {puzzle[5]} |')
    print('---------------')
    print(f'| {puzzle[6]} | {puzzle[7]} | {puzzle[8]} |')
    print('---------------')

# print algorithm selection and get option
def menu():
    print('1. Uniform Cost Search.')
    print('2. A* with the Misplaced Tile heuristic.')
    print('3. A* with the Manhattan Distance heuristic.')
    print('4. Rum all 3 Algorithm without print intermediate states')
    try:
        select = int(input())
        while (select > 4 or select < 1):
            print('Enter selection 1-3')
            select = int(input())
    except ValueError:
        print("input not integer")
    return select

# calculate number of misplaced tiles
def Misplaced_Tile_Heuristic(problem):
    compare = 1
    count = 0
    for i in range(len(problem)-1):
        if problem[i] != compare:
            count += 1
        compare+=1

    return count

# calculate manhattan distance
def Manhattan_Distance_Heuristic(problem):
    # not including distance for '0'
    goal = [1, 2, 3, 4, 5, 6, 7, 8, 0]
    sum = 0
    for i in range(1, len(problem)):
        prob_index = problem.index(i)
```

```
        prob_col = prob_index % 3
        prob_row = prob_index // 3
        goal_index = goal.index(i)
        goal_col = goal_index % 3
        goal_row = goal_index // 3
        sum += abs(prob_row - goal_row) + abs(prob_col - goal_col)
    return sum

# helper function convert puzzle to integer, for later comparison
def ltoi(puzzle):
    s = [str(integer) for integer in puzzle]
    return int("".join(s))

# check if input puzzle has valid format
def validpuzzle(problem):
    if len(problem) != 9:
        return False
    for i in range(len(problem)):
        if problem.count(i) != 1:
            return False
    return True

#search algorithm for all 3 searches
def generalsearch(problem, QUEUEING_FUNCTION):
    myqueue = []
    visited = []
    rootnode = node(problem)
    nodecount = 0
    myqueue.append(copy(rootnode))
    qtracker = 1

    while len(myqueue):

        temp_node = myqueue.pop(0)  # dequeue min cost
        if printpuzzle:
            print(f'Expanding node with g(n) = {temp_node.gval} h(n) = {temp_node.hval} Puzzle:')
            print_puzzle(temp_node.puzzle)
        myqueue = sorted(myqueue, key=lambda node: node.fval)

        if (len(temp_node.puzzle)) == 0:
            return "failure"

        puzzlenumber = ltoi(temp_node.puzzle)
        if puzzlenumber == ltoi(goal):  # if reach goal
            print('===== Goal State ! ======')
            print(f"Solution Depth: {temp_node.gval}")
            print(f"Number of nodes expanded: {nodecount}")
            print(f"Max queue size: {qtracker}")
            return 0

        visited.append(copy(puzzlenumber))

        children_nodes = copy(EXPAND(temp_node))

        for i in range(children_nodes.num_children):
            tempn = node(children_nodes.children[i])
            if ltoi(tempn.puzzle) not in visited:
                nodecount += 1
                if(nodecount>100000): return print("failure, complexity exceeds 100K")
                # print_puzzle(tempn.puzzle)
                tempn.gval = temp_node.gval + 1
```

```
                if QUEUEING_FUNCTION == 2:
                    tempn.hval = Misplaced_Tile_Heuristic(tempn.puzzle)
                if QUEUEING_FUNCTION == 3:
                    tempn.hval = Manhattan_Distance_Heuristic(tempn.puzzle)

                tempn.fval = tempn.hval + tempn.gval
                visited.append(copy(ltoi(tempn.puzzle)))
                myqueue.append(copy(tempn))

                if qtracker < len(myqueue):
                    qtracker = len(myqueue)


    print('Error. search stoped')

# expand children node/puzzle for valid moves
def EXPAND(n):
    cstate = n.puzzle
    index = cstate.index(0)  # get index of '0'
    col = index % 3
    row = index // 3
    nstate = []
    cindex = 0

    if col != 0:  # left
        nstate.append(copy(cstate))
        nstate[cindex][index], nstate[cindex][index - 1] = nstate[cindex][index - 1], nstate[cindex]
        n.children.append(copy(nstate[cindex]))
        cindex += 1
        n.num_children += 1

    if col != 2:  # right
        nstate.append(copy(cstate))
        nstate[cindex][index], nstate[cindex][index + 1] = nstate[cindex][index + 1], nstate[cindex]
        n.children.append(copy(nstate[cindex]))
        cindex += 1
        n.num_children += 1

    if row != 0:  # 0 can go up
        nstate.append(copy(cstate))
        nstate[cindex][index], nstate[cindex][index - 3] = nstate[cindex][index - 3], nstate[cindex]
        n.children.append(copy(nstate[cindex]))
        cindex += 1
        n.num_children += 1

    if row != 2:  # down
        nstate.append(copy(cstate))
        nstate[cindex][index], nstate[cindex][index + 3] = nstate[cindex][index + 3], nstate[cindex]
        n.children.append(copy(nstate[cindex]))
        cindex += 1
        n.num_children += 1

    return n


if __name__ == '__main__':
    #sample_in = [8, 6, 4, 0, 7, 2, 5, 1, 3]
    #sample_in = [1,2,3,7,4,8,6,5,0]
    #sample_in = [1,2,3,0,5,6,4,7,8]
    sample_in = [1, 2, 3, 5,0,6,4,7,8]
    #sample_in = [1,2,3,7,4,0,8,6,5,]
```

```
#sample_in = [7, 1, 3, 0, 2, 5, 8, 4, 6]
#sample_in = [1, 2, 3, 4, 5, 6, 7, 8, 0]
userin = []
puzzle = []
print('1. use default puzzle')
print('2. create a puzzle')

puzzle_selection = int(input())

if (puzzle_selection == 1):
    print("your puzzle: ")
    print_puzzle(sample_in)
    puzzle = sample_in
else:
    print("Enter 9 tiles, EX: 1 2 3 4 5 6 7 8 0")
    userin = list(map ( int, (input().split())))
    while not validpuzzle(userin):
        print("Invalid input, Try again: ")
        userin = list(map ( int, (input().split())))
    print("your puzzle: ")
    print_puzzle(userin)
    puzzle = userin
selection = menu()  # just print 3 options and get selection

if selection != 4:
    generalsearch(puzzle, selection)
else: # run all 3 search without print puzzle
    printpuzzle = False
    print('=== Search with Manhattan Distance ===')
    generalsearch(puzzle, 3)
    print('=== Search with Misplaced Tile === ')
    generalsearch(puzzle, 2)
    print('=== Search with Uniform Cost === ')
    generalsearch(puzzle, 1)
```