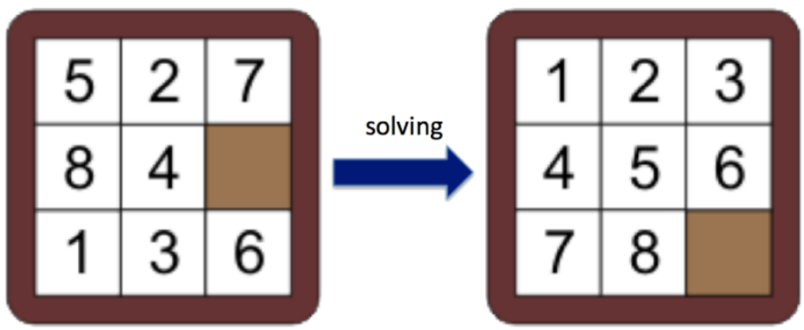
The Eight Puzzle

Name: Siddhant Arya
Email: siddhantarya0707@gmail.com
LinkedIn: https://www.linkedin.com/in/siddhantarya07

Introduction

According to Wikipedia, an Eight puzzle is a sliding block puzzle that challenges the player to arrange the numbers in increasing order starting from 1 to 8. The last tile is an empty block, which helps the player to arrange the tiles. In our project, the Eight puzzles are generalized into the N*N puzzle where we have N*N squared tiles with N*N-1 numbers engraved on them and the last tile is a blank tile. The same row and same positioned tiles can be moved horizontally or vertically by sliding them respectively. The aim of the N*N puzzle is to place the tiles in numerical order.



N*N puzzle is a marvelous problem to demonstrate search algorithms involving heuristics. This is our first project under Dr. Eamonn Keogh's CS 205 course. The aim of the project is to solve write generalized algorithms that will help to solve any N*N puzzle.
We are going to solve the puzzle using three search algorithms

- Uniform Cost Search
- A* with Misplaced Tile Heuristics
- A* with Manhattan Distance Heuristics

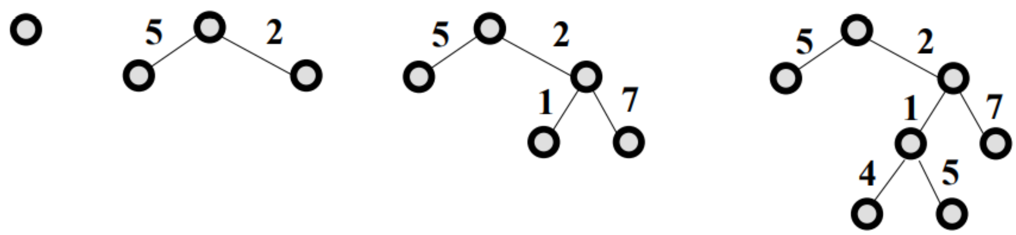
I have used Python 3.9.7 to write the algorithms and the code can be retrieved from the following GitHub link
"https://github.com/siddhantarya07/AI_The_Eight_Puzzle"

Comparison of Algorithms

1. **Uniform Cost Search**

   A uniform cost search is an uninformed cost search that uses the lowest cost to find the path from the initial state to the goal state. The algorithm operates in a brute-force manner and does not use any heuristics.

The goal of the algorithm is to expand the cheapest node. H(n) is always 0. The algorithm is complete and even optimal to some extent but it explores nodes in all the directions and does not have any information about the goal state.
The Time and Space complexity of the algorithm is O(b^d) where b is the branching factor and d is the depth of the solution.

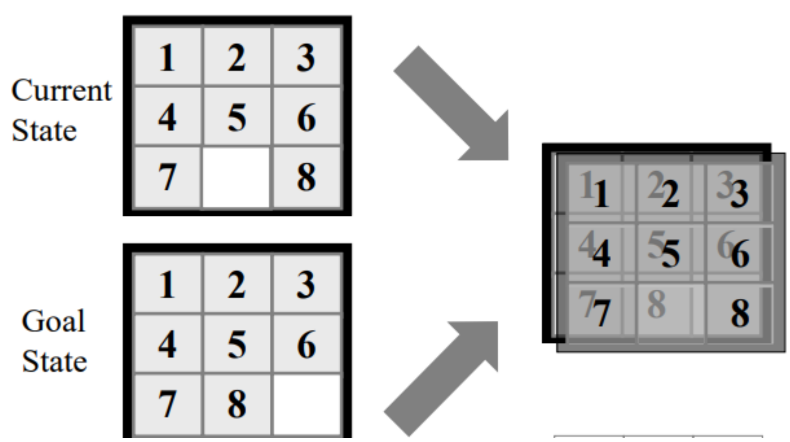


The above figure represents the generic working of the Uniform Cost Search algorithm.

For the N*N puzzle, the cost is constant and assumed to the 1 for each expansion. It is assumed that the same effort is required to move the tile in any direction.

2. A* with Misplaced Tile Heuristic

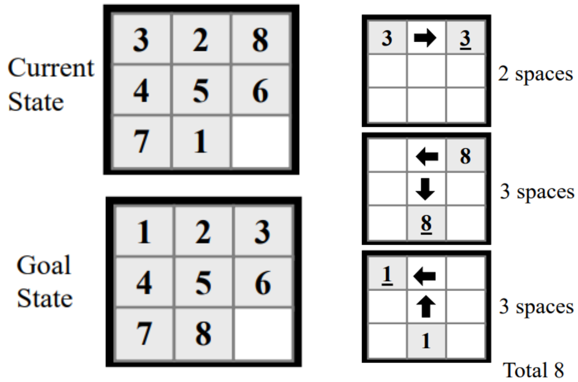
The above algorithm uses a heuristic known as Misplaced tile. The current state of the puzzle is matched with the goal state to find out the number of misplaced tiles. The number of misplaced tiles excluding 0 is returned as a heuristic value. The least value of the heuristic function is directly proportional to the best state of the puzzle since it will be closest to the goal state. When this heuristic is applied to the N*N puzzle, the node with less heuristic value (the cheapest node will be selected for expansion).



Heuristic function h(n) returns 1 as only one tile with value 8 is misplaced.

3. A* with Manhattan Distance Heuristic

The Manhattan Distance heuristic used in this algorithm computes a heuristic value by counting the number of moves along the grid that each tile is displaced from its goal position and sums up all these values over all the tiles.

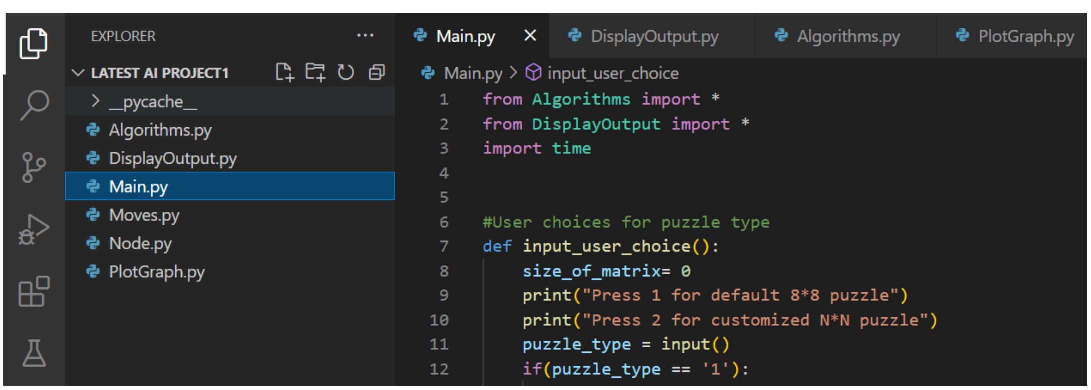


Tiles “3” “8” and “1” are misplaced, thus h(n) = 8

Project Structure

Our project structure contains 5 files to increase the modularity of the application.

- **Main.py:** This file contains the main method which is the starting point of the application. It will accept all the required user inputs and pass the selected algorithms to further solve the puzzle.
- **Algorithms.py:** This file consists of all the 3 algorithms that we have implemented to solve our puzzle. Misplaced tile and Manhattan heuristics are also calculated in the same file and passed the value back to the calling algorithm.
- **Node.py:** It contains a Node class that creates the object with the required parameters passed. Explored moves method explores all the possible movements of the nodes and if possible, it creates a node and adds to the explored set.
- **Moves.py:** This file contains all 4 possible moves methods. For any tile movement, it checks whether the particular movement of tile is possible is not. These are the generic methods that work excellent with a puzzle with any N size.
- **DisplayOutput.py:** This file contains the methods that help to display the initial state or the intermediate and final output in N*N matrix form.
- **PlotGraph.py:** This file is used to plot the graph from the data we received from the comparison of algorithms.



Passing Puzzle Input Parameters to the Application

The application can be started by running the “python Main.py” command in the project folder cmd prompt.

- As the program runs, the User is asked to enter his choice to solve a default 8*8 puzzle or create his own customized puzzle.
- Our application needs to enter data column-wise separated by spaces between the digits.
- To enter a blank space, the User needs to enter 0.
- After entering the data, the User can select one of the three algorithm types to solve the puzzle.
- Once the user selects the algorithm type, the puzzle will be solved and the output will be displayed in a console window with the required steps.

```
E:\UCR Projects\Latest AI Project1>python Main.py
Press 1 for default 8*8 puzzle
Press 2 for customized N*N puzzle
1

You have selected to solve default 8*8 puzzle

Please enter space to seperate the numbers.
Please enter 0 for the blank tile.
Enter data for column 1:
1 2 3
Enter data for column 2:
4 5 6
Enter data for column 3:
0 7 8
Please select type of algorithm to solve the puzzle
Type 1 for Uniform Cost Search
Type 2 for A* with Misplaced Tile Heuristic
Type 3 for A* with Manhattan Distance Heuristic
1
Initial State of the puzzle is:
-------------
| 1 | 2 | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 0 | 7 | 8 |
-------------
```

Passing input to the default 8*8 Puzzle

```
E:\UCR Projects\Latest AI Project1>python Main.py
Press 1 for default 8*8 puzzle
Press 2 for customized N*N puzzle
2

You have selected to solve N*N puzzle

Please enter space to seperate the numbers.
Please enter 0 for the blank tile.
Enter the size of the matrix puzzle:
4
Enter data for column 1:
1 2 3 4
Enter data for column 2:
5 6 7 8
Enter data for column 3:
9 10 11 12
Enter data for column 4:
13 14 0 15
Please select type of algorithm to solve the puzzle:
Type 1 for Uniform Cost Search
Type 2 for A* with Misplaced Tile Heuristic
Type 3 for A* with Manhattan Distance Heuristic
1
Initial State of the puzzle is:
------------------
| 1 | 2 | 3 | 4 |
------------------
| 5 | 6 | 7 | 8 |
------------------
| 9 | 10 | 11 | 12 |
------------------
| 13 | 14 | 0 | 15 |
------------------
```

Passing input to the customized N*N Puzzle

Snippets of the algorithms implemented

1. Uniform Cost Search

```python
#Uniform cost search algorithm
def uniform_cost_search(initial_state, goal_state, customized_puzzle_size):
    initial_node= create_node(initial_state, None, None,0,0)
    trace_list = []
    frontier_queue =[]
    expanded_nodes_count = 0
    working_node = initial_node
    final_depth= 0
    orignal_queue = [] # To prevent expansion of repeated nodes
    while(working_node.state != goal_state):
        tempMoves = explored_moves(working_node,customized_puzzle_size)
        for moves in tempMoves:
            if moves.state not in orignal_queue:
                moves.depth = working_node.depth + 1
                moves.heuristic =0  #H(n) is always 0 for this puzzle problem
                orignal_queue.append(moves.state)
                frontier_queue.append(moves)
                expanded_nodes_count = expanded_nodes_count +1
                final_depth = moves.depth
        frontier_queue.sort(key =lambda x: x.depth)
        working_node = frontier_queue.pop(0)
    while(working_node.parent != None):
        trace_list.append([working_node.operation,working_node.state])
        working_node = working_node.parent

    trace_list.reverse()
    return Result_Set(trace_list, expanded_nodes_count,final_depth)

```

2. A* with Misplaced Tile Heuristic

```python
# A Star algorithm with Misplaced tile heuristic
def a_star_misplaced_tile_heuristic(initial_state,goal_state,customized_puzzle_size):
    initial_node= create_node(initial_state, None, None,0,0)
    trace_list = []
    frontier_queue =[]
    expanded_nodes_count = 0
    working_node = initial_node
    final_depth= 0
    orignal_queue = [] # To prevent expansion of repeated nodes
    while(working_node.state != goal_state):
        tempMoves= explored_moves(working_node,customized_puzzle_size)
        for moves in tempMoves:
            find_misplaced_tile_heuristic(moves, goal_state)
            if moves.state not in orignal_queue:
                moves.depth = working_node.depth + 1
                orignal_queue.append(moves.state)
                frontier_queue.append(moves)
                expanded_nodes_count = expanded_nodes_count +1
                final_depth= moves.depth
        frontier_queue.sort(key =lambda x: x.heuristic + x.depth)
        working_node = frontier_queue.pop(0)
    while(working_node.parent != None):
        trace_list.append([working_node.operation,working_node.state])
        working_node = working_node.parent
    # traverse the list
    trace_list.reverse()
    return Result_Set(trace_list, expanded_nodes_count,final_depth)
```

```python
#Calculates the misplaced heuristic value for A* algorithm
#Counts measures the amount of misplaced tile by comparing initial state to goal state
def find_misplaced_tile_heuristic(current_state, goal_state):
    count = 0 #amount of misplaced tile number
    for i in range(0,9):
        if current_state.state[i] != 0 and current_state.state[i] != goal_state[i]:
            count = count +1
    current_state.heuristic = count
```

3. A* with Manhattan distance heuristic

```python
# A Star algorithm with Manhattan distance heuristic
def a_star_manhattan_distance_heuristic(initial_state,goal_state,customized_puzzle_size):
    initial_node= create_node(initial_state, None, None,0,0)
    trace_list = []
    frontier_queue =[]
    orignal_queue = [] # To prevent expansion of repeated nodes
    expanded_nodes_count = 0
    final_depth =0
    working_node = initial_node
    while(working_node.state != goal_state):
        tempMoves= explored_moves(working_node,customized_puzzle_size)
        for moves in tempMoves:
             calculate_manhattan_distance_heuristic(moves, goal_state,customized_puzzle_size)
             if moves.state not in orignal_queue:
                moves.depth = working_node.depth + 1
                orignal_queue.append(moves.state)
                frontier_queue.append(moves)
                expanded_nodes_count = expanded_nodes_count +1
                final_depth = moves.depth
        frontier_queue.sort(key =lambda x: x.heuristic  + x.depth)
        working_node = frontier_queue.pop(0)
    while(working_node.parent != None):
        trace_list.append([working_node.operation,working_node.state])
        working_node = working_node.parent
    # traverse the list
    trace_list.reverse()
    return Result_Set(trace_list, expanded_nodes_count,final_depth)
```

```python
#Calculates the manhattan heuristic value for A* algorithm
def calculate_manhattan_distance_heuristic(node, goal_state,customized_puzzle_size):
    current_state = node.state
    total_manhattan = 0
    # for manhattan distance, convert a 1d array to N-D array
    initial_array =np.array(current_state)
    goal_array = np.array(goal_state)
    n_d_current_array = initial_array.reshape(customized_puzzle_size,customized_puzzle_size)
    n_d_goal_state = goal_array.reshape(customized_puzzle_size,customized_puzzle_size)
    for i in range(0,customized_puzzle_size):
        for j in range(0,customized_puzzle_size):
            if (n_d_current_array[i][j] != n_d_goal_state[i][j]) and n_d_current_array[i][j] != 0:
                current_item_index_in_goal= np.where(n_d_goal_state == n_d_current_array[i][j])
                current_tile_index_in_goal = list(zip(current_item_index_in_goal[0], current_item_index_in_goal[1]))
                current_item_index_in_initial= np.where(n_d_current_array == n_d_current_array[i][j])
                current_tile_index_in_initial = list(zip(current_item_index_in_initial[0], current_item_index_in_initial[1]))
                current_move_required= abs(current_tile_index_in_goal[0][0] -current_tile_index_in_initial[0][0]) + abs(current_tile_index_
                total_manhattan = total_manhattan + current_move_required
    node.heuristic= total_manhattan
```

Puzzle Operators Implementation

Operator 1: Move Up

```
# Swap the tile up and return as new node
# If possible move is not available, return None
def move_up(state,customized_puzzle_size):
    next_state = state[:]
    next_state_index = next_state.index(0)
    state_not_to_consider = []
    start =0
    while(start < customized_puzzle_size):
        state_not_to_consider.append(start)
        start = start +1
    if next_state_index not in state_not_to_consider:
        temp = next_state[next_state_index - customized_puzzle_size]
        next_state[next_state_index - customized_puzzle_size] = next_state[next_state_index]
        next_state[next_state_index] = temp
        return next_state
    else:
        return None

```

Operator 2: Move Down

```
# Swap the tile down and return as new node
# If possible move is not available
def move_down(state, customized_puzzle_size):
    next_state = state[:]
    next_state_index = next_state.index(0)
    state_not_to_consider = []
    start = len(state) -customized_puzzle_size
    for i in range(0,customized_puzzle_size):
        state_not_to_consider.append(start)
        start = start +1
    # if next_state_index not in [6, 7, 8]:
    if next_state_index not in state_not_to_consider:
        temp = next_state[next_state_index + customized_puzzle_size]
        next_state[next_state_index + customized_puzzle_size] = next_state[next_state_index]
        next_state[next_state_index] = temp
        return next_state
    else:
        return None

```

Operator 3: Move Left

```python

# Moves the tile left and return as new node
# If possible move is not available, return None
def move_left(state, customized_puzzle_size):
    next_state = state[:]
    next_state_index = next_state.index(0)
    state_not_to_consider = []
    start = 0
    while(start < len(state)):
        state_not_to_consider.append(start)
        start = start + customized_puzzle_size
    # if next_state_index not in [0, 3, 6]:
    if next_state_index not in state_not_to_consider:
        temp = next_state[next_state_index - 1]
        next_state[next_state_index - 1] = next_state[next_state_index]
        next_state[next_state_index] = temp
        return next_state
    else:
        return None

```

Operator 4: Move Right

```python
# Swap the tile right and return as new node
# If possible move is not available, return None
def move_right(state, customized_puzzle_size):
    next_state = state[:]
    next_state_index = next_state.index(0)
    state_not_to_consider = []
    start = customized_puzzle_size -1
    while(start < len(state)):
        state_not_to_consider.append(start)
        start = start + customized_puzzle_size
    # if next_state_index not in [2, 5, 8]:
    if next_state_index not in state_not_to_consider:
        temp = next_state[next_state_index + 1]
        next_state[next_state_index + 1] = next_state[next_state_index]
        next_state[next_state_index] = temp
        return next_state
    else:
        return None

```

Displaying Output of the Application

- The output of the puzzle is neatly depicted in the console window in the N*N matrix form. Even the console window shows the trace path of the puzzle solved step by step.
- Even we are showing the statistics of each algorithm like the number of nodes explored, Solution depth of the puzzle, and time taken by the algorithm to solve the puzzle.

```
Type 3 for A* with Manhattan Distance Heuristic
1
Initial State of the puzzle is:
-------------
| 1 | 2 | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 0 | 7 | 8 |
-------------
Step 1: Move Right
-------------
| 1 | 2 | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 0 | 8 |
-------------
Step 2: Move Right
-------------
| 1 | 2 | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | 0 |
-------------
---------Analysis----------
The puzzle has been solved.
The total number of nodes explored: 15
Solution Depth is at: 2
The time by the algorithm is: 0.0
```

Sample output of an easy puzzle solved by Uniform Cost Search

Analysis of Algorithms with Sample Puzzles

The sample puzzles were provided by the professor and we are testing our algorithm on these sample puzzles. These sample puzzles are arranged in order of their increasing depth.

Uniform Cost Search

S No	Sample Input	Solution Depth	Number of nodes explored	Time Taken
1	123 456 078	2	14	0.00105834
2	123 506 478	4	60	0.00099658
3	136 502 478	8	500	0.03923344
4	136 507 482	12	3607	0.35713315
5	167 503 482	16	20293	9.72216749
6	712 485 630	20	71780	266.45185589

A* with Misplaced Tile Heuristic

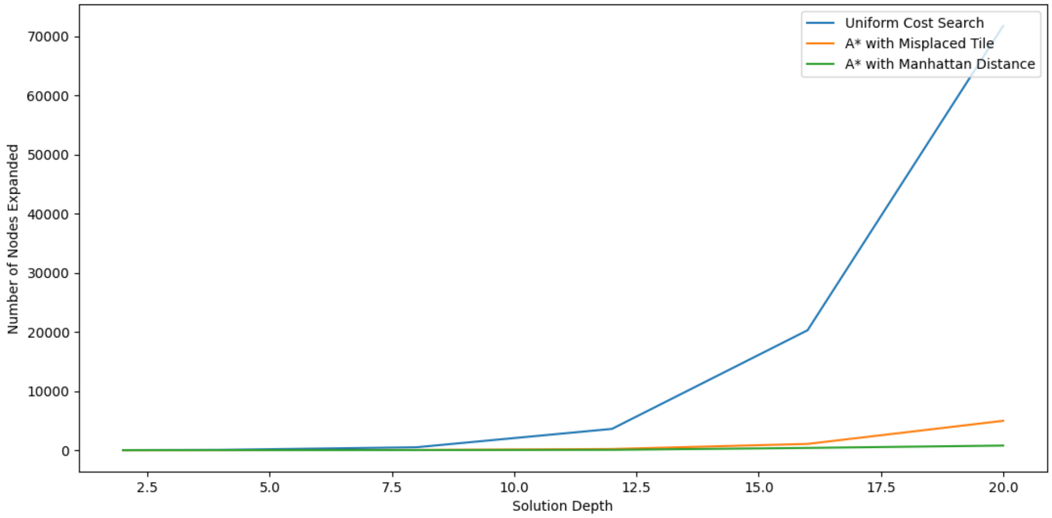
S No	Sample Input	Solution Depth	Number of nodes explored	Time Taken
1	123 456 078	2	5	0.0
2	123 506 478	4	12	0.0
3	136 502 478	8	34	0.00100827
4	136 507 482	12	203	0.00199007
5	167 503 482	16	1065	0.05316710
6	712 485 630	20	4974	0.70125699

A* with Manhattan Distance Heuristic

S No	Sample Input	Solution Depth	Number of nodes explored	Time Taken
1	123 456 078	2	5	0.0009975
2	123 506 478	4	10	0.00223922
3	136 502 478	8	24	0.00297236
4	136 507 482	12	64	0.02707028
5	167 503 482	16	388	0.06337380
6	712 485 630	20	786	0.12660217

Comparison of Algorithms

The graph below depicts the comparison of all three algorithms in terms of nodes explored vs Solution depth.



Conclusion

Graphs and Statistics depict that for puzzles with lower solution depth, heuristics are not too important. But as solution depth increases, we can observe that the number of nodes expanded increases tremendously. So, Heuristics makes our search very fast.

References

1. CS 205 AI slides by Prof. Eamonn Keogh
2. Wikipedia
3. Artificial Intelligence: A modern approach by Stuart Russell and Peter Norvig