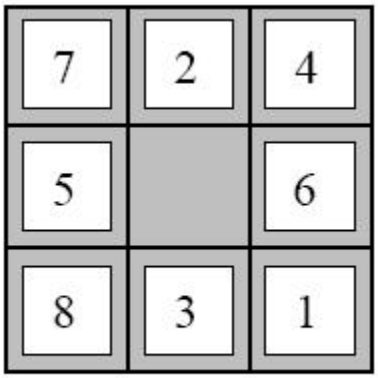
Project 1: 8 Puzzle Problem

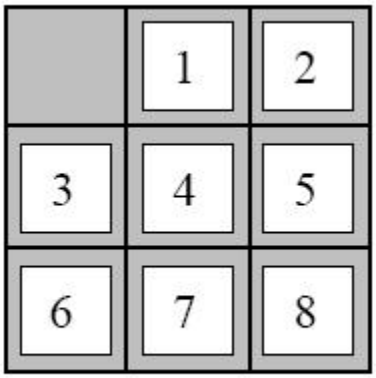
I. The 8 puzzle problem description:

The 8 puzzle is a puzzle that have a 3*3 board and an empty space. The space can swap place with the tile next to it. In other words, it can move up, down, left and right.

In an 8 puzzle problem, the puzzle is often shuffle to the different configuration. Our objective is to find a solution to solve the puzzle back to its original state.



Start State



Goal State

(Source: Rohan Pandare's Google sites

II. Code development:

a. Analyze the problem:

An initial state of the puzzle is given. From any state, we could have maximum 4 states derive from it. Thus, technically speaking, all states that can be derive from the initial state can be put into a tree structure. Then, an optimal path can be chosen. However, this approach is not desirable as it would be slow, and waste lots of resources. Thus, other graph/tree search method a highly recommended.

b. A* algorithm:

In this particular project, A* algorithm is being used to implement the solution. A* algorithm is a search algorithm that is a lot better than other search algorithm such as depth first search or breadth first search as it does take into account how close the current state is to the final goal state. Thus, it would avoid traversing to other state that is "obviously" longer and less desirable.

How desirable a state is determined by a function:

$$f = g + h;$$

In which, g is the cost to reach the current state from the original state and h is the heuristic function that estimate cost to reach the goal state. Thus the lower f is, the less resource we need to get to that node. It means that lower f are more desirable.

This is how A* algorithm are being implemented to my solution code:

- We have 2 lists: a closed list and a list called frontier. Closed list are for the list of visited node and frontier is a list of node waiting to be traverse to.
- Then we put the initial state into the frontier.

- While the first node in frontier is not goal:
    - We remove that first node in the frontier to the closed list
    - Than we expand the children of the node that just has been put in the closed list.
    - If the children are not in the closed list, we put it in frontier. (This is being done to prevent infinite loops)
    - Sort frontier by f in ascending order. If there are equals in f value, newer node are being put in higher priority.
- This process will continue until the first node in the frontier is the goal state the we can backtrack to get the steps to the solution.

c. Heuristic:

The way that a* function is implemented, the better the f function is, the better result we are going to get. But f is dependent on g and h. While g is something concrete and can be calculated, h is an estimation, a guess. Thus A* algorithm effectiveness is dependent on our h (heuristic function). In this case 4 heuristic function are being chose:

H0: we don't have a heuristic function at all. Thus essentially, this is brute forcing to the answer using breath first search.

H1: the number of tiles that are misplace from the goal state.

H2: the total Manhattan distance of each node. The Manhattan distance of each node is calculating by how many steps both horizontally and vertically the tile has to be moving to get to its position in the goal state

H3: a mix of heuristic 1 and heuristic 2. If a tile is misplaced, H3=1+the horizontal distance to the goal state.

By looking at how these heuristic works, it is obvious that H2 is the best heuristic as it considers both the vertical and horizontal distance from goal. H3 works better than H1 as it does consider the horizontal distance from goal and H0 is the worst as it is essentially brute forcing.

This is shown in my result from solving randomly generated board.

H1

	V(Nodes visited)	N(Maximum nodes stored in memory)	D(depth of the optimal solution)	b(Effective branching factor where N=b^d)
Minimum	6	13	5	1.46902
Median	782	1277.5	17	1.52169
Mean	2511.38	3928.04	16.5	1.51406
Maximum	18558	28270	24	1.67028
Standard deviations	4517.55	6935.36	4.61586	0.03094

H2

	V(Nodes visited)	N(Maximum nodes stored in memory)	D(depth of the optimal solution)	b(Effective branching factor where N=b^d)
Minimum	7	15	5	1.2426
Median	237.5	397.5	17	1.42949
Mean	479.24	782.68	16.5	1.4319
Maximum	2883	4660	24	1.7411
Standard deviations	648.05	1046.50	4.61586	0.0683

H3

	V(Nodes visited)	N(Maximum nodes stored in memory)	D(depth of the optimal solution)	b(Effective branching factor where N=b^d)
Minimum	6	13	5	1.31672
Median	309.5	508	17	1.441545
Mean	689.48	1105.74	16.5	1.441728
Maximum	5379	8464	24	1.67028
Standard deviations	1064.328	1679.452	4.61586	0.05761