8 Puzzle Project Report

Name: Yueh-Lin Tsou ID: 012843142

1. Approach

a. ***User Selection:*** A user interface provide three menu options for the program.
(1) Input a specific 8-puzzle configuration
(2) Generate random 8-puzzle problem
(3) Read 8-puzzle input from the .txt file

b. ***Data Structure:*** In order to improve the performance and speed of the program, I use 'character array' instead of 'integer array' and used "1-dimensional" array instead of "2-dimensional" array.

c. ***Heuristic Function:***
(1) h1: the number of misplaced tiles
- Using "for loop" to check if the number is in the right position or not.

(2) h2: the sum of the distances of the tiles from their goal position
- Use equation: |(i / 3) - char[i] / 3| + |(i % 3) - char[i] % 3| to check how many steps should move to the goal position.

d. ***Open Set and Close Set:***
(1) Open set: the board which has been created but doesn't consider yet.
- Method: add the board into **priority queue** and use **comparator** to sort the queue by consider it's heuristic function and depths.

(2) Close Set: the board which already been considered as solution.
- Method: put the board into **hash map**, then the program will check if the new board is already visited or not.

e. ***Search Cost:*** Counter add 1 when new child nodes been created.

f. ***Board and Node Information:***
(1) Board information: only contain **heuristic value**.
(2) Node information: make every board as a node, the node information contain **moves**(which is depths), **previous node**(which is it's parent).
- The final depths for the goal steps is the final node's moves value, and the program can follow the previous node information to print out the each step from the initial state to the final state.

2. Comparison

For the comparison, I generate around 5000 random cases for using different heuristic function and sort the output with different depth, then calculate their search cost and average run time by using python to do data analysis.

- Note: 5000 random cases' result are in file **h1_result.csv** and **h2_result.csv**

(a) The average solution depth for a randomly generated 8 puzzle instance by using h1 is *around 22* and h2 is *around 21*, both are close to the average depth 22 which describe in the project description.

(b) In the figure below, I generate extra test cases for depths 2 to 8 because the program only generate a few for these cases.

- For example, for depth 2 the program only meet 2 cases after generated 10000 random instance. The percentage for depth 2 is only around 0.02%.

Compare different between two heuristic function

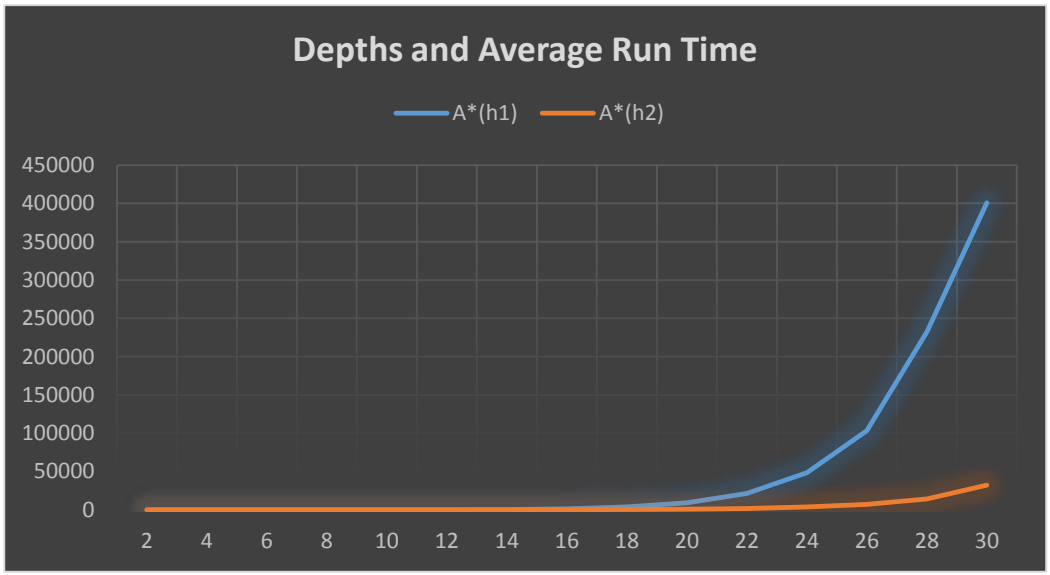
	Search Cost		Average Run Time (nanoseconds)		# of Cases	# of Cases
Depth	**A*(h1)**	**A*(h2)**	**A*(h1)**	**A*(h2)**	**A*(h1)**	**A*(h2)**
2	6	6	93805	8087	100	102
4	11	11	106795	9261	100	127
6	20	19	118862	14715	102	126
8	45	29	195197	34975	104	206
10	106	50	263130	45426	109	208
12	245	87	257697	64452	128	226
14	612	167	425023	93415	152	250
16	1440	285	459368	105760	140	127
18	3672	581	1142912	194948	286	274
20	9316	1058	2916703	346270	509	504
22	21519	1940	6864120	646996	629	581
24	48750	3779	16599192	1253119	604	680
26	103364	7218	38606029	2393443	383	410
28	232314	14380	96910806	4740205	110	121
30	400728	32183	178312582	10930378	5	9

3. Analysis

By comparing the data in the above table, we can find that the average search cost and the average run time from depth 2 to depth 6 for A* search algorithm with using different heuristic function h1 and h2 are almost the same.

As the table shows, starting from depth 8, the average search cost and average run time using h1 (the number of misplaced tiles) cost more than the one which using h2 (Manhattan Distance). And after depth 20 the average run time which using h1 increase dramatically which shows in the figure below.

Therefore, after testing 5000 random puzzles we can conclude that using A* search algorithm with h2 generates less node and cost less time to solve 8-puzzle problem. Thus, A* search algorithm with Manhattan distance heuristic is more efficient.



4. Finding

(a) Find the next step for the current state:

- By checking the result for (zero position / 3) and (zero position % 3)
  (1) Move up: if (zero position / 3) != 0
  (2) Move down: if (zero position / 3) != 2
  (3) Move left: if (zero position % 3) != 0
  (4) Move right: if (zero position % 3) != 2

(b) Find the best solution from the open set:

- Using priority queue and sort the element by consider heuristic value + depths.