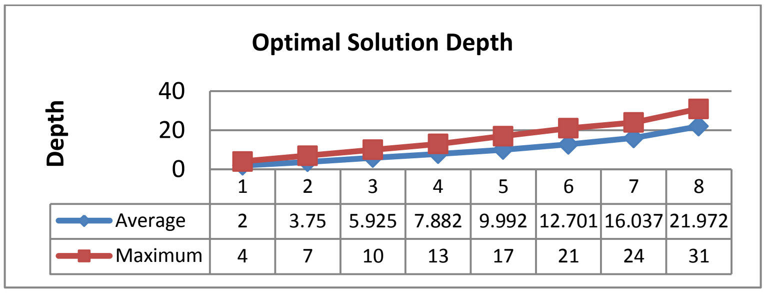
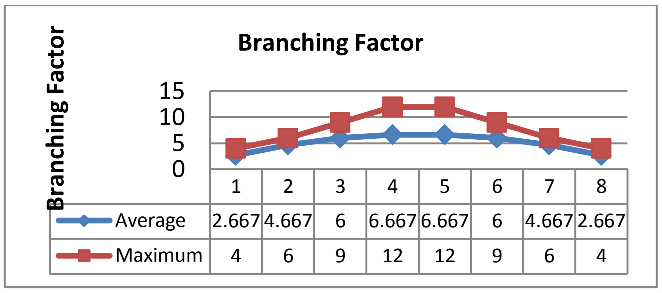
1~8 Puzzles

Problem Analysis

The complexities of puzzles with different number of tiles can be measured by two variables: (1) the average **depth** to get to the solution, (2) the **branching factor** of each step.



	1	2	3	4	5	6	7	8
Average	2	3.75	5.925	7.882	9.992	12.701	16.037	21.972
Maximum	4	7	10	13	17	21	24	31



	1	2	3	4	5	6	7	8
Average	2.667	4.667	6	6.667	6.667	6	4.667	2.667
Maximum	4	6	9	12	12	9	6	4

From our result of experiments, we can see that the optimal solution depth increases steadily as the number of tiles goes up. However, the 4, 5-puzzle have the highest branching factor, while the 1, 8-puzzle have the lowest.

Unsolvable States

	1	4
2	3	5
6	7	8

Counting from left to right and top to bottom, and let the parity be the numbers of pairs that have reverse order. For example, we can see the figure on the left as “14325678”, and the pairs with reverse order is (4, 2), (4, 3), so the parity is 2. If we divide the states of 8-puzzle into **two disjoint sets**, one with odd parity, and the other with even parity, we can show that states in one set cannot reach any states in another set.

1	3	4
2	7	5
6	7	8

When moving a tile horizontally, the orders of all numbers are unaffected, thus the parity is unchanged. When moving a tile vertically, the orders of three numbers would be affected. For example, when moving the ‘7’ up, only the orders of 5, 6, and 7 are affected. And we see that **exactly two pairs are exchanged**, this causes the parity to increase by 2, to decrease by 2, or to stay the same. Therefore, the odd or even property does not change.

The experiments I have done show that, any states with the same parity can reach each other. Therefore, exactly half states in 8-puzzle can reach the goal, which has even parity, and they cannot reach any states with odd parity. We can check whether an initial state can reach the goal by simply counting the parity.

	8	1
2	3	4
5	6	7

If we see one of the space as number ‘8’ in the corresponding 8-puzzle, we can change the parity from even to odd or vice versa by simply exchanging the ‘8’ with the tile besides it. As mentioned earlier, any states with the same parity of the goal state can reach the goal even with only the moves possible in 8-puzzle. Since all initial states of 7-puzzle can change its parity to match with the goal state, all of them can reach the goal state. Using the same argument, we can show that there are no unsolvable initial states in 1~6 puzzles. Experiment data gained when calculating optimal solution depth also confirms with this.

Notice that, because half of states in 8-puzzle are unreachable, the number of valid states is **9!/2**, which is the same as 7-puzzle. However, the branching factor of 8-puzzle is lesser. This has an interesting effect that would be discussed later.

Experiment Methods

Algorithms Tested

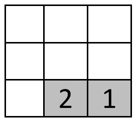
Three versions of algorithms are tested: (1) **IDS**, (2) **A* search graph version**, (3) **A* search tree version**. For A* search, three heuristic functions are tested: (1) the number of tiles **misplaced**, (2) **Manhattan** distances of all tiles from correct place, (3) **pattern database** storing the exact cost of a subset of tiles to the goal state.

Heuristic h_3
The pattern database of heuristic h_3 is produced by doing BFS from the goal state. Exact costs of all 1~6 puzzles are generated. When a puzzle with higher number of tiles is encountered, h_3 converts the tiles into correspondent problem of 6-puzzle. “102345678”, for example, is converted into “000123456”, so that only the 6 largest numbers are considered.

H_3 is admissible. For 1~6 puzzles, it is the exact cost, and for 7, 8 puzzles, it never overestimates because the optimal moves must at least put the 6 largest numbers into correct place. Also notice that the moves in 7, 8 puzzles are actually more restrictive than correspondent 6-puzzle problem.

H_3 is consistent. For any single move, h_3 decreases at most by one, but any move costs 1. That is, $h_3(n) \le c(n, a, n') + h_3(n')$ for any successor n’ of n, since h_3(n) is at most 1 greater than h_3(n’) while c(n, a, n’) is 1.

Properties of heuristic functions
As we can see, h_3 is both admissible and consistent, and it is the exact cost for up to 6-puzzle. For h_1 and h_2, they are both admissible and consistent as described in the textbook. For h_1, it is never the exact cost unless only one move is left to reach the goal. For h_2, it is the exact cost in 1-puzzle, but not in 2~8 puzzles. To see why this is the case, consider the 2-puzzle on the right. The exact cost is 4, while the value of h_2 is 2. We cannot avoid this kind of behavior in 2~8 puzzles.



Data Collection
The source code is attached, and comments are provided to explain what has been done. Most of the time, data is collected by enumerating all possible initial states, but when doing so is impossible, 100 random tests are carried out.

The tree version of A* search consumes extremely large amount of memory. It’s basically impossible to test it on the machines I have access to when the problem space is large. Even if random tests are performed, memory exhaustion still cannot be avoided. Indeed, random inputs only decreases the time needed to perform the experiments, not the memory.

The full statistical data can be found in **proj1appendix.xlsx**. (Raw data is not included because it’s too big.)

Results & Compares

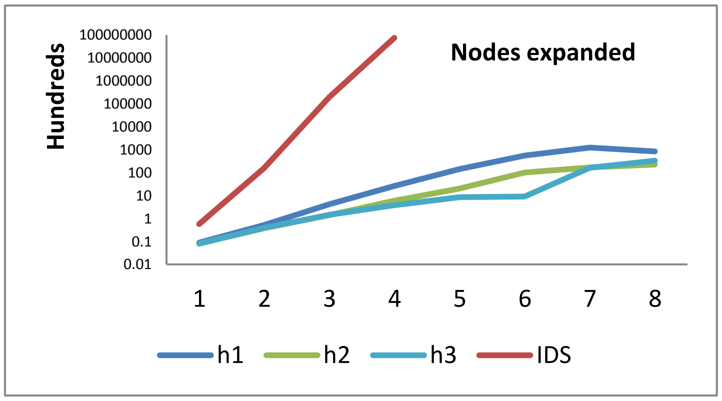
IDS versus A* graph search

Puzzle	1	2	3	4	5	6	7	8
Nodes expanded								
IDS Average	21.111	1326.847	677944.2	2.07E+08	-	-	-	-
IDS Maximum	58	16025	19746145	7.31E+09	-	-	-	-
A*h1 Maximum	9	52	427	2590	14496	54741	122312	83697
A*h2 Maximum	8	38	140	590	2022	9977	16796	22127
A*h3 Maximum	8	39	148	384	854	906	16380	33300
Nodes Stored / in Frontier								
IDS Average	4.555	15.472	32.312	52.01	-	-	-	-
IDS Maximum	7	25	50	82	-	-	-	-
A*h1 Maximum	5	32	193	951	4021	13955	32128	23913
A*h2 Maximum	5	36	177	886	2905	10442	13975	9470
A*h3 Maximum	5	36	179	667	1548	1888	14365	14125

- The data of A search with h1 in 7, 8 puzzles and IDS in 4-puzzle are randomly sampled.*
- A search is the A* graph search.*
- Notice that h_2 performs better than h_3 in 8-puzzle.

Comparing IDS with A*, we see that IDS spends much more time than A*, but IDS uses lesser space. However, we see that the space used by A* doesn’t grow fast. This is because the explored states wouldn’t be added to the frontier, so the size is limited by the number of valid states. We even notice that **in 8-puzzle, A* actually uses less space than in 7-puzzle.** This due to the fact that 7, 8 puzzles have the same number of valid states while the branching factor of 8-puzzle is lesser.

Also we can see that the nodes expanded grow asymptomatically like O((b*)^d), where b* is the effective branching factor and d is the depth, this is consistent with the textbook. Notice that branching factor is decreasing starting from 5 to 8, but depth is increasing.



Because of the increasing time used, some data is collected by doing random sampling. The time spent by IDS is so much that we can't really collect the data. Indeed, using the average branch factor and maximum depth for 5-puzzle, we can calculate that in the worst case, about 6.67^{17} nodes must be expanded. Even if we can expand 10^6 nodes per second, it still takes approximately **1185 days** to complete, which is impractical even for random sampling. Indeed, if we carefully choose sampled data, we may avoid the worst cases, but **such data collected will not be accurately representing the true situation.**

Graph vs. Tree

A graph search avoids expanding the same nodes but must use additional space to stored explored set. We see that IDS uses surprisingly small space compared with A*. We also see that IDS would have to do a large amount of repeated work once the depth is large. In the case of A* search, we see that if we use a tree search, the frontier would grow extremely large that it becomes larger than the number of all possible states, so storing explored set should be more reasonable.

Puzzle	1	2	3	4	5	6	7	8
Nodes expanded								
A* tree h1 Max	22	1056	191818	-	-	-	-	-
A* tree h2 Max	9	65	344	12059	1494557	6717390	-	-
A* tree h3 Max	9	39	379	524	2695	3093	-	-
Nodes in Frontier								
A* tree h1 Max	40	4170	1049590	-	-	-	-	-
A* tree h2 Max	15	267	1785	66809	8389795	33797053	-	-
A* tree h3 Max	15	152	2078	3174	16580	16428	-	-

It's interesting to see that h_3 works very well in 1~6 puzzle because it's the exact cost, but in 7, 8 puzzles, A* tree becomes so inefficient that it **exhausts all memory**.

Conclusion

During the experiments, I discovered how a seemingly small increase in depth can cause huge change in the time spent by IDS. The comparison between A* tree and A* graph also gives me insight into choosing between graph search and tree search. Several test routines have been designed for this project. During the process, I learned how to test the algorithms that I implemented and how to use statistical data to analyze the efficiency.

I often found strange results from the data I collected, and later figured out a bug in my program. This process takes me much time because it takes so long to generate and analyze the data. It teaches me the importance of testing before actually generating statistics data. Hopefully I can do better next time.

Remaining Questions & Future Investigation

It seems strange why A* tree can perform better than A* graph with h_3 in 8-puzzle. Some unnoticed problems may exist. Also, it should be interesting to analyze the difference of the behavior of h_3 between 1~6 puzzles and 7~8 puzzles. When the heuristic is not exact, what has changed during the A* search? What caused such a big increase in complexity? Setting $h_4 = \max(h_2,h_3)$ and we see substantial improvement over both h_2 and h_3. This gives us some hints about the problem of h_3.

Possible Improvements in A* Search Implementation

A heap is used for A* graph search. A linear search is performed to see whether the old node has higher f value when the same state is already in the frontier. Such implementation is rather slow. It may be more efficient if a hash table is used. For the memory exhaustion problem, it may be possible to manually store some nodes into a file, thereby reducing the use of memory while still be able to run the A* tree search.