Allan Gershon

March 15, 2021

CSCI 363 – Artificial Intelligence

Changhe Yuan

Assignment 1 Report

Program Design – Added to Clarify Design Decisions

For the regular A* algorithm, I may have overcomplicated my data structures in order to seek efficiency. I maintained a priority queue of pair objects containing a certain f-value and a HashMap full of nodes with that f-value. These pair objects would be put into the priority queue with the minimum f-value HashMap being on top of it. I also needed to maintain a HashMap of different f-values pointing to different maps within the priority queue to allow for constant time removal and addition, as opposed to needing to search through the whole queue. Whenever any of the priority queue's maps became empty, I removed it, and if a map corresponding to a certain f-value did not exist, I had to add it. Working with hash maps this way seemed to have randomized the number of nodes expanded and time elapsed before finding the optimal solution too. Admittedly I made fragile code in my attempt to optimize as much as possible, because when I made these design decisions, I did not know exactly what was bottlenecking my runtime. The other parts of the algorithm work as they originally should.

Across the different search algorithms, I called different helper methods since they all shared similar code, such as to expand nodes and generate children. If a procedure was done repeatedly in an algorithm, I turned it into a helper function too. For IDA* and DFBnB, my algorithm was the same as the pseudocode, more or less, except for some optimizations that are described later.

Data from Search Algorithms – separate tables for different pieces of data

Table for number of nodes expanded, time taken, and number of moves for optimal solution for each algorithm using different start states.

Algorithm Used	Starting State	Nodes Expanded	Solve Time (ms)	Number Moves
A* using Misplaced Tile Heuristic	Easy	5	8.48 ms	5
	Medium	27	4.86 ms	9
	Hard	68	6.66 ms	12
	Worst	119,756	1624.12 ms	30
A* using Manhattan Distance Heuristic	Easy	5	0.912 ms	5
	Medium	13	0.386 ms	9
	Hard	25	0.829 ms	12
	Worst	2,033	23.56 ms	30
Iterative-Deepening A* (Manhattan Distance)	Easy	5	3.82 ms	5
	Medium	10	0.18 ms	9
	Hard	29	0.468 ms	12
	Worst	592	7.12 ms	30
Depth-First Branch + Bound (Manhattan Distance)	Easy (L is infinite)	11	2.33 ms	5
	Medium (L is infinite)	21	1.17 ms	9
	Hard (L is infinite)	156	6.506 ms	12
	Worst (L = 5000)	461,722	386.978 ms	30

Table for DFBnB's Time to Find Optimal Solution

Start State	Time to Find Optimal Solution
Easy (L is infinite)	2.12 ms
Medium (L is infinite)	0.695 ms
Hard (L is infinite)	6.298 ms
Worst (L = 5000)	378.926 ms

The table for the different sequence of moves in each algorithm's optimal solution was moved to the bottom because it took a great amount of space.

Data Analysis

For each of the number of nodes expanded for each algorithm, the amount increases by a certain factor, and the increase is greater depending on the algorithm used. For example, for DFBnB and A* misplaced tile, the number of nodes needed to be expanded jumps into the hundred thousand when using the worst start state. Oddly, some of my runtimes for harder start states were lower for than some of those for easier ones. I want to guess that this is due to my computer caching the results of previous times the algorithm was run. However, when it comes to comparing runtimes between different algorithms for different start states, the runtimes for easy, medium, and hard start states are mostly in the same range while runtimes for the worst-case once again make a big jump. However, the consistent results in the number of moves needed to reach the optimal solution suggests that my algorithms were implemented correctly. Between the runtimes for A* for Misplaced and Manhattan heuristics, the latter were less than the former by almost a factor of 10. Based on my data for IDA* and DFBnB I feel as though they are too inconsistent to compare them properly. However, I can say that the worst-case for DFBnB would not terminate, unless I set a limit for the starting L-value when calling it.

Experimental Results and Analysis

Implementing the algorithms themselves was not too bad but optimizing them took a significant amount of thought. For example, for the A* algorithm, I had horrible runtime, especially for the worst-case state when I was checking for duplicate states in the open and closed lists. This is because I brute forced the search for duplicate states by checking each individual state and comparing it to a newly generated child using my isSameConfig() function repeatedly. Instead, I eventually used a HashMap to keep track of a state's string representation and only needed a single query in constant time to check if the state already existed. This reduced my worst-case runtime from 50 minutes down to a few seconds.

For my branch-and-bound portion of my IDA* and DFBnB algorithms, first, I kept a HashMap of created states and their respective costs to get there. This capped the f-values of states with which I worked for DFBnB, but still let it run for a long time. I eventually removed this HashMap because it consumed a significant amount of memory, which goes against DFBnB having to have linear space due to a lack of a closed list or any form of memorization.

For the IDA* algorithm, I had decent runtime for all given cases, except for the worst case because of the algorithm's lack of memorization. I added an optimization to check if any child state is the same as the current state's parent, to decrease the chances of looping, since this form of depth-first search could lead to that. A loop is possible this way because a number tile can move into an empty space in one step, then in the next step it moves back to the same space it was previously. Before this, my IDA* algorithm would either run in a seemingly infinite loop or take a few minutes to complete depending on different methods I tried. Other than this, I did not have much difficulty with this algorithm.

The DFBnB algorithm however gave me a lot of trouble. I tried many guesses to optimize it, none of which really worked. For example, before expanding the current state, I checked its cost so far versus the f-value of the starting state, and multiplied factors of it, and if the former was greater than the latter, I continued to the next iteration of the loop to eliminate unnecessary states. I also tried to check newly added children in the stack to see if any of them happened to be the goal state, because I figured it would be possible that a goal state child was added but would not be on top of the stack. Eventually, I was able to make this algorithm work for the easy, medium, and hard states with an infinite L value, but had to limit my starting L value for the worst-case to 5000, which still allowed me to find an optimal solution with decent runtime.

<u>Answers to Questions</u>

1) There are 9!, or 362880, possible states for the board, half of which are solvable.
2) The following 3 x 3 board below (next to the calculation for average) gives values in each square for how many possible moves are possible depending on where the empty tile space is in the current state's configuration. For example, for the center tile space, the number there is 4, because when the empty tile is in that position, there are four possible succeeding states to be made from it.

Finding the average number by adding them up and dividing by nine, we get:

$$\frac{2+2+2+2+3+3+3+3+4}{9} = \frac{24}{9} = \frac{8}{3} = 2.\overline{66} = 2.67$$

2	3	2
3	4	3
2	3	2

5	6	7

4		8
3	2	1

3) I would assume the worst-case configuration for the eight-puzzle problem would be so that each number tile is as far away as possible from the opposite end of the board, as shown in the board on the right. With this being starting state, we have made no moves so far, so I would calculate the Manhattan Distance heuristic to estimate how many moves left is there. I add up how many squares each number tile is away from its goal place horizontally or vertically (not diagonally).

$$Manhattan\ Distance = 4 + 4 + 4 + 4 + 2 + 2 + 2 + 2 = 24$$

4) Estimate of number of nodes examined with BFS: $2.\overline{66}^{24} = 16{,}720{,}512{,}102.6$

5) Assuming each move is a possible state node generated by BFS, it would take about $16{,}720{,}512{,}102.6\ ms$ to terminate. This is equivalent to about 6.36 months. The semester ends in about 2 months, so the algorithm would not be able to terminate by then.

6) The worst case is easy for an actual human to solve because when the direction from which you look at the starting state is ignored, the worst case start state is the goal state. A human can turn the puzzle facing upside down so that all the numbers are in the same relative order as the goal state, except the numbers are upside down. This would give a human the idea that they just need to swap individual squares around the sides of the puzzle to get to the given goal state. What AI can learn from this is that AI programs can also be adjusted to consider the overall details of each state at once, and not just one characteristic about it to judge future moves (i.e., the f-value, which is derived from the heuristic value). Even though depending on the problem, this can cause a lot of overhead, the eight-puzzle problem may benefit from a program being able to make judgments based on each individual number's placements for each state. Doing this may allow it to see that the worst case's starting state is very much like the goal state, just not in terms of desired orientation.

7) Discussing A*: The A* algorithm is the most reliable for its ability to find the optimal solution as its first goal state and makes effective use of h-values and g-values.

- Pros: This algorithm uses a priority queue to help find the lowest f-value each time a state node is searched. This allows the first goal state to be found to be the optimal solution, allowing the algorithm to terminate immediately. This allows the algorithm to be complete and optimal.
- Cons: Storing past states in both open and closed lists uses a significant amount of memory to the point that the program may have to terminate due to RAM memory running out.

Discussing IDA*: The IDA* algorithm is complete and optimal for its more careful approach to finding the solution, even if it uses a form of depth-first search.

- Pros: This form of DFBnB is more likely to be complete and optimal because it repeatedly searches up to a certain f-value as its depth. Therefore, instead of going down different states limitlessly, it bounds itself to see what it can find given a certain f-value. This also allows it so that the first solution found is optimal, allowing no longer need to search when it is found, unlike in DFBnB.
- Cons: Starting the search from scratch repeatedly can make the runtime increase unnecessarily depending on the depth of the solution state.

Discussing DFBnB: DFBnB can be difficult to work with, depending on the sort of problem you are working on and can easily lead to an infinite loop.

- Pros: This algorithm takes linear space because it does not maintain a closed list of expanded nodes. Instead, it drops expanded nodes from memory. Also, when it expands a node, it becomes picky about which nodes can stay (bound) based on the overall upper bound.

- Cons: The overall upper bound as the algorithm runs begins as infinite, and only lowers if any solution is found. Because of this, we would have to hope that the algorithm runs to eventually find a solution, so that eventually the optimal solution can be found. If no goal state is found at all, the lack of any closed list or memorization can cause the algorithm to run in an infinite loop. Therefore, DFBnB can only be complete and optimal if either the algorithm's upper bound or its state space depth is less than infinite.

Table for Different Algorithms and steps taken between each state for different start states.

Algorithm Used	Starting State	Step Sequence
A* using Misplaced Tile Heuristic	Easy	State #1 1 3 4 8 6 2 7 5 ----- State #2 1 3 4 8 2 7 6 5 ----- State #3 1 3 4 8 2 7 6 5 ----- State #4 1 3 8 2 4 7 6 5 ----- State #5 1 3

		8 2 4 7 6 5 ----- State #6 1 2 3 8 4 7 6 5
	Medium	State #1 2 8 1 4 3 7 6 5 ----- State #2 8 1 2 4 3 7 6 5 ----- State #3 8 1 2 4 3 7 6 5 ----- State #4 8 1 2 4 3 7 6 5 ----- State #5 8 1 3 2 4 7 6 5 ----- State #6 8 1 3 2 4 7 6 5 ----- State #7 8 1 3 2 4 7 6 5 ----- State #8 1 3

		8 2 4 7 6 5 ----- State #9 1 3 8 2 4 7 6 5 ----- State #10 1 2 3 8 4 7 6 5
	Hard	State #1 2 8 1 4 6 3 7 5 ----- State #2 2 8 1 4 6 3 7 5 ----- State #3 2 8 1 4 3 7 6 5 ----- State #4 2 8 1 4 3 7 6 5 ----- State #5 8 1 2 4 3 7 6 5 ----- State #6 8 1 2 4 3 7 6 5 ----- State #7 8 1

		2 4 3 7 6 5 ----- State #8 8 1 3 2 4 7 6 5 ----- State #9 8 1 3 2 4 7 6 5 ----- State #10 8 1 3 2 4 7 6 5 ----- State #11 1 3 8 2 4 7 6 5 ----- State #12 1 3 8 2 4 7 6 5 ----- State #13 1 2 3 8 4 7 6 5
	Worst	State #1 5 6 7 4 8 3 2 1 ----- State #2 5 7 4 6 8 3 2 1 ----- State #3 5 7

		4 6 8 3 2 1 ----- State #4 4 5 7 6 8 3 2 1 ----- State #5 4 5 7 3 6 8 2 1 ----- State #6 4 5 7 3 6 8 2 1 ----- State #7 4 5 7 3 6 8 2 1 ----- State #8 4 5 7 3 6 2 1 8 ----- State #9 4 5 3 6 7 2 1 8 ----- State #10 4 5 3 6 7 2 1 8 ----- State #11 4 5 3 6 7 2 1 8 ----- State #12 3 4 5

		6 7 2 1 8 ----- State #13 3 4 5 2 6 7 1 8 ----- State #14 3 4 5 2 6 7 1 8 ----- State #15 3 4 5 2 6 7 1 8 ----- State #16 3 4 5 2 6 1 8 7 ----- State #17 3 4 2 6 5 1 8 7 ----- State #18 3 4 2 6 5 1 8 7 ----- State #19 3 4 2 6 5 1 8 7 ----- State #20 2 3 4 6 5 1 8 7 ----- State #21 2 3 4

		1 6 5 8 7 ----- State #22 2 3 4 1 6 5 8 7 ----- State #23 2 3 4 1 6 5 8 7 ----- State #24 2 3 4 1 6 8 7 5 ----- State #25 2 3 1 6 4 8 7 5 ----- State #26 2 3 1 6 4 8 7 5 ----- State #27 2 3 1 6 4 8 7 5 ----- State #28 1 2 3 6 4 8 7 5 ----- State #29 1 2 3 8 6 4 7 5 ----- State #30 1 2 3

		8 6 4 7 5 ----- State #31 1 2 3 8 4 7 6 5
A* using Manhattan Distance Heuristic	Easy	State #1 1 3 4 8 6 2 7 5 ----- State #2 1 3 4 8 2 7 6 5 ----- State #3 1 3 4 8 2 7 6 5 ----- State #4 1 3 8 2 4 7 6 5 ----- State #5 1 3 8 2 4 7 6 5 ----- State #6 1 2 3 8 4 7 6 5
	Medium	State #1 2 8 1 4 3 7 6 5 ----- State #2 8 1

		2 4 3 7 6 5 ----- State #3 8 1 2 4 3 7 6 5 ----- State #4 8 1 2 4 3 7 6 5 ----- State #5 8 1 3 2 4 7 6 5 ----- State #6 8 1 3 2 4 7 6 5 ----- State #7 8 1 3 2 4 7 6 5 ----- State #8 1 3 8 2 4 7 6 5 ----- State #9 1 3 8 2 4 7 6 5 ----- State #10 1 2 3 8 4 7 6 5
	Hard	State #1 2 8 1

```
4 6 3
  7 5
-----
State #2
2 8 1
4 6 3
7   5
-----
State #3
2 8 1
4   3
7 6 5
-----
State #4
2 8 1
  4 3
7 6 5
-----
State #5
  8 1
2 4 3
7 6 5
-----
State #6
8   1
2 4 3
7 6 5
-----
State #7
8 1
2 4 3
7 6 5
-----


State #8
8 1 3
2 4
7 6 5
-----
State #9
8 1 3
2   4
7 6 5
-----
State #10
```

		8 1 3 2 4 7 6 5 ----- State #11 1 3 8 2 4 7 6 5 ----- State #12 1 3 8 2 4 7 6 5 ----- State #13 1 2 3 8 4 7 6 5
	Worst	State #1 5 6 7 4 8 3 2 1 ----- State #2 5 6 7 4 8 3 2 1 ----- State #3 5 6 7 4 8 1 3 2 ----- State #4 5 6 7 4 8 1 3 2 ----- State #5 5 6 7 4 8 1 3 2 ----- State #6

```
5 6 7
  8 1
4 3 2
-----
State #7
  6 7
5 8 1
4 3 2
-----
State #8
6   7
5 8 1
4 3 2
-----
State #9
6 7
5 8 1
4 3 2
-----
State #10
6 7 1
5 8
4 3 2
-----
State #11
6 7 1
5 8 2
4 3
-----
State #12
6 7 1
5 8 2
4   3
-----

State #13
6 7 1
5 8 2
  4 3
-----
State #14
6 7 1
  8 2
5 4 3
-----
State #15
```

		7 1 6 8 2 5 4 3 ----- State #16 7 1 6 8 2 5 4 3 ----- State #17 7 1 6 8 2 5 4 3 ----- State #18 7 1 2 6 8 5 4 3 ----- State #19 7 1 2 6 8 3 5 4 ----- State #20 7 1 2 6 8 3 5 4 ----- State #21 7 1 2 6 8 3 5 4 ----- State #22 7 1 2 8 3 6 5 4 ----- State #23 1 2 7 8 3 6 5 4 ----- State #24

		1 2 7 8 3 6 5 4 ----- State #25 1 2 7 8 3 6 5 4 ----- State #26 1 2 3 7 8 6 5 4 ----- State #27 1 2 3 7 8 4 6 5 ----- State #28 1 2 3 7 8 4 6 5 ----- State #29 1 2 3 7 8 4 6 5 ----- State #30 1 2 3 8 4 7 6 5 ----- State #31 1 2 3 8 4 7 6 5
Iterative- Deepening A* (Manhattan	Easy	State #1 1 3 4 8 6 2 7 5 ----- State #2 1 3 4

Distance)		8 2 7 6 5 ----- State #3 1 3 4 8 2 7 6 5 ----- State #4 1 3 8 2 4 7 6 5 ----- State #5 1 3 8 2 4 7 6 5 ----- State #6 1 2 3 8 4 7 6 5
	Medium	State #1 2 8 1 4 3 7 6 5 ----- State #2 8 1 2 4 3 7 6 5 ----- State #3 8 1 2 4 3 7 6 5 ----- State #4 8 1 2 4 3 7 6 5 ----- State #5

		8 1 3 2 4 7 6 5 ----- State #6 8 1 3 2 4 7 6 5 ----- State #7 8 1 3 2 4 7 6 5 ----- State #8 1 3 8 2 4 7 6 5 ----- State #9 1 3 8 2 4 7 6 5 ----- State #10 1 2 3 8 4 7 6 5
	Hard	State #1 2 8 1 4 6 3 7 5 ----- State #2 2 8 1 4 6 3 7 5 ----- State #3 2 8 1 4 3 7 6 5 ----- State #4

		2 8 1 4 3 7 6 5 ----- State #5 8 1 2 4 3 7 6 5 ----- State #6 8 1 2 4 3 7 6 5 ----- State #7 8 1 2 4 3 7 6 5 ----- State #8 8 1 3 2 4 7 6 5 ----- State #9 8 1 3 2 4 7 6 5 ----- State #10 8 1 3 2 4 7 6 5 ----- State #11 1 3 8 2 4 7 6 5 ----- State #12 1 3 8 2 4 7 6 5 ----- State #13

		1 2 3
		8 4
		7 6 5

	Worst	State #1 5 6 7 4 8 3 2 1 ----- State #2 5 7 4 6 8 3 2 1 ----- State #3 5 7 4 6 8 3 2 1 ----- State #4 4 5 7 6 8 3 2 1 ----- State #5 4 5 7 3 6 8 2 1 ----- State #6 4 5 7 3 6 8 2 1 ----- State #7 4 5 7 3 6 8 2 1 ----- State #8 4 5 7 3 6 2 1 8 ----- State #9

```
4 5
3 6 7
2 1 8
-----
State #10
4   5
3 6 7
2 1 8
-----
State #11
  4 5
3 6 7
2 1 8
-----
State #12
3 4 5
  6 7
2 1 8
-----
State #13
3 4 5
2 6 7
  1 8
-----
State #14
3 4 5
2 6 7
1   8
-----
State #15
3 4 5
2 6 7
1 8
-----

State #16
3 4 5
2 6
1 8 7
-----
State #17
3 4
2 6 5
1 8 7
-----
State #18
```

		3 4 2 6 5 1 8 7 ----- State #19 3 4 2 6 5 1 8 7 ----- State #20 2 3 4 6 5 1 8 7 ----- State #21 2 3 4 1 6 5 8 7 ----- State #22 2 3 4 1 6 5 8 7 ----- State #23 2 3 4 1 6 5 8 7 ----- State #24 2 3 4 1 6 8 7 5 ----- State #25 2 3 1 6 4 8 7 5 ----- State #26 2 3 1 6 4 8 7 5 ----- State #27

		2 3 1 6 4 8 7 5 ----- State #28 1 2 3 6 4 8 7 5 ----- State #29 1 2 3 8 6 4 7 5 ----- State #30 1 2 3 8 6 4 7 5 ----- State #31 1 2 3 8 4 7 6 5
Depth-First Branch + Bound (Manhattan Distance)	Easy (L is infinite)	State #1 1 3 4 8 6 2 7 5 ----- State #2 1 3 4 8 2 7 6 5 ----- State #3 1 3 4 8 2 7 6 5 ----- State #4 1 3 8 2 4 7 6 5 ----- State #5

		1 3 8 2 4 7 6 5 ----- State #6 1 2 3 8 4 7 6 5
	Medium (L is infinite)	State #1 2 8 1 4 3 7 6 5 ----- State #2 8 1 2 4 3 7 6 5 ----- State #3 8 1 2 4 3 7 6 5 ----- State #4 8 1 2 4 3 7 6 5 ----- State #5 8 1 3 2 4 7 6 5 ----- State #6 8 1 3 2 4 7 6 5 ----- State #7 8 1 3 2 4 7 6 5 ----- State #8

		1 3 8 2 4 7 6 5 ----- State #9 1 3 8 2 4 7 6 5 ----- State #10 1 2 3 8 4 7 6 5
	Hard (L is infinite)	State #1 2 8 1 4 6 3 7 5 ----- State #2 2 8 1 4 6 3 7 5 ----- State #3 2 8 1 4 3 7 6 5 ----- State #4 2 8 1 4 3 7 6 5 ----- State #5 8 1 2 4 3 7 6 5 ----- State #6 8 1 2 4 3 7 6 5 ----- State #7

		8 1 2 4 3 7 6 5 ----- State #8 8 1 3 2 4 7 6 5 ----- State #9 8 1 3 2 4 7 6 5 ----- State #10 8 1 3 2 4 7 6 5 ----- State #11 1 3 8 2 4 7 6 5 ----- State #12 1 3 8 2 4 7 6 5 ----- State #13 1 2 3 8 4 7 6 5
	Worst (L = 5000)	State #1 5 6 7 4 8 3 2 1 ----- State #2 5 7 4 6 8 3 2 1 ----- State #3 5 7

		4 6 8 3 2 1 ----- State #4 4 5 7 6 8 3 2 1 ----- State #5 4 5 7 3 6 8 2 1 ----- State #6 4 5 7 3 6 8 2 1 ----- State #7 4 5 7 3 6 8 2 1 ----- State #8 4 5 7 3 6 2 1 8 ----- State #9 4 5 3 6 7 2 1 8 ----- State #10 4 5 3 6 7 2 1 8 ----- State #11 4 5 3 6 7 2 1 8 ----- State #12 3 4 5

		6 7 2 1 8 ----- State #13 3 4 5 2 6 7 1 8 ----- State #14 3 4 5 2 6 7 1 8 ----- State #15 3 4 5 2 6 7 1 8 ----- State #16 3 4 5 2 6 1 8 7 ----- State #17 3 4 2 6 5 1 8 7 ----- State #18 3 4 2 6 5 1 8 7 ----- State #19 3 4 2 6 5 1 8 7 ----- State #20 2 3 4 6 5 1 8 7 ----- State #21 2 3 4

```
1 6 5
  8 7
-----
State #22
2 3 4
1 6 5
8   7
-----
State #23
2 3 4
1 6 5
8 7
-----
State #24
2 3 4
1 6
8 7 5
-----
State #25
2 3
1 6 4
8 7 5
-----
State #26
2   3
1 6 4
8 7 5
-----
State #27
  2 3
1 6 4
8 7 5
-----

State #28
1 2 3
  6 4
8 7 5
-----
State #29
1 2 3
8 6 4
  7 5
-----
State #30
1 2 3
```

		8 6 4 7 5 ----- State #31 1 2 3 8 4 7 6 5