Steven Perry
4/26/2022

Video
https://youtu.be/0pUGWk4qBeo

Easy	Length	Cost	Time	Space
BFS	5	17	41	36
DFS	47	235	49096	42
UCS	5	17	23	16
GBF	5	17	9	2
A1	5	17	6	8
A2	5	17	6	8

Medium	Length	Cost	Time	Space
BFS	9	31	385	278
DFS	49	223	208893	43
UCS	9	31	144	107
GBF	25	95	860	299
A1	9	31	13	15
A2	9	31	13	15

Hard	Length	Cost	Time	Space
BFS	N/A	N/A	N/A	N/A
DFS	48	202	3840410	43
UCS	N/A	N/A	N/A	N/A
GBF	N/A	N/A	N/A	N/A
A1	118	584	4658	3267
A2	118	584	4658	3267

This assignment was actually an assignment that I have already attempted before during an undergraduate class. However, as an undergraduate I was unable to properly implement most of the search functions. For this attempt I made major changes to my previous work and rewrote every search function. I also rewrote the Board class and implemented new classes to properly handle the search functions. I reused portions of my previous code that dealt with user input and the certain board related functions. My code was written in Java and utilized five different class objects. Three of the five were simple implementations of the comparator interface where I overwrote the compare function. These were used for custom priority queues that I will detail further in the write-up. The other two classes were a class that represents individual states of the board and a class to run the search algorithms on user input.

The Board class is used as a bookkeeping tool along with other important functions. Importantly, it keeps track of each parent Board that comes before it. Within the Board class is a method possibleSwaps() that returns an Iterable list of Boards that are possible children of a Board. This method checks that the boards in the Iterable list do not contain any Boards that are in the same state as one of its parent Boards. This is what I used to check for repeats to ensure there would be no infinite loops. Checking if a Board is in its' solution state is done outside of the Board class and uses one of Board's methods: check(). This method returns a score between 0-9 that indicates how many tiles are in the correct place, if it returns 9 that indicates the Board is in a solution state. This method is also used for implementing A*1 search as a means for comparison in the priorityqueue. Another important method was manhattanDistanceBoard(), which was used similarly in implementing A*2 search. I chose to put these methods in the Board class so I could easily implement the comparators for these search functions.

The EightPuzzle class is where the main method is and asks the user for their desired board and algorithm. Then it creates an instance of itself and uses this to pass the board through the desired function. My bfs function is pretty straightforward and uses a queue to make

sure each child of the original node is checked before checking their children. I had more trouble with dfs and decided to use a stack to implement it. Initially I tried to just keep checking the child of the examined node, hoping that checking for repeats in Board was enough to make it work. Eventually I realized I needed to limit the algorithm to examining only nodes that had less than 50 parent nodes. I chose 50 as I found out that the optimal solution will always be under 50. Ucs was similar to bfs, except that I used a custom priority queue instead of a normal queue. This queue used the UCSComparator class which overwrote the compare function so that comparisons between Boards were done based on the score. Lower scores were pushed to the front of the queue. I took the exact same approach for A1 and A2, except the comparator for A1 pushed higher check() values to the front and the comparator for A2 pushed lower manhattanDistanceBoard() values to the front. Gbf also used check() values, but no priority queue, so it only added Boards to the queue if it had the maximum check() value between the other Boards in possibleSwaps().

On the easy board, the most expensive in all categories was dfs. Interestingly, these values changed significantly depending on the limit number I chose for dfs. It seemed that dfs would find a better solution, with the length residing close to the limit number, if I set the limiter lower. However, setting certain limits made dfs not work on the harder puzzles so I decided to keep it at 50. While dfs seemed to have exponentially higher time scores for harder puzzles, it seemed that the space, cost and length seemed relatively constant. It seemed that the value I chose for the limiter had a bigger impact on these scores than the difficulty of the puzzle itself. With the exception of dfs, all algorithms found the exact same solution set for the easy puzzle. GBF did it with the least space and A1/A2 did it with the lowest time. Interestingly, A1 and A2 performed the exact same on all three puzzles which could be a consequence of the way I choose to order children and the way I checked for repeat Boards. For the medium puzzle dfs and gbf were the exceptions to the solution set obtained by the other algorithms. A1/A2 performed the best on this puzzle in all categories. On both the easy and medium puzzles ucs

did the same work as bfs in under half the time/space. While I did expect ucs to arrive at a better solution compared to bfs as it is able to consider the cost of moving pieces, I was surprised that bfs both reached the same solution and examined more nodes to do so. When I attempted to run the searches on the hard puzzle, my computer ran into memory problems and the program crashed for half of the algorithms. A1/A2 did work and somehow dfs worked as well. I believe the reason dfs worked was because while it does examine a lot of nodes, the space size is relatively constant so it did not lead to a memory related crash.