Documentation

Introduction and Explanation:

This puzzle solver is designed to solve a sliding puzzle game. We chose the A\* and BFS methods to solve this problem. Of course, DFS can also handle this problem, but we think it requires a lot of computation. The A\* algorithm is a search algorithm that finds the shortest path between two points, meaning that we expect the shortest path to solve the problem, rather than calculating all the paths as BFS does. A\* The algorithm works by utilizing a heuristic function, denoted as h(n), that estimates the cost from the current position to the destination, and we think it can be used in a variety of applications as well. We also took different situations into account, and used the BFS method after we tried the A\* method many times and failed for larger size. We have considered the characteristics of BFS. For example, BFS is completeness, meaning that in a finite graph, if there is a path from the started point to the target, BFS would be able to find that path. Greedy Search is important way to our consideration, which is leading us to expand our idea for puzzle question.

Here is the process of our project:

We tried to use BFS as our main method to solve the puzzle because we believed that it will always find the answer. However, after we wrote the complete code, we found it only works well for some puzzles in 3 by3 or 4 by 4 dimensions. So, we realized BFS only work for a puzzle with less steps. We re-do our code in A\* method instead. For the A\* code, we found it could handle 3 by 3 up to 5 by 5. For higher dimension, an out of memory error appeared. So, as we discovered, we found a greedy best-first search. After we applied this method, it could handle puzzles with dimension 6 and dimension7. Our code do works some of dimension 8 puzzle like board31.

We have one part only solves dimension 3 -5, another part for 6,7. Board.java is the part for heuristics and basic methods for puzzles. Node.java and Node1.java are the classes used for solving dimensions 6 and 7 puzzle. Solver.java read input, then solve it base on dimension. If less than 6, apply A\* in the main. Else, apply greedy greedy best-first search. Finally, get output in Solver.java.

Was one of the heuristics always better? Or different heuristics solved different puzzles?

We only tried Manhattan distance. This heuristic calculates the minimum number of moves required to move from the current node to the goal node in a grid world, assuming that the movement is only possible in four directions: up, down, left, and right. We know there’re other heuristics such as Euclidean Distance and Octile Distance, but we didn’t use them. Euclidean calculated the straight distance so we didn’t try that one. Octile as we know it’s an extension of Manhattan, so we thought Manhattan is fine since it worked well in A\* and greedy best-first search.

EASIER part:

Solving A normal 3 by 3 or 4 by 4 puzzle is not very difficult, we can solve this kind of board with concepts like A\* and the Manhattan distance that we have learned. It doesn't really require a lot of computation. Across a variety of methods and concepts, algorithms for smaller size of puzzle are more likely to succeed than algorithms for larger one, even if their core concepts are similar.

HARDER part

The most difficult part is how to optimize the algorithm to achieve the desired answers. In other words, we have to know how to dig deeper into the Heuristic section. It was a big challenge because it took us a long time to improve it, and a lot of mistakes happened. Since heuristic algorithms need to find acceptable solutions in a limited amount of time, they may not be guaranteed to find optimal solutions, so how to strike a balance between efficiency and solution is a key issue. If there's too much computation in this problem, it's hard to get the answer we want. After we tried many times the A\* algorithm, we found that the normal A\* was difficult to get answers in 5\*5 puzzles or above. After we understood the principle of BFS, we decided to try A new approach instead of using only A\*. We decided to use these two methods together to solve the puzzle, and it turned out to be more effective than the single method.

This is how we run the code

We need to compile both Puzzle.java and Solver.java

javac -cp src src/fifteenpuzzle/Puzzle.java src/fifteenpuzzle/Solver.java

java -cp src fifteenpuzzle.Solver board22.txt output.tx

Distribution: We almost did everything together. We worked together for the code part. We both did research for the heuristics. We wrote the report together as well.

Describe what you wrote but then didn’t include in the final project:

We tried to use the structure of the puzzle to solve this problem. For example, we first find the numbers in the first row and column and keep them. We then move on to locate the small 3x3 puzzles with the side of 1, 2, 3 (Figure 1), and move in other directions to sort out the other 3x3 small puzzles, and so on. We believe we can get the right answer this way, because the 3x3 solution is pretty reliable.After sorting the numbers in it, we can immediately determine the movement of each number. We also tried a similar approach (Figure 2), first identifying the numbers in the outermost row and column, then searching the next row and column one by one. As for other queued or unwanted numbers, we did not consider their movements.

Figure 1 Figure2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |
| 6 | x | x |  |  |
| 11 | x | 0 |  |  |
| 16 |  |  |  |  |
| 21 |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |
| 6 | x | x | x | x |
| 11 | x |  |  |  |
| 16 | x |  |  |  |
| 21 | x |  |  |  |