Members:

1. Rohit Bhardwaj 301450331
2. Aswin Budi Rahardja 301439005

Distribution

We were working on the codes together, splitting the works equally. Almost every function is done together. As such, we would like to split the score equally.

Solver.java

In this file, we have implemented the A* algorithm to solve 15 puzzles. There is a main Solver class under which there is an inner class 'Node' which represents the node and implements the Comparable interface. Each Node object stores the state of the puzzle, the previous node, the number of moves made, the priority value of this node, and the direction that was taken to reach this node. It also contains the 'MinPQ' class which implements a minimum-valued priority queue to store nodes using a binary heap data structure. The 'MinPQ' class has a number of methods for working with the priority queue. This class is being used to keep track of nodes in the search tree and to order them by their priority values, which are based on the sum of a combination of heuristics and the number of moves to reach that state.

We are using two instances of MinPQ class, which is, the original puzzle and one for its twin in two defined constructors of the Solver class taking different arguments. In these constructors, we have initialized two minimum priority queues pq1, and pq2, and then looped them which continuously removes minimum priority nodes from two queues, and then added the neighbors of each node to their respective priority queues until the solution is found.

There are different helper functions created like *getDirection()* to get L, R, D, and U as directed to reach the goal state; *moves()* which returns the minimum number of moves to solve the puzzle; *solution()* which returns iterable of 'Fifteen Puzzle' object which represents the sequence of states that lead to a solution and this method uses 'Stack' to keep track of the sequence of states, *solutionDirection()* which returns a list of strings representing sequences of moves required to solve the puzzle. Each string in the list represents a move and consists of two parts: the tile number and the direction of the move; *writeToFile()* which is used to write the solution of the puzzle to an output file. Implementing each function in the solver is quite something to do, requiring us a lot of creativity and idea, making it the harder side of the project.

FifteenPuzzle.java

In FifteenPuzzle class, we have done the implementation of a 2D board and created a lot of helper functions that help us in checking the board, getting BadBoardException, if any, and manipulating the board by moving tiles in different directions. The *makemove()* takes a tile and direction as its arguments and moves the tile in the given direction, only if it is a legal move. There is a *twin()* method that returns a new 'FifteenPuzzle' object with two tiles swapped. There is another method called *neighbors()* which generates and returns a list of neighboring 'FifteenPuzzle' objects. This part is basically an improvement from the second assignment, making it the easier part of the project.

Heuristics

For heuristics, we have made *sumDistance()* which is basically a combination of simpler heuristics using Manhattan distance between each tile's current position and goal position in the puzzle. We kinda changed our heuristics before and then to see how much time it takes to solve different boards. Some heuristics were solving a few boards and some were not. For example, with one of heuristics, board08 (4x4) was solving and board10 (4x4) was not but when using a combination of distances it was vice versa. This is one of the parts that requires more thinking and time, along with functions in the solver.

Problems that we faced

The problem that was the main issue for us was the Heap Space errors that keep happening every time we try to compile our codes. We tried with several methods including A* and deepening A* (IDA), but we apparently failed to implement the methods properly as we keep getting Java Heap Space: out of memory error. Moreover, when using A*, we have defined priority queue and hashSet of different class types which were creating errors in functions and solved that error.

Conclusion

Overall, this was an interesting project that helps us to learn about A* algorithm while implementing it to solve the puzzle. The difficulty level for the project varies, it can be so easy if we do not do it the efficient way, but it is so hard to achieve the efficient implementation for the algorithm, as we do not think we achieved it after working for more or less 2 weeks. We think that this project is helpful enough to prepare us for future learning, especially in Java.