FIFTEEN PUZZLES SOLVER DOCUMENTATION

(The journey to success)

Student 1:Leang Paul KHO (Paul)
ID: 301563533

Student 2: Do Hoang Quan Le (Kevin)
ID: 301452147

The journal is divided into 3 parts and a summary which explains the process and the logic behind our code. Further explanations are explained as comments in the code.

Part 1: Approaching stage

- *March 25-26, 2023:*
    - Get to know the project, brainstorm and do as much research as possible.
    - We got the general ideas:
        - Fifteen puzzles and graph theory are related by states as vertices. We have decided to put each state as a child from a class called State , give them something to identify between different stages which is our hashCode(at the time we still didn't know how to implement it).
        - During this time, we also watch many videos on how to solve 15 puzzles as well as research on A* algorithm.
            - In order for A* algorithm to work, we need to have something that keeps the distance from start as well as the heuristic which we found out needed to use Manhattan.
            - After these days, we conclude that there needs to be 2 classes, one already given which is Solver and the second one is State.
            - **<u>Conclusion:</u>**
                - Inside the State class, we will keep the distanceFromStart, calculate the heuristic and return the cost f() function by adding those 2.
                - Inside the Solver, we will keep a goal board, and the dimension of the board.

- *March 27- April 2th, 2023:*
    - We finished doing more research and divided up the work:
    - In term of research, we concluded that we need implements:
        - PriorityQueue
        - HashSet: to keep track of whether a state is visited.
        - hashCode function
        - Generate possible states
        - Read the board from the file into a matrix
        - Stack: to reverse and store the moves. We decided that the solver will return a node with final moves then we just keep getting the parents which are the moves to get to it.
        - Calculate manhattan.

    - In term of dividing up the work:
        - After listing out all the requirements, we see that it is quite easy to just each person work in one class.
            - Paul worked on the Solver which includes:
                - Reading files into a int[][] matrix. For this, we used the code from assignment 1.
                - Set up the Priority Queue,HashSet: built in.
                - Generate the goal board

        - o implemented the hashCode function in State class as well as the manhattan.
    - Kevin worked on State which includes:
        - Generate different states of the board and also update the distanceFromStart of the children by adding 1 to its parent.
        - Finished up implementing the State class by adding setters and getters, compareTo, getF...
    - We also worked together on implementing the A* algorithm which pseudo-code was given by the Professor during class. We decided to have a hashmap to check whether the state is already in the Queue.
    - **<u>Conclusion:</u>**
        - After implementing and hours of debugging as well as merging, we were able to solve the very easy board really fast.
        - Data structure used: Stack, Priority Queue, hashMap, HashSet, hashCode, matrix.
        - Algorithm: Manhattan + distanceFromStart for cost function, A*. For generating possible boards, we find the position of the blank and then work with it to get different possible stages.
        - For the Manhattan, we just the location of the current node and subtract it to its goal's location then add them all up to get the heuristic.
        - Problem: only work for easy up to board up to 15 (all 4X4 boards and some 5X5).

Part 2: Debugging stages

- *April 3rd to 7th:*
    - <u>Big changes:</u>
        - After days of debugging, recopying the A* pseudo code, changing hashCode, manhattan and it still did not work for any harder board.
        - We found out that we should not include the distanceFromStart into the cost function.
        - Therefore, we are using pure heuristic search with linear conflicts instead.
            - For most of this stage, we mostly worked together and did not really divide up the work.
                - Technically , the changes were quite small which is just to remove the distanceFromStart and anything that related to it.
                - During this period, we tried to use different ways to hashCode
                    - Using the built in function, using prime number 31, convert to string and then make the hashcode. We decided to keep it simple by using the prime number.
                - For Manhattan, instead of using the old way which requires 2 for loops to find the location of one node and then another 2

for loops to find the location of that node on the goal board. We decided to use a HashMap <Integer, {x,y}> to get the location.

- Conclusion:
    - With Great progress, we were able to solve all boards except 7x7 very fast(up to board 25 in the test case), however anything above that failed.

Part 3: Final stage

- *April 7 - April 10*
    - Final changes :
        - After using only heuristic values that consisto linear conflicts and Manhattan we found out that it works on the smaller board i.e 3x3 and 4x4 board but it does not work well on the larger board i.e. 5x5, 6x6, 7x7 board.
        - We continue doing further research and come up with a better way to calculate the heuristic value which consist of :
            - Manhattan distance
            - Number of misplace tile
            - Euclidean of the board
            - Linear conflicts
        - We implement the method to calculate the linear conflicts , the method to calculate the number of misplace tile and the method to calculate the euclidean of the board
        - After making the changes above, we realise that we can solve boards up to 7X7 without any issue.
        - **Conclusion:**
            - We were able to solve all the boards up to board with dimension 7X7 without any issue.

Summary

1) *Data structure used :*
    a) PriorityQueue
        i) To store all the possible state
        ii) All the state in the priority queue is sorted based on their heuristic value
    b) HashSet
        i) To keep track of the element in the priority queue
        ii) act as a quick lookup table for the board that are currently in the priority queue
        iii) Optimize the code
    c) hashCode function
        i) Hash the matrix into a “int” data type
    d) Stack
        i) Since all the move is get by using the “getParent()” method, we use stack to store the move in remove order then we use the built-in “pop()” method to get the move and store the move into a .txt file
    e) HashMap
        i) Store all visited state
    f) Matrix
        i) Use to store all the different combination of board

2) *Algorithm used:*
    a) A*
        i) To find the optimal solution as possible to the goal states.
    b) Pure heuristic search with linear conflicts, misplaced titles and ECD and manhattan distance
        i) After multiple attempts , we found out that the more parameters we add to the board, the better and more efficiently it solves.
            (1) Pure heuristic is calculated using manhattan
            (2) The linear conflicts are calculated based on conditions about row and column.
            (3) Misplaced title checks the number of nodes that is not in the correct position.
            (4) Euclidean: Similar to Manhattan distance, but now distances are calculated using the euclidean formula.

3) *Takeaways:*
    a) This is a very fun project for both of us because we have never experienced coding in pairs before. Lots of fun, lots of bugs especially when merging the code.
    b) Understand A*, a very powerful algorithm for path finding.
    c) Learn and experience so many useful things, especially googling/research skills.
    d) Learned how to deal with conflict and teamwork