Kevin Chang

Part 3 Write Up

* Description of your data structures for the tray and blocks
  + Solver
    - Description:
      * The Solver class attempts to manipulate a the blocks in a board with a starting configuration determined by an input file until the board's configuration is the same as the blocks given by a goal file.
    - Important Instance Variables:
      * ArrayList<Block> goal
        + An ArrayList of blocks determined by the goal file. This is used to check whether a board has the same configuration as the goal configuration.
      * HashSet<Board> hash
        + A HashSet of Boards to keep track of which Board configurations have been visited. This makes sure that the solver does go into an infinite Board loop.
      * PriorityQueue<Board> bQueue
        + A PriorityQueue of Boards to keep track of which Boards to consider next. Since a pseudo-BFS is implemented, all the Boards at the current depth of the graph of Boards are analyzed before moving on. A PriorityQueue is used instead of a normal Queue to make the solver “smarter” by analyzing the Boards closer to the goal configuration first.
  + Block
    - Description:
      * The Block class represents a Block that will be present in the Board. The Block functions mainly as a placeholder as possible moves are considered and executed. Multiple Blocks are also used to verify whether the goal state has been reached.
    - Important Instance Variables:
      * int height
        + The height of the block
      * int width
        + The width of the block
      * int row
        + The row position occupied by the upper-left corner of the block
      * int col
        + The column position occupied by the upper-left corner of the block
  + Board
    - Description:
      * The Board class represents a board of blocks that can slide into empty spaces in the board as long as it does not collide with other blocks
    - Important Instance Variables:
      * Block [][] board
        + A 2-D array of Blocks to keep track of the position of the Blocks in a Board and to quickly determine empty spaces.
      * ArrayList<Block> goal
        + An ArrayList of blocks determined by the goal file. In the Board class, this is used to determine the “score” of the Board, which is a numeric representation of how close the Board configuration is to the goal configuration
      * HashSet<Block> bContent
        + A HashSet of Blocks that stores the Blocks in the Board. This is for quick access of the Blocks without the need to search the entire 2-D array.
      * HashSet<Coordinates> adjEmptySpaces
        + A HashSet of Coordinates that keeps track of the empty coordinate spaces adjacent to blocks. This is most effective when the Board is mostly empty and there is no reason to analyze all the empty spaces in the Board.
  + Coordinates
    - Description:
      * The Coordinates class represents the position of a space in the Board by storing row and column values.
    - Important Instance Variables:
      * int row
        + The row position of the coordinate
      * int col
        + The column position of the coordinate
* A list of the operations on blocks, boards, and the collection of boards seen earlier in the solution search
  + Blocks
    - getAdjacentSpaces()
      * Generates an ArrayList of nonnegative Coordinates that are adjacent to the block. Spaces touching the corner of the block are not included because the blocks cannot move diagonally
  + Boards
    - moveBlock( int row, int col, int dir)
      * Attempts to move the block at (row, int) in the direction dir. Returns the new Board configuration or the original if the move was impossible
  + Solver
    - generateNextBoards()
      * Iterates through the empty spaces adjacent to blocks, and attempts to move adjacent blocks into itself (e.g. a block above an empty space would attempt to move down). If the block is present and the move is possible, meaning there is no collision with other blocks, and the new Board configuration generated has not been seen before, add the new configuration to the priority queue and hash set.
    - solve()
      * While the priority queue is not empty and the current Board being analyzed is not the goal configuration, set the currBoard to the next Board in the priority queue, generate the new board configurations stemming from currBoard, and add them to the priority queue.
      * Once the solution has been found, or the puzzle is determined to be impossible, print the moves taken to attain this conclusion and the final board configuration.
* A list of alternative strategies for selecting moves and alternatives for representing blocks, boards, and the collection of already-seen boards. Analysis of the advantages and disadvantages of each alternative in these lists
  + Breadth First Search of possible moves from a starting configuration
    - Advantages
      * Makes sure that all the possible movements are considered at the current depth
      * If the search is directed, meaning the configurations closest to the goal are looked at first, this search type can be made more efficient
    - Disadvantages
      * Must consider all the possible moves even if a move takes the board configuration further away from the goal
  + Depth First Search of possible moves from a starting configuration
    - Advantages
      * Efficiently finds the board configuration if the moves are chosen intelligently, meaning that the board will choose the move that makes the board closer to the goal configuration than the other possible moves
    - Disadvantages
      * Can overlook the goal configuration if the solver is not intelligent and would thus require many undo’s
  + 2-D array representation of the board
    - Advantages
      * Allows for quick and easier access to empty spaces since the solver only needs to look for spaces equal to null
      * Can help visualize the state of the board
    - Disadvantages
      * Costs a lot of memory to store all the boards especially as they get very large
      * Takes a long time to determine whether two boards are equal and whether the board is the goal configuration
  + List of blocks to represent the board
    - Advantages
      * Very memory efficient to store the state of a board
      * Can easily verify whether two boards are equal and whether the board is the goal configuration
    - Disadvantages
      * Inefficient when determining the empty spaces in the board
  + Only consider empty spaces adjacent to blocks when determining possible moves
    - Advantages
      * Does not waste time looking at empty spaces that cannot be occupied by moving a block once
    - Disadvantages
      * Is more inefficient when the board does not have a lot of empty spaces since the program needs to filter out the empty spaces from all the possible spaces adjacent to a single block for every block in the board
* A description of the alternatives you adopted and of evidence you collected that support your choices.
  + Directed Breadth First Search of possible moves from a starting configuration
    - I implemented this using a priority queue instead of a normal when storing the Board configurations. This allowed the solver to look at the Board configurations in the priority queue that is the closest to the goal configuration instead of in the order they are put into the queue. This results in a more efficient search for the goal configuration because the board configurations most likely to lead to the goal configuration are considered first. Testing the use of a priority queue v. a normal queue showed that the solver using the priority queue had a solve runtime that was a bit faster than the runtime using a normal queue.
  + Only considering the empty spaces in the board that are adjacent to blocks when making moves
    - Instead of looking at all the empty spaces in the board, the solver only looks at the empty spaces that touch the edges of a block. Since blocks can only move one space in a single direct per move, there is no reason to look at empty spaces that do not touch blocks since nothing can move into that space in a single move. This is most effective when the board is very large but with very few blocks, so there are many empty spaces that do not have adjacent spaces that are occupied by blocks. For example, in big.tray.2 there is 100x100 board with one 1x1 block. In this puzzle, a maximum of 4 moves can be made form that one configuration, so there is no reason to consider the other 95 empty spaces that the block does not touch. After implementing this alternative method, the runtime of the solver decreased over 5 times.
  + Using both a 2-D array of blocks and list of blocks to represent the board
    - Certain methods required of the board can be accomplished more easily by looking at the 2-D representation, such as checking if moving a block would be cause an overlap. On the other hand, a list representation can be more effective in certain cases, such as determine how close the board configuration is to the goal configuration and checking whether two boards are equivalent. In addition, there are methods such as determining empty spaces that are adjacent to blocks that need to implement both representations. In this case looking at the adjacent of all the blocks generates the possible adjacent spaces and these spaces are checked to determine whether they are empty or not by looking at whether the space in the board is empty.