Division of Labor

Samuel Chia – solver class & main method, additional tray methods, efficiency improvements, hashcode methods, solve method, readme

Caroline Chen – base code & methods for tray class, piece class, minor efficiency improvements, solve method, readme

Elton – solve method, efficiency improvements, bulletproofing the solver, readme

Design

Design consists of a Solver class and a Tray class. The Tray class contains a nested Piece class.

A Piece is an object that contains a single primitive integer array of four integers. These four numbers represent coordinates of the top left corner and the bottom right corner of a piece in the tray. The Piece class then has methods that return the top left x-coordinate, the top left y-coordinate, the height of the piece, and the width of the piece, all of which can be derived from the two pairs of coordinates. There is also a method that returns the whole array of the coordinates. Lastly, we overwrote the equals and hashCode methods. Equals used the equals method of the Java Arrays class to check for equality between the coordinates of one Piece and the coordinates of another. The hashcode of a Piece is generated by creating a string of numbers—the first number is the y-coordinate of the Piece, the second is its height, the third is the x-coordinate, and the fourth is its width. The string is then parsed into an integer and returned as the hashcode of the Piece. This method is deterministic and takes into account the location and size of the Piece when generating the hashcode.

The Tray consists of a 2D array of Pieces. From the main method of the Solver class, a scanner reads lines from the initial tray configuration file. The first line is passed in to construct a Tray with the specified dimensions. Subsequent lines are passed in as arguments to the Tray's add method, which populates the array with new Piece objects at the given coordinates. This works similarly to a 2D array representation of a checkers board. However, since pieces in a sliding block puzzle can be of varying sizes as opposed to checkers pieces which take up only one spot on the board, multiple coordinate positions of the Tray may be occupied by the same

Piece. This is accomplished in the Tray's add method by first creating the Piece, and then setting the array at the given coordinates to contain the Piece.

Trays also have parent and moveMade instance variables. These variables may be null in the case of the initial tray configuration, but later generated Trays will contain parents, which are the tray configurations from which the current configuration is derived from (between any Tray and its parent Tray, there should be a difference of only one Piece moved in one direction), and moveMades, which are primitive integer arrays containing the first, two integers representing the initial top left coordinates of a Piece, and second, another two integers representing the moved top left coordinates of the same Piece.

The Tray class also contains canMove methods that return a Boolean indicating whether a specified argument Piece can move up, down, left, or right on the board depending on whether there is enough empty space for it to do so. These canMove methods are utilized within the getMoves method, which adds possible tray configurations that can result from a given configuration to a Tray's possibleMoves LinkedList instance variable. It does so by iterating through every single Piece in the Tray. For each Piece, if a canMove method returns true for a particular direction, the corresponding move method is called, which returns and adds a Tray to possibleMoves. Directional move methods take a Piece as an argument, generate new coordinates for the Piece if it were to move one spot in the specified direction, generate a move integer array to represent the move made, and then call a common move method. The common move method takes in three arguments: a Piece, which is to be moved, an array of coordinates, which are to be the new coordinates of the specified Piece, and an array representing the move that will be made. The method creates a new Tray with the current Tray as the parent and the move as the moveMade instance variable. Then, it iterates through the Pieces of the current tray. If the Piece is the Piece that was passed in as an argument to move, it adds the Piece to the new Tray in at the new coordinates. Otherwise, for all other Pieces, it adds the Piece in at the same coordinates as in the current Tray.

The trayMatch method compares two Trays—the current one and a given one—and returns a Boolean depending on whether the current Tray contains the same Pieces in the same places as the goal Tray. The current Tray may have additional Pieces that the goal Tray does not contain, but the method will return true so long as every Piece in the goal Tray has a matching Piece in the current Tray at the same coordinates. This is accomplished by iterating through every element of the 2D Piece array of the goal tray: if the spot contains a Piece, the method checks the same spot in the current Tray. If there is a Piece present, the method compares the height and width of the two Pieces. If the Piece location, height, and width match, the iteration continues. If any of these conditions do not match, the method returns false.

The Solver class's main method creates two Trays right off the bat—a Tray representing a given initial configuration and a Tray representing the goal configuration. It scans in the lines from the two given files, making two ArrayLists of strings and passing each ArrayList into the static addToTray method, which populates a given Tray with Pieces at the coordinates in the ArrayList.

The Solver class itself contains two instance variables: a fringe Stack and a visited HashSet of Trays. These variables come into play after the main method has initialized the initial and goal Trays and initializes a new Solver, and then calls its solve method.

Solve takes in an initial Tray and a goal Tray. It iterates through all possible configurations stemming from the initial configuration—which is automatically added to the fringe when a Solver is created—by getting the moves of the initial Tray and adding them to the fringe. When a Tray is popped off the fringe, it is added to the visited HashSet, and its possible resulting Trays are also generated and added to the fringe. While the fringe is not empty, this process is repeated, although only Trays that have not yet been seen (i.e. are not contained within visited) will be added to the fringe to limit configuration repetitions. Furthermore, before getting possible moves, the first thing executed within the while loop is a check for whether the popped Tray matches the goal Tray, using the Tray class's trayMatch method. If this condition is satisfied, a new solution Stack is created, and the Tray that matched the goal configuration is pushed in. From there, we follow the path backwards by continuously pushing in the parents of the Tray until the initial Tray is reached. Then, from the solution Stack, we pop off Trays and print out their moveMade variables, ultimately printing the path from the initial configuration to the goal configuration.

Experimental Results

Experiment 1

Summary: For our first experiment, our implementation used a 2D array as a representation of the tray. Our tray kept track of its own blocks by creating Piece objects. Each Piece object is only responsible for holding onto its coordinates, which were stored as int arrays. We then created a 2D array of Piece objects, storing the Piece in the array at its respective X and Y coordinates. Our Solver class reads in configuration files line by line and uses the coordinates provided to create a new Tray object and add each Piece into the Tray. To solve the puzzle, each Tray object also keeps an ArrayList of Trays that contain all possible Tray configurations that can be obtained by moving any Piece by one index. Our Tray object has a method getMoves(), which gets every Piece on the tray and checks if each one has a valid position it can move into. If a valid move is present, the move is recorded and added to a new Tray object, which is then added to the ArrayList of possible Trays from each move. Each time a new Tray is created by a move, its parent is also recorded and passed through its constructor. Our Solver takes advantage of the possible moves by iterating through them via DFS. First the initial Tray is added to the fringe, then popped off and the possible moves are created. If the Tray popped off the fringe matches the goal configuration file, a stack of moves is created iterating backwards from the last Tray by getting its parents until the initial configuration is reached. The moves recorded in the stack are then popped off one by one and printed out. To prevent infinite looping by checking Trays we have already gone through, we created a HashSet of trays that keeps track of the ones visited already. This was made possible by creating a new hashCode and equals method for both the Piece objects and the Tray objects. Our Tray object creates its hashCode by taking advantage of the Arrays.deepHashCode method, creating its hashCode out of the 2D array of pieces that we implemented. This is made possible because our Piece object has a unique hashCode based on the X, Y coordinates of the top left corner and the height and width of the Piece. This ensures that the hashcodes of each tray are relatively unique to the pieces on the board. Our equals method for the Tray objects is relatively slow but effective, because it checks every spot on the 2D array against the other to see if the Pieces at every coordinate are the same. To ensure our Piece equals method works properly, we compare the coordinate arrays saved in each Piece.

Results: Our first experiment was relatively successful because we were able to pass all but 2 of the hard tests provided in the script. The two tests we were unable to pass were test 5 and test 7.

Conclusion: Pretty good Tray implementation, but not good enough. Runtime is too slow because hashing and checking if the trays are equal involves iterating through the entire board and checking each piece individually. Other arrays that may also slow down the process is the creation of the new Trays based on possible moves. Each tray has to call the add method on every Piece already included in the old Tray.

Experiment 2

Summary: Our second experiment involved using a 1D array of Pieces instead of a 2D array. The implementation was basically the exact same methods and variables of our first experiment, except that we heard 1D arrays have faster loading and getting speeds so we wanted to see if that was a significant enough speed increase to beat tests 5 and 7. The 1D array works the same way, except requiring simple calculations to find the index of each Piece. The Piece array is of length max X * max Y, and coordinates are accessed by x + (y * x length).

Results: Our implementation was still slow that it could not pass test 5 and 7, so we reverted back to our original design.

Conclusion: Speeding up our add method by using the 1D array was still not a significant enough speed increase. Equals and hashing is most likely the bottleneck of our code.

Program Development

We began by first creating the Tray and Piece classes, because we believed that visualizing the puzzle itself and understanding how our implementation functioned would allow us to better understand how to solve it. The Tray's 2D array was declared, a Piece class was made with the top left x- and y-coordinates and height and width measurements, and the add method was put into the Tray class, as well as the trayMatch method to determine whether two Trays matched each other. Then, we began to work on the Solver class's main method, the first part of which was to read the two argument files and from them, create initial and goal Trays.

With a basic visualization of Trays complete, we moved on to program canMove methods, and from there, the actual move methods. These were then both utilized in the getMoves method, which we made because we knew that we would later want to iterate through all possible Trays stemming from an initial tray configuration.

Finished, save for some refinements and efficiency improvements, with Tray functionality, we began to write the Solver class's solve method. We'd had a basic idea of how we were going to go about solving the puzzle, and had written prior classes and methods conducive to that logic, and now that those methods were written, it was much more clear as to how we could iterate through our generated Trays and build a path from initial Tray to goal Tray. Of course, things were not perfect right off the bat. It was while we were writing the solve method that we added parent instance variables to the Tray class, writing a new Tray constructor for that purpose.

We tested our code by running Checker.jar to get a general sense of whether our code was working. Once we'd narrowed down the amount of tests we were failing, we'd open up the test files individually to examine the tray configuration so we could identify what kinds of cases we were not considering with our code. We'd then fix our code accordingly.

After we passed a satisfactory amount of tests, we looked towards how we could improve our runtime efficiency. We removed unnecessary variables and processes where we could. For example, it was at this point that the Piece class lost instance variables for every single feature and instead only contained a coordinates variable—all other features could be calculated from these coordinates. Where we were previously creating an additional result ArrayList from a result Stack in our solve method, we removed the redundant result ArrayList and used only the Stack.

Disclaimers

Our Solver does not properly solve test 5 and 7 in under 80 cpu seconds of the hard test script. We believe that it should still get the proper move output, it is just too slow to finish finding all moves within the time limit. We also spent too much time tweaking the current design of our Tray class to make it faster (finding new hashcodes, faster equals methods), which was not good enough to solve the 2 tests. We should have spent more time finding a completely new design, which is why we are one experiment short in the experimental results.

Improvements

One other way we could have tried to speed up the program is to use heuristic search along with Depth First Search. We can calculate the total distance from one next tray to the goal tray by adding up all the estimated distances between pieces in the goal tray and the corresponding pieces in a next tray. Among all the possible next moves (next trays), our program will select a next move that has the shortest total distance to the goal tray and repeat the same algorithm when finding every next move, and therefore we will obtain a shortest path to the goal tray instead of going through all possible paths. For the estimated distance, we can calculate the Manhattan distance of two reference points of pieces, where a reference point can be the top left coordinate of a piece. For example, p1 at (x1, y1) and p1' at (x2, y2), the Manhattan distance between them is |x1 - x2| + |y1 - y2|. After implementing this method, we expect our program will run more efficiently, because we would be attempting to select the shortest path towards the goal as opposed to randomly iterating through all possible paths stemming from the initial configuration.