James Gupta
David Phan
Michael Hester
Max Schridde
Section 107

Project 3 Readme

DIVISION OF LABOR

For the last project, we found that working together as a group through everything worked best for us. Therefore, there was not much division of labor between any of us as a group for this project. Initially, we all sat down and conceptualized different design frameworks for the project as a whole. Once we agreed on what we believed to be the best design choice, we would all work on one computer, debugging and writing code together as a group. If we found multiple bugs at once, or wanted to research some sort of new functionality or class that was a possible implementation, we would divide and conquer. However, most time was spent together talking through ideas and programming as a team. Sometimes, our colleagues Max and Michael would work on a separate computers doing individual tasks related to the project. But as a whole, we all contributed to our design document and our ReadMe. Overall, work was distributed equally as a result.

DESIGN

Our program has 3 classes: Solver.java, TrayConfig.java, and Block.java. We will now discuss our implementation of these three classes in depth.

TrayConfig

TrayConfig objects represent a specific configuration of blocks. Two important instance variables in TrayConfig are HashSet<Block> blocks and HashSet<Point> emptySpaces. The blocks hash set contains all of the blocks in a certain TrayConfig. Blocks are represented by block objects, and are defined by their top-left and bottom-right coordinates. emptySpaces is a hash set that contains all of the points that are empty within the tray configuration.

Another very important part of the TrayConfig class is the implementation of heuristic. This function assigns a TrayConfig object a heuristic, which is basically a value that represents how close it is to the goal configuration. This is extremely useful, as we can determine which possible tray configurations are the most promising. This allows us to work on the most promising TrayConfigs before investigating ones that are further from our goal. The implementation of this will be discussed more in the future.

TrayConfig is mostly made up of one large method, possibleMoves. This method checks to see what possible moves can be made given the current tray configuration and the block that we want to move. For every move that can be made, we create a new TrayConfig object that would result from making that move (either a move left, right, up, or down of the block).

Solver

Solver is what ultimately solves the puzzle. Therefore, it has many important components. Its more basic instant variables are “initial”, which stores the initial tray configuration, “goal”, which stores the goal tray configuration, “width”, which stores the width of the tray; and “height”, which stores the height of the tray. The more complicated and important variables are

the HashSet<TrayConfig> alreadyConsidered and PriorityQueue<TrayConfig> toConsider. alreadyConsidered keeps track of the tray configurations that we have already considered, keeping us from trying identical tray configurations again and therefore improving our code's runtime. toConsider keeps track of the tray configurations that we have created but have not yet "considered", or investigated fully. These are kept in a priority queue that is based on each tray configuration's heuristic value, as mentioned previously. Therefore, tray configurations with better heuristic values are considered before ones with worst heuristic values are. This allows us to get to the correct answer faster.

The Solver class only has two methods: consider and printSolution. The consider method is where Solver actually solves the puzzle. This method checks to see if we have reached our goal, and if we haven't it investigates the next best tray configuration. It continues to do this until we either reach our goal or run out of options. The printSolution method simply prints out the solution by starting with the tray configuration that reached our goal, and following its parents all the way back to the initial tray configuration, printing out every move made to get to each of the tray configurations along this path.

The Solver class' constructor does a significant amount of setup before it actually begins solving the puzzle. First, it takes in the first file and determines the tray's width and height based on the first line. Then, it goes through the rest of the lines and reads where the blocks are supposed to be located, and keeps track of them in a hash set. Then, using the locations of these blocks, it determines where the empty spaces are on the board, and keeps track of them in a hash set of points. Once this is done, it repeats this process for the second file, which contains the goal

configuration information. Finally, it creates two TrayConfig objects: one that represents the initial configuration, and one that represents the goal configuration.

Block

The Block class is very simple in relation to the other two classes. Each Block object has instance variables that keep track of the coordinates of its top left and bottom right points, and also has variables that contain its width and height. We also created a specific hashcode for blocks that made it so there was less collision when storing them in our TrayConfigs. Our hashcode makes sure that blocks with similar coordinates in different orders do not collide.

How It Comes Together

The program revolves around the actions of our Solver class. First, the Solver class takes in the files and, as explained before, transforms them into an initial TrayConfig and a goal TrayConfig. Then, it runs the consider method on the intial TrayConfig. Consider initially checks if our initial TrayConfig is equal to our goal, and if its not, it runs possibleMoves on this initial TrayConfig to get the tray configurations that would result from all of the possibleMoves that can be made from it. Then, it takes these TrayConfigs and adds them to a priority queue, where it then chooses the best one based on its heuristic value, and does the same process for that. This process continues until we find a TrayConfig that is equal to our goal. When we reach this point, we print the solution.

EXPERIMENTS

Experiment 1: Recursion vs. Iteration in the "consider" method

When we were first working on this project, we first thought it would be best to do the considering of each tray configuration recursively. What we did was we first called our consider method on the initial tray configuration to see if it matched the goal configuration. If it was, then great, we finished the puzzle. If not, then we look at the possible moves from that initial configuration and recursively called the "consider" method on each of the possible moves that could be made from the first block that the consider method looks at. We would then add each TrayConfig object to a hash set titled AlreadyConsidered to prevent any repeated tray configurations to be shown in our recursive calls. The generation of possible moves were done by shifting each block in the tray config by one space and determining whether each shift was a legal move or not. This would then be added to a hash set of results containing possible tray configurations that reflect possible moves from an initial tray setup. The consider method would be called recursively on every tray configuration returned from the hash set of possible moves, and wouldn't stop until the possible moves were all exhausted, or the goal has been reached.

This method worked until we hit the puzzles of the hard difficulty. What happened was we were getting runtime errors in our program. At first, we were confused on what was happening. Because the error messages were flooding the terminal display, we couldn't figure out what was happening or what the specific error was. We finally figured out what the error was when we decided to put a try-catch statement when running the recursive step, and it turned out to be a StackOverflow error. It turned out that our consider method was taking in too many calls and the memory overflowed. Upon this realization, we knew we had to take a different approach.

We considered iteration as a possible alternative approach to looking at each possible move to match with the goal configuration. Instead of recursion, we used a “while” loop that would stop when a tray configuration matched the goal configuration. We also incorporated a PriorityQueue that would look at the heuristic of each tray configuration (more on this experiment later). We would add the initial configuration to the front of the queue. Then, the tray configuration in the front would be polled and its possible moves would be added to the queue. Afterwards, if the polled configuration does not match the goal configuration, then the next value would be polled. The process continued until the goal configuration was found or there was no solution.

This process worked much more efficiently than the latter. We found that we were able to pass more tests and take advantage and incorporate of our heuristic better to find the most efficient way to get to the goal. What we learned was that although recursion is very powerful, it takes up a lot of space and if a function or method is called recursively too many times, then an error will most likely (if not always) be thrown as a result of this. Though we did not pass all tests, an error was never raised, we just ran out of time.

Tests Passed with Given Method:

	Recursion	Iteration
Easy	All tests passed	All tests passed
Medium	5 tests failed	1 test failed
Hard	All tests failed from error	20/28 tests passed

Experiment 2: Heuristic vs No Heuristic

Upon our initial start on the project, we thought mostly about how to make our search algorithm solve puzzles. Although the thought of a heuristic seemed appealing and could greatly optimize our code, we figured we wouldn't get to it until our program was actually solving puzzles. When we were finally able to solve the easy puzzles and some of the hard ones, we then began to think of how we could come up with a heuristic to help expedite the search algorithm's selection of configurations.

The heuristic we ended up setting up was based on how far away it was from the goal configuration (which was set as a public static variable). Initially, every heuristic has a value set to Integer.MAX_VALUE. Then, for every block within the tray configuration, if the block matches a block in the goal configuration, the distance between that block and the location of the goal is calculated. This is set to a different integer that is then compared to its current heuristic (which is initially Integer.MAX_VALUE). This guarantees that every goal configuration has a heuristic with respect to its goal.

However, merely setting a heuristic was not enough. We had to override the compareTo method from the object class so it could reflect heuristic comparisons. Whichever had the lower heuristic was the better tray configuration with respect to the goal. This implementation of comparable allowed us to use a PriorityQueue. Whenever a cluster of tray configurations were added, the front of the queue would always have the best heuristic as a result.

Upon its application, we found that our search for the goal configuration within our possible moves were greatly expedited. The time it took to solve all easy puzzles greatly diminished, and we found that we were also able to solve medium and hard puzzles at a faster time as well. The use of a heuristic greatly benefited our search algorithm when looking at each possible goal configuration.

Though there was no accurate measure to record the time for each test, we figured that our estimates of how much each group of tests took would be enough:

	No Heuristic	Heuristic
Easy	1-2 minutes	Less than 5 seconds
Medium	5 minutes with no errors	1-2 minutes with no errors
Hard	8 minutes with no errors	5 minutes with no errors

Experiment 3: The Representation of Empty Spaces

When we first started with our project, the first immediate idea we had was using a 2D boolean array to represent our board. However, this later shifted and we began to represent blocks as separate objects with two arrays representing their top-left coordinates and bottom-right coordinates. These values from the coordinates would then be used to account for spaces that have been filled (these values in the array would be set to false). The rest of the values in the 2D array would all be true, as there is no block occupying that specific space. This process worked for the most part, but it was inefficient due to the following ways.

In our possible moves method, when we were making new tray configurations as possible moves, we couldn't use the clone method to copy the array values from the previous tray configurations. This was due to a pointer issue, as making any changes to a clone of an object would also affect any variable pointing to it as well. So, in the end, we had to create a new 2D boolean array that reflected the previous configuration and then the empty spaces accordingly to the movement of the blocks.

We came up with another alternative to using a 2D boolean array. Instead of representing empty spaces like that, we used a hash set of points to represent each space that has not been taken up by a block. This implementation removed the hassle of hard-coding each empty space from the previous tray configuration, but points still had to be removed and/or added to reflected the blocks that had moved.

After this was implemented, we believed that this changed our code and made it more efficient for the better, but there was actually no change in runtime. The tests from easy, medium, and hard still ran at the same pace on the hive computers. However, we do believe that this change was memory efficient, as it did remove a lot of lines from the possible moves method in the TrayConfig class. We were also able to use the .clone() method to copy over empty spaces from the previous tray configuration because its instance variables were primitives, and therefore could be copied over successfully without having any pointer issues. In conclusion, though there was no change in runtime, it did made our code look much cleaner and less redundant than before, so we believe that this change was for the better.

	2D Boolean Array	HashSet<Point>
Runtime Improvement?	N/A	No
Memory Improvement?	No	Yes

PROGRAM DEVELOPMENT

Our program is made up of three classes: the Block, TrayConfig, and Solver classes. We initially started developing the Block class. Our reasoning behind this was that the block object essentially represents the most primitive piece of the program. We needed to define how we would represent the blocks before we were able to move on to creating objects that would represent the puzzle tray. After solidifying the instance variables for each block, we designed the TrayConfig class. This class represents the puzzle tray. Lastly, the Solver class was defined. This class contains the main method and converts the inputted init and goal files into TrayConfigs.

At first, our goal was to come up with a complete program that we could then run the easy test cases against. However, we then realized that we needed to test the TrayConfig class before running the program as a whole. The reason for this is that we needed to confirm that each TrayConfig correctly represented the specific puzzle tray with correct block placement. Due to the fact that each TrayConfig holds a HashSet of blocks and that block positions are represented by coordinate positions in each block, to test representation we would construct a TrayConfig inputting a HashSet of blocks with specific positions. We would then attempt to input new blocks with positions that were already occupied but already contained blocks. This would ensure that our TrayConfig would not allow conflicting / overlapping blocks on the puzzle tray. We would then also change the positions of certain blocks in the HashSet and attempt to add

blocks to those initial positions. This was to test the fact that after a block was moved, its initial position now did not hold a block and was available for a new move.

After we ensured that our TrayConfig class correctly represented puzzle boards, we created our Solver class in which we would consider possible moves and verify whether or not these moves would present the goal TrayConfig. After we designed and programmed all three of these classes, we tested our entire program against the easy test cases provided in the project files. Once we passed these and determined that our design framework was sound and ran successfully, we focused the rest of our attention on optimizing the program so that it would run more efficiently and faster.

IMPROVEMENTS

One way we believe we could have improved our program is through implementation of iterative deepening in our consider method (Solver class). As of right now, we iterate through our graph of possible moves using heuristics. This is under the assumption that a board in which all blocks are closer (distance wise) to their desired position in the goal puzzle board is preferable. However, if it is better to at first move blocks away from their desired positions to open up space on the board and then to move them closer, our heuristic method would not work. This is where we believe iterative deepening, in which depth first search to a certain depth is undertaken and then breadth first search takes over might be a better solution. This way, boards in which blocks are moved away from their goal position first will still be considered. We are not certain that this would be faster or more efficient, but we do believe it would allow us to examine more creative, and not necessarily intuitive solutions to the puzzle.