Jiahao Huang

Bryan Lopez

Will Park

Cary Schwartzstein

Project 3 Readme

**Division of Labor (half a page)**

For our division of labor, we decided that Jiahao would be responsible for the design and code, Bryan would be responsible for the design and researched file parsing and scanner streams, Will would be responsible for design and the Readme file, and Cary would be responsible for the design and code. In short, each member contributed to the design and code of Project 3, but had his own tasks to focus on.

We wanted every member to contribute to the design and code of the project because we felt that those two tasks were the most important to gain experience and knowledge in programming, but we also wanted to finish the project in a timely and efficient manner. To that end, we tried to place each member in an area that he was most strongly suited for accomplishing. However, we communicated with each other frequently regarding what progress we had made in our respective areas to make sure everyone understood the gist of what was going on in the areas outside of their responsibility.

**Design (2 - 3 pages)**

*A description of the overall organization of your submitted program (algorithms and data structures) that lists operations on blocks, trays, and the collection of trays seen earlier in the solution search. Diagrams will be useful here to show the correspondence between an abstract tray and your tray implementation. This description should contain enough detail for another CS 61BL student to understand clearly how the corresponding code would work.*

Our project consists of two classes - Solver and Block. Most of what we have written resides within the Solver class, though Solver makes extensive use of Block.

The Block class represents a block that might be contained within a Node (tray), and has instance variables size, width, height, and coords. Size is simply the dimensions of the block, and coords is an ArrayList of integers that are the coordinates of the block’s location within its tray. It also has two methods of note: a hashcode method that will return the hashcode of its coordinates, and an equals method that returns true if the object that it is being compared to is an instance of block and if the two block’s coordinates are the same. Otherwise, it returns false.

The Solver class represents the resulting data structure when our program is run, which is an undirected graph that points to each node’s children and parents. Each node has its own state, namely the path that each one represents thus far. Once the goal node has been reached, it should have the path to itself as one of its instance variables.

The Solver class also has a Node class nested inside, which constitutes each tray within our graph and will be discussed in detail later.

The Solver class has numerous instance variables, the most important of which are an instance variable of type Node called myHead (the beginning of our graph structure), a HashSet called visited (the hash codes of trays that we have already traversed), a HashSet called blocksInitial (the blocks contained within our initial tray), a HashSet called finalblocks (the blocks contained within our goal tray), a boolean called finished (set to true if we have found a path to our goal configuration or if our queue of possible trays is empty), a Comparator called comparator (organizes our queue for us and makes our program heuristic), and a PriorityQueue that contains Nodes called priority; priority will regularly be updated with possible trays that branch out from a current tray.

Solver’s constructor takes in two arguments, both of which are Strings representing the directory that the initial tray configuration file and the goal tray configuration file are contained in respectively. Once initialized, the Solver will in turn initialize a Comparator, a HashSet called “blocksinitial”, a HashSet called “finalblocks”, and a Scanner which will parse through the initial tray configuration file and set up the instance variables width, height, and blocksinitial accordingly. It will repeat the same process for the goal tray configuration file, except this time it will update finalblocks.

Once these variables have been set up, a priority queue will be established, finished will be set to false, blocksinitial’s hashcode will be added to visited, myHead will be instantiated as a new node with blocksInitial as its argument, and myHead will then be added as the first node to our priority queue. Then we will call a method that will start the process of finding the goal tray called startChildren.

startChildren is a method within our Solver class, and will continuously remove a node from our priority queue (which has been organized according to our heuristic), and then call another method children on the removed node. startChildren will continue to do this so long as the boolean variable finished is set to false and our priority queue of nodes is not empty. This means that once finished is set to true (when we have found a path to the goal tray) or our queue is empty, startChildren will cease.

children is an auxiliary method to startChildren, and is contained within the Node class. For each block contained within the node, it will check first if it can move left and if it can, it will create a new HashSet of blocks “n” and a new ArrayList of Integers “t”. For the new HashSet of blocks “n”, it will add all of the blocks from the current Node, except for the one that will be moved. The new ArrayList of Integers “t” will be used to represent the moved block by adding to it the original block’s coordinates moved one space to the left. Then, the moved block will be instantiated by passing in as arguments the ArrayList of Integers “t”, which represents the new coordinates of the moved block. This moved block will be added to the new HashSet of blocks “n”. Immediately after this step, “n’s” hash code will be calculated and if our HashSet “visited” contains this hash code, we will not do anything further with “n”. If visited does not contain this hash code however, we will add n’s hash code to “visited,” and we will instantiate a new node, passing in as its arguments the new HashSet of blocks “n”, and the direction of the move that just occurred. This node will then be added to our priority queue, and we will repeat this same exact process for the same block, except this time checking if it can move right, then up, and lastly down. Then we will do the same for the next block in the current node’s HashSet of blocks, checking if it can move left, right, up, or down, adding to the priority queue a new node reflecting the moved block each time.

As stated above, the class Node is meant to represent a particular tray in our graph data structure, and is nested within our Solver class. It’s instance variables are an array of arrays containing Strings called myMatrix (prints out a visual representation of the tray; useful for debugging), an ArrayList of Integers called move (indicating the coordinates of the move that just occurred), an ArrayList of Nodes called myChildren, a Node called myPrev (points to its parent), and a HashSet of Blocks called blocks (containing the blocks which are in the node’s tray).

Node also makes use of several important methods. Its method addBlock will update instance variable myMatrix to include the name of the block that is passed in as an argument, and its directional methods moveLeft, moveRight, moveUp, and moveDown will take in a block’s coordinates and return the block’s resulting coordinates had the block been moved left, right, up, or down. It is important to note that these methods do not actually move the blocks themselves; they simply return the coordinates of the blocks *if* they are moved left, right, up, or down. And as discussed earlier, Node has a children() method that plays an integral role in traversing our graph data structure and locating the goal node (if there is one).

\*\*Check my description of heuristic and iterative here

Lastly, two crucial aspects of our program are that 1). it utilizes a heuristic function to organize its priority queue, enabling a “best-first” traversal rather than a depth-first or breadth-first, and 2). it is iterative in nature, rather than recursive. As mentioned earlier, we pass in a Comparator object when we instantiate our priority queue, and this Comparator in turn uses a Heuristic function that takes in a HashSet of blocks and returns a “score” based on how close the HashSet of blocks is to the goal tray configuration, the lower score, the closer the HashSet is. The Comparator uses this information to organize the priority queue in order of “closest node” to “furthest node”.

\*\* Mention iterative aspect here

We did not implement these two changes until the very end, but after we did we noticed significant changes in the runtime of our program and the lack of stack overflow errors that had given us a considerable amount of trouble before. By changing our program to use a best-first traversal and to be iterative rather than recursive, we made our program more able to locate the goal node quickly, and reduce the number of pending operations to zero, so that each time we called children our stack was empty.

**Experimental results (1 - 2 pages per experiment)**

*Three experiments comparing results of a design choice from the project. Each experiment should include the following sections and content, written in a way that a fellow CS 61BL student would understand:*

* *Summary: description of the test and the results*
* *Results: graphs and/or tables with the results of the test*
* *Conclusions: explanation and interpretation of the results*

*Here are some questions to get you thinking about appropriate tests:*

* *What data structure choices did you consider for the board? What operations did you optimize? (Generation of possible moves? Comparison of the current configuration with the goal? Making a move?) How did these considerations conflict?*
* *If you used a data structure that involved hashing, how did you choose a hash function for the boards? How did your choice balance the need for fast computations, minimal collisions, and economical use of memory?*
* *How did you choose between moving blocks one square at a time and making longer block moves?*
* *How did you choose between breadth-first and depth-first (or some best-first) searching of the graph of move sequences? If you took a different approach, what was it and why did you take that approach?*

**Experiment 1:**

Summary: At the time of this experiment, we were using a dubious hash function that we suspected might be faulty: it worked by hashing a HashMap, which would hash each block’s name (a String), and multiply it by each block’s coordinates, which had been multiplied by prime numbers. Each resulting product would be summed, and returned as the hash function. We decided to see if it worked by running all of the easy tests, with the help of Checker.jar.

Results: Failed fifteen of the easy tests, listed below.

|  |  |  |
| --- | --- | --- |
| Test Name | Tray Size | Number of Blocks |
| big.block.1 | 3 x 4 | 4 |
| big.block.2 | 3 x 4 | 4 |
| big.block.3 | 4 x 3 | 4 |
| big.tray.2 | 100 x 100 | 1 |
| big.search.1 | 3 x 3 | 8 |
| big.search.2 | 3 x 3 | 8 |
| enormous.full.1 | 140 x 130 | 2 |
| 140x140 | 140 x 140 | 140 |
| 140x140 2 | 140 x 140 | 140 |
| enormous.3 | 140 x 130 | 2 |
| 140x140 with impossible goal | 140 x 140 | 2 |
| impossible.1+90 | 3 x 3 | 2 |
| impossible.4+180 | 3 x 4 | 2 |
| no.move.2 | 2 x 2 | 3 |
| one.move.1 | 3 x 2 | 2 |

Conclusions:

We were already suspicious of our hash function since we had cobbled it together without much thought: Here, we had further reason to inspect our hash function, since a significant portion of the tests we were failing involved large block dimensions or large tray dimensions. So, we investigated further and confirmed that the hash code was indeed the root of the problem. What was happening was that the hashcode being returned often became zero, because one term would become zero and would be multiplied with another term. The resulting product would in turn be multiplied with another term, eventually leading to the whole hash code of the tray being returned as zero. As a result, many blocks would have the same hash code, and when we would check if a block with hash code zero had been visited before, we would find that there would already be another block with the hash code zero there, tricking the program into thinking that we had already visited a block when in fact we had not.

**Experiment 2:**

Summary: Having successfully passed all of the easy tests, we were now on the medium tests that had been provided to us. We had fixed our hash function which had been the root of the problem for the easy tests and having done so, we turned our attention to other problems in our implementation, such as our children method, which would find the children of each node in our graph and was thus the main driving force behind our Solver class. We were worried that it was not finding the children of each node efficiently - that is, each node would call children and children would call node, which would lead to twice as many operations as necessary, all of which would be pending.

Results:

We failed nineteen of the medium tests, listed below.

|  |  |  |
| --- | --- | --- |
| Test Name | Tray Size | Number of Blocks |
| c15 | 5x4 | 10 |
| c27a | 5x4 | 10 |
| c27b | 5x4 | 10 |
| c27c | 5x4 | 10 |
| c27d | 5x4 | 10 |
| c28 | 5x4 | 10 |
| c29 | 5x4 | 10 |
| c30 | 5x4 | 10 |
| c31 | 5x4 | 10 |
| c34 | 5x4 | 10 |
| c35 | 5x4 | 10 |
| c38 | 5x4 | 10 |
| c39 | 5x4 | 10 |
| c40 | 5x4 | 10 |
| c41 | 5x4 | 10 |
| c43 | 5x4 | 10 |
| c44 | 5x4 | 10 |
| c50 | 6x4 | 11 |
| c56 | 5x6 | 14 |

Conclusions:

Looking at the tests that we had failed in the medium folder, we reasoned that the main problem this time was probably not the hash code, since we had just fixed our hash code. Instead, this time we were suspicious of how our Solver class was recursing and finding new children at each level of the tree. This would also explain why our error type had changed from having invalid trays to stack overflows. In other words, our method of graph creation was simply inefficient.

Upon further inspection of our code, we realized that we were making twice as many calls/pending operations than we need. Rather than having a node call the children method and the children method create a new node, we could simply call children on the result of itself recursively.

**Experiment 3:**

Summary: We made our program more efficient by having it rely on fewer pending recursive calls. This made it pass virtually all of the tests in the medium folder, but we had now turned our attention to the hard tests, nearly all of which we were still failing.

Results:

|  |  |  |
| --- | --- | --- |
| Test Name | Tray Size | Number of Blocks |
| test6 | 100x100 | >100 |
| test7 | 100x100 | >100 |
| test8 | 100x100 | 1 |
| test9 | 100x100 | 2 |
| test10 | 5x4 | 12 |
| test11 | 5x4 | 11 |
| test12 | 5x4 | 11 |
| test15 | 5x6 | 15 |
| test17 | 5x6 | 15 |
| test18 | 5x6 | 16 |
| test19 | 6x6 | 19 |
| test21 | 5x4 | 10 |
| test22 | 5x4 | 10 |
| test23 | 5x4 | 10 |
| test24 | 5x4 | 10 |
| test25 | 5x4 | 10 |
| test26 | 5x4 | 10 |
| test27 | 5x4 | 9 |
| test28 | 6x4 | 11 |

Conclusions:

Based on our own knowledge of how our program worked and how long it was taking for our Solver class to pass these tests, we were fairly certain that our program was still not efficient enough, despite the numerous adjustments we had made towards that goal. Based on the advice of our TA and a lab assistant, we decided to make our program heuristic and iterative, two design choices that were the capstone to our finished program. Please see “Program Development” below for how we implemented these changes.

**Program Development (1 page)**

*An explanation of the process by which you constructed a working program.*

* *What did you code and test first, and what did you postpone?*
* *Why did you build the program in this sequence?*
* *How did your team test your program? What test cases did you use for each of your classes, and how did you choose them?*

First, we created the Node class, then the matrix to represent the Board. We did this first because we felt that it was the most fundamental component to our project, and made the most sense in trying to solidify our data structure before worrying about other intricacies.

Next, we developed a method that would parse a text file containing initial or goal configurations and would turn it into an initial or goal matrix within our program. We tested this by passing in configuration files from the easy folder, and then having the resulting matrix print itself out using auxiliary functions that we ourselves created. We would then compare this result with our own hand-drawn visual representation of the same configuration file that we had passed in.

Afterwards, we turned our attention to graph creation and traversal. We knew that in order to have a graph data structure, we would need some sort of function that would find the children of each node by making every possible move for each block that could be moved. Therefore, we created a series of helper functions that would check if our blocks had space to move in various directions - moveRight, moveLeft, moveUp, and moveDown. Each of these methods would return true if the block could be moved one space in the direction indicated in its name, or otherwise would return false. We were planning to have a method called children that would use these four methods as helper methods, and this method would take care of the recursion by repeatedly calling the helper methods on each individual block that could be moved, and then on each result (the children). We tested these methods by using very simple initial configuration files, and then calling the four methods on them. We would then hand-draw the results ourselves and confirm that our drawings corresponded with the output of our program.

We then moved on to creating our children function. In its first incarnation, we had four while loops that would check if a piece could be moved in any direction - left, right, up, or down. If any of the blocks could be moved in any of these directions, they would do so in every combination possible, and these new resulting tray configurations would be added to a HashSet called visited, but only if they were not there previously. We would only call children recursively on trays that had not been visited, in order to prevent our program from finding the same tray configurations over and over again.

Our next logical step was to actually implement the visited HashSet. This HashSet would contain Integer objects, which were actually the hashcodes of our previously visited trays. We were initially unsure how to come up with a good hash function at first, so we simply decided to do various combinations of operations involving prime numbers, which we hoped would even out the range of distribution in the HashSet. This hash function would have negative consequences, as we would find in the future.

Having pieced together most of the necessary elements for our Solver class, we began a long and arduous process of testing and debugging. We started out with the easy tests that had been provided for us along with Checker.jar, and found that most of them failed with the message “Invalid test state.” This particular test actually corresponded with Experiment 1, which is detailed above.

After fixing the hash code and passing the easy tests, we decided to make a few optimization adjustments. First, we made Node a nested class within Solver, in order to keep the HashSet in the overarching class and see if the Node had been visited before. Next, we changed our four boolean functions moveLeft, moveRight, etc. into functions that returned the movement a block would make if it were able to move.

Next, we ran the medium tests, of which we failed approximately half (see Experiment 2). In order to pass them, we made more optimization adjustments that would involve less runtime and memory. For example, rather than having a node call children and children create a new node whenever we found a new set of blocks that hadn’t been visited (which would call children), we created a new method called startChildren that would continue to call the children method on the nodes in our priority queue, while we had not found the goal tray and the priority queue was not empty. This particular enhancement made our Solver class less memory-intensive and as a result, it passed more tests without getting a stack overflow error. We also moved our checkFinal method (which would check if a node’s tray configuration matched the goal’s tray configuration) into the constructor, to check if it was the final block immediately after construction. The end result of these design choices was a program that was no longer recursive, but iterative.

Finally, we made the program heuristic: rather than exploring nodes indiscriminately, our priority queue would now take in a Comparator, which would organize the queue based on a heuristic function, which we defined ourselves. It would first check to see if a tray has a number of blocks greater than 200. If not, it would iterate through all the final blocks and compare them to each of the blocks in the current node’s set of blocks. If one of the final blocks is the same size as one of the blocks in the current set, then it would find the difference in x coordinates and the difference in y coordinates and square both of them. It would then repeat this process for all of the blocks that it can and return the minimum of all of them.

**Disclaimers**

*In this section, describe parts of your solution that don't work, if any. You will lose fewer points for bugs in your project that are listed here.*

Our solution does not pass test 19 in hard, we think it is because there are too many possibilities for this one and our code is a little bit run out of time.

**Improvements (half a page)**

*If you were to make one more improvement to speed up your program, what would it be, and what is your evidence for expecting a significant speedup?*

If we were to make one more improvement, we would make the program store the values for the heuristic function. This would allow the program to only compute these values once per hashset of blocks state instead of having to compute each one every time it entered into the priority queue. Another way that we could implement this would be instead of calculating the heuristic value for each hashset of blocks, it could instead take the value computed from previous states and only change it slightly if the block set was actually closer to its final state. Right now, it changes it every time, sometimes unnecessarily. This would speed it up significantly because right now, the program has to iterate through all of the blocks in the final and current block sets in order to calculate a heuristic value. By changing this, I am confident that our program would run more quickly than it does currently since it would not have to calculate these values as many times.