cs61bl-db Suk Kyu (Alex) Kong
cs61bl-dc Anthony Joel Castro
cs61bl-eq Gerald Park
cs61bl-er Monica Wang

Project 3 ReadMe File

1. **Division of Labor**

   The engineering of this project was the collective effort of all four members. After deciding as a team on the project's overarching design, Anthony implemented the Block class and all of its defining characteristics, which involved deliberating between different ways to represent Blocks efficiently in terms of memory. Monica took charge in scripting the original prototype of the Tray class including data structures to store information and methods to find next possible moves given a tray and its blocks; Anthony and Alex helped debug and code auxiliary functions in Tray as various needs arose. Alex and Gerald collaborated on the Solver class and its subclasses; Alex authored the command-line argument processing facilities including the debug options as well as the main solving mechanism while Gerald seeded the Tree representation of tray configurations and also directed the writing of the README.

2. **Design**

   The three main portions of our project are as follows: 1) our solver method that solves the puzzle in `Solver.java` 2) the `Tray` class in `Tray.java` which creates `Tray` objects, and 3) the `Block` class in `Block.java` which creates `Block` objects.

   `Solver.java` is the tool that we use to solve for a desired Tray configuration (what the tray would look like if the goal block was in the goal position). The general algorithm that we used for our solver systematically tried out different configurations until we reach our desired or "goal" configuration. In order to get optimal performance we used our own self-defined `Tree` (called `TrayConfigTree`) and a `HashSet` (called `cache`). The general structure of our trees are below (we wrote our version, and then the lab equivalent of our versions for better understanding).

   Clarification Chart for our Tree Structure

Our Version	The CS61BL Lab Equivalent
Name	Name
`initialTray`	`myRoot`
`TrayConfigNode`	`Node`
`myItem`	`myItem`
`possibleMoves`	`myChildren`

Each TrayConfigNode consists of an item (which are Tray objects), and if it has children, it is saved through an ArrayList of Trays (which we call possibleMoves). Each different configuration consists of the movement of a single Block in its given tray. Initially, we saved the given initial tray configuration into the HashSet and set the it as the node (nodes are called TrayConfigNode) initialTray, the equivalent of myRoot in the Tree class implemented in lab.

These different configurations were saved through a HashSet of Tray objects. This memoization of tray configurations allowed us to keep track of which configurations we have seen before. If the tray configuration matched with an existing tray in the cache, then we do not repeat this move and we ignore it completely. We ignore these nodes because we know that it will not lead to a solution. For instance, if a particular node and its descendents did not end up producing the goal configuration that we wanted, then we would avoid running this configuration ever again. This is because it never produced the right moves that would lead to the correct "goal" configuration. Thus, moves that return to an already-seen configuration will not yield the "goal" configuration. However, if there were no matches, that tray configuration is added onto our cache as well as a new node whose item is that particular tray configuration and the next possible configurations were designed based off of the newly added configuration.

A Tree was used mainly because of its organization of data. The link between the parent node and child node in a tree structure, gave us an organized way to print out the sequence of moves that would help lead us to a desired goal configuration. The main idea was to do as little as Depth First Searches as possible until we get our desired goal configuration and organizing the structure of these searches like a stack. We used a stack over recursive calls solely because of the memory efficiency that it gives to avoid heap overflow. The method running this algorithm is solve(), which uses a helper function called solveHelper(). Given a current Tray, solveHelper() checks to see if the current Tray matches the goal configuration. If it does, then the current Tray is the solution and solve() prints the corresponding moves leading up to the solution. If not, then finds possible moves that can be made from current Tray, and pushes these possible moves to the top of the stack. If no solution can be found, then it returns null to solve(). solveHelper() uses satisfiesGoal(), a boolean method which returns true if the current Tray has every Block in the correct positions as specified by the goal Tray configuration and false otherwise, to check if the solution has been found. solveHelper() repeats this entire process until either the stack is empty or a it reaches a solution.

The Tray class is our biggest class. A Tray contains an ArrayList of Blocks called myBlocks, to store all of the references to Block objects that are in the tray. The addBlock() method checks if there are no Blocks at the desired (row, column) or within the new Block's dimensions and if so, adds the Block object to myBlocks using the ArrayList.add() method. Another function used was positionOccupied() which takes in an integer row and integer column as its arguments, and then returns true or false depending whether a Block occupies or does not occupy a position, by running a function from the Block class called occupiesPosition() (which checks the dimensions and positioning of the block) upon each Block in myBlocks. If a Block's position and dimension uses up the designated column and and row, this function returns false, else true. We used an ArrayList data structure rather than a Linked List or Tree to store the Block objects because it has a constant add() run time provided that we have the desired index. This behavior is desirable for our design because our Block objects are immutable meaning the instance variables that define a Block's positions cannot be changed. When we "move" a Block from one position to another, we simply replaced the old Block with a new one at its moved position. In order to conserve memory, we cached every instantiated Block in a HashMap called

blockCache; if the Block with the desired dimensions and position already exists in blockCache, we simply point the reference in myBlocks to this already existing object rather than create a new Block object. This is a relatively inexpensive process since HashMaps have constant lookup time.

One of the most important methods in the Tray class is findNextConfigurations() which returns an ArrayList of Trays that represent all the possible configurations of trays that could result from moving a single Block in the Tray. We defined/restricted the movement of a Block to moving only one space to the left, right, down or up in order to minimize the number of potential moves to keep track of. We used an ArrayList data structure because of its constant add() run time. This fast run time for add() is important because the number of next possible configurations could be very large (potentially up to four times the number of Blocks on the Tray) and slow down total run time. To find next possible configurations, the method takes a Block and checks if it can move one space to the left, to the right, upwards, and downwards on the Tray by calling canMove() . If canMove() returns true for that position, the Tray that would result from this move is added to the ArrayList of Trays, or "configurations", that is returned. This makes use of the method, trayAfterMovedBlock(), which returns the resulting Tray when a specified Block is moved to a specified position. findNextConfiguration() does this for every Block in the Tray and returns configurations which can later be used by the Solver class.

Another key method in the Tray class is isOK(). This checks the Tray's contents to see if any of the Blocks are overlapping. To do this, it takes each Block in myBlocks, records its position in a new temporary boolean matrix, tempBlockLocations, which contains true for each position in the Tray that a Block occupies. It returns false if a specified (row, column) is already true when adding a new Block to that position. This method ensures that all Blocks that are added to the Tray are legal.

We also overrode three functions within the Tray and the Block class in order to implement our two caches: hashcode(), toString(), and equals(). We overrode these methods because of the way we structured our code. For the intents of our project we needed to redefine what it meant for two Trays or two Blocks to be equal; the default equals() method would simply return whether two objects are the same. For our purposes, equals() returns true if the two objects' hashCode were equal and false otherwise. The toString() methods in both classes were overridden as well. The Block's toString() method returns a String representing four integers: the height, width, row, and column of the Block respectively (ex. "2 2 0 0"). Our hashCode() in Block simply took advantage of the String class's hashCode() method, which we know to be reliable and efficient. Moreover, since we created our Block objects to be immutable, we were able to save both the toString() representation and the hashCode() of a Block upon instantiation within two immutable instance variables; this meant that we did not need to create new String objects each time we called toString() or hashCode() on a Block. A Tray's toString() representation is simply the result of concatenating its Blocks' toString() representations together. The hashCode() for Tray simply uses the String class's hashCode() method. We made the assumption that any given Block representation in a Tray maintains the same index at all times. This allows us to simply compare two Tray's toString() representations when we test for equality.

We created the Block class to represent the Blocks that occupy positions in a given Tray. Blocks contain immutable instance variables that store its dimensions (myHeight and myWidth), position (where the block is on the tray in terms of its top-left corner represented by

myRow and myCol), and the toString() representation and hashCode() (the variables mySignature and myHash respectively) as well as methods such as toString which simply returns mySignature, and equals() which returns whether or not this Block object has the same dimensions and positions as another Block object. A StringBuilder is used to concatenate Strings for its memory efficiency when a Block creates its mySignature upon instantiation. Then toString() can simply return mySignature instead of creating a new String object. myHash too is created upon instantiation, so that the Block does not need to reevaluate its hashCode() every time hashCode() is called.

3. **Debugging**

The debugging options that our team has provided for you are as follows:

Debugging Options	Notes
"-ocommentate"	This commentates all the moves done when the solver is run.
"-ooptions"	Prints out all of the debugging options available.
"-odebug"	Shows all of the debugging options.
"-odebugBlock"	Displays all of the moves done by the Blocks by listing out Block and position
"-odebugTray"	Displays the moves done by the Tray in comparison to the desired goal Tray configuration.
"-odebugBlockFull"	Same as -odebugBlock except it also adds comments in what the Block did did.
"-odebugTrayFull"	Same as -odebugTray except it adds comments of how the move was made.
"-odebugFull"	Does -odebugBlockFull and -odebugTrayFull.

Our isOK() method (as stated earlier) checks this one very important invariant: no Blocks are overlapping one another within a given Tray.To do this, it takes each Block in myBlocks, records its position in a new temporary boolean matrix, tempBlockLocations, which contains true for each position in the Tray that a Block occupies. It returns false if a specified (row, column) is already true when adding a new Block to that position. This method ensures that all Blocks that are added to the Tray are legal.

4. **Evaluating Tradeoffs**

Experiment #1: Use of a stack for the implementation of `Solver.java`

Summary: This experiment was designed to prove the efficiency of `Solver.java` as opposed to our previous project design in which we did not use a `stack`. Our previous design that didn't use a `stack`, but rather used a recursive call.

Methods: We tried to run and time our experiments given the two different implementation styles. Each trial escalates in difficulty with regards for the tests (i.e. trial #1 uses an easy puzzle, trial #2 uses a medium, and trial #3 uses hard).

Results:

Recursive Design		vs.	Stack Design	
Attempts	Time (seconds)		Attempts	Time (seconds)
Trial #1	2		Trial #1	3
Trial #2	Stack Overflow		Trial #2	6
Trial #3	Stack Overflow		Trial #3	18

Conclusions: As the results shows us, the recursive design/implementation simply used too much space and memory, probably due to the fact that there were so many recursive calls waiting upon each other that the stack overloaded with too many tasks to handle (as shown by the trials when there were more blocks to manipulate and move). Our iterative solution clearly performed much better.

Experiment #2: Deciding between `Blocks` moving one square at a time VS multiple squares

Summary: This experiment was designed to see the consequence of choosing which style of movement for the blocks were the most practical. The implementations of comparison that we decided to use was between a `Block` moving one square at a time, and free-moving `Block` that is capable of moving to its max capability of movement.

Methods: We simply tried to run and time our experiments given the two different implementation styles. Each trial escalates in difficulty with regards for the tests (i.e. trial #1 uses an easy puzzle, trial #2 uses a medium, and trial #3 uses hard).

Results:

1-Square Move Design		vs.	Multiple Square Move Design	
Attempts	Time		Attempts	Time
Trial #1	1 Seconds		Trial #1	1 Seconds
Trial #2	7 Seconds		Trial #2	30 Seconds
Trial #3	24 Seconds		Trial #3	10 Minutes (Did not finish)

Conclusions: The reason as to why the design with one square movement of `Blocks` were better was because it structured itself in a way to perform a depth first search style. This would limit our branching factor per node. Though moving one space at a time is tedious versus a single move multiple spaces in any direction, we reduce our branching factor and allow our program to fail fast.

Experiment #3: The usage of `2D boolean array` to keep track of vacant squares VS usage of a boolean function to track vacant squares

Summary: This experiment was designed to test and see the memory efficiency with regards to the usage of the `2D boolean array`. The other alternative that we had was to use a function, in our project called `positionsOccupied()` which determined vacancy of a square by accessing information about blocks that were previously stored in an `ArrayList` and then determining the vacancy based upon that. At this stage we also implemented a `cache` for our immutable `Blocks`.

Methods: We simply tried to run and time our experiments given the two different implementation styles. Each trial escalates in difficulty with regards for the tests (i.e. trial #1 uses an easy puzzle, trial #2 uses a medium, and trial #3 uses hard).

Results:

Functional Design		2D Boolean Array Design	
Attempts	Time	Attempts	Time
Trial #1	1	Trial #1	10
Trial #2	6	Trial #2	Heap Overflow
Trial #3	24	Trial #3	Heap Overflow

Conclusions: The usage of the `2D boolean array` simply required the creation of too many objects that it caused the heap to overflow and cause the program to crash, as shown by the results. The use of a functional approach with regards to determining the vacancy of a square was much more memory efficient and allowed the program to actually run. Moreover implementing a `Block cache` saved us a great deal of memory.

5. **Disclaimer (N/A)**
    a. Our project is awesome!!! :DDDDD