PROJECT III
README.pdf

cs61bl-ge, gg, gm, gp

Division of Labor

There is really no particular reason why we split the work the way we did. Right after the first project checkpoint, we pretty much assigned each individual a class to work on so we can get things under our way. Corrina did the Solver class. Hao-Wei did the parsing. Andrea did the parsing and the Tray class. Aim did the Block class. Then we joined together and see how the objects are going to interact. After the first implementation, we realized that our Block class was really not needed, so we changed the implementation to making the Tray represent blocks by numbers using a 2-dimensional array. Corrina and Andrea changed codes from the implementation so that they work with the new design. Hao-Wei and Aim dealt with the tracing of the block, printing the moves from the goal to the initial point. Then we joined together again and make sure that everything works the way it should. Then together we do various testings and debugging.

Design

Overall, we have two java classes— Tray.java and Solver.java. Solver.java is where the main method is and where the puzzle solving takes place, including the parsing of files and optional debugging command. Tray represents a configuration, how the Tray looks like, at a certain step. The overall implementation we choose to traverse through different moves is through depth first search with the implementation of a stack. Every time we make a new move, we check whether or not the Tray was seen before. If so, we just pop the Tray and move on to the next unvisited Tray. if at any point the current Tray fulfill the final configuration, namely, if the current Tray contain all the lines inside the final configuration file, then we change the boolean variable solutionFound to true, print out the moves, and terminate the program.

The following is the descriptions of the constructors, methods, variables used in Tray and Solver:

Tray.java

Tray Constructors:

Tray(int row, int column)

-instantiates a row*column Tray object

Tray(Tray copy)

-instantiates a copy of a existing Tray object

Tray Instance Variables:

Tray **parent** = null; //the previous Tray
int[][] **board**; //the board representation
HashMap<String, ArrayList<Integer>> **coordinates**; //coordinates stores the blocks
int **blockCount**; //keeps the number of blocks, and acts as special key for each block
int t**rayRow, trayColumn**; //numbers of row and column
String **stringRep**; //String representation of the Tray (in one line)

Tray Methods:

toString()

-returns the String representation of the Tray

getString()

-returns the String representation of the Tray and sets stringRep to the return value

copy(int[][] toCopy)

-takes in a board representation toCopy and mutate the Tray's board so that it is identicalto toCopy

isAdjacent(ArrayList<Integer> blockPos, String direction)

-returns true if all integers at the direction (e.g. "left") of blockPos (representation of block) are zero's re turns false otherwise

addBlock(int row1, int column1, int row2, int column2)

-adds block to the Tray by passing in coordinates
-gives each block a unique number (blockCount)
-uses the blockCount number as the key, and an ArrayList of the block coordinates as the value, and put into coordinatehashmap
-add the block to the "board" grid of arrays. Fill in the top leftblock as the area of the block, and fill in the rest as -1

More Tray Methods:

printBoard()

-print out the entire board in String format, using system.out.println();

move(ArrayList<Integer> blockPos, String num, String direction)

-moves a given block a given direction (precondition: the block CAN be moved in the given direction)

-changes the state of the board grid, and change the coordinates corresponding to the block in the coordi nates hashmap

checkGoal(ArrayList<ArrayList<Integer>> goalConfig)

-checks whether the tray contains all the block positions of the goal configuration

moveToTray(Tray t)

-Checks the "difference" between two trays and returns the top left block positions of the first differing blocks

-To be used for tracing through the sequence of moves

isOk()

-checks that the tray is valid

Solver.java

Solver Constructors:

Solver()

-instantiates a Solver object with its finalTray and stack instantiated

Solver Instance Variables:

Tray **initialTray**;
Tray **current**;
ArrayList<ArrayList<Integer>> **finalTray**; //final Tray representation for checking if solution found
boolean **isBigTray**; //true if the Tray are is greater than 1000
HashSet<String> **alreadySeen**; //contain representation of Trays in Strings that have already been seen
HashSet<HashMap<String,ArrayList<Integer>>> **alreadySeen2**; //the alternative alreadySeen for big Tray
LinkedList<Tray> **stack**; //the stack we used to traverse through different configurations
boolean **solutionFound**;

//// ********** DEBUGGING FIELDS ********** ////
String **debugOptions**;
private int **numTraysDebug**;
boolean **iAmDebugging** = false;

Solver Methods:

initializeGame(String[] args)

-handles comman line inputs and sets up the initial and final Trays

main(String[] args)

-runs the solver program

makeMoves(Tray currentTray)

-Checks each block in the current configuration to see if it may be moved LEFT, RIGHT, UP and/or DOWN

-If the move is valid, create a new Tray object, and make that move.

-Push new configurations on to the stack

More Solver Methods:

haveSeen(Tray currentConfig)

-Checks whether we have seen the current configuration before

-If the isBigTray field is true, we will check the alreadySeen hashset of hashmaps for previous configura tions

-If the isBigTray field is false, we will check the alreadySeen hashset of strings for previous configurations

-If we have not seen this configuration yet, add it to the appropriate hashset

addToStack(Tray currentConfig)

-Add the current configuration to the stack

-Will almost always be added to the top as usual; but if a block is the farthest possible distance away

-from its goal position, add the configuration to the bottom of the stack

produceTrace (Tray currTray)

-The input should be the solution configuration.

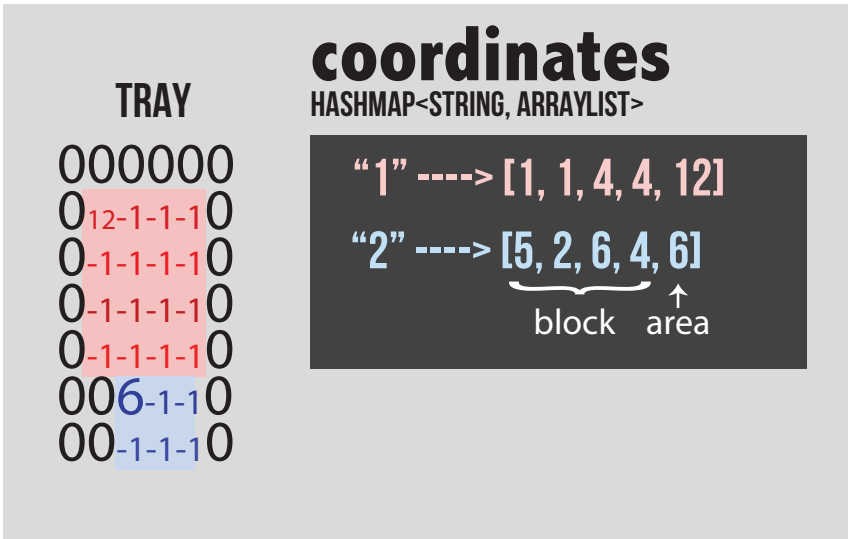
-Create an arrayList that has the reverse order of tray steps

printTrace (ArrayList trace)

-Prints out the sequence of moves from the initial configuration to the solution

Visual Representation

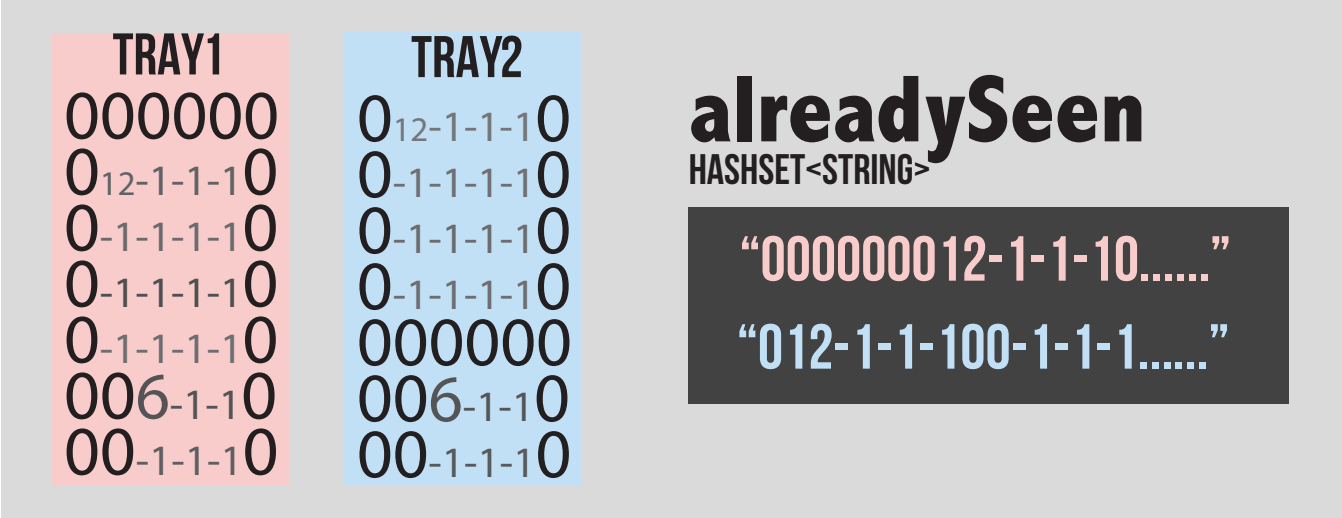
The following is the visual representation of the board and coordinates variables in Tray.java:



(up) each Tray has a 2D array called board that represents the configuration. Upper left corner of each block is its area.

(left) each Tray has a coordinates Hash-Map that stores all the blocks; each block has a unique key number generated by blockCounts; the value the key points to is an ArrayList

The following is the visual representation of the alreadySeen variable in Solver.java:



(up) Inside the Solver, alreadySeen keep track of all the seen configuration in Strings

Debugging

Our debugging output facility is enabled by inputting the first argument as -o following by a debug option. We implemented error checking to ensure that this first argument has been correctly inputted. If not, the code would stop running and an error message would print out. We came up with our debugging options based off of what print statements were using throughout the design of our code. -oinitialtray was helpful in revealing bugs that occurred in addBlock(), for if the representation of the board was inaccurate, we knew that it's because the blocks weren't initialized correctly. We encountered a slight roadblock when we realized our interpretation of coordinates was wrong, and initialtray was helpful in making sure we corrected our mistake. -ofinaltray was helpful in revealing bugs in checkGoal(). Similar to what we used -oinitialtray for, we would compare the representation of the board printed by -ofinaltray to what we knew the final configuration was supposed to look like.

-oruntime, memused, and numtrays were used similarly to analyze different aspects of what our code was doing. By calling these options on puzzles we had solved, we knew the capacity of our code. -oruntime was used often to make sure every puzzle was solved under 80 seconds. When the time ran past 80 seconds, we knew we had to either implement a better strategy or there was a bug somewhere in the program. memused helped us learn the limits of our program. Many puzzles resulted in running out of java memory, so by getting the memory used of solvable puzzles, and using other debugging options, we figured out what we needed to change in our code. numtrays was often used in accordance with memused to see how many instances of tray were made in a given a certain puzzle. With these three debugging options, we had an idea of how we optimize either one of the three options.

-oprinteachmove, checkokay, stopformem, stopfortime are options that affect our code while it's running, as opposed to all the previous options which either do something only in the beginning or only in the end. stopformem and stopfortime were especially helpful when we were unable to solve a certain puzzle. Instead of waiting for the code to terminate on its own, stopformem stops the program when it crosses the threshold of 95% memory having beenn lased. checkokay calls on isOK(), which checks whether the variables trayRow and trayCol are valid, and whether the blocks within the board are accurate. This was very useful in debugging when running the code through the Checker class, because it made sure that each move was valid (it resulted in a valid configuration).

Evaluating Tradeoffs

Experiment1

Summary:
This experiment compares the capacity of different data structures. We want to know how many objects can a hash-map, a hashset, or an array can hold. We also want to know how does the object type affect the structures' ability to hold them.

Methods:
Create a class called MemoryTest. This class contains a HashMap, a HashSet, and an ArrayList. Set a for loop with some large number (billions) and repeatedly insert objects into one of the data structures. Print out the i-th iteration and wait until the program quits the loop due to overflow of memory. The last index printed should be the amount of a particular object the object can hold.

Sameple Pseudocode (for the HashMap case; similar for other data structures):

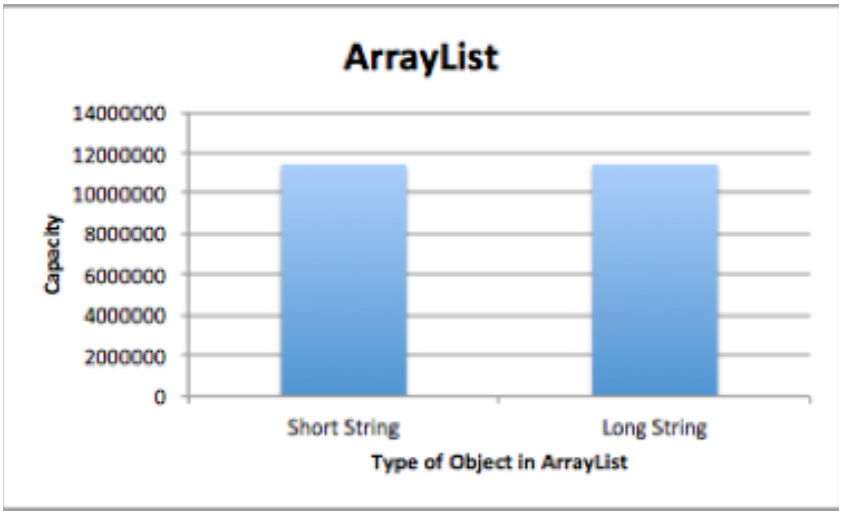
```
public class MemoryTest {
        private HashMap<Integer, String> hash;
        private HashSet hashset;

        public MemoryTest() {
        }
        public HashMap memory(int x) {
                hash = new HashMap<Integer, String> (x);
                return hash;
        }
        public static void main(String[] args) {
                MemoryTest m = new MemoryTest();
                HashMap h = m.memory(1000000000);
                for (int k = 0; k < 1000000000; k++) {
                        h.put(k, A REALLY LONG STRING);
                        System.out.print(k);
                }
        }
}
```

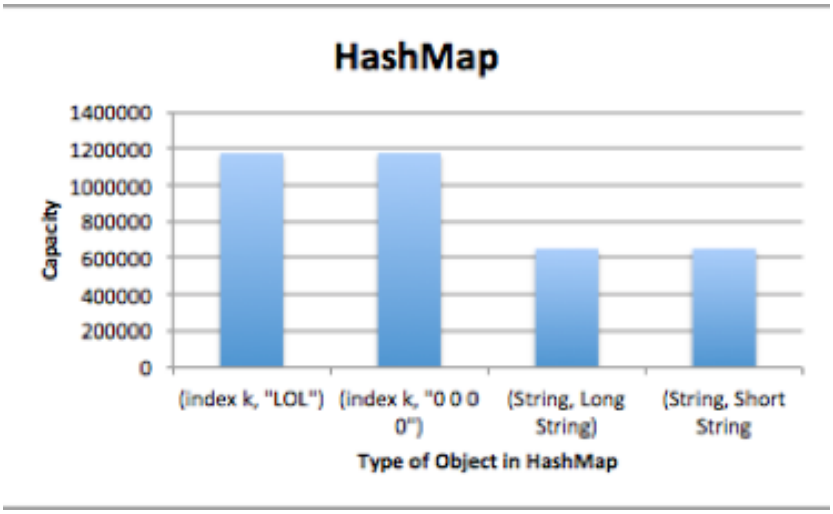
Result(refer to the graphs on the next page):
The size of a string does not affect how many of them we can put into an ArrayList. The same is true for HashMaps with <String, String> type; however, when a hashmap has the type <Integer, String>, there was a slight difference in the number of strings it could hold depending on the size of the string.

The experiment on HashSet is a little different from others where we held only strings. In the hashset experiment, we first tried putting strings into the hashset, then we tried putting a string into arraylists and then put them into the hashset. The iteration kept on going up to 2.1 billion.
The iteration actually stopped not because the hashSet can contain only 2.1 billion objects but because any integer value greater than 2.1 billion would cause a compiler error due to the number being too large. In terms of capacity, HashSet is the best data structure because it can hold much more than any other structures. Therefore, if we ever need to store a lot of objects, a hashset would be the best choice.

Graphs:

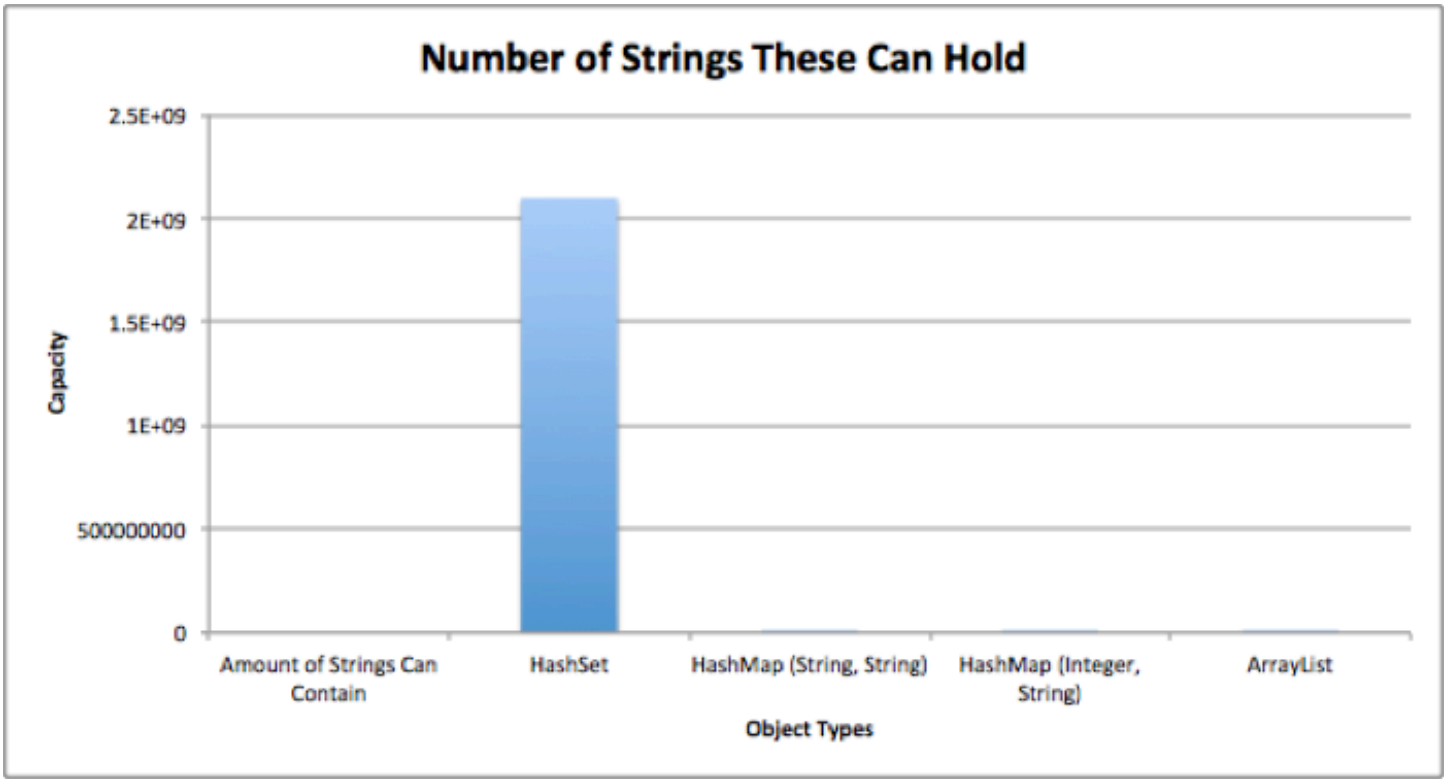


(left) capacities for ArrayList to hold Strings with different lengths



(left) capacities for HashMap to hold different types of keys and values

(down) comparison of capacity of holding Strings



Experiment2

Summary:
Experimenting to determine whether using a stack or queue is most time efficient (in essence, deciding between a type of DFS or BFS), or whether a decision-based version of a stack/queue would improve results. Based on the outcomes, it would seem using a decision-based version of a stack/queue is most efficient, which implies using both DFS and BFS depending on the puzzle.

Methods:
Testing a selection of 6 puzzles (1 easy, 2 medium, 3 hard) in which at least one is a "big tray" using a normal stack, a normal queue, and a decision-based blend of a stack/queue. The latter involves a check to see where all of the blocks included in the goal configuration are currently positioned. If any of these blocks are farther than half the maximum distance (the tray width + the tray height) away from their goal position, we will add the configuration to the bottom (or, alternatively, the end) of the stack/queue instead of the top/beginning. This is so that we examine configurations that are "closer" to the goal configuration sooner than we examine configurations that are not close to being solved. With this strategy, we will also test to see whether configurations are usually added to the front/top or to the end/bottom to determine whether its behavior is closer to a stack(DFS) or a queue(BFS). In order to assess time efficiency we will record the time it takes to solve (or not solve) each puzzle, and compare results.

Result:

TEST 1 – EASY	big.tray.1 (BIG TRAY)		
Tray: 100x100	# of blocks: 9999	# of goal blocks: 9999	
	Time (ms)	**Time (s)**	**Solved?**
QUEUE	4184	4.18	yes
STACK	-	0	no, out of memory
DECISION-BASED	4195	4.2	yes
-# added to front	0		
-# added to end	2		

TEST 2 – MEDIUM blockado			
Tray: 6x4	# of blocks: 11	# of goal blocks: 1	
	Time (ms)	**Time (s)**	**Solved?**
QUEUE	963	0.96	yes
STACK	1074	1.07	yes
DECISION-BASED	575	0.58	yes
-# added to front	4474		
-# added to end	9		

TEST 3 – HARD	little house		
Tray: 5x4	# of blocks: 10	# of goal blocks: 10	
	Time (ms)	**Time (s)**	**Solved?**
QUEUE	3420	3.42	yes
STACK	3746	3.75	yes
DECISION-BASED	3392	3.39	yes
-# added to front	0		
-# added to end	260057		

TEST 4 – MEDIUM	56		
Tray: 5x6	# of blocks: 14	# of goal blocks: 1	
	Time (ms)	**Time (s)**	**Solved?**
QUEUE	3160	3.16	yes
STACK	731	0.73	yes
DECISION-BASED	1686	1.69	yes
-# added to front	55236		
-# added to end	6522		

TEST 5 – HARD	46		
Tray: 4x5	# of blocks: 11	# of goal blocks: 11	
	Time (ms)	**Time (s)**	**Solved?**
QUEUE	8655	8.66	yes
STACK	-	0	no, out of memory
DECISION-BASED	8356	8.36	yes
-# added to front	0		
-# added to end	513286		

TEST 6 – HARD	53		
5x4	# of blocks: 10	# of goal blocks: 1	
	Time (ms)	**Time (s)**	**Solved?**
QUEUE	-	0	no, >>>>80s
STACK	2232	2.23	yes
DECISION-BASED	1987	1.99	yes
-# added to front	13893		
-# added to end	19322		