Final Analyzing Report

Group: 37
Members: Yang Yang
Qiheng Chen

Project Result: Able to solve and print the moves for all puzzles in the "easy" folder.

Files submitted: `Solver.java  Tray.java  Block.java`

Sample Output: solving a puzzle from the "easy" folder with -o command line argument enabled

```
D:\study\UI\core courses\10.Spring 2017\Data Structures\HW8\slidingblockpuzzlesolver\srcj
avac Solver.java Tray.java Block.java

D:\study\UI\core courses\10.Spring 2017\Data Structures\HW8\slidingblockpuzzlesolver\src>
java Solver -ototaltrayschecked easy\tree+90 easy\tree+90.goal
1 0 0 0
0 2 1 2
1 2 2 2
0 1 1 1
0 0 0 1
0 1 0 2
0 2 1 2
Total number of trays checked: 136
```

Division of Labor

In this project, we worked together with a lot of discussion and cooperation. We often discuss our plan and the outline of the whole project. Sometimes, we may start over because of new ideas and potential bugs and problems. In coding part, we explained our code and talk to each other in order to find out if there were problems so we can fix it in time. We worked together in the same room in Maclean Hall on most of the time. We spent over 25 hours on this project. In division of labor, Yang mainly wrote IsOk, goalCheck and most code. He also came up with the draft version of constructing this project. Qiheng is mainly responsible for finding bugs and testing the code and writing report. Considering the effort and participation, we both agree that we contribute equally to the project, and what we did in the project demonstrated teamwork and cooperation, and we worked well together!

Design and Logic

Design:

First, we created 3 classes: **Block, Tray and Solver**.

Block class stores the information of blocks in each tray. It contains the size and coordinates of each block.

Methods implemented:

`Constructor`

`Copy constructor`

`moveUp, moveDown, moveLeft, moveRight`

`equals` *// we override the equals method in Block class in order to compare the contents in blocks instead of the reference.*

`necessary getters and setters`

`hashCode` *// details will be explained later*

Tray class store the information of the tray layout. It contains the size of the tray and a list of blocks in the tray.

Methods implemented:

`Constructor`

`Copy constructor`

`isOK` *// if there are oversized or overlapped blocks, then it is an invalid configuration*

`goalCheck` *// after each move, check whether the current configuration satisfy the goal config*

`validMove` *// generate valid tray configs after one move from the current tray config*

`hashCode` *// hash a tray configuration into an integer for hashset use later, details explained later*

`printMoves` *// if there is a solution, the method will print the moves by tracking the parents of each tray config that leads to the goal*

`equals` *// we override the equal method in order to compare the Tray contents instead of reference*

`getters and setters`

Solver class is the class to run all the code in other classes to get the result.

- -o command line arguments enabled (`-ototaltrayschecked` is implemented, others yet to be implemented). We can handle multiple -o command line arguments.
- We are also able to read and parse the initial configurations and goal configurations from the file and create the corresponding Tray objects.
- BFS: we use the Breath-First Search to traverse the tree of possible tray configurations until we find the goal configuration
- DFS: Depth-First Search is also implemented

Logic:

Step 1: We made a move method by pushing every block forward in every direction (4 directions, i.e., up, down, left, right).

Step2: We check the if the movement is valid (i.e., not out of tray bound, no overlap with other blocks) by calling the `isOK` method.

Step3: Check to see if the configuration after moving satisfies the goal configuration by calling `goalCheck` method.

Step4: Applying `BFS` and `DFS` search to solve this problem.

Step5: Print the moves that lead to the goal configuration by calling `printMoves` method.

Detail information:

For BFS search, we created a queue and hashset. The queue is used for the traversal according to the BFS order, and the hashset checks repeated tray configurations in order to avoid unnecessary search. In other words, the hashset stores the configurations that already appeared in the BFS search. If the queue is empty and we still haven't found the solution, then there is no solution.

The hashCode method that we used for the hashset overrides the default method. For the tray hashcode, it is illustrated through the following example:

Suppose we have a following tray configuration:

2 2 0 0

1 1 2 0

1 1 2 1

1 1 2 2

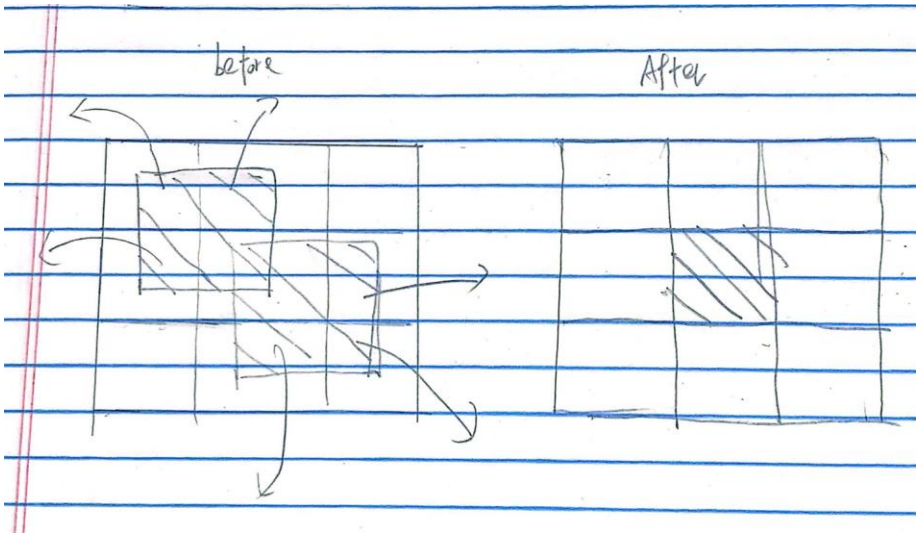
Our hashCode method essentially treat all the 4-tuples as 4-digit numbers and add them together: 2200+1120+1121+1122 = 5563.

We believe this way of hashing reduces the chance of collision for different tray configurations (of course in some extreme cases, it is possible that different tray configurations are hashed to the same integer, but it is very unlikely).

For DFS search, the idea is similar except that we used a stack instead of a queue to maintain the DFS order of search.

Discussions

1. First, the running time of the isOK method is $O(n^2)$, where n is the number of blocks in a particular tray. We check whether there exist overlapped blocks or blocks that exceed the bound of a tray through mutual comparison. So, if there are many blocks, it may talk a long time to process. We believe that there exist better ways to check tray validity, which takes lower than n^2. One possible way that we didn't try but seems promising is illustrated in the following picture. We can think of a tray as a grid, and we cover it will all the blocks in their current positions. Then we remove the uppermost layer of the blocks from the grid, if afterwards there is still something left on or outside the grid, it means that there would be an overlap or out-of-boundary error. The running time of this method could possibly be linear in O (n) instead of $O(n^2)$, if implemented properly.



2. We made improvement on coding styles which helps us solve more puzzles. For example, look at the following two code snippets in the BFS method in the Solver class:

```
for (int i = 0; i < current.validMove().size(); i++) {
    Tray n = current.validMove().get(i);
    if(!s.contains(n)){
        s.add(n);
        n.setParent(current);
        q.add(n);
```

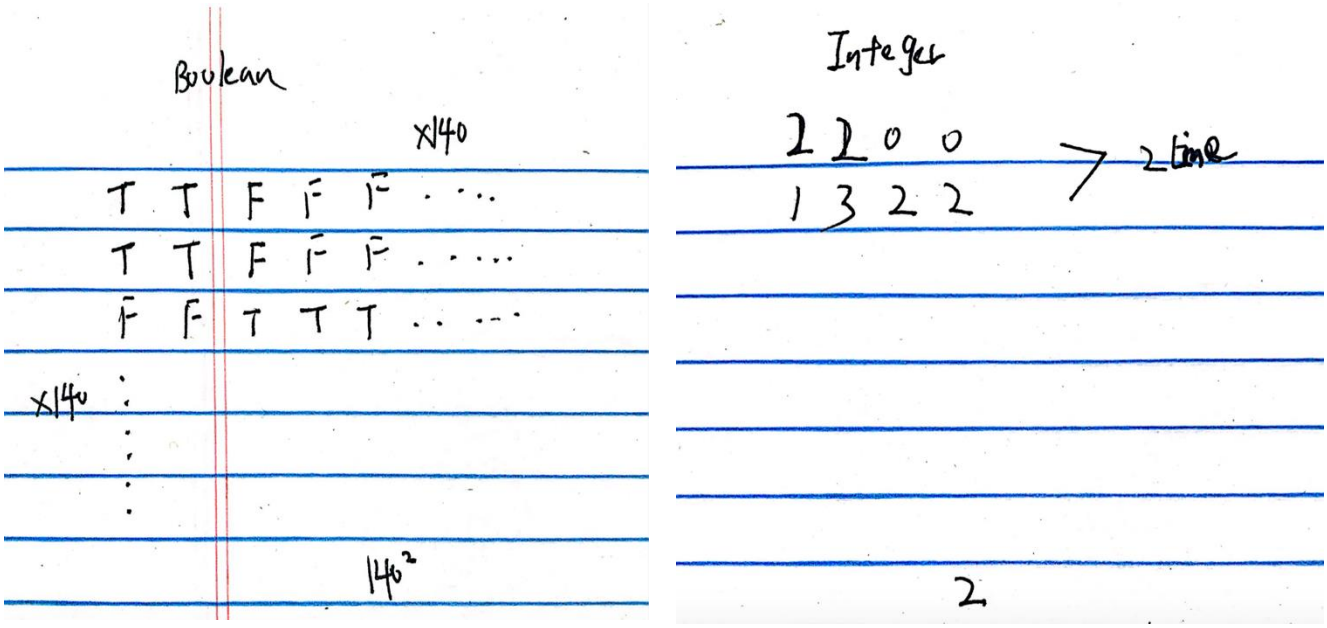
Fig 1: before improvement

```
List<Tray> possible_candidates = current.validMove();
int len = possible_candidates.size();
for (int i = 0; i < len; i++) {
    Tray n = possible_candidates.get(i);
    if(!s.contains(n)){
        s.add(n);
        n.setParent(current);
        q.add(n);
```

Fig 2: after improvement

We first programmed as in Fig. 1. We can see that the `validMove` method was called twice during each iteration in the for-loop. Actually, `validMove` is a main source of time consumption in our code (as discussed later). This would dramatically increase the time and space requirement. After the improvement, as illustrated in Fig. 2, we first store all the valid moves from the current tray configuration into a list of trays `possible_candidates`, and then run the subsequent code which is equivalent to the code in Fig. 1. This change allowed us to solve the 140x140 puzzle in the easy folder, which we were unable to solve using the coding style in Fig. 1.

3. We compared two different ways to store a tray configuration. For example, suppose we need to store 140 x 140 tray with 2 blocks in it. If we use boolean variables for each cell in the 140 * 140 grid (i.e., a cell that has a block there is true, otherwise false), we need 140^2 boolean variables. If we represent the tray configuration by using integers to represent the size and the location of the upper left corner of blocks, we just need 2*4 = 8 integers.



4. At the beginning of our coding, we only made a Tray class and insert every block as an array of 4 integers into a tray represented by a linkedlist. However, when we noticed that we need to move every block in that linkedlist, and we were unable to achieve this goal based on the current setting, which motivated us to create a new class called “Block”. Now in the tray linkedlist, each element is a Block object instead of an integer array. And we can use the method developed in the Block class to move the block into different directions.

5. In terms of running time, for easy puzzles, our code typically prints out the solution moves in less than 5s. We did try some of the puzzles in the medium folder and hard folder, but unfortunately it didn't print out the moves in 2 minutes. For one of the puzzles in the medium folder, we kept our program running for about 90 mins and it threw the OutofMemory error. We believe that one of the potential improvement is in the `Tray.validMove` method. By checking the code, we can see that we created a lot of tray objects in this method for testing moves of each block in 4 directions and then return these objects. In the `Solver.DFS` and `Solver.BFS` methods, the hashset also contains tray objects that have appeared in the search. In the DFS and BFS tree, the branches grow exponentially because starting from each tray configuration there could be a lot of valid moves. The memory is likely to overflow quickly, especially for BFS. BFS always gives the minimum number of moves if it could find a solution before running out of memory. But it will search all possible moves from current configuration before getting into the next layer of the tree, which give rise to the memory explosion. Nevertheless, we tried both DFS and BFS on puzzles from the medium and hard folder. It seemed that both ways won't be able to handle these puzzles. Therefore, we think that if we can find a way to substantially reduce the memory cost in calling the `Tray.validMove` method, we might be able to dramatically increase the efficiency and solve some puzzles from the medium folder and the hard folder.