CS2230 Homework 8 Report
Sliding Block Puzzle Solver
Jack Geati and Madeline Silva
May 3, 2017

Question 1: Description of our program

We represent game trays using several classes and data structures. The Tray class is a container for all the data structures used to represent a tray. It holds a HashMap of (Coord : Block) pairs, representing all the blocks on the tray and their positions. It also holds a 2D integer array of 0s and 1s, representing where blocks are and where there's empty space on the board. (See Tray Representation diagram)

The main method of the program is in Solver. It starts by creating two Trays, the main game tray and the goal tray, from the inputted text files. It also processes any optional arguments at this time. It then checks if the main tray already matches the goal tray - if this is the case the program exits with no output. Otherwise, Solver calls Tray.solve on the main tray with the goal tray as input.

Tray.solve returns either an ArrayList of integer arrays representing a sequence of moves to make the main tray match the goal tray, or an empty ArrayList if the puzzle is found to be unsolvable. To generate this list it first creates a root TreeNode containing the main tray. This will be the parent of all the possible Tray configurations we're going to check. Next, it calls the internal method breadthFirst with the root and the goal tray as its parameters.

BreadthFirst creates a tree of possible tray configurations and searches the tree for a goal configuration at the same time. (see Tree of Trays diagram) It looks at a parent, creates children of the parent, then looks at all those children before going on to the next depth. For each node it looks at, it generates a list of possible moves that could be applied to the current tray. Then it creates new child trays that are one move different from the current node by making copies of the current node and applying each move in the list to them. Then it goes through the new child trays and attempts to add them to the HashSet visited. If they are not already in visited, they are added as children to the current node. If they already are in visited, then that configuration has already been looked at and that tray is not added as a child. Finally, we add all the children to the queue. This continues until either: a child tray matches the goal and is returned, or the queue is emptied, which means there are no more possible unvisited configurations, and null is returned.

Tray.solve returns either an empty ArrayList if breadthFirst returned null or an ArrayList of int[] representing moves to achieve the goal. Solver then either calls System.exit(1) if the list is empty or prints out the moves in the list if it has them. Then it

prints out various data about the solving process depending on which optional arguments were inputted.

Question 2: Description of other files

Block.java- The Block class holds data for an individual block in a tray. It holds the dimensions of the block and its upper leftmost space's coordinates. It has a method which takes a 2d array and input and returns a list of moves a block can take based on that array (see Move Detection diagram). It also has equals and hashCode methods so that Trays can be effectively put into a HashSet by hashing their HashMaps of Blocks.

TreeNode.java- TreeNodes are used to build the tree of trays. Each parent node holds one instance of Tray and an ArrayList of children, where each child is a tray one move away from the tray held in the parent node. It has methods to get all children of a node and to add a child.

Coord.java- Coords are used as keys for Blocks in Tray HashMaps. It allows the lookup of Blocks using their xy position. The only fields it has are the row number and column number of the top left coordinate of a block. It has equals and hashCode so it works effectively as a key.

Question 3: Explanation of how we addressed efficiency concerns

Our first decision was about how to represent the tray. We decided to use both hashMap of Block objects and a 2d array of 1s and 0s. By using block objects, we can change the tray by changing a single object and then fixing the tray to correctly represent the blocks. In our 2d array, an empty space in the tray would be represented by a 0 and an occupied space would be represented by a 1. This allows us to easily check for possible moves, as we can see if the space along one side of a block is completely 0, and register that it is possible to move in that direction. It also allows us to easily check if there are any overlapping blocks in a tray by attempting to generate a 2d array from the Block HashMap and checking for collisions. This is what our isOk method does. We considered using only a HashMap or list of Blocks, but the 2d array made move detection and isOk much easier to implement.

The greatest way we increased efficiency was through our HashMap and HashSet data structures. We considered storing Blocks in an ArrayList, but this would result in O(n) lookup when searching for a block to perform a move. With our hashMap, using Coords as keys, we can find blocks in O(1) time, which helps with quickly comparing trays and moving blocks without iterating through all the blocks in the tray. We also use a hashSet, visited, as a static variable in the Tray class for holding all of the tray configurations we have already looked at. Rather than have to compare every tray, we can find the hash of a tray and see if it is already in our set, which is much

more efficient. These do their job effectively, allowing us to solve, for example, the d209 puzzle in (effectively) 0 seconds and big.tray.4 in 8 seconds.

The problem with these is that they are very memory heavy, and while we weren't running into the 4 GB memory cap just yet, we wanted to prevent the issue before it ever happened. To do so, we made sure to take steps to mitigate this across our program without making less efficient in runtime. Using an ArrayList to hold the list of moves to solve the program and and to hold the pointers to children of a node show this, as ArrayLists take less memory per element than Linked Lists.

We wrote both breadth-first and depth-first search, adding children to each node as we went along. However, we ran all of the puzzles on both types and breadth-first was faster in every single scenario. Breadth first also didn't cause the stack overflow errors we were having on harder puzzles with depth first. It allowed us to solve more hard puzzles without error so it was the obvious choice.

Time and space usage of selected puzzles

| Puzzle name | Puzzle difficulty | Runtime in seconds | Space used in mb (approx.) |
|---|---|---|---|
| easy | easy | 0 | 1 |
| tree | easy | 0 | 2 |
| blockado | medium | 0 | 17 |
| dads | medium | 0 | 9 |
| little.house | hard | 11 | 339 |
| handout.config.2 | hard | 3 | 106 |
| c22 | hard | 19 | 379 |
| big.tray.4 | hard | 4 | 366 |

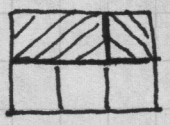
# Tray Representation

**Tray game**

```
int[] tray = [1 1 1]
             [0 0 0]

HashMap blocks =
{(coord(0,0): Block(1,2,0,0),
 (coord(0,2): Block(1,1,0,2)}
```

=

Check.diff.blocks

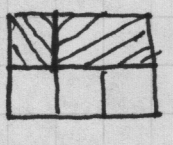


**Tray goal**

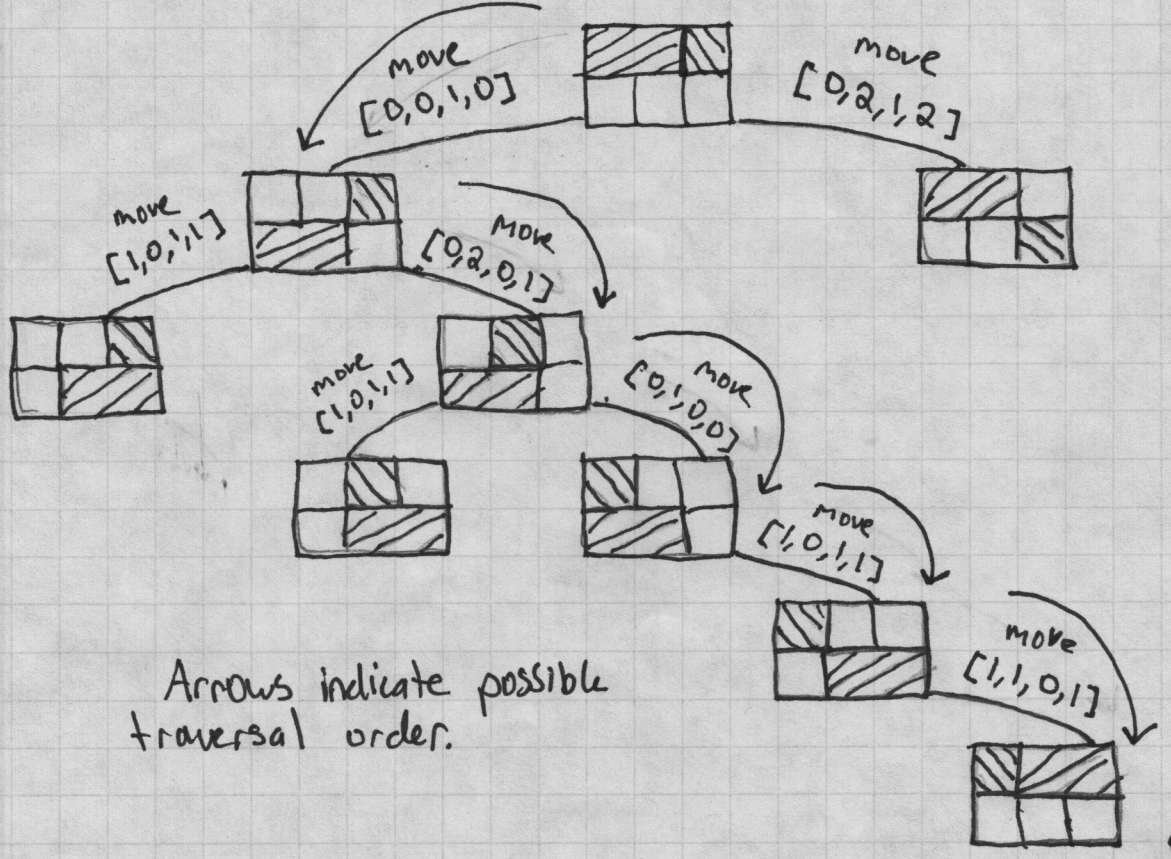
```
int[] tray = [1 1 1]
             [0 0 0]

HashMap blocks =
{(coord(0,1): Block(1,2,0,1),
 (coord(0,0): Block(1,1,0,0)}
```

=

Check.diff.blocks.goal



## Tree of Trays

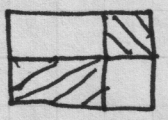



Arrows indicate possible traversal order.

goal reached!

# Move Detection

example tray config:



tray:

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

Block(1,1,0,2). FindMoves(tray)

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

check move left - yes, add [0,2,0,1] to list
check move right - no, OOB
check move up - no, OOB
check move down - yes, add [0,2,1,2] to list

return Arraylist<int[]> containing ([0,2,0,1],[0,2,1,2])