CS2230 Homework 8 Report
Sliding Block Puzzle Solver
Jack Geati and Madeline Silva
May 4, 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 tray data. It hold 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.

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 checks if the main tray already matches the goal tray - if this is the case then 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 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 as well as keep a queue of tray configurations to be looked at. First 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. Finally, we iterate through all the children of the current node and add them to the queue.

Question 2: Description of other files (no clue???)

TreeNode.java- This file is for the TreeNode class, which we use to build the Tree data structure. Each parent node holds one instance of the tray can have an unlimited number of children, where each child is a tray one move away from the tray held in the parent node. Each node also has a pointer to its parent (where the parent of the root is null). An ArrayList field named children holds pointers to all of the children of a node, and is iterated through in the recurse function of the tray class.

Coord.java- The Coord class holds Coord objects which are held in a hashMap for blocks. The only fields it has are the row number and column number of the top left coordinate of a block. These are used to make a hash so that when we need to look up a single block, we can do so by finding the hash of it in O(1) time. This is used in comparing trays, and adding, removing and moving blocks.

Question 3: Explanation of how we addressed efficiency concerns

Our first decision was about how to represent the tray. We combine using a hashMap of block objects with a 2d array of integers. 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 blank tray, if while iterating through the blocks we pass over a space that already has a 1, we can return false.

The largest way we increased efficiency was through our hashMap and hashSet data structures. With our hashMap of Coords, we can find blocks in O(1) time, which helps with quickly comparing trays and moving blocks without iterating through the entire 2d array representation of the tray. The hashSet is used for holding all of the trays we have already visited. 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 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, causing us to run into out of memory errors in the 4GB we are limited to. We are currently looking into how to fix this, but have taken steps to mitigate this across our program. 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, as well as completed puzzles that would result in stack overflow errors in depth-first search.

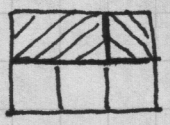
Tray Representation

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

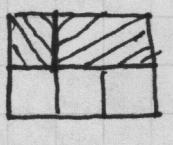


Tray goal

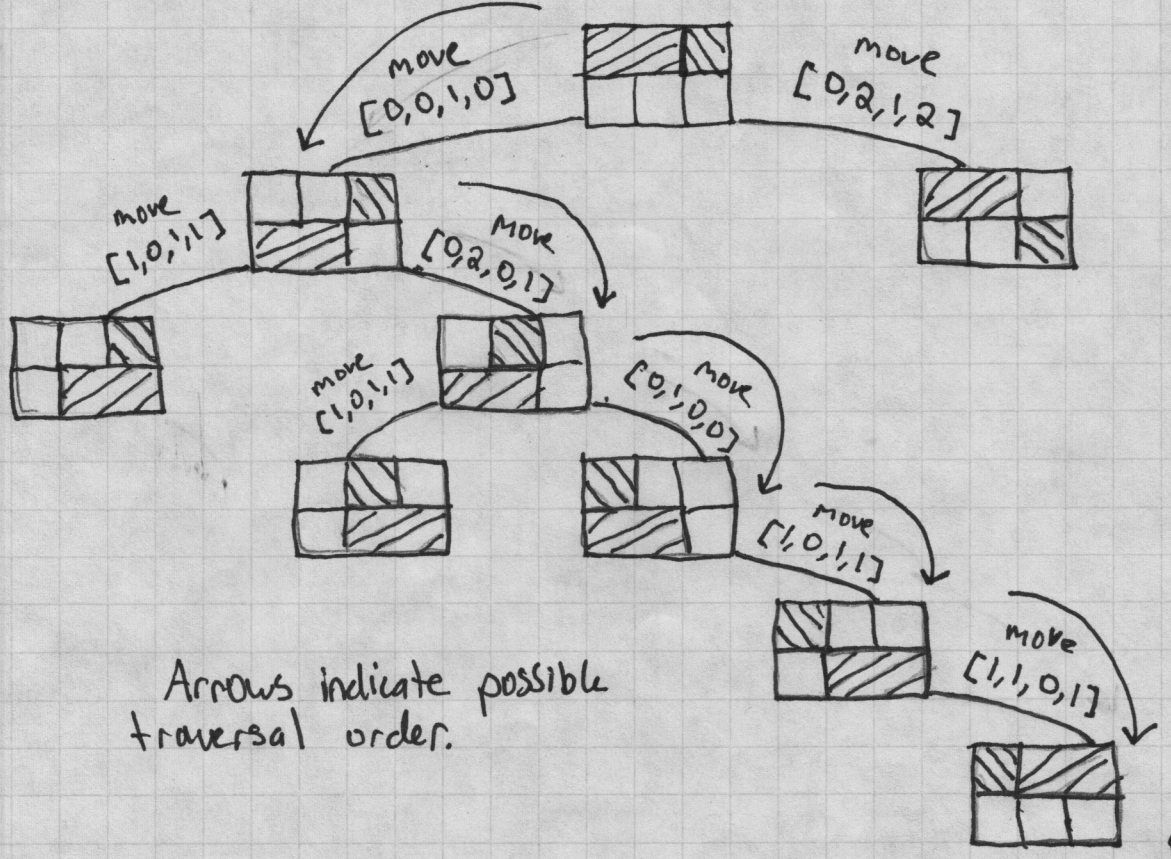
```
int[] tray = [1 1 1]
             [0 0 0]

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

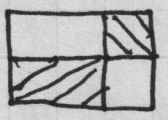


Arrows indicate possible traversal order.

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])