**COMS 2270  
Fall 2024  
Assignment 3 Addendum  
The Puzzle Solver**

## Introduction

In theory this problem is an easy recursive search. For each possible move, grab the block and move it in one of four directions. If the move is possible, then make another move. If we solved the puzzle, then record the moves. Here is a simple solution approach.

***solve(board)***

***if the game is over***

***record the move history as a solution***

***else***

***get a list of all possible moves***

***for each possible move***

***grab the block to move***

***move the block in one direction***

***solve(board)***

***undo the move***

But there is not a chance this will actually work. Here are the key problems.

* There are infinitely many, infinitely long solutions, because you can keep returning to the same grid layout over and over again. We need to stop when we've seen the same grid layout already.
* Even if you don't return to the same grid layout within an attempt to play the game, there are way too many solutions to find them all. But if a game can be solved in less than 10 moves, we really don't care about the solutions of 10000 moves that also solve it. So we should impose an upper bound on the number of moves.

The first problem can be solved by keeping a list of grid layouts that have already been encountered. If you find one you've seen before, just return. The second problem can be solved by placing an upper bound on the length of a potential solution. If the number of moves has exceeded that upper bound, just return.

Unfortunately, the solution to the first problem is at odds with our interest in finding the *shortest* solution. For example, if you first reach a certain grid layout after 15 moves, and then find a solution three moves later, well great. But what if at a later point in the search you reach that same grid position after only 7 moves? In that case, you *do* want the search to continue, because you'll then reach a solution in a total of 10 moves.

## Solution Approach

Basically, we need to associate each grid layout we find with the number of moves we took to get to it. If we get to that same layout again, but do so in fewer moves, *then we should continue solving*. So the updated algorithm we have come up with looks more like this:

***solve(board)***

***if the number of moves is over the max***

***return***

***else if the game is over***

***record the move history as a solution***

***return***

***else if the grid layout is one we've seen before***

***if number of moves is greater than or equal to how we got there before***

***return***

***else***

***update the number of moves for that grid***

***else***

***record grid layout and number of moves in the list of those we've seen***

***// if we get here, we keep searching!***

***get a list of all possible moves***

***for each possible move***

***grab the block to move***

***move the block in one direction***

***solve(board)***

***undo the move***

There are several methods in **Board** that help support implementing this solutions.

**void grabBlockAtCell(int row, int col)**

Grab one of the blocks so it can be moved.

**boolean isGameOver()**

Check if the game is over.

**void moveGrabbedBlock(Direction dir)**

Move the grabbed block one cell in the given direction.

**ArrayList<Move> getAllPossibleMoves()**

Get a list of all legal moves.

**getMoveHistory()**

Get a history of all moves taken to get to the current position.

**undoMove()**

Undo the previous move.

**How to record a grid layout as already seen?**

You may have noticed the declaration:

**private Map<String, Integer> seen = new HashMap<String, Integer>();**

Huh what? One thing we need in the algorithm is an efficient way to store the grid layouts we've seen before, and to associate with each one the number of moves it took to get there. A HashMap is an ideal data structure for this. You will learn much more about HashMap and hashtables in general in Com S 228, but for now we just need the following operations:

**seen.put(board.toString(), board.getMoveCount());**

Record the signature description of the board as seen with a particular move count.

**if (seen.containsKey(board.toString()))**

Check if the current board position has already been seen.

**int moves = seen.get(board.toString());**

Get the number of moves recorded for the current board position.

**How to record the history of moves as a solution?**

A shallow copy of the list of moves is sufficient.

**solutions.add((ArrayList<Move>)board.getMoveHistory().clone());**

## Undoing a Move

The only method you need to implement other than **solve()**, is **undoMove()** in **Board**.

Undo is likely to be a common source of bugs, so it is important to get it right. The pseudocode for **undoMove()** follows.

**undoMove()**

**if the move history is greater than zero**

**set game over to false**

**remove the last move from the move history**

**grab the block for the move**

**if move direction is up**

**move down**

**else if move direction down**

**move up**

**else if move direction right**

**move left**

**else**

**move right**

**remove the last move from the move history\***

**decrease the move count by two\***

\* It may seem strange to remove an element twice from move count and decrease the move by two. These instructions are assuming you are calling **moveGrabbedBock** to move the block. Because that function adds the move to the history and increase the move count by one every time, we must counteract the effects of the extra “undo” move.

## Testing

The recursive method can be difficult to debug, so start with *very* simple boards. Here is an example for a new file named *simple-games.txt*.

**\* \* \* \* \* \***

**\* . . . . \***

**\* . . . . \***

**\* [ ] . . e**

**\* \* \* \* \* \***

**\* \* \* \* \* \***

**\* . . . . \***

**\* . . ^ . \*  
\* [ ] v . e  
\* \* \* \* \* \***

Here is a simple setup for testing the solver.

**import java.io.FileNotFoundException;  
import java.util.ArrayList;  
import api.DescriptionUtil;  
import api.Move;  
import hw3.Board;  
import hw3.Solver;**

**public class SolverTests {  
 public static void main(String args[]) throws FileNotFoundException {  
 ArrayList<String[][]> gameDescriptions = DescriptionUtil  
 .*readBoardDescriptionsFromFile*("simple-games.txt");  
 int boardIndex = 0; // change to select different board setup  
 Board board = new Board(gameDescriptions.get(boardIndex));  
  
 Solver solver = new Solver(6); // set higher for larger puzzles  
 solver.solve(board);  
 ArrayList<ArrayList<Move>> solutions = solver.getSolutions();  
  
 System.*out*.println("Number of solutions found: " + solutions.size());  
 solver.printSolutions();  
 }  
}**

The expected output of the above code is the following.

**Number of solutions found: 1  
Solution:  
(3, 1) one cell RIGHT  
(3, 1) one cell RIGHT  
(3, 1) one cell RIGHT**

When setting boardIndex to 1, the expected output is the following.

**Number of solutions found: 1  
Solution:  
(2, 3) one cell UP  
(3, 1) one cell RIGHT  
(3, 1) one cell RIGHT  
(3, 1) one cell RIGHT**

## Some Final Fun

There are no extra points for this part, but for your own enjoyment when you get the solver fully working, you can improve the hint button. Currently the hint button selects a random move from all possible moves starting at line 85 in **ButtonPanel.java**. Change it to call the solver and give the user the next move in one of the solutions.