CS47B Block Project Write-up

Name: Zhaohua Gao, Pei Jun Chen

Sponsor: Hugh

Date: 4/25/2012

1. Representations:
2. Tray:

Because the Tray class contains all of the blocks, our primary goal in implementing the tray was to keep track of each block’s position as well as the methods that generate moves and traverse the state tree. The attributes and methods are listed below:

Attributes:

private int rows;

private int cols;

private ArrayList<Block> blockList;

private int[][] twoDimRepr;

private HashSet<String> previousConfigs;

private String solution;

private boolean checkBlanks;

The rows and cols attributes represent the length and width of the tray. The blockList data structure is a list of all of the Block objects in the tray. The twoDimRepr is the actual game board, with each integer representing the index of the corresponding Block object in our blockList. We chose the number -1 to represent an empty space in twoDimRepr, since -1 could never be an index in blockList. previousConfigs is a HashSet used for the purpose of avoiding tray configuration that we have already encountered, so as to avoid infinite loops when looking for solutions. The solution string is used as a buffer to store the sequence of moves leading up to a solution, and we use it to print out the solution at the end. The checkBlanks is to check whether the block is blank.

Methods:

private Scanner openFile(String)

public Tray(String filename, int numrows, int numcols)

public Tray(Tray tray)

void isOK(Block newBlock)

void updateTwoDimRepr(Block newBlock)

void removeTwoDimRepr(Block delBlock)

public String toString()

public String flattenTwoDimRepr()

public ArrayList<Tray> generateMoves()

public ArrayList<Tray> generateMovesByCheckBlanksWith2Dim()

public boolean checkGoal(Tray targetTray)

public void solve(Tray targetTray)

when the solve method is invoked. Solve will generate moves in a breadth-first tree and store them in a queue, called moveQueue. We use a while loop to continue solving until our breadth-first queue is empty. At each step of iteration, we first check if the current tray configuration happens to be the solution by invoking our checkGoal method. If so, then the program terminates. Othersize, we proceed to check if the current tray configuration has already been encountered by checking our PreviousConfigs HashSet. If we have seen it before, we skip the current state and keep generating new states. Otherwise, if the current state is unique, we add it to PreviousConfigs and then generate the next depth level of moves and place into our queue only the ones that aren’t found in PreviousConfigs.

## Block:

The Block class is a representation of the block we use in the program. The attributes and methods are listed below:

Attibutes:

private int length;

private int width;

private int row;

private int col;

private int id;

private Tray parentTray;

id is the index of the block in the blockList array in its parentTray; we need it in order to easily update tray when we have inserted a block (when reading the file) or when moving a block.

Methods:

public Block(int length, int width, int row, int col, int id, Tray parentTray)

public Block(Block block, Tray parentTray)

public String move(int direction)

public String toString()

public boolean equals(Object object)

The second constructor is a copy constructor, used when we make duplicate trays during the course of execution of the program. The move method takes an integer argument between 1 and 4 inclusive, with 1, 2, 3, 4 representing up, right, down, and left, respectively. This move method will check whether or not the move is allowed. The equals method is used to compare two blocks, and is used inside Tray’s checkGoal.

1. Alternative Strategies, Advantages, and Disadvantages:
2. Traversing the Solution Tree

We had to decide between using breadth-first and depth-first traversal. We were guided by our sponsor to use breadth-first traversal based on the fact that it would provide the shortest solutions. Further, we realized that breadth-first traversal would not only find the shortest solutions, but it would find *any* solution using the least amount of time and memory resources as opposed to depth-first traversal.

1. Representation of Blocks and Boards

For the representation of blocks, we toyed with the idea of using only a two dimensional integer array, but quickly realized we would need an additional data structure for convenience. Similarly, we were thinking of only using the blockList to represent all the blocks, but then realized that it would be unnecessarily to keep track of where each block is in the tray, and whether it is being placed on top of another block or out of bounds. In the end we chose to use both the list of blocks and two dimensional integer array to represent the game board as this gave us the most flexibility in terms of reusing our code.

1. Hashcode

We added a comma between blocks of different ids and added a period after each row. This avoid collision in Hashset for sure.

1. Representation of Solution

Initially, we decided to represent the solution as an ordered list of strings, with each string being formatted to match the desired output in the program spec. Later, we chose to simply use a string and keep concatenating to it, to avoid using the unnecessary list of strings. By choosing to implement the latter, we found an increase in speed of almost 300% when solving particular puzzles.

1. Evidence of using the strategy
2. Since most game trays appear to contain far more occupied spaces than empty spaces, we figured that we could generate moves around blank spaces instead of attempting to move every block on the board in every direction. This, we thought, should significantly improve our performance.