**README**

**DIVISION OF LABOR:**

Hung (cs61bl-dm) was responsible for creating the structure of the whole codes, and most of the coding part, which included hash code, tray structure, search methods as well as debugging. Hung also did the half the write up. Yuan (cs61bl-cu) was responsible for the rest of the write-up, the design of the program structure and algorithm, modifying classes and solver and writing helper methods such as isOK, input, output, and hash code. Yuan also provided algorithm insight. Andy (cs61bl-bf) was responsible for testing.

The group had many complications with division or labor, but someone had to do something or the project would have never been completed.

**DESIGN:**

Our program has several main parts:

1. InputSource.java:

This file that was already provided gives and object that reads input source files. This enables the initialization of a tray.

1. Tray.java:

A Tray object is specified by its size and the blocks distribution inside of it.

Naturally, we have to use a 2-D occupancy matrix to represent the empty spaces, and an ADT to record the blocks information. With the ADT, we can change or get the block distribution information at constant time if we want to make a move on a block. We chose to use a hash map as the ADT.

Also we included a tray object previous to reference the previous tray that moves to the current tray. Whenever we move a block, we record the prevMove and direction information in the new Tray after move. This variable will facilitate us to find and output the solution at the end. A tray also contains a depth variable which indicates how many moves it took so far.

The tray can be constructed in two ways, by an input file or by an existing tray at which a duplicate is made. It also has 3 important methods: move(), equals(), and hashCode(). move() moves a duplicated tray by modifying its internal data such as the hashed blocks, the occupancy matrix, prevMove, depth, and prev only if isOK() passes. isOK() checks for block collisions and out of tray bounds. If the check fails, it will throw an exception. equals() checks if two trays are equal by comparing blocks in each other’s hash map. hashCode() returns the tray’s hash code by first getting all its blocks hash code and putting it into a sorted array and calling a deep hash code on the array. Assuming each unique block has its own unique hash code, each tray configuration will have a unique hash code.

1. Block.java:

This class specifies the data structure of a block, with basic instance variables like: row, col, length, width with some getter methods and basic operations like duplicate and move methods in it. We overrode the Object’s equals() method and hashCode() method for a block according to its instance variables’ information.

equals() checks if two blocks are equal by checking if the instance variables are the same. hashCode() returns the code by appending row, col, length, and width into a single string and calling for the string’s hash code. Because it is likely that different strings have different hash code, different blocks will produce different strings which will produce different hash code.

1. Goal.java:

Goal objects contain a list that contains all goal blocks. It has a isSolution() method which checks if the goal is reached by looking up if blocks match on the tray’s occupancy matrix.

1. Solver.java:

This is the main part of finding the solution. With the help of a debugger argument, we are able to use four searching methods: Depth-First Search (DFS), Breadth-First Search (BFS), Priority Depth-First Search (PDFS), and Iterative Depth-First Search (IDFS).

DFS:

We used a Stack implementation to do the DFS. Starting from the initial configuration tray, we check all possible trays that are 1-step moves from current tray, if we reached the goal configuration, return, otherwise add the new trays into the stack, and start searching from the next tray in the stack.

BFS:

For the BFS, just replace the stack with a queue ADT by using a Linked List implementation. This will move all existing trays once before moving it again.

PDFS:

However later on we found that when we move from one direction, we do not need to move back, instead it might be better to continue moving in that direction. So we implemented a priority queue to rate next possible moves, depending on the previous move, and call this PDFS. In short, it will move a block in one direction until it hits a dead end before moving another.

IDFS:

IDFS harnesses depth to quicken the search. Given that most puzzles take 80 or less moves to solve, the depth threshold is set to 81. This means all trays after the depth of 81 will be put to the back of the queue. This enables all shallower trays to be moved. This cuts down all unnecessary moves and also unnecessary searches. Some trays even go to depth of 4000.

Once we found the solution, we can use the previous link (prev instance variable) to trace back the path from the initial tray. Meanwhile, with both the produceSolution() and print() methods we can output a sequence of moves toward the goal configuration.

To prevent infinite looping, we need to tell if we have already visited the tray. We used the hash set structure to record all the trays we have encountered before. The hash code which depends on the in-the-tray block’s hash codes should produce little collision.

1. PriorityQueue.java:

It is implemented by a Linked List. The queue supports both PDFS and IDFS. The element with the highest priority will be dequeued first.

1. Debugger.java

Contains the predefined commands for a debugger, provide methods to enable certain action and checks if the user has properly defined the debugger argument with isOK(). If it is a bad argument, it throws an illegal argument exception.

**DEBUGGING:**

Our debugging output facility is very convenient and helpful. We predefine several variables in a debugger.java. The debugging function is imbedded in the solver.java, and can be used as a command for running the solver by inputting arguments at the beginning, which enable us to run DFS (Depth-First Search), BFS (Breadth-First Search), PDFS (Priority Depth-First Search), or IDFS (Iterative Depth-First Search) separately, and outputs timing results if necessary.

The debugger menu can be called by passing the “–oinfo” argument.

For our isOK method, we used it to check whether the next move is valid, that is, whether the Tray and Block invariants hold after we move a certain block. It will try to make a “move” by checking if there is space for a block to move there before actually moving there.

The invariants we check are as follows:

(1) the block will still be in the tray’s boundaries after it has been moved;

(2) the block will be only moved to empty spaces, not the spaces that are pre-occupied by other blocks. Also, invariants like block to move must be present in the current tray and block’s size does not change after each move are imbedded in the design of our program.

One thing to note is that since we check isOK before every move of block, we have transplant the throw IllegalStateException into the move method, whenever the isOK fails.

**EVALUATING TRADEOFFS**

**Experiment 1:**

**Summary:**

This experiment examines a better way to output the path from initial configuration to goal configuration, between using data structure of an ArrayList memo variable in the tray class that records all the previous moves, and using data structure of a link within a tray denoting its parent tray, while only recording the very last move to current tray.

**Methods:**

We coded both structure and tested in real puzzle solving environment, by comparing the output time and solver’s runtime error that sometimes occur when solving big size complex trays.

**Results:**

When there is no runtime error, the run times of both structures are similar, however since our solver have to check many moves to find the solution, a run time error often occurs when we are using the first data structure. On the other hand, if we use a parent link to track the path, this problem never occurs. So empirically, the longer it takes the solver to find the solution, the more likely we would have a runtime error.

Long Arraylist, out of memory

More moves to record

Longer time

**Conclusions:**

On the one hand, both structures enable us to output the path in linear time, if we could record the moves successfully.

However on the other hand, as described in the graph above, the longer it takes to find the solution, the more moves we are likely to record in following trays’ memo, and it could consume all the memory and cause a runtime error. In addition, it is far more space economic to add a single reference than add a long growing arraylist in the tray object.

So we chose the second structure to keep track of the path for the output file.

**Experiment 2:**

**Summary:**

This experiment is to determine what data structures we need to fast generate possible moves, and fast compare the current configuration with the goal configuration, and make a move. Since the data structure representing a Tray is of great importance affecting the speed of the algorithm, we did the experiment at the very beginning of this project.

**Methods:**

Since comparison of two trays is crucial, the data structure should have a fast and unique way of representing the blocks distribution information.

We considered three possible data structures to represent trays: (1) an arraylist of block objects, (2) a hashtable generated by the blocks distribution information in the tray(mapping each spots (i,j) into a table index, and pointing to the block object that covers it), (3) a hash map of blocks in the tray. So based on the three data structure, we rewrite three equals() method for the tray and measure the time it takes to generate the representation and compare two tray objects.

**Results:**

For generating the data structure: (initialize a tray)

Speed: (2) < (1) ≈ (3)

For comparing two trays:

Speed: (2) < (1) ≈ (3)

For making a move: (change one block’s location)

Speed: (2) < (1) < (3)

**Conclusion:**

The result is as we expected. Since for (2) we tried to make a very long array to contain the mapped information, and many for-loops to write and compare, it takes far longer than the other two structures in all three functions. For (1), it takes constant time to add a block, while for (3), it takes constant time to calculate the hashCode and add to the hash map too, so the speeds are similar when generating the data structure. When comparing two trays, for (3) it can just compare the hashCode of the tray, which takes linear time to generate the hashCode from (3), while for (1) we need to compare each element in the arraylist, which takes linear time too. When making a move, it takes an arraylist linear time to find the block object and change its information, while for the hash map, the operation can be done in constant time.

So we choose to use a hash map to represent blocks information in a tray.

**Experiment 3:**

**Summary:**

This experiment compares different hash functions for trays to see whether the hash function will generate collisions, or whether the hash function can generate two same codes for two equal objects, that is A.equals(B) => A.hashCode()==B.hashCode().

Also, in this experiment, we examine the memory we need to calculate the hashCode and the speed of computation.

**Methods:**

We use tests to check the correctness of the hash function. We use debugger to measure the memory space involved and the time it takes to compute the hashCode.

**Results:**

Hash function 1: Using the imbedded static hash function of Array to calculate the hashCode of an array of blocks.

Speed: fast Correctness: bad Memory requirement: huge

Hash function 2: A user-defined hash function that returns the hashCode of an int array of block’s hashCode, where each individual block’s hashCode is calculated by adding up each block’s toString information.

Speed: fast Correctness: good Memory requirement: reasonable

**Conclusion:**

It is hard to make an imbedded static hash function work. Because Arrays.equals(obj[] a, obj[] b) does not equivalent to a.equals(obj[] b). We need to define a hash function that is computed by every block’s information, fast while keeping as few collisions as possible. Hash function 2 satisfies the purpose.

**Experiment 4**

**Summary:**

This experiment is to determine which way of search is fastest to reach the goal. We have 4 options: DFS, BFS, PDFS, and IDFS.

**Methods:**

We use debugger in the solver to switch search options and measure the times. All search methods were tested with puzzles of varying difficulty, steps to solve, and tray size. The time calculated only accounts for the search and excludes all initialization times in the solver.

**Results:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Difficulty | Initial File | Goal File | Search Method | Time Taken |
| Easy | init.from.handout | goal.2.from.handout | DFS | 1 ms |
| Easy | init.from.handout | goal.2.from.handout | BFS | 3 ms |
| Easy | init.from.handout | goal.2.from.handout | PDFS | 2 ms |
| Easy | init.from.handout | goal.2.from.handout | IDFS | 2 ms |
| Easy | big.tray.2 | big.tray.2.goal | DFS | 4 ms |
| Easy | big.tray.2 | big.tray.2.goal | BFS | 3 ms |
| Easy | big.tray.2 | big.tray.2.goal | PDFS | 4 ms |
| Easy | big.tray.2 | big.tray.2.goal | IDFS | 4 ms |
| Hard | century+180 | century+180.goal | DFS | 2887 ms |
| Hard | century+180 | century+180.goal | BFS | 4195 ms |
| Hard | century+180 | century+180.goal | PDFS | 2888 ms |
| Hard | century+180 | century+180.goal | IDFS | 3277 ms |
| Hard | big.tray.4 | many.blocks.20.goal | DFS | Too long |
| Hard | big.tray.4 | many.blocks.20.goal | BFS | Too long |
| Hard | big.tray.4 | many.blocks.20.goal | PDFS | Too long |
| Hard | big.tray.4 | many.blocks.20.goal | IDFS | Too long |

**Conclusions:**

DFS methods are usually faster in larger puzzles that require more steps. However, with the bigger trace and the larger amount of blocks, the way the tray is represented and the algorithm causes it to be much slower. This is because the program has to search for an empty space in the occupancy matrix. This runs in O(n2) and it takes a long time as the number of spaces exist. A better data structure and algorithm for the tray would be to keep track of all the empty spaces as well, just like the blocks. This would allow the retrieval of empty spaces in constant time rather than searching the entire matrix. DFS is faster in most cases for medium and larger sized puzzles because BFS requires more move iterations as more steps are needed to solve the puzzle. On the other hand, DFS would be more likely to reach the goal first.

**DISCLAIMER:**

The solver works successfully for most puzzles except for the larger trays with large columns and rows and also trays with many blocks. This would also cause insufficient heap memory errors.