Remy Orans -dt

Anna Carey -ii

Luis Torres -in

Quoc Thai Nguyen Truong -on

Readme

Division of Labor:

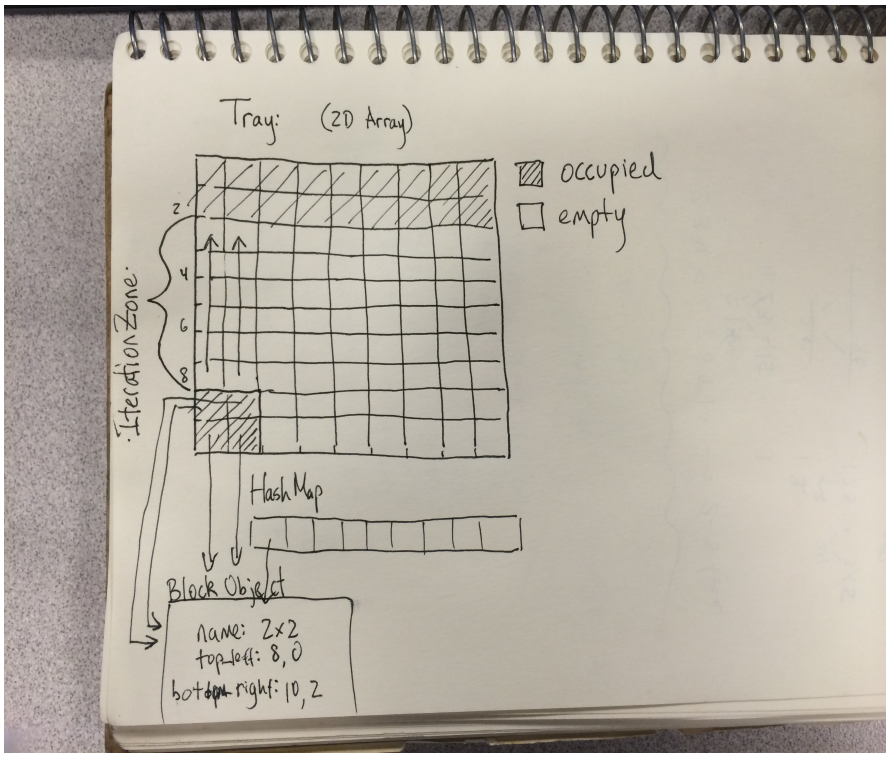
Our process for this project involved a lot of preliminary outlining. We found last project that jumping headfirst into coding without extensive planning was a poor strategy. Although we moved quickly through the code writing process, our lack of foresight made for a time-consuming and difficult debugging process. This time, we met before we started coding anything and discussed and debated which algorithms and data structures we would use. For about two hours, we went back and forth until we arrived at some consensus. We subsequently sat around and delved deeper intot he planning process. We mapped out which methods we wanted to use and wrote pseudocode. We all agreed upon input arguments and output arguments and the purpose of each method and documented it in our outline together. After this process we felt it was safe to code separately because we had spent so long discussing our ideas and were all on the same page. Luis and Thai did the majority of the coding for Checker, while Anna and Remy tackled most of Solver. Everyone pitched in for writing tests and thinking of edge cases at the beginning of the coding process. We all did most of the debugging together and split up the readme according to which parts different people completed. Overall, although it was challenging working with a larger group of four, our team worked well together and communicated effectively.

Design:

Two primary data structures are at the core of our design: a 2D-Array of Coordinates, and a HashMap that maps Coordinates to Block objects to provide more complete information about the occupied area on the Board. Together, these two data structures represent the Board.

The two main objects that are central to our implementation are Coordinates and Blocks. The Coordinate class is similar to the Points data structure found in the Java library, but we have added some instance variables, a boolean called occupied and an int called name. If a coordinates occupied boolean is set to true, the coordinate is hashed to a Block in the HashMap which allows for efficient lookup to see what the actual dimensions of the Block are. The combination of these two data structures allow us to easily iterate over the Board differentiating free and occupied space. If a space is occupied we are able to tell how it is being occupied with the HashMap.

The Block class has instance variables name, top_left, and bottom_right. The name is an int that remains consistent across blocks of the same size but differs across Block size. The name is useful in calculating a block's "state" which is a unique represent ation of a Tray Configuratio n that allows us to memoize seen



configurations in the solver. The most important methods in the Block class are move, canMakeMove, and updateMove. Move just calls canMakeMove and upDateMove. CanMakeMove works by iterating over the 2d array of Coordinates across a block's path in each possible direction of movement. In Solver it goes as far as possible until breaking. If we are moving a block in Checker it checks until the block's new position. If canMakeMove returns true, meaning a possible move is valid, it is updateMove's turn at the plate. UpdateMove needs to remove all the block's current occupied coordinates and reinitalize coordinates in the block's new position. Additionally it has to hash these new coordinates to the block object they represent.

We used graphs to implement solver. The vertices of the graph are Tray configurations and the edges are moves between different configurations. Solver has a Node called goal, a HashSet of Integers called seenInt, a HashSet called goalBlocks, a Stack called fringe, and an ArrayList of Integers called prime. We created a Node class that represent the vertices. A Node is a hacked Checker object, only using instance variables in checker that are applicable to Solver. In the constructor, we create an initial Node, and push it onto the fringe. When we solve, we pop a Node off the stack and call generateMoves on it. GenerateMoves generates all the possible moves by direction and distance for each block and creates new Nodes out of them. New Nodes are created using Checker's Copy method which does the work of changing the 2d array and hashmap for us. Once we have these new nodes they are pushed onto the fringe.

Experimental Results :

Experiment Number 1:

This experiment compares Depth First Search vs. A* search. Depth first search is an algorithm that will generate tray configurations as deeply in the decision graph as possible before moving onto a new branch. A* search uses a heuristic to “intelligently” expand the graph, searching areas it deems to be close to the solution. The heuristic consists of “distance from start” vs. “distance to solution”. Distance from start is the number of vertices needed to progress to the current state. Distance to solution each block's “manhattan distance” from its position in the solution state. Manhattan distance means the distance is calculated moving parallel to the x and y axis in a cartesian coordinate plane. A* balances the depth first and breadth first approaches so you never go far out along a branch unless your actually progressing to a solution. The challenge of A* is that you have to calculate the heuristic for each node which can be resource and time intensive. Also A* is used to find the quickest path to the solution which we don't care about in this case. Our implementation of A* search involved using a priority queue that ordered tray configurations based on the heuristic. Each node had a cost instance variable which was calculated when it was initialized by comparing iterating through the Node's hashset of blocks and comparing it to the goal hashset of blocks. We had consistent naming conventions for our blocks so we could compare a 1x1's position to a goal 1x1's position.

Conclusion:

Depth First Search ended up being much faster. This was due to our convoluted implementation of calculating the heuristic which ran in n^2, n being the number of blocks in the HashSet of blocks for each node in the graph. The overall running time would be V*N^2, v being the number of vertices in the graph. The overall lesson learned was that even though

one algorithm can be “smarter,” unless your implementation is also smart the net result will be a slower program.

Experiment Number 2:

Generating Moves:

This experiment centered around different ways of generating moves. When generating a move you can move one “unit” to the next occupied position in increments of one in any direction. For any given blocks this leaves a bunch of different possible tray configurations that could result from any given vertice. Would it be faster to generate new nodes by only expanding tray configurations created by moving blocks one unit of distance at a time and adding to the fringe, or should we add all possible Nodes created from all possible moves to the stack. This experiment offers conclusions regarding which is faster.

Conclusion:

As you can see from Figure 1, it is much faster to find all possible moves instead of using incremental moves. This is because the all possible moves implementation goes to the end of the decision branch of the graph, where it may encounter another block or the edge of the board. There are more moves in the incremental moves because we are just going one move by one of distance one, rather than going as far as we can until we can no longer move.

Experiment 3:

DFS vs BFS

Summary: This experiment centered around using a DFS implementation to solve versus using a BFS implementation. In DFS, it will solve faster if the goal configuration is far away

from the original configuration. If this is the case, then we do not use up as much memory. In BFS, it needs a lot of memory to store each move that neighbors the current location of the block while trying to move. BFS should be faster than DFS if the goal configuration is close to the original configuration. So, we wish to check the times of both traveral implementations for a close goal and a far goal.

Results:

Close goal (Start from top left and go to middle):

Far goal (Start from top left and go to bottom left):

Conclusion:

BFS is faster when the goal is close to the initial configuration, and DFS is faster when the goal is far from the initial configuration.

Program Development :

Outlining proved to be a huge focus of this project. As we discussed in our division of labor section we learned a lot of hard truths coding the second project. The lazy way of coding is to write out the first solution arrived upon without thought of future code that must build on that solution as well as boundary cases. The lazy method works for short scripts, or homework assignments, but when writing giant programs it eventually fails. One can't build the foundation of a house unless they have some idea what the house will look like. While outlining we figured out our main data structures to represent all the pertinent information. Once we figured out which data structures would lead to the most efficient implementation, we decided upon methods to manipulate those data structures. After agreeing upon methods

we further specified the input and output of each method. After finishing our outlining crusade, we decided to add some test driven development to our process.

Because of problems we had in our last project, we decided to strictly adhere to test-driven development. Then, we wrote tests and together brainstormed some edge cases to add to our test. (Once we completed our code and had a more thorough understanding of the project, we were able to think of more edge cases.) In previous projects, we had to submit test code but rarely even used JUnit to test our code. Instead, we would write almost all of the code and then attempt to debug either using the Eclipse Debugger or print statements. This proved to be very inefficient. In many cases, we couldn't find the exact location of the bug for hours. We also still would have to write test files to turn in. In this case, our test-driven development proved more efficient and helped us more quickly gain a deep understanding of the project spec and our code.

Obviously Murphy's law quickly interrupted our idyllic approach to writing code. Despite our best efforts to write test-driven code with impeccable style, one can't predict all the possible challenges involved in implementing a solution and ultimately our implementation suffered from some problems.

Disclaimer :

The code doesn't pass the hard tests. We suspect this is due to our generating move method, which doesn't properly generate all resulting nodes. Due to this error we are probably missing some of the search space. Additionally, our code suffered from some redundancy. It was hard to predict the necessary overlap between Checker and Solver which used a lot of the same

functionality but had slightly different demands. Because of this we ended up writing some stylistically bad functions that look quite similar but are for different things.

Improvements:

After completing our code, there are several aspects which we could have improved. There is extensive overlap between Checker and Solver. The two classes contain many very similar or even identical methods with similar names which clouds the logic of the code. Even though they contain similar methods, their primary jobs are quite different. It would have been better to write more all-purpose methods that could work seamlessly with both Checker and Solver. Additionally another place we could improve would be to hash anytime we use ArrayList. This change would speed up our code.