Project 3 Readme

Members

Section 101 (Edwin)
`cs61bl-al` Jingchen (James) Wu
`cs61bl-ae` Kevin Chian
`cs61bl-ak` Jungyoun (Josh) Kwak
`cs61bl-ag` Ethan Yi Sheng Chan

Contents

Division of Labor

We did our best to divide the labor equally. We had two project planning meetings ahead of time in order to plot out the code structure, the general strategy for solving, and the division of labor. The result of dividing the labor was laid out thus:

James - Works on Tray.java and Block.java
Kevin - Works on Tray.java and Solver.java
Ethan - Works on Unit tests for Tray.java and Block.java
Jung - Works on Benchmarking various functions of Tray and Solver

We would commit the code together to github and work on the project asynchronously and together. We were able to exchange messages via Google Hangout and Gmail on when to meet and work together. It took us about two weekends of working together on separate files to come

up with a complete solution to the project. SInce we always worked together through Google Hangout, we all spent approximately the same amount of time.

Design

We decided to split up classes in accordance to abstractions: A Block object would go on a Tray, and a Solver would move pieces on the Tray to solve a puzzle. We ended up with 3 public classes: Block.java, Tray.java, and Solver.java.

Block.java

Block.java represents a single block and stores the information for its current position. A few getter functions are available to access its whereabouts, and it has a move() function that generates a new Block based on the current block and a desired movement position. Blocks are static objects that are not meant to be changed internally upon creation, but instead have methods that create a copy and change the clone's internal information instead. This allows us to use the same Block instances within different Tray instances in order to save memory, without having to worry about how the movement of one Block in a tray would affect Blocks in other trays. To distinguish blocks from each other, we gave each block a unique ID corresponding to the order they were created. The hashcode of the block is created from the unique set of coordinates each Block has, by representing its coordinates as a four digit (one for each dimension) base-256 number.

Tray.java

Tray.java is similar to Block.java in design. Instances of Tray are meant to be static and the blocks within are not meant to be moved. Each instance of Tray has a method moves() that can generate a list of all other Trays() that are possible succeeding moves of the instance Tray. Each tray is constant, and succeeding Trays are generated, not modified from existing trays. The information of a Tray contains the whereabouts of its blocks. It stores this information redundantly in two data structures: a HashMap of Block IDs to Blocks, and a 2D array of Block IDs that corresponds to the layout of the board. The reason for this redundancy is for performance in calculating succeeding trays and comparing against another Tray. It checks for internal consistency upon removal or addition of a Block through an isOK method.

For an example Tray, the following lines would result in the following 2D array of Block IDs:

Input Init File	Input Goal File	Initialize Tray	Add blocks	Goal:
54 3232	3232	|0|0|0|0| |0|0|0|0|	|0|0|0|0| |0|0|0|0|	|0|0|0|0| |0|0|0|0|

3131 4242 3232		|0|0|0|0| |0|0|0|0| |0|0|0|0|	|0|0|0|0| |0|1|3|0| |0|0|2|0|	|0|0|0|0| |0|0|1|0| |0|0|0|0|

Solver.java

Solver.java has the responsibility of solving a problem from an initial tray and a goal tray, and to print out all steps if possible, or exit quietly otherwise. It also contains the important main loop that reads from the command line args and printing out debugging information if necessary. **Solver effectively works by running Dijkstra's algorithm on a tree of nodes representing trays and edges representing moves.**

The actual class Solver starts with the initial and goal trays, and solves the problem by running a graph search (without storing the whole graph). If stored, the graph would represent all possible states of the Tray, and paths between two nodes would represent a possibility of reaching one tray from another tray by sliding a single block one distance.

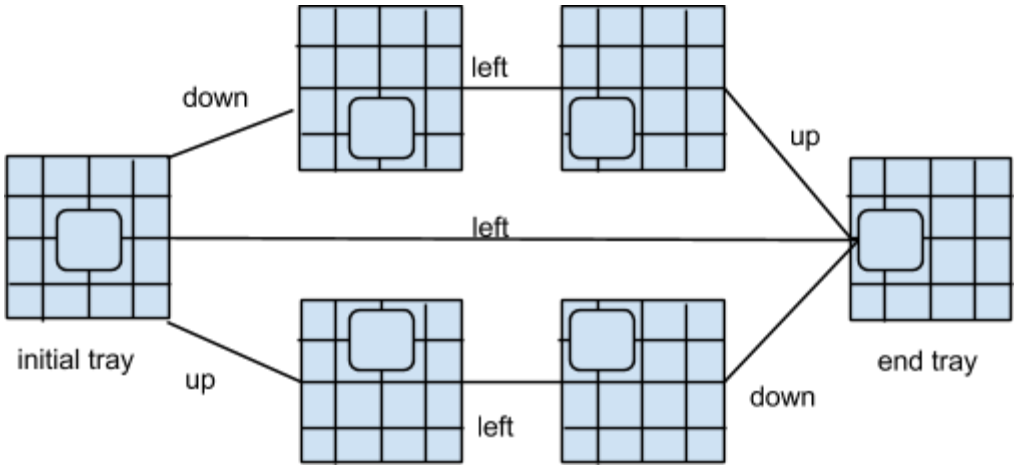


Fig. Diagram showing possible nodes on the graph and the edges represent moves to get from the initial tray to the end tray.

Instead of storing the entire graph, the program uses a fringe object, represented by a priority queue. The program continuously pops off the highest-priority object from the queue and processes it by finding all the possible moves (edges in the graph) and storing the nodes connected to those edges into the fringe, along with their priorities. The first object in the fringe is the initial tray. If the goal state is reached while popping nodes off of the fringe, the program exits because the problem is solved. The path from the initial tray to the goal tray is stored in Tray.java as an ArrayList of moves. All visited nodes are stored in a HashSet to avoid storing objects on the Fringe that have already been there.

Heuristic Strategy

The priority of each node and which ones get processed first in the fringe is determined by a heuristic() function in Tray.java. The heuristic function returns a number between 0 and 1, where 0 means that the Tray satisfies the goal Tray, and 1 represents that it is impossible to reach the goal state from the current Tray. Any number in between 0 and 1 represents how close the current tray is to the goal tray, with lower numbers meaning closer to the goal state.

The heuristic is calculated by finding the average manhattan distance between initial blocks and goal blocks. For each goal block, the program tries to match up an initial block that has the same size and has the closest distance to the goal block, and assign that goal block a score calculated by the ratio of that distance to the maximum manhattan distance (i.e. width of tray + height of tray). Then it averages the scores of all the goal blocks and spits out a score that represents how close each goal block is to having an initial block to fulfilling it.

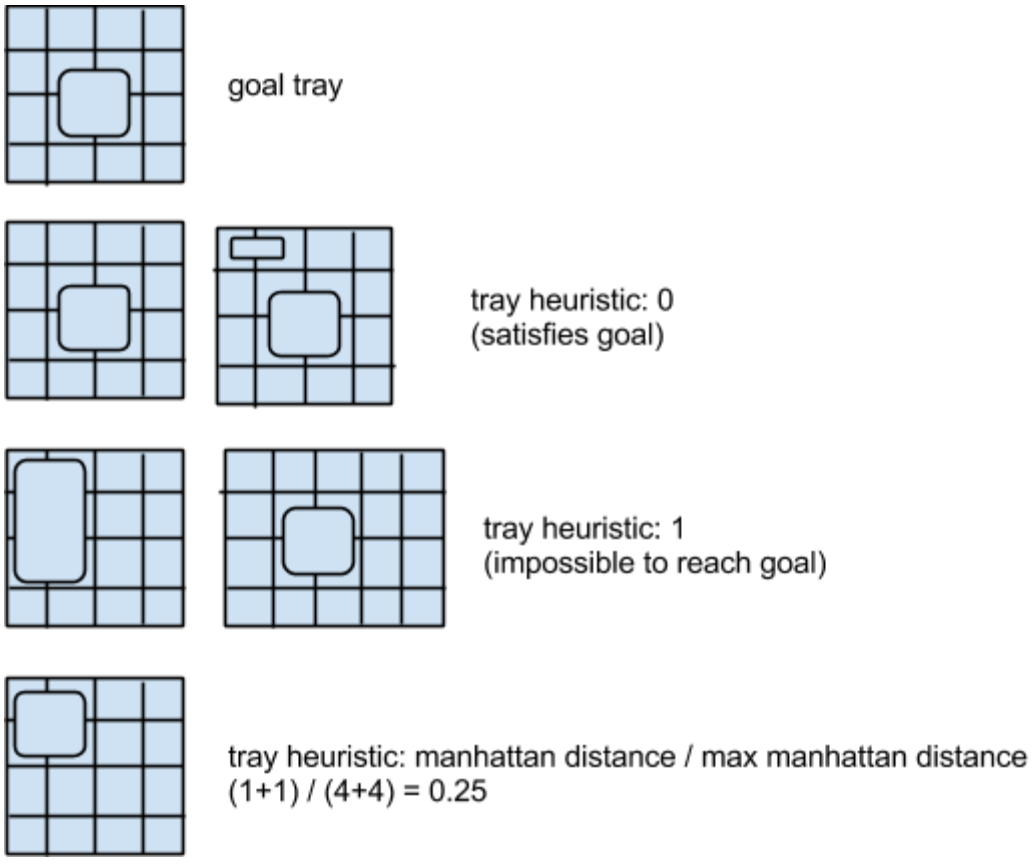


Fig. Heuristics for various setups, compared to a goal tray.

When nodes (Trays) are placed into the fringe, they are ordered in accordance to their heuristics. Each Tray has a reference to the goal tray so they can compare against each other and compare against the goal tray at the same time. Heuristics are calculated once and then

memoized in a Tray for future use, since Tray objects are static, heuristics will never change.

Debugging

Our solver method contained –o methods, which are similar to Project 2 and Project 1's iAmDebugging boolean. A user could call these –o options by inputting it as the first argument for solver and our program would recognize it and run the setParams method to check which command was being called. Then it would execute the following options:

`-ooptions`	print out all possible –o choices and exits out of the solver
`-odebug`	print out ALL output at each step of the process: creating the trays, calling solver, beginning to check a tray, finishing checking a tray, finding a winner, et
`-otime`	output the time it took to run a puzzle after solving it
`-oprint`	print out each tray to solve the puzzle before solving it

Our `Tray.isOK` method in our tray class checked if the board state was valid. It checked the following things:

- Are all coordinates valid for the blocks? We didn't want to input a 5x4 block in a 1x1 tray, or put a small block outside of the bounds of the tray. This was really to check if the blocks stored in the hashmap corresponded to legal positions in the 2D array.
- Are any blocks overlapping? If they were, then we violated an invariant
- Do all blocks in the hashmap correspond to the correct position in the 2D integer array?

These debugging options saved us numerous headaches when writing each step. It helped avoid problems where something would go wrong when calling the solver, but we didn't know whether the tray was buggy or something in the solver was buggy. The debugging options helped isolate these issues each step of the way, so we knew we were building upon solid foundations and coding correctly. It also helped when we were inputting hard tests to see how long it took to solve in order to gauge how much faster we needed to make our solver.

Unit tests were written and also saved us a lot of debugging time. They are basic and do not cover 100% of the lines in the code, but cover basic functions and give us a piece of mind. They also give us hints as to what might be wrong if they pass but the experiments fail.

Evaluating Tradeoffs

Data Structures

What data structures choices did you consider for the tray? What operations did you optimize: fast generation of possible moves, fast comparison of the current configuration with the goal, or making a move? How did these considerations conflict?

Summary: To represent the tray, we considered the following implementations:
a) A 2D int array
b) An adjacency list where each block would have its neighbors (including empty spaces) in a linked list with the following order: above the block, right of the block, below the block, and left of the block. This would also include a linked list of all the spaces.
3) An adjacency matrix with the same idea as the adjacency list, with its neighbors marked true and the rest marked false.
After our experiment, we found that a 2D int array allowed us to compare goal states the fastest, and made it easier to understand what was going on in a board; however, finding moves took a while and needed to be processed separately by finding empty spaces, finding neighbors, then getting all possible moves. An adjacency list allowed us to find neighbors quickly, and then isolate all possible moves from that list, but it was slow to compare goal states and incredibly difficult to think about while coding. An adjacency matrix was just highly space inefficient, slower in finding neighbors, and wasn't as clear as a 2D integer array, so it offered almost benefits compared to the other two.

Methods:

We ran our three implementations by doing time tests for each basic implementation. We followed the following steps for each data structure:
1) We used the same sample puzzles for each data structure to avoid variance. These puzzles included one tray with lots of space and one tray with lots of blocks.
2) We created the data structure we were testing, then initialized the sample puzzles starting tray.
3) We used System.currentTimeMillis() to test how long it took for each data structure to initialize the tray, generate moves, and compare if done.
4) To initialize the tray for 2D int array, we did it the same way we implemented our project. Generating moves and comparing goals were also similar to our final project.
5) To initialize an adjacency list tray, I needed to first use a 2D int array then get the neighbors of each block by iterating through the blocks. Generating moves was just checking if there was a linkedlist of value zero in a certain blocks neighbors. Comparing took a while to think of, but our final idea was to check if any block had the top, bottom, left, and right variables of the goal state, which involved iterating through each block.

Results: (We averaged the times for these trays over 3 puzzles)

	Initialize Tray	Generate Moves	Compare to goal

2D int array	533 ms	3524 ms	332 ms
Adjacency List	5042 ms	435 ms	924 ms
Adjacency Matrix	4037 ms	372 ms	950 ms

Conclusions: The adjacency lists took much longer to initialize because we couldn't think of a way to get neighbor without using an initial 2D int tray state then getting each blocks neighbors from that comparison. This same issue happened with adjacency matrices too. While generating moves only required to check each block and seeing which direction (aka what order it appears in the linked list) had a block id of 0, comparing the goal configuration took much longer than anticipated because I ran through the entire matrix to see if it matched. Note that the most probable reason comparing goal states took so long was because of my implementation of checking for the goal state, and not because of the adjacency list itself. However, our final conclusions were that while adjacency lists did in fact save time generating moves due to the fact that they only needed to iterate through the existing list, the level of confusion and workaround involved was not worth the time save. Also, initializing and comparing to board states were drastically slower for smaller boards, but I recognize that as boards get larger both goal comparison methods will be about the same. We ultimately agreed that with the limited advantages an adjacency list offered and the easy to understand intuitiveness of the 2D int array, it was better to use the 2D int array even if it meant giving up efficiencies in generating moves. One concern that we did have were board states that involved few blocks and lots of space. We would be both wasting space and time, but recognized that most puzzles have more blocks than spaces and a 2D array would be better at handling these more pragmatic and harder puzzles.

Storing Tray States

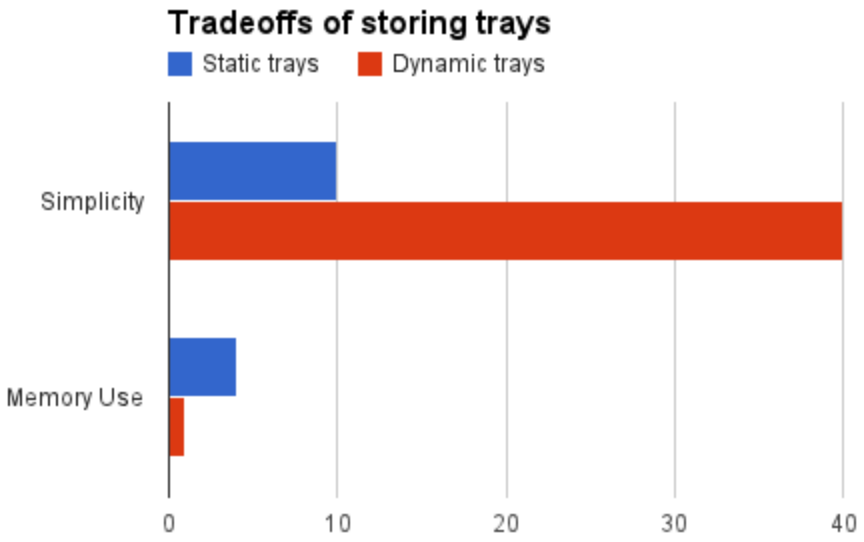
Summary: In evaluating the tradeoffs for the different methods of moving blocks around, we considered many options and instead went with the one that would be easiest to implement rather than the one with the best memory usage or the best runtime.

Methods: We considered many different types of methods for moving blocks around in order to solve the problem. It was clear from the spec that we were required to solve the problem by simply trying every combination of moves until every possible state of the tray has been explored, but there were multiple ways of doing this:

Option #1: We could have a non-static Tray object that recorded a list of moves that it took to get to the current state, and was able to backtrack any of its moves. This would require that the Tray be dynamic and add and remove blocks very often (whenever a block is moved or backtracked). This would also require the Tray to have many many methods for tracking which moves have already been made in the past, and would require the Tray to be aware of all past states that it has already explored.

Option #2: We could have a static Tray object that generated more Tray objects from its current state, but would not modify its internal data, but instead create new ones from scratch. This would give the Tray the responsibility of only generating new moves, but does not require it to keep track of previous Trays that are visited or future trays that have already been explored. This option would require a lot of memory whenever new Trays are generated, but the runtime and simplicity in code design and implementation would be worth the tradeoff. Since the specs said to not worry about memory space, we thought this was a viable option.

Results: The tradeoffs between using a lot of memory vs. overloading a class with function can be represented with the following graph:



Note: Simplicity is measured by the **number of functions** we would have to write for Tray.java. Memory space is measured by the **number of different Trays** we would store for every call to moves().

Conclusions: We went with the second option for the sake of pure simplicity in code. Solver.java was able to contain most of the code for running Dijkstra's Algorithm, and we were able to keep Tray simple and easy to implement by having it only generate moves. This was also beneficial for us to work asynchronously because we could work on moves generation and the graph search without touching the same files.

Tray Hash Functions

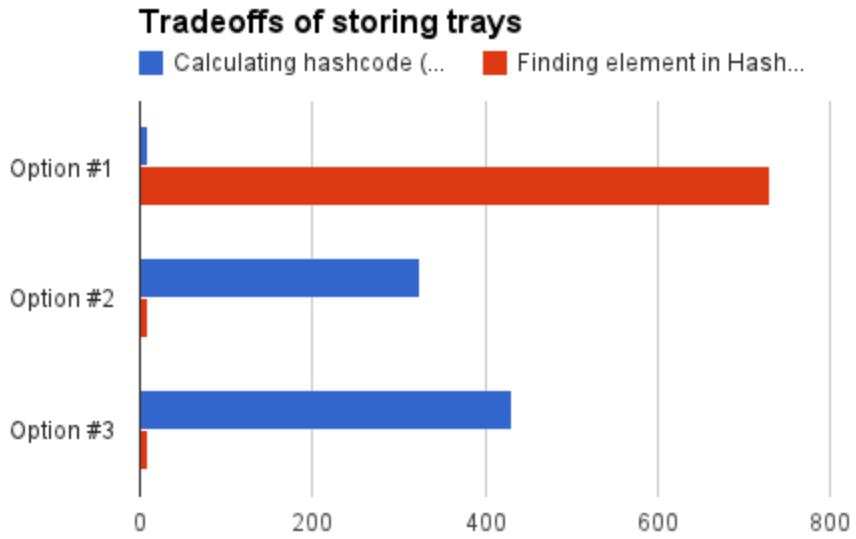
Summary: The implementation of Solver.java stores the list of visited nodes inside of a HashSet to avoid visiting nodes that have already been visited (avoiding cycles in the graph). In order to

compare future nodes against visited nodes, we have to have a good Hash function to minimize memory usage and maximize HashSet performance. If all the nodes have the same hashcode, the set is no more efficient than a linked list. After testing different types of Hash functions, we used one that's relatively easy to implement and fast to perform.

Method: We had a few options for implementing Tray.hashSet. Our goals were that it runs O(1), or as best of big-O that we can make it, and that it gives us a hashcode that's spread out in a good distribution and not cluttered around one center. We also had to make sure that the hashcode for equivalent trays are equal (i.e. hashcode is the same if equals returns true) so that HashSet can visit the correct bucket right away before it calls .equals.

Options #1: Keep the default Object.hashCode method, which computes the hashCode based on memory address.
Option #2: Compute the hashCode from the ids of the current tray (in the 2d array).
Option #3: Compute the hashCode from the hashcodes of the blocks (from the hashmap).

Results: We timed the time it took to compute the hashCode of 1000 random trays, and the results were thus:



Conclusions: While option #1 had the best runtime, it gave the most headaches because Trays that were equivalent did not have the best hashcode, so that the HashSet was not able to quickly locate the bucket associated with the Tray. Option #2 and #3 were a close tie, but #3 was rather slow for Trays that had a lot of Blocks, whereas Option #2 ran in constant time.

Graph Traversal

Summary: Solver.java had the job of picking which nodes on the fringe will get explored next.

Two common approaches would have been depth-first search and breadth-first search. While breadth-first guarantees the shortest path, depth-first prioritizes the best runtime, if you get lucky. We used an alternative method, which is to implement Dijkstra's algorithm using a PriorityQueue.

Method: Instead of using a regular Queue to do breadth-first search in the node space of possible Trays, which would result in a breadth first search, we pick the node in the fringe that has a state that's most like the goal state. This is done by running a heuristic function to compare the goal state and the tray state. The heuristic function was described in details earlier, but basically calculates a heuristic based on the manhattan distance between goal blocks and current blocks. The solver then picks the Tray that has the best heuristic to explore and put on the fringe. This method does not guarantee the fastest solution, because if the heuristic is wrong, then a shorter path is possible, but it does guarantee a very stellar runtime.

Results: After benchmarking PriorityQueue, Queue, and Stack as data models for our fringe, we came up with the following results for a hard puzzle:

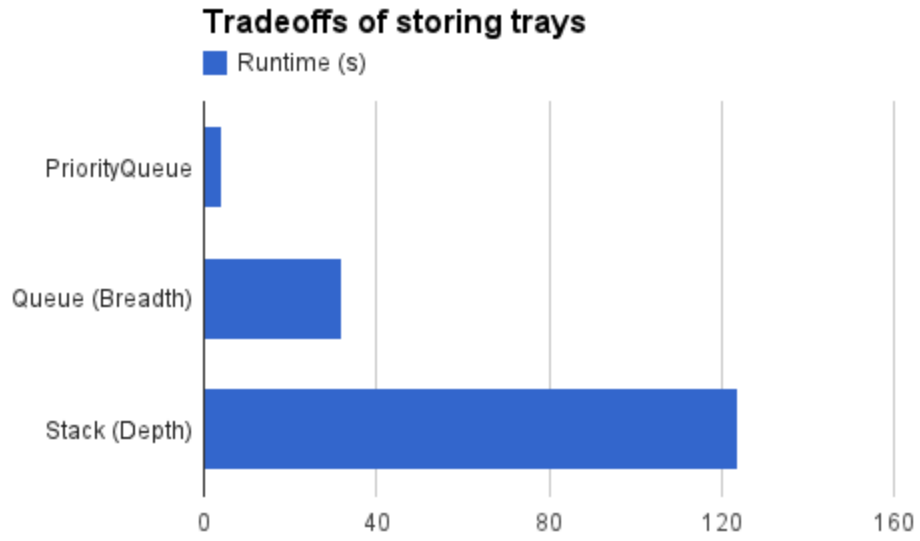


Fig. The runtimes of solving a hard puzzle. Depth first didn't get very lucky.

Conclusion: We used a PriorityQueue with the heuristic as the priority code, which maximized the ability for our Solver to pick good nodes, with a slight trade off in runtime in order to calculate the heuristic. It was worth the effort.

Disclaimers

Our code works relatively well for most of the tests. One major weakness in our code is the handling of many empty spaces. Our moves method finds all the empty spaces and checks if

blocks can move into it, but if there are too many empty spaces then it slows our code drastically down, as we have seen when running big.tray test cases. If we were given more time, we would further improve the code by having a generate moves method that checks all the blocks and their possible movements on a space heavy tray to avoid searching through every single space. Then we would have a third method that chooses which generate moves to call depending on the space to block ratio in that tray. We would save large amounts of time and memory by doing so when running programs with lots of spaces, because then we can reduce the number of items we need to iterate through.

Program Development

We did all tests using JUnit in Eclipse.

We wanted to code from the ground up, so we started with the most fundamental parts then built up. Since solver used trays and trays used blocks, we decided to make block.java first. Then, we tested block thoroughly before progressing to tray in order to ensure that later on we wouldn't have too much confusion on what was breaking and what was working. We tested the following things in block:

1. creating legal and illegal blocks
2. if the equals method worked when comparing two blocks. We tested two different block objects with the same dimensions and two different blocks with different dimensions to see if equals worked.
3. The hashcode for blocks.
4. The basic return functions for blocks (top, bottom, left, right, id)

Then, after making sure blocks was bulletproof, we moved on to the tray.java and coded the constructor, tested it, then moved onto each method writing and testing each one as we went along. We made sure that each method worked the way we wanted it to before moving on to new methods to avoid confusion and possible future error. Some of us coded then tested while others wrote tests without seeing the code to try and account for all edge cases and find all possible ways to break the code. At this point, we didn't code the heuristic functions yet, because we weren't completely positive on how to implement it. We decided to do this after solver worked for some easier cases.

After writing Tray.java, we timed how long it took for trays to initialize, to find moves, and to check if it matches the goal state. If the times weren't as fast as we wanted we would go back and find ways to increase speed, but overall there weren't any large issues needing fixing.

After the block and tray classes were complete, we moved onto Solver.java and created a preliminary solver class using a stack. This step probably took the longest, because our testing and debugging involved running lots of easy tests and seeing if it executed, then checking our solutions against the Checker.class. It was difficult to debug and involved lots of careful traversal

through Eclipse's debug feature. It had no heuristics and worked rather slowly, but after finally successfully doing a few easy puzzles, we could move onto improving its run time.

After all preliminary steps were completed, we researched heuristics and designed a heuristic that checked how far blocks with the same size as the goal blocks were from the goal state. Then, we coded the manhattan distance method in the block class, the heuristic method in the tray class, and corrected our stack fringe implementation into a priority heap to work with heuristics. We timed and tested our solver again, and after some debugging found that the heuristic drastically improved run times.

Afterwards, we constantly tested our program with the provided puzzles to iron out any major issues, and worked to make our code cleaner until we were sure everything worked and was easy to understand.

We decided to run our project this way because we felt it would be beneficial in the long run to test one step at a time rather than having all the code done first then testing for bugs. We also avoided test driven development because we found that we were more efficient coding, testing, and debugging in that order. We tested all public methods because if a user or outside source were using our code we didn't want the code to break in any possible way. Private methods, since outside sources can't use them, only had to work with the given input and output and were not tested explicitly.