Project 3: Sliding Block Puzzles

Team Members: Lilly Chou (JL), Rashmi Vidyasagar (JB), Joanna Chau (IV)

Lab: Section 108

Division of Labor

We worked on the project as a group with everyone present at every meeting. Together, we came up with potential algorithms to solve the problem and rotated typing of the code. We decided to work in this manner to prevent inconsistencies and to ensure that the different classes and methods all worked together properly. Additionally, doing the project this way allowed us to all learn together!

Regarding the division of the report, Lilly wrote up the design for the entire algorithm, Joanna wrote up the experimental results, including graphics, and Rashmi wrote up the program development, disclaimers, and improvements that could be made to this project.

Design Overview

The logic of our implementation is divided into two parts: Solver.java is responsible for creating Tray objects, generating next moves by creating new Tray objects with new block locations, and printing out the solution; Tray.java is the representation of the puzzle board by storing information about the blocks on the board through Block objects.

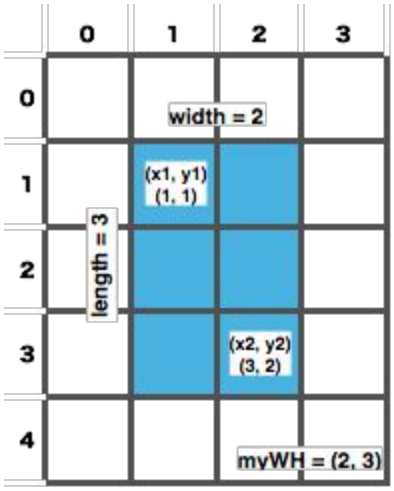
Our solution make use of class Block to represent the blocks in the puzzles. Every Block object stores information about its location and size, as well as its priority and the type of Tray it exists in (see Class Block). Our Tray class contains information about the its parent (previous move) and children (next possible moves) Trays, location of the blocks, and various data structures containing information about its blocks.

Class Block:

Instance Variable

- ***coords*** (ArrayList<Integer>): Store the location of the block this object represents in the same format as given by the init/goal files.
- ***height*** (int): height of the block
- ***width*** (int): width of the block
- ***myWH*** (ArrayList<Integer>): The width and height of the block in respective order
- ***x1*** (int), ***x2*** (int), ***y1*** (int), ***y1*** (int): The corresponding location of the block (see image below)
- ***ID*** (int): Cached hashCode of the block
- ***myPriority*** (int): The smallest distance from current Block to the closest Goal Block of the same dimension. This is used to calculate the heuristics of the Tray.

- **BIG** (boolean): If either the width or height of the Tray dimension is greater than 100, BIG is true, else false. (See *makeHash()* under Clas Block for the purpose of this implementation)



Methods

- **Block constructor #1:** *public Block(ArrayList<Integer> location, Integer[] TraySize)*
  This constructor is meant for the creation of blocks for the goal Tray. All other Tray objects requires information about goal Tray to calculate its Block objects' distances from goal Tray's Block objects for heuristics.
  The constructor initiates all the instance variables.
- **Block constructor #2:**
  public Block(ArrayList<Integer> location, LinkedList<Block> sameSizedGoal, Integer[] TraySize)
  This constructor is meant for the creation of blocks for Tray objects other than the goal Tray. The additional parameter *sameSizedGoal* takes in an LinkedList of the current goal Tray's Block objects with the same dimension in order to calculate the heuristics. (See minDist).
- **public int minDist(LinkedList<Block> myGoals)**
  This method takes in *sameSizedGoal* from the second Block constructor and returns the smallest distance from the current Block to a closest Goal Block with the same dimension.
- **public int makeHash()**
  This method creates customized hash code, which is stored as the ID. Dependent on the size of the Tray, we generate different hash codes. The general logic is to create a hash code corresponding to the coordinates of the Block. However, when BIG is true, using all four coordinates will cause result in a *long* rather than an *int*. Therefore, we use the first three coordinates of the Block and fill in 0's to maintain each coordinates' place.
- **public int hashCode()**
  We override the default hashCode() method to return the ID of this Block object to limit the call to the calculation of hash code to once per creation of Block object.
- **public boolean equals(Object o)**

Two Block objects are equal if the have the same ID, which is determined by their coordinates.

- **public String toString()**
  We override the *toString* method to return the coordinates of the Block for identification purposes for the debugging process.
- ***get methods***
  The following ***get*** methods returns the respective instance variables for easy access outside of the Block object:
  - *getCoords, getPriority, getX1, getX2, getY1, getY2, getHeight, getWidth, getWH, getID.*

Class Tray implements Comparable<Tray>:

Instance Variables:

- ***blocks*** (boolean[ ] [ ]): A boolean representation of the current puzzle board. Occupied coordinates are represented by *true* while empty coordinates are represented by *false*.

	0	1	2	3
0	FALSE	FALSE	FALSE	FALSE
1	FALSE	TRUE	TRUE	FALSE
2	FALSE	TRUE	TRUE	FALSE
3	FALSE	TRUE	TRUE	FALSE
4	FALSE	FALSE	FALSE	FALSE

(Continuing the example from above)

- ***blockSet*** (ArrayList<Block>: An ArrayList of Blocks on the current puzzle board. This instance variable is used to create next moves and new Tray objects. On the off chance when Block objects have colliding hash codes, we decided to use ArrayList rather than HashSet in order to make sure we store all Blocks in the set.
- ***blockIDSet*** (HashSet<Integer>): An HashSet of Block's ID's. This is mainly used for the *.contains* method.
- ***hashByBlockID*** (HashMap<Integer, Block>): A HashMap mapping Block ID to the corresponding Block objects.
- ***hashBySize*** (HashMap<ArrayList<Integer>>, LinkedList<Block>>): A HashMap mapping Block's *myWH* to a list of Blocks with the same *myWH*. This is used to create a list of Goal Blocks with the same size, which is passed into Block's constructor to generate its *myPriority*.

- ***moveToGetHere*** (ArrayList<Integer>): The move from previous Tray to the current Tray in standard format. (*i.e. moving from [1, 0, 1, 0] to [1, 1, 1, 1] is represented as [1, 0, 1, 1]).*
- ***myChildren*** (HashSet<Tray> myChildren): a HashSet of Trays as a result of moving a Block from the current Tray's configuration to a new one. This is generated by the *createMoves* method in Solver.java.
- ***myParent*** (Tray): the Tray representing the previous move.
- ***ID*** (int): the hash code of the Tray.
- ***myPriority*** (int): The priority of the Tray for the purpose of sorting Trays in the PriorityQueue in Solver.java using the .*compareTo* method.

Methods:

- ***Tray Constructor #1:***
  public Tray(Integer[] dimension, ArrayList<ArrayList<Integer>> myCoords, Tray goalTray)
  The first constructor is used to create the *init* and *goal* Tray. Within the method, we created new ArrayList of Blocks using *myCoords*. All other initiation of the instance variables happen in the *init* method.
- ***Tray Constructor #2:***
  public Tray(Tray parent, ArrayList<Integer> prevMove, Integer[] dimension, ArrayList<Block> newBlocks)
  The second constructor is used to create all the Tray objects following the *init* and *goal* Tray. Instead of taking in ArrayList of coordinates, this constructor takes in an ArrayList of Blocks. All other initiation of the instance variables happen in the *init* method.
- ***Tray init:***
  public void init(Tray parent, ArrayList<Integer> prevMove, Integer[] dimension, ArrayList<Block> newBlocks)
  Initialize all instance variables. The notable initiation is the calculation for *myPriority*. We calculated the heuristics by adding up the priorities of all the Blocks, which were generated by the distance between the particular Block and the closest Block of the same size. In addition, the ID caches the customized hash code to limited the call to calculation to once per instantiation of a Tray.
- ***public void reset()***
  To minimize the data stored during our search for puzzle solution, we reset all the visited Trays by setting all irrelevant pointers to null.
- ***public void forceUP()***
  During the process of creating children Trays in Solver.java, we want to speed up the process of search if a child Tray created is the goal Tray. By setting this particular child Tray's priority to 0, it will be the next Tray to be examined in the Priority Queue.
- ***public void addChildren(HashSet<Tray> child)***
  Set myChildren to a child.
- ***public boolean isEmpty(HashSet<ArrayList<Integer>> spaces)***
  Returns whether a list of coordinates are empty spots by checking against blocks.
- ***public int compareTo(Tray myTray)***

Compare Trays by comparing their priority.

- ***public int hashCode()***
  The hash code is calculated by keeping a sum while iterating through all the block's hash codes. During every iteration, the sum is updated as the current sum multiplied by 19 and added by the Block's hash code.
- **public boolean equals(Object o)**
  For Trays with only one type of goal Blocks, we iterate through all the blocks in the *o* Tray to see if the current Tray contains all the items. In this case, two Trays are equal if they have the same Blocks.
  For other cases, two Trays are equal if they have the same hash code.
- ***public boolean GoalEquals(Tray goal)***
  A tray is equal to Goal if it contains all the Blocks in Goal Tray.
- ***get methods***
  The following ***get*** methods returns the respective instance variables for easy access outside of the Tray object:
    - *getID, getPriority, getParent, getBlocks, getBlockSet, getBlockIDSet, getHashBySize, getHashByBlockID, getChildren, getMoveToGetHere.*

Class Solver:

Instance Variables:

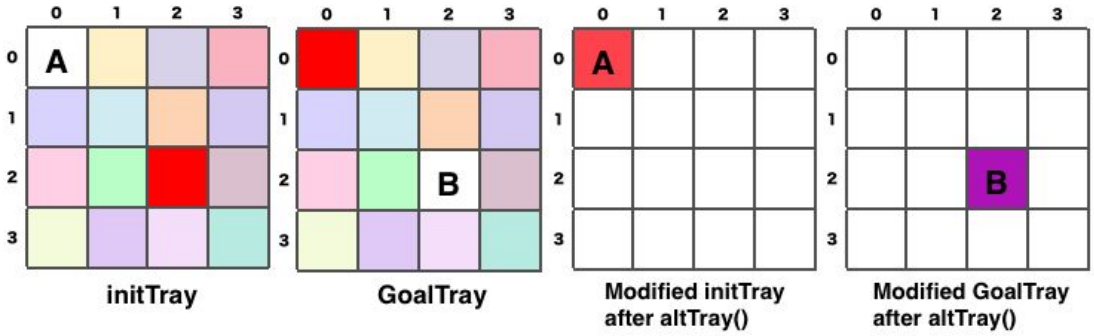
- **initTray** (Tray): the initial Tray
- **GoalTray** (Tray): the Goal Tray
- **BoardSize** (Integer[ ]): the dimension of the board
- **Visited** (HashSet<Tray>): a HashSet of visited Trays that are not the GoalTray
- **goalWHtoBlock** (HashMap<ArrayList<Integer>, LinkedList<Block>>): a HashMap mapping the (width, height) of goal Blocks, represented as an ArrayList<Integer> to a list of Goal Blocks with the same dimension.
- ***searchType*** (int): If a puzzle has only one type of Goal Blocks that are one by one in dimension and there are the same amount of blocks given in the initTray and GoalTray, this puzzle uses searchType 1, otherwise, it is searchType 2.

Methods:

- ***Solver Constructor:***
  public Solver(ArrayList<ArrayList<Integer>> init, ArrayList<ArrayList<Integer>> goal, Integer[] dimension)
  Creates the GoalTray and initTray first, instantiates all instance variables, then determines the searchType of the current puzzle.
- ***public Tray move()***
  This method conducts the main searching process. If this puzzle board is searchType1, it reconfigures the board by calling *altTray()* (see altTray). Next, it instantiates a PriorityQueue<Tray> *fringe* and iteratively *polls* Tray out of the fringe, adds it to visited, and adding all of its children into the fringe if they are not in visited or in the fringe. The process of creating children is elaborated in the *createMove* method described below. At the end of each search, if the Tray

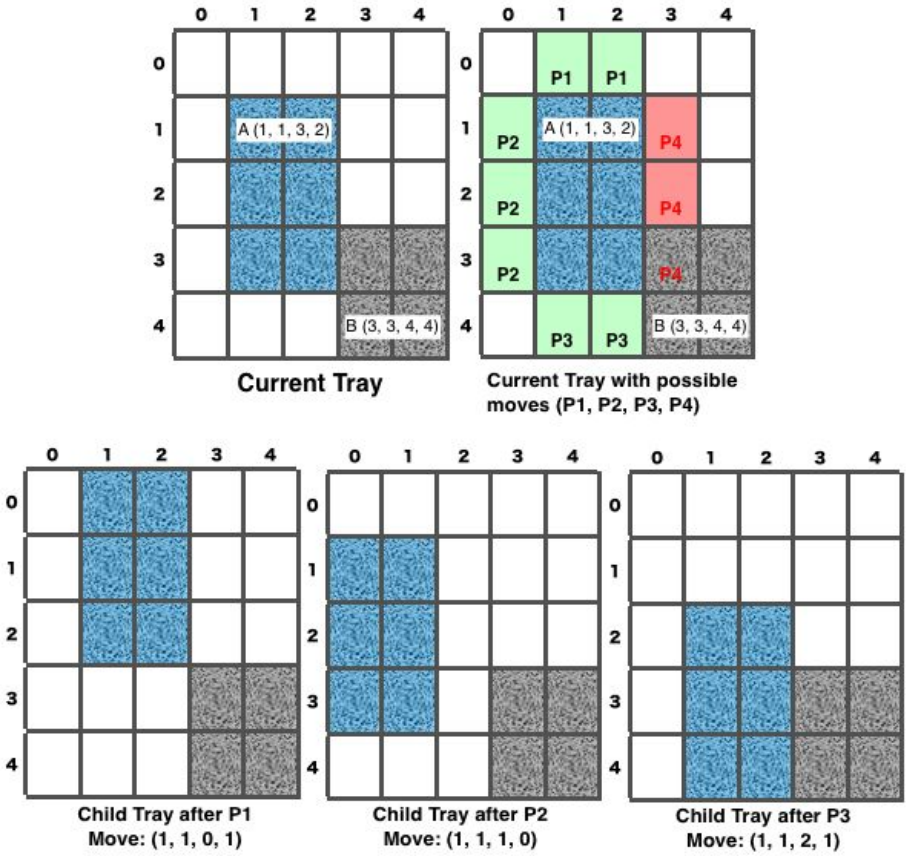
equals the GoalTray, this method returns the Tray, else it calls *reset* on the Tray to discard all unneeded data. If no solution is found, it returns *null*.

- ***public void altTray()***
  For searchType1 where all the goal Blocks are one by one in dimension and that there are exactly the same amount in both initTray and GoalTray, we can imagine that the only Block on the board is the empty spaces in the original Tray:



  This simplifies the problem by scale n, where n is the amount of Blocks we eliminated. In this new Tray, all we have to do is document A's movement to B and reverse the move (see *reverseMove)*. For example, in the modified version: moving A from (0, 0, 0, 0) to (1, 0, 1, 0) generates move-coordinates (0, 0, 1, 0). In the actual puzzle, however, the move is really from (1, 0, 1, 0) to (0, 0, 0, 0), which has a move-coordinates (1, 0, 0, 0). In order to print out the actual movements, just need to reverse the first two numbers of the modified move-coordinates with the second two numbers.

- ***public void createMoves(Tray myTray)***
  This method iterates through all the Blocks in current Tray and see if there any of them can move to a valid spot (a valid spot is a spot that is empty). Possible moves (*possibleMoves)* are generated by calling *getValidMoves* on the current Block during the iteration (see *getValidMoves*). This method then creates new Tray children using the result in possibleMoves by calling *moveBlock* and add them as children of current Tray.
- ***public HashSet<ArrayList<Integer>> getValidMoves(Tray myTray, Block b)***
  This method takes in current Tray and a Block b to be moved and checks b's surrounding coordinates for empty spots. *getValidMoves* checks whether moving up, down, left, and right are valid and returns a HashSet of the location of the Block after moving as an ArrayList of Integers.



- ***public ArrayList<Block> moveBlock(ArrayList<Block> mylist, ArrayList<Integer> origin, ArrayList<Integer> moveToGetHere)***
  moveBlock takes in a list of Blocks of the current Tray and replaces the Block to be moved at coordinates *origin* with a new Block representing the next move using the information from *moveToGetHere.* This method then returns new list of Blocks with the updated information regarding the next move.
- ***public void printPath()***
  This method prints the solution of the current puzzle by printing out the *moveToGetHere* coordinates starting at the current Tray and continuing with its parent until parent is null. If the puzzle is searchType1, it calls *reverseMove* to reverse the move-coordinates prior to printing out the move.
- ***public ArrayList<Integer> reverseMove(ArrayList<Integer> myMove)***
  This method takes in a move-coordinate and reverses the first two digits with the second two.
- ***public static void main(String[] args)***
  The main method parses the *init* and *goal* files and turns them into ArrayList<ArrayList<Integer>>. To get the solution of the current puzzle, it instantiates a Solver class and calls printPath().

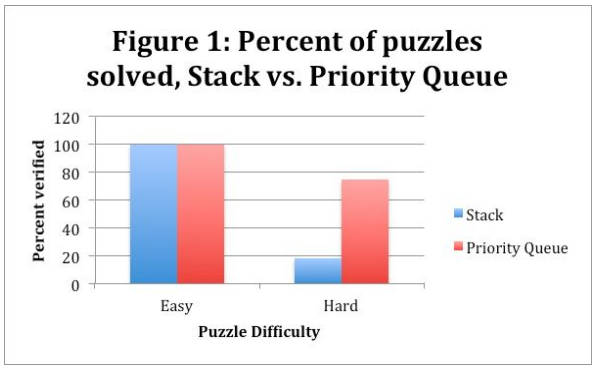
Experimental results

Since our program was expected to succeed under 80 seconds without taking up too much memory, we chose data structures that helped us balance memory and time efficiency. Throughout this project, we tried three major design choices: using a priority queue, using hash tables and resetting instance variables to null (in addition to using heuristics), and editing our heuristics and hashCode method. These experiments are not necessarily in chronological order.

The Priority Queue Approach

Our initial design relied on depth-first traversal using a stack of Trays, which allowed us to traverse through the graph and determine if each move fit the goal configuration. Since Stack and Queue have the same runtime, we decided to choose the stack to traverse through the possible moves. However, our puzzles could not meet the time constraints and took a significant amount of time when it was run in Eclipse. We decided that the best way to optimize our time efficiency was to use a Priority Queue to help us choose Trays that were likely to get us to the goal configuration faster. Since using the stack took far too long to time and most ended in an OutOfMemoryException, the success of the program was based on the percent of puzzles verified. As expected, our program was able to solve many more puzzles with a priority queue than with a stack.

Figure 1 shows us that the priority queue did not affect the success rate of the easy puzzles, though it did improve the time efficiency significantly. Before we added the priority queue the “hard/handout.config.2” puzzle was solved at an average time of 5848.4 ms, compared to an improved 722 ms after adding the priority queue (results not pictured). Although the priority queue did not help us solve all of the hard puzzles, we were able to improve the success rate four-fold.



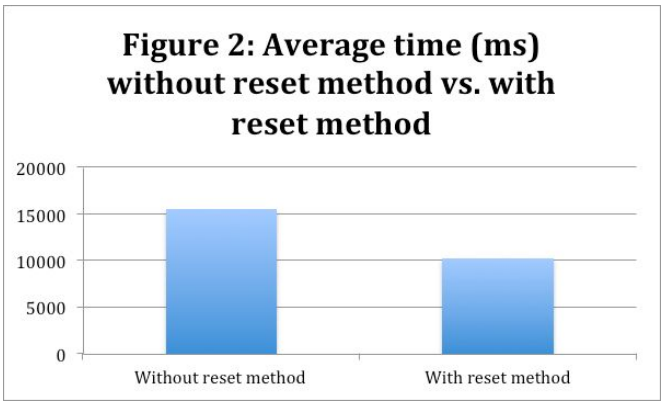
Although our primary reason for using a priority queue was to optimize time efficiency, the priority queue should also optimize the memory efficiency because we would be traversing through fewer moves before we get to the goal configuration. According to the success rates in Figure 1, we can see that the priority queue helps us omit unnecessary moves that are unlikely to generate the moves that will help us solve the puzzle. We used the distance between a block and its nearest possible goal position as our heuristic in the final program. We also went through some trials to test other

combinations of heuristics (see the “Editing Heuristics and hashCode” experiment). Overall, this experiment attests to the advantages of using a priority queue when we want to optimize our search through a data structure.

The Hash and Reset Approach

After realizing that our primary problem was running into *OutOfMemoryException* for our puzzles, we decided to improve our memory efficiency by using hash tables rather than ArrayLists to hold our information and by removing data structures that were no longer needed. To implement the second change, we wrote a reset method for the Tray class that sets all of its instance variables to null. This approach showed a dramatic improvement in memory and time efficiency because it eliminated the *OutOfMemoryException* from a few of our puzzles and our previous puzzles were solved in less time.

Before we removed the data structures, each of our Trays were holding a number of data structures, such as the boolean matrix, a reference to the Tray’s “parent,” and several ArrayLists. These data structures, multiplied by the number of moves that are required to get to the goal configuration, take up a significant portion of space on the heap. After changing ArrayLists to hash tables and using the reset method where it was necessary, we were able to reduce the CPU time by around five seconds. Figure 1 shows the average time that each method took to solve the puzzle “hard/handout.config.2,” with or without the reset method.



We decided that replacing everything with hash tables would optimize time efficiency because we would be able to check if the table contained the object and to get the object with ease. As we can see from Figure 1, our time efficiency showed significant improvement after eliminating unnecessary objects from the heap. Without the reset method, our program could solve the puzzle within an average of 15568 milliseconds. With the reset method, we could solve the puzzle at an average of 10226.5 ms. Additionally, we were able to solve two additional puzzles after all of our changes, which implies that we were able to eliminate the “OutOfMemoryException” for two puzzles. In the big picture, this experiment shows the advantages of using hash

tables and the importance of eliminating objects from the heap so that other objects can be made.

Editing Heuristics and hashCode

After we improved our program by cleaning up the heap, we decided to come up with a strategy to further optimize our memory efficiency. Since we were already minimizing the number of moves we were checking and cleaning up useless objects, we decided to tweak our heuristic and Tray's hashCode method. After trying different heuristics and ways to implement the hashCode method, we figured out the best combination that allowed us to pass all the hard puzzles. Distance was the best criteria for comparing priority, and 19 was the best multiplier to calculate the hash code.

Figure 3 shows the difference in success rates between the different heuristics we tried for our program. Distance refers to the distance between a block and its nearest possible goal position, whereas total distance refers to the sum of the distance from a block's position to its initial position and the distance between a block's position and its goal position. The number of moves refers to the number of moves that were made from the initial configuration to the current configuration. Percent verified is the success rate of the hard puzzles.

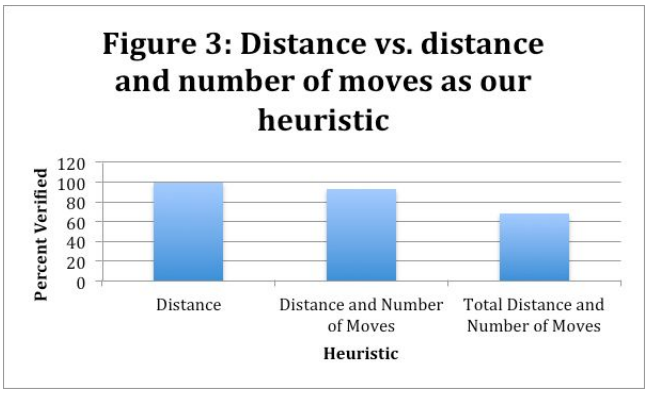
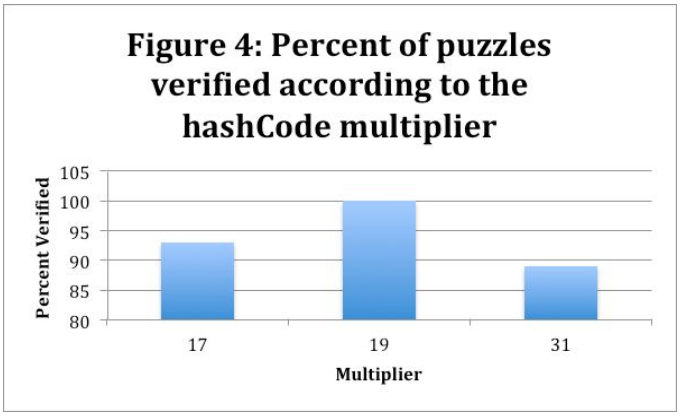


Figure 4 shows the success rate for different multipliers in the hashCode method of the Tray class. We needed a hashCode that would allow Solver to recognize that Trays with the same configuration were the Tray, but also prevent Trays with different configurations from having the same hash code. We ended up testing prime multipliers to prevent possible collisions. The results indicate that 19 is an optimal option for our program.



Editing the heuristics and hashCode method was the final step in completing our program. Figure 3 shows that the combination of more than one heuristic caused our program to fail some puzzles, possibly because the consideration of multiple factors increased the chance of collisions for some puzzles. Figure 4 demonstrates the importance of implementing a powerful hashCode method that minimizes the number of collisions so that the Solver does not go through the same Trays or omit Trays that could possibly lead to the goal configuration.

Program Development

Initially, we created a simple outline of the classes we would write, along with their instance variables and methods. Then, using this outline, we wrote up the basic framework of our three classes, Solver, Tray, and Block. We decided to focus on Tray and Block first, since those would be the building blocks of our program. We then came up with a means of determining whether a block could move to certain spaces or not, which led to the creation of our boolean[][] blocks variable in the Tray class, which represents an entire board. If the value at any position blocks[i][j] was false, that meant that it was an empty space on the board, otherwise, there was a block in that location. We used this to determine a Block's surroundings and to write the method, *getValidMoves*, which determines whether any Block in a Tray could move up, down, left, or right. We considered these moves to be the children of the current Tray and created a new variable in the Tray class, a HashSet<Tray> myChildren, that stored all of a Tray's possible moves. The hashcode we used to store the Trays in our HashSet<Tray> was dependent on that of the Block class. Since each ArrayList<Integer> is supposed to have its own unique hashcode, we decided that the hashcode for Block should be the hashcode of the ArrayList<Integer> of the Block's coordinates and that the hashcode for the Tray class should be the sum of the hashcodes of all of the Blocks in it. After creating this basic implementation, we proceeded to the Solver class, where we implemented the *move*() method. Initially, we just created a Stack<Tray> fringe, which took in the initial Tray configuration. This Tray would then be popped off and we would check if it was the goal configuration. If it was,

we would return it, otherwise we would add all of the Tray's children to the fringe and add the current Tray to the HashSet<Tray> visited to prevent us from looking at it again and causing an infinite loop. This appeared to work decently for most of the easy puzzles, but it was too slow for the medium and hard puzzles, meaning it needed to be optimized.

At this point, now that our algorithm worked, we had to figure out how to improve it and make it more efficient. We decided to implement a PriorityQueue<Tray> for the fringe, instead of the Stack<Tray>, where the priority was determined by a Tray's Blocks' distances to the goal configuration's Blocks. The Trays with smaller distances would have higher priority, so they would be popped off first. This would allow us to find the shortest path to the goal configuration faster and more efficiently. However, although this version of our program improved our project's runtime, it still wasn't passing all of the tests. To further improve our implementation, we created a *reset*() method that would set certain instance variables of a Tray object to null after the Tray had been analyzed and moved to the HashSet<Tray> visited. This would prevent our program from holding on to information it no longer needed and helped better our Solver's runtime for the harder puzzles. However, even after all of these modifications, we were still having issues with some of the hard puzzles. Upon inspection, we learned that our Solver was having a difficult time determining the solution to Trays of many one-by-one Blocks. As a result, we created a separate method to handle the case where the initial configuration was a large board full of one-by-one Blocks. By having a method that could handle this case specifically, we were able to get those series of hard puzzles to work quite quickly.

However, during this whole process we noticed an issue when our Solver tried to determine the solution to the 100 x 100 puzzle with multiple one-by-one Blocks. When we ran our program, our HashSet<Tray> visited did not contain as many items as we expected. After performing some tests, we determined that our hashcode function was the problem. There were several Trays that had the same hashcode, so whenever we checked if the HashSet<Tray> visited contained some new Tray, if there was already a Tray in it with the same hashcode, the new Tray would not be added, even if its configuration was completely different. This meant that the hashcode of the ArrayList<Integer> was not completely unique, so the hashcode of our Block class would have to be modified. We then decided that the best alternative would be to parse the four values in the ArrayList<Integer> of coordinates and make that the new hashcode. Additionally, we modified the hashcode of Tray such that it used a multiplier, further reducing the chance of collision. We ran the puzzles again and many of them worked, but now there was an different error. When the size of the Tray was greater than or equal to 100 x 100, the Block's hashcode function could not parse the value we were giving it into an Integer, because we were providing it with a value that contained more than ten digits. (The Java class, Integer only supports values up to 2^31 - 1.) As a result, we had to modify our hashcode once again. This time, we made two separate cases: one where the size of the board was less than 100 x 100 and the other where it was greater than or equal to 100 x 100. (Our program determined whether the size was greater than this bound by using the boolean BIG in the Block class. The size of the Tray was passed into the Block's constructor and if it was greater than or equal to 100 x

100, BIG was set to true.) If the size was less than 100 x 100, our hashcode would use the previous method (parsing the four values together). Otherwise, it would parse the first three values together, modifying those values by certain criteria, such as leading 0's if the value was less than 100, etc. As soon as this was adjusted, all of our tests ran successfully, verifying all of the easy, medium, and hard puzzles.

We decided to build the program in this sequence, because it made the most sense logically. In the beginning, we wanted to make sure we were approaching the problem correctly, so we didn't really focus on optimization. We primarily focused on writing a program that could solve at least the easy sliding block puzzles. After accomplishing this portion of the project, we started focusing on optimization and gradually made modifications to increase the speed and efficiency of our program. However, all the while, our algorithm remained unchanged. We simply added to it when necessary, and made certain other aspects more time and memory efficient.

We tested our program primarily by looking at some of the individual puzzles given in the easy, medium, and hard folders. We would run our program and see which puzzles didn't work; then using the debugger and the "run configurations" option, examine what exactly our program did with each of those puzzles individually. This allowed us to better understand what our code was doing and how to fix whatever problems we were having. Additionally, we used the function .*getCpuTime*() to determine the CPU time our program was taking for different puzzles. This gave us a good estimate of our program's time efficiency for different types of puzzles, allowing us to better optimize our project.

Disclaimers

Our hashcode function for the Block class, and thus the Tray class, was not entirely bulletproof. While creating our algorithm and writing code, we found that for Trays with dimensions greater than or equal to 100 x 100, our hashcode had some issues. At this point, we had decided to use the coordinates of each Block concatenated into a String, which was then parsed into an Integer to represent the hashcode. However, this became a problem when the Solver tried to solve puzzles that were of dimensions 100 x 100 or larger, because Integers cannot hold values where the total number of digits is greater than 10. (The maximum value for an Integer in Java is 2^31 - 1.) Because of this, we made a separate hashcode for this specific case which also depended on the coordinates, but rather than parse a concatenated String of all four values in the ArrayList<Integer> of coordinates, it parsed the String of the first three values, such that each of the values fulfills certain criteria, such as leading 0's for values less than 100, etc. This appears to work with all of the test cases given, but there is chance of collision, especially as the Trays get larger. (Tray's hashcode function is dependent on that of Block.)

Improvements

If we were to make an improvement to speed up our program, we would come up with a better, more generic, fool-proof hashcode. At the moment, our hashcode still appears to have some collisions, but we would ideally like to fix this such that each Tray, unless it has the exact same dimensions and the exact same Block positions, would have a different, unique hashcode. This should allow our program to speed up significantly, because when comparing Blocks and Trays, we would only need to compare hashcodes (Integers), rather than the objects themselves, which would greatly reduce the amount of memory taken by running the program and the amount of time needed for the program to solve the puzzle. Evidence of this was gathered from running different implementations of our code. Previously, for our Block class' .*equals* method, we compared the ArrayList<Integer> of coordinates themselves and the program appeared to take longer. However, when we started comparing hashcodes, the program was able to determine a solution for the puzzle (if there was one) much faster than before. We encountered the same scenario with the Tray class as well. In addition, in our current implementation, we have two cases for our hashcode: one for Trays with dimensions greater than or equal to 100 x 100 and the other for Trays with dimensions less than 100 x 100. Ideally, we would have only one version of calculating the hashcode for a Tray, no matter what its size, which would clean up our code.