CS61BL Project 3: Solver Readme

Section 110, Doug Wreden

Project Members:

- Valerie Cook `cs61bl-ne`
- Jennifer Dai `cs61bl-nd`
- Yangzi He `cs61bl-nc`
- Daniel Radding `cs61bl-ns`

Division of Labor

For this project, Valerie and Jennifer worked more closely together on `Solver.java`, while Daniel and Yangzi did the same for `Tray.java`. Daniel and Yangzi wrote an initial Tray file that used a 2D array as its data structure along with multiple methods; however, we later decided that instead of just having a **tray** object, we also wanted **block** objects to place into a **tray**, and our 2D array implementation made it difficult to distinguish between distinct **block** objects. Valerie reimplemented Tray using an `ArrayList`, and wrote a new `Block` class.

We decided to split the work in this manner because `Solver` and `Tray` are the two most important files that need to be completed in order to get the project up and running. `Solver`'s logic is needed in order for us to be able to tell blocks how to move on the tray, such that the block configuration will eventually be the same as the goal configuration. `Tray` was necessary because without deciding on what data structures we were going to use for both the tray and the blocks, `Solver` could only be written using pseudo-code.

Design

Solver

Data Structures

Solver uses two Data Structures we created to create abstraction. The first is a `BlockCollection` object called `myGoals`. `BlockCollection` objects hold `Block` objects and `myGoals` specifically holds `BlockCollection` that represent the solution to the puzzle. The second structure is an `AlreadySeenCollection` named `alreadySeen`. `AlreadySeenCollection` objects store `BlockCollection` objects the have previously been tried as a possible a solution. Finally we also use a Graph structure to find actually solutions to the puzzle. We use `node` objects we wrote ourselves.

Methods

`Constructor`

The Constructor takes in two arguments. One is a `Tray` object named `tray` and the other is a "BlockCollection `object named` goals `. Constructor are sets five of the instance variables. The first is a` Tray `object named` myTray `which is set the to value of the argument` tray. The next is a `Block Collection` object named `myGoals` and is set to the value of the argument `goals`. This is followed by a `Node` object named `myRoot` which set to equal a `Node` with `null` values. The next variable is a also a `Node` object and is called `currentNode`, this `Node` is set to equal `myRoot`. Lastly is an `AlreadySeenCollection` object named `alreadySeen` is initialized.

`MatchesGoal`

This method returns a boolean that checks if the current "BlockCollection" matches the solution to the puzzle.

`addPossibleMoves`

This is a `void` method that calls the `Tray` method `getPossibleMoves` and to fill an `ArrayList` named `moves`. The method then loops through `moves` to create a `node` for each `Move` object and then add it to the children of the `currentNode`.

`makeMove`

This method actually starts the motions of making a move. The method checks to see if the `currentNode` has a `Move` stored in `myMove`. If it doesn't then it returns `false`. If there is a `Move` object then it calls `makeMove`. From there the method checks to see if the new `BlockCollection` has been seen before. If it has then the `Move` is undone by calling `undoMove` and the method returns `false`. If the `Move` hasn't been scene then the `BlockCollection` is added to `alreadySeen` and the method returns `true`.

`getNextNode`

This `void` method finds the next `Node` to be tried. If `currentNode` does not have children then `undoMove` is called. Otherwise `currentNode` is set to the first `Node` stored in `myChildren` of the `currentNode`.

`undoMove`

This `void` method undoes a `move`. It first looks at the `currentNode` and reverses the `myDir` value of its `myMove` variable. It then calls the `Tray` `makeMove` method to move to the most recently moved `Block` back to its previous spot. After that it resets `currentNode` to the

`myParent` value of the `currentNode`. It then removes the first value in `myChildren` of the `currentMove` so that `Move` is not made again.

`findSolution`

The method `findSolution` is a `void` method. To start, the boolean `solved` is set to equal the result of a call of the method `matchGoal`. If `solved` is `true` then solved is immediately return. Otherwise if `solved` equals `false` then the current `BlockCollection` is added to `alreadySeen`. Then there is a call to `addPossibleMoves` followed by a call to `getNextNode`. Next there is a `while loop` that runs while `currentNode` does not equal `myRoot` and `solved` is `false`. Inside is an `if` statement that calls `getNextNode` if `makeMove` is `false`. Otherwise there is an `else` statement. Inside the `else` statement in `if` statement that sets `solved` to `true` if a call to `matchesGoal` return true. Otherwise there is an `else` statement that makes calls to `addPossibleMoves` and `getNextNode`. The `while Loop` then finishes.

`printSolution`

This method is a `void` method that only works is the `boolean` `solved` is `true`. `currentNode` is then set to equal `myRoot`. The method then uses a `while loop` that runs while the `currentNode` has children `. Inside the loop` currentNode `is set to equal the child at the 0th index of` myChildren `. Then the` myMove `variable of the` currentNode` is printed.

`Node`

The `Node` class creates the objects we use for the vertices of the graph used to find the solution to the puzzle. `Nodes` have three instances variables: `myMove` which stores a `Move` object, `myParent` which stores a `Node` object, and `myChildren` which is an `arrayList` that stores `Node` objects. The `Node` class also has three methods:

`Constructors`

The `Node` class has two constructors. One takes no arguments and sets `myMove` and `myParent` to `null` and initializes `myChildren`. The other constructor which overloads the first constructor takes in two arguments, a `Move` object named `m` and a `Node` object named `previousMove`. `myMove` is set to `m` and `myParent` is set to `previousMove.` myChildren` is then initialized.

`getParent`

This method returns the value of `myParent`

`getMove`

This method returns the value of `myMove`.

`main`

The Solver class also holds the `main` method for our program. The method first creates a `SolverChecker` object named `checker` with the `String array` `args` as the provided argument. `main` then creates a `Tray` object named `tray` with `length`, `width`, and `blocks` values of `checker` as the arguments. Then a `try` `block` is opened. Inside there is a call to the `findSolution` method of `solver` which is the followed by a call to the `printSolution` method of `solver`. If an `exception` is thrown by any method called the `exception` is caught in the `catch` `block` and the result of calling the `getMessage` method of the `exception` is printed.

Block

Data Structures

A 'block' object represents any block that can legally be put into the tray. It has four instance variables: `myUpperRow`, `myLowerRow`, `myUpperColumn`, and `myLowerColumn`, all of which are the primitive type `int`.

Methods

`Constructor`

The constructor takes in four arguments, which are `upperRow`, `upperColumn`, `lowerRow`, and `lowerColumn`, and sets the instance variables of the object accordingly.

`length`

A method that calculates and returns the length of the block by subtracting `myUpperRow` from `myLowerRow`, and then adding one.

`width`

A method that calculates and returns the width of the block by subtracting `myUpperColumn` from `myLowerColumn`, and then adding one.

`getters`

Four methods that each return one of the block's instance variables, which have the `private` access modifier before them.

`setters`

Four methods that each lather one of the block's instance variables, based on the first `int`

argument, `coord`. The second argument is an `int` named `limit`, which is passed either the max length or width of the Tray in which the setters are called, in order to make sure that the instance variables of block objects don't go out of bounds.

`isValid`

Method that is called by the setters to determine whether or not the desired change to a block's instance variables is consistent with the dimensions of the tray. Returns a boolean.

`clone`

Makes a copy of a block and returns it.

`equals`

Overrides Object's equals method. The method takes in an argument of type `Object`, casts it to a `Block`, and determines equality by checking whether or not the instance variables are equal.

`hashCode`

Returns an `int`, which is the hashCode for a block object. The hashCode is calculated by multiplying the length and width of the block together.

`setDebugLevel`

This 'void' method takes an 'int' argument and sets the debug options based on the 'int' value provided.

`startTimer`

This 'void' method takes in a DebugEvent object as an argument and then starts the timer for a specific event.

`stopTimer`

This 'void' method takes in a DebugEvent object as an argument and then stops the timer for a specific event.

`printDebugResults`

This 'void' method prints out the results of the previously specified debugging options.

`actOnInvalidInput`

This 'void' method catches incorrect user input, prints a useful message, then exits.

`debuggingOptions`

This 'void' method Changes our boolean flags to true when we call our debugging options and makes sure our debugging options do what they're supposed to.

`printOptions`

This 'void' method prints the options when -ooptions is called.

Space

Data Structures

A `space` object represents an empty cell in a tray. Similar to `Block`, it also has `int` instance variables: `myRow` and `myColumn`. Unlike a block, the only size that a space can be is a 1 x 1 cell.

Methods

`Constructor`

Takes in two `int` arguments, row and column, and sets `myRow` and `myColumn` equal to them respectively.

`getPosition`

A method that creates and returns an `int []` named `rtn` of size 2. The first element is `myRow`, while the second is `myColumn`. This is necessary because the instance variables are set to `private`.

`equals`

This method overrides `Object`'s equals method. It determines equality by checking whether or not the instance variables are equal between two space objects.

Move

Data Structures

`Move` objects are used to create and store possible moves for a `Block`. The move object stores a block and the direction that it could potentially move in.

Methods

`Constructor`

The Constructor takes in two arguments, a `Block` object `b` and a `String` object `dir` which represents a direction the `Block` might move. The Constructor sets six different instance variables. The first two are a `Block` object named `myBlock` and a `String` object named `myDir`. `myBlock` is set equal to `b` while `myDir` is set equal to `dir`. The next two variables are `ints` named `OldUpperRow` and `OldUpperColumn`. These two variables store the original `UpperRow` and `UpperColumn` values of the given `Block` object. The last two variables are `int` values also and are named `newUpperRow` and `newUpperColumn`. These two variables are set dependent on the value of `dir` which can equal "up", "down", "left", or "right". Through four different if statements `newUpperRow` and `newUpperColumn` are set to values the would move the given block in the direction of `dir`.

`toString`

This method returns the `String` representation of a `Move` object. The `String` returned is the `String` representation of the following four values each separated by a space: `oldUpperRow`, `oldUpperColumn`, `newUpperRow`, and `newUpperColumn`.

Tray

Data Structures

Tray is not actually implemented with it's own separate data structure; instead it relies `myBlocks` and `myEmptySpaces` instance variables, which are a `BlockCollection` and a `SpaceCollection` respectively, to tell it where everything is located. In other words, a `Tray` object cannot be physically represented independently of the `block`s and `space`s. It is initialized with `length` and `width` instance variables, and a `BlockCollection`, which represents the current configuration of the blocks on the tray.

Methods

`Constructor`

It takes in three arguments: a length, a width, and a `BlockCollection`. The `BlockCollection` is set to `myBlocks`, and length and width are stored as the instance variables `myLength` and `myWidth`, which are referenced by other `Tray` methods to make sure that the blocks and spaces created are legal.

`findEmptySpaces`

Within `Tray`'s constructor is the method `findEmptySpaces()`, which given the initial `BlockCollection`, creates `space` objects for all the cells in the tray that aren't occupied by a `block`, and stores everything in `myEmptySpaces`. `findEmptySpaces()` works by filling

`myEmptySpaces` with every possible 1 x 1 cell in the tray. Then, it iterates through `myBlocks`, and for each block, it deletes the `space` objects that correspond with the position of the block by using a nested for loop.

`getMyBlocks`

A getter for the blocks that are placed into the tray. It returns an `ArrayList<Block>` object.

`addBlock`

A method that takes in a 'Block' b as input and add its to the end of `myBlocks`.

`getPossibleMoves`

This method is called when the user wants to get a collection of all the valid moves given a specific configuration of blocks in the tray. It returns an `ArrayList` of `Move` objects. The method creates a local variable called `possibleMoves` of type `ArrayList<Move>`. It then iterates through `myBlocks`, and for each block in the `ArrayList`, it checks whether or not it can move in the up, down, left, and right directions by calling `canMove`. If `canMove` returns true for a specific direction, then a new `move` object is created with the corresponding `block` as the first argument, and the direction of the valid move as the second argument. The `move` object is then added to `possibleMoves`. When the `for` loop concludes, it returns `possibleMoves`.

`canMove`

`canMove` takes in two arguments: a `block` and a direction, which is passed in as a `string` object. It returns `true` if it is possible for the block to move one cell in the indicated direction, and false otherwise. It does this by checking that the cell necessary for the move are in `myEmptySpaces`, as this indicates that the cell is actually empty, and not being occupied by another block. For `up` and `down`, this involves making sure that the row *(either immediately above or below the block depending on direction)* starting from the `block`'s upper column number with a length equal to the `block`'s length is within `myEmptySpaces`. The same process is used for `left` and `right` directions, but instead, the columns immediately to the `left` and `right` of the blocks are checked, and the columns are determined based on the width of the block rather than the length.

`makeMove`

This method also takes in a `block` and `string` argument called direction. Depending on the direction, it either changes the upper and lower rows of a block or the upper *(left)* and lower *(right)* columns by one. It then calls `findEmptySpaces()` to determine where new `space` objects are.

Debugging

Debugging options

Our SolverChecker class contains all the debugging options. To get all the debugging options that you can type in, just type in –ooptions. Otherwise, there are different levels of tray and solver debugging options that you can enable, basically denoted as –o(insert level)(insert t or s or both depending on which options you want to enable). The levels are 1 for counts, 2 for times, 3 for times and counts, and 4 for output about every move. Below is a comprehensive list of what options are enabled.

- o1t: counts of important events in Tray
- o2t: the times of important events in Tray
- o3t:both the counts and the times of important events in Tray
- o4t:both the counts and times of important events in Tray and the blocks, spaces and possible moves for each call to getPossibleMoves
- o1s:the counts of important events in Solver
- o2s: the times of important events in Solver
- o3s: both the counts and the times of important events in Solver
- o4s: both the counts and the times of important events in Solver and the successful and unsuccessful moves that Solver makes
- o1ts or -o1st: the counts of important events in both Tray and Solver
- o2ts or -o2st:the times of important events in both Tray and Solver
- o3ts or -o3st:both the counts and the times of important events in both Tray and Solver
- o4ts or -o4s: both the counts and the times of important events in both Tray and Solver and possible moves for each call to getPossibleMoves and the successful and unsuccessful moves that Solver makes

isOK

Our isOK method checks to make sure that each block does not have dimensions that are impossible relative to each other, for example, if the upper column is greater than the lower column. It also checks to make sure that there are less empty spaces than there are total spaces in the entire tray. It then checks to make sure that there are no overlapping blocks and that the block is not out of bounds. To be complete, there is also a check to see if any block spaces are in empty spaces. The isOK method was somewhat useful in checking for program bugs when we were first writing the program because we realized that we had done our upper row, upper column, lower column, and lower row inconsistently from one another. However, the check is really slow and therefore makes running the hard puzzles a lot longer. It is also not suggested that you run level 4 debugging options with any hard puzzles unless you're using breakpoints. This is because there is so much output that it gets lost and stops being useful. Level 4 was mainly used to make sure the program's methods were working properly and getting the correct possible moves given the tray's current blocks and empty spaces.

Evaluating Tradeoffs

Experiment 1: HashSets

Summary

When we first began the coding we used arrayLists to store just about everything. It was a quick and dirty way to store objects while we focused on getting our code to actually solving puzzles. However once we got a fucntioning program we needed to find ways to optimize. Something that was immediately obvious was replacing the arrayLists with `HashSet`s where possible. The arrayLists were clearly slowing down the program and we knew, with an efficient hashCode, we could probably make the biggest difference in speed with this change. Specifically we changed `AlreadySeenCollection`, which stores all of the previous `BlockCollection`s, to use a HashSet. The results were dramatic. We went from solving only through about half the Medium puzzles to solving all of the Medium puzzles except one and many of the Hards.

Methods

Using our debugging options we were able to see real differences in a the amount of time it took to run any given test. The relevant test from out debugging options is a test called `checkAlreadySeen` which times how long it takes to see if a `BlockCollection` is a the `AlreadySeenCollection` object's collection of previous `BlockCOllection`s. We recorded the time it took our program to solve puzzles with and without HashSets implemented. Across the board we saw dramatic improvements.

Results

The follwing is a sample of the differences in times for various tests on a Laptop with 4 GB of RAM and a 1.7GHz processor. Below is a chart of that shows the test we ran as well as the time our previously mentioned `checkAlreadySeen` debugging option recorded when `AlreadySeenCollection` used an arrayList versus a HashSet.

Test Name	ArrayList	HashSet
big.tray.2	right-aligned	$1600
140x140	4ms	1ms
c38	531960	3149ms
big.tray.4	630033 ms	1384 ms
supercompo	5209 ms	85 ms
c54	944 ms	51ms

Conclusion

The reason why a Hashset was superior to an ArrayList comes down to the fundamentals of the `contains` method for both Data Structures. The `contains` method for an `ArrayList` has to iterate through every element to look for a match which takes `O(N)` time where `N` is the number elements in the arrayList. This can work okay for a small number of elements but quickly becomes slow and cumbersome with a large number of elements. The `contains` method for a `HashSet` on the other hand is more or less efficient as possible if the objects it is storing has a quality hashCode function. In fact the `contains` method for a HashSet can effectively search for an object in constant time. This is because a HashSet stores and searches for objects by the index of the object's compressed HashCode which can be done in constant time. If the object's HashCode collides with the hashCode of other object's then the HashSet must iterate through a `LinkedList` of objects at the given index. However if the HashCode function is efficient then these collisions happen so infrequently that any LinkedLists are so small the time to iterate through them is negligible.

While the difference in time in the easy tests was often small we saw huge differences in the medium and hard tests. Some of the largest puzzles can generate thousands of **BlockCollection**. Having to iterate through thousands of elements to check for a match is simply unacceptable and we were very pleased with the result of this experiment.

Experiment 2: Hash codes

Summary:

In order to speed up our program, we noticed that making a move was taking a longer time than we would have liked. Specifically, the check to `alreadySeen` was taking up almost half the total time to find the solution to the puzzle. Therefore, we started to experiment with our hashCode. As a result, our program went from failing 13 of the medium puzzles to passing all of them and passing 11 more of the hard ones than it had before.

Methods:

At first, we started with our block collection as an `ArrayList`, so we just called the built-in arrayList hashCodes for `BlockCollection` and `Block`. We realized that all the blocks were being given the same hashCode because the arrayList hashCode was just using the Block's position in the arrayList, which we weren't changing because we were simply setting the new values of block's corners when we moved it. We decided we needed to use the block's position on the tray (it's upper and lower column) for our new hashCode, so that it would change when we moved the blocks. This changed proved to be very effective and was taking almost no time. Unfortunately, we started running it against some larger puzzles and realized that we weren't catching all of the repeat trays. This is because the hashCode used by arrayList was using the block's position in the array to determine the hashCode. This meant that if two blocks of the

same size had switched locations on the board, we gave them separate hashCodes. To fix this we set up the block collection hashCode to add up hashCodes of each block in the block collection. Then we started to optimize our block hashCode.

We changed the hash codes in our `Block` and `BlockCollection` classes to see if we could make our program faster because we were trying to optimize when we checked our `alreadySeenCollection`, which is a class that has a `HashSet` of `block` collections.

To optimize our block hashcode, we started with (`myUpperRow` + 1) + 31 * (`myLowerColumn` + 1). We timed a couple of different puzzles with this `hashCode` a couple of times just to make sure that we got a good indication of time. All units are in milliseconds.

C22	Trial 1	Trial 2	Trial 3
Making a move	6216	6087	6109
Adding possible moves	364	359	299
Checking already seen trays	3669	3507	3587
Finding a solution	6682	6528	6506
Lane rouge	Trial 1	Trial 2	Trial 3
Making a move	326	424	344
Adding possible moves	121	102	86
Checking already seen trays	184	251	173
Finding a solution	496	617	478
Big.tray.3	Trial 1	Trial 2	Trial 3
Making a move	225	220	211
Adding possible moves	1401	1347	1301
Checking already seen trays	3	4	1
Finding a solution	1627	1575	1515

We then changed our hashcode to (int)(myUpperRow * 1.5) + 31 * (int)(myLowerColumn * 2.34) + (int)(width() * 3.13) + (int)(length() * 9.087129); because we realized that we had too many collisions, and this hashcode helped us to solve the rest of the medium puzzles that we weren't solving.

C22	Trial 1	Trial 2	Trial 3
Making a move	2709	2604	2589
Adding possible moves	335	282	275
Checking already seen trays	1581	1478	1417
Finding a solution	3128	2982	2955
Lane rouge	Trial 1	Trial 2	Trial 3
Making a move	188	246	200
Adding possible moves	140	70	143
Checking already seen trays	100	100	95
Finding a solution	368	370	395
Big.tray.3	Trial 1	Trial 2	Trial 3
Making a move	220	230	197
Adding possible moves	1249	1306	1262
Checking already seen trays	0	5	2
Finding a solution	1471	1539	1466

We then tried making sure our hashcode changed based on how large our board was. if (myLowerRow < 16 && myUpperColumn < 7)⇒ (int) Math.pow(2, myLowerRow) + (int) Math.pow(31, myUpperColumn); Otherwise, (int)(myUpperRow * 1.5) + 31 * (int) (myLowerColumn * 2.34) + (int)(width() * 3.13)+(int)(length() * 9.087129);

C22	Trial 1	Trial 2	Trial 3
Making a move	1195	1418	1265
Adding possible moves	351	271	328
Checking already seen trays	738	785	706
Finding a solution	1641	1803	1679
Lane rouge	Trial 1	Trial 2	Trial 3
Making a move	204	164	161
Adding possible moves	19	150	96
Checking already seen trays	69	68	66
Finding a solution	341	372	344
Big.tray.3	Trial 1	Trial 2	Trial 3
Making a move	211	202	233
Adding possible moves	1269	1289	1314
Checking already seen trays	1	2	1
Finding a solution	1481	1491	1551

Conclusion

We realized that in order to get the least amount of collisions, we should use unique numbers to multiply by the dimensions of our block so that we could spread it out as far as possible. Also, the multiplying by 31 is important because the block dimensions are so close together that they would just immediately collide otherwise. We also realized that if our tray is smaller, we can get better results by using a different hashcode than when the tray is large. For example, we use powers when we have smaller trays. With bigger trays, using powers would take up a lot more

memory and make our program a lot slower.

Experiment 3: Data Structure Change

Note: Shu Zhong said that this experiment was okay for the required data structure experiment in Tray.

Summary:

We tested a different implementation of our nodes in our tree to see if by storing a tray, we could get our program to run faster. As a result, it does run faster but runs out of memory before it can finish the puzzles for the harder puzzles that require more moves.

Methods:

The way we implemented our project was to look through a tree of possible moves. Each `node` in the tree contained a `move` object, a `parent` node, and an `arraylist` of children nodes. When we traversed our tree, we changed the `BlockCollection` that was stored within our tray object by moving the blocks in that collection around. We decided to change the way that was implemented by concentrating on time efficiency rather than space efficiency. Since whenever we make or undo a move, we have to move the blocks within the `BlockCollection`, we decided to store all the possible children's trays so that when we undo a move, we just have to go to the parent and get the tray from the parent node and repeatedly try moves on the next children's trays. This would in theory save time because we wouldn't have to undo the move, we would just already have that information stored. The first table is data collected where we create new trays every time. This first algorithm fails for tests such as c31 and for most of all the hard puzzles because it runs out of memory. The following units are all in milliseconds.

Dads+180	Trial 1	Trial 2	Trial 3
MakeMoves	50	39	78
getNextNode	5	4	4
findSolution	450	471	452
Handout.config.1+90	Trial 1	Trial 2	Trial 3
MakeMoves	36	39	76
getNextNode	3	4	4
findSolution	323	310	327
blockado	Trial 1	Trial 2	Trial 3
MakeMoves	5	4	7
getNextNode	0	0	0
findSolution	63	42	41

This following table is the data collected from when we simply undo our moves.

Dads+180	Trial 1	Trial 2	Trial 3
MakeMoves	36	27	25
getNextNode	3	5	7
findSolution	76	80	80
Handout.config.1+90	Trial 1	Trial 2	Trial 3
MakeMoves	225	189	179
getNextNode	28	37	38
findSolution	394	237	399
blockado	Trial 1	Trial 2	Trial 3
MakeMoves	120	115	139
getNextNode	31	25	33
findSolution	264	264	265

Conclusion:

In order for this to work, we would have to have a much better way to check getPossibleMoves that was optimized so that we wouldn't be checking as many moves. This way, it would be less likely to run out of memory. However, it does run faster in some cases where it doesn't run out of memory. For some puzzles, the changing the block collections within tray causes it to run a lot faster while for other puzzles, creating new trays every time you want to store a new move causes it to run faster. However, creating new trays every time causes us to run out of memory a lot faster.

Time/Space tradeoffs

Q:

The program will generate moves possible from a given configuration. This will involve examination either of the blocks in the tray or of the empty space in the tray. Should the tray be stored as a list of blocks/empty spaces to optimize move generation, or should the locations in the tray be represented explicitly? If the former, should blocks/spaces in the list be sorted?

A:

We opted to use `Block` objects and `Spaces` separately to represent both blocks and spaces. Each type of object held its coordinates and then was stored in the `ArrayList` of its own object type. The Blocks were not stored in any particular order other than the order they were parsed in.

Q:

Prior to each move, the program must check whether the desired configuration has been achieved. What tray representation optimizes this operation? If this representation is incompatible with implementations that optimize move generation, how should the conflict be

resolved?

A:

We used the class `SolverChecker` to check whether or not the current `BlockCollection` contained a match to the goal configuration. We were able to optimize quite easily throughout the project because we created a lot of abstractions so method names never changed even if the code behind them did.

Q:

Once it has a collection of possible next moves, the program will choose one to examine next. Should the tree of possible move sequences be processed depth first, breadth first, or some other way?

A:

We pursue solutions depth first in hopes of finding a solution quickly. We believe this is faster because going breadth first would require us to make a lot of moves without getting very deep into subtrees which is very inefficient.

Q:

Should block moves of more than one space be considered? Why or why not?

A:

They should not be considered because if we start considering moves of more than one space, it will take more time to find all the possible moves since each direction possibly has three or four moves each depending on where the empty spaces are. Also, it would take more space to store the moves into our collections, and thus, we decided that the time saved from possibly moving a block more than one space in the correct direction is not worth the time and space spent to figure out and keep track of all additional possible moves.

Q:

The program needs to make and unmake moves. Again, a representation that optimizes these operations may not be so good for others. Determine how to evaluate tradeoffs among representations.

A:

We determined that evaluating tradeoffs should be done by looking at how many times methods key to solving the puzzles were called, and the total time taken to execute those methods, as well as the total time of the puzzle. Thus, whenever we tried a different representation, we looked at which parts of the program sped up, which ones slowed down, and whether or not

the total time of each puzzle was faster or slower.

Q:

The program must detect configurations that have previously been seen in order to avoid infinite cycling. Hashing is a good technique to apply here. What's a good hash function for configurations? The default limits for Java memory allocation may limit the maximum number of configurations that the table can contain. How can this constraint be accommodated, and what effect does it have on other operations?

A:

See Experiment 2: HashCodes

Disclaimers

Problems

There were four tests that we did not pass, `pandemonium` and the three tests in `big.tray.4`. In both cases we ran out of memory due to the large size of these boards in combination of a large number of blocks. We believe this is because our program doesn't make smart enough decisions when deciding the next move so we waste time and memory going down bad subtrees as we look for possible answers.

Possible Improvements

A possible improvement could be using the Manhattan distance of goal block to its goal position. We could have focused on moving the goal block along this line and moving any block that is in the way of its path. We believe this could have prevented us from going down bad and unnecessary subtrees by helping us focus on which blocks were the most relevant to move. We believe this would not only significantly speed up our ability to solve puzzles, but also solve some of our memory issues because we would be creating fewer `BlockCollection` objects.

Program Development

The first thing we did was write `main` in `Solver` in pseudo code to get an idea of how we were going to run our program and which tasks would be done in which classes. We then decided to have the tray class keep track of the entire tray, which includes the tray's collection of block and empty spaces, and the length and width of the tray. And we chose to have the `solver` class keep track of the goals, the current tray, the already seen trays, and the tree of possible moves.

Once we had a general idea of how both the solver and tray class worked, we started working on the methods each of the classes would require. Since tray keeps track of the tray, we wrote headers for methods to find moves, make moves, and find empty spaces. And solver keeps track of finding the solution, so we wrote headers for methods to see if the current tray matches

the board, to add all the possible moves to the tree, to find the solution, and to print the solution.

Then, we wrote each method in pseudo code and made new headers for methods that would help our current methods, like a can move method in tray that retuned true if a block could move in a given direction, and a make and undo move in solver, which adjusts the tree when we make or undo a move.

Finally, once everything was filled out in pseudo code and we mostly understood what each method needed to do, we added classes like `Block`, `Move`, `Space`, `BlockCollection`, `SpaceCollection`, and `AlreadySeenCollection`. We decided to make all of these things separate classes so that we would be able to modify them easily without affecting the rest of our program.

Then, we went back to `Solver` and `Tray` and started writing in actual code, adding small methods in the different classes as necessary (like getters and setters and equals methods for the `Block`, `Move`, `Space`, `BlockCollection`, `SpaceCollection`, and `AlreadySeenCollection`). As we were writing the code, our primary focus was on getting a working program, and not worrying too much about the efficiency.

We built the program in this sequence because we wanted to make sure we understood the big picture of how everything would work first, and then build accordingly. We also wanted to make sure it was set up in a way that would make changing certain aspects fairly simple and not too destructive to the rest of the code. And we decided to get a working program together first, so that we could then just use the debugging options we had to write to find where the program was slowest, and modify it accordingly.

For test cases, we tried to consider all the edge cases and unexpected behavior or input that could cause our program to stop or break.

- For the `Solver` class, we tested that when adding possible moves, you can't add a duplicate move to the tray. Also, when we get the next node and it's corresponding move, we can't retrieve a move that is in the `alreadySeen` collection. We tested to make sure that our program exits correctly; if there are no more possible moves then it exits with `System.exit(1)`. However, if there is a solution, then it will exit normally and print out the moves taken to get to the solution.
- In the `Block` class, we tested the constructor and confirmed that it was making the blocks with the dimensions that we input into the constructor. We also had tests for setting and changing the length and width of the block objects. We ensured that our `clone` method made a deep clone instead of a shallow clone, because whenever we make and store a `BlockCollection` inside `alreadySeen`, we need to clone all the blocks currently in the tray. We had a test for the `equals` method so that when searching for blocks inside different types of collections, we would be able to retrieve the correct block. We had tests to make sure that the instance variables of block objects

made sense *(no negatives and lower row and column should always be bigger than upper row and column)*.

- In the `Space` class, we tested the contractor and made sure that we were getting space objects with the desired instance variables and dimensions. We included tests for the getters and setters to ensure that we're setting and retrieving the correct instance variables. We also tested to make sure that the instance variables of spaces were consistent and made sense.
- In the `Tray` class, we checked for a lot of the same things that we did in our `isOK` method. Things that we tested for are that Tray can correctly identify when blocks and/or spaces would be overlapping. We checked that the tray is counting the correct number of blocks and spaces, and that it doesn't exceed the dimensions of the tray. We also test to make sure that the Tray recognizes if and when the dimensions of a block or space is outside the bounds of the tray's dimensions.
- As we created `BlockCollection` we systematically tested each method as we created it. We added tested `add` by adding block objects to a `BlockCollection` object and then checking too see if the block object was actually in `myBlocks` for that specific `BlockCollection`. To test `contains` we called it for a block object the `BlockCollection` did not contain and it returned `false` and when we did the opposite it return `true`. For `size` we added block objects to the `BlockCollection` and called the `size` method which returned the right amount. We tested the Iterator method and found that it correctly returned all of the elements in `BlockCollection`. To test the `clone` method we called the `equals` method on the returned `BlockCollection`. For `containsAll` we tested to see if the method returned `true` for a given subset of a given `BlockCollection` and `false` for an unrelated `BlockCollection`. We tested `equals` by adding the same `Block` objects to two different `BlockCollection` objects and then testing whether or not they were equal. Finally we were able to test `getBlockWithCorner` by creating `BlockCollection` objects and then testing the method with coordinates we knew to be either `True` or `false`. In the creation of this class we simply made and tested methods as we needed them.
- As we created `SpaceCollection` we systematically tested each method as we created it. We added tested `add` by adding space objects to a `SpaceCollection` object and then checking too see if the `Space` object was actually in `myEmptySpaces` for that specific `SpaceCollection`. We then tested `remove` to see if the space object we wanted removed was no longer in `myEmptySpaces` for the specific `SpaceCollection`. To test `contains` we called it for a space object the `SpaceCollection` did not contain and it returned `false` and when we did the opposite it return `true`. For `size` we added spaces objects to the `SpaceCollection` and called the `size` method which returned the right amount. Finally we tested the Iterator method and found that it correctly returned all of the elements in `SpaceCollection`. In the creation of this class we simply made and tested methods as we needed them.

- In the `SolverChecker` class, we tested that it used the correct constructor depending on the arguments that we gave it, and that it gave the appropriate error messages when the number of arguments were outside the bounds of what was allowed for `SolverChecker`. We tested the debugging options and made sure that when `-o` was entered along with any of the options specified and the correct statements were printed.
- When we created the `move` object class we tested it immediately since it would be crucial to our code as a whole. The only method that needed testing was the `Constructor`. We tested that the coordinates were properly changed in the appropriate direction for each direction with numerous block objects and found the `Constructor` to be flawless.