Project 3 Readme

CS61BL Group:

Yang Liu (cs61bl-hc)

Michelle Ling (cs61bl-hi)

Teeranan Pokaprakarn (cs61bl-he)

Quinton Dang (cs61bl-hf)

Division of Labor

In our project, we based our work off a lot of discussion and cooperation. We would often discuss our plan of attack, and partners who felt more comfortable or had a better idea of what to do would often volunteer to tackle a portion of the project. We would then split off remaining tasks with whoever was left and tackle these coding problems day by day. We constantly updated each other through a group chat, and uploaded new files onto our Dropbox folder with all of the old (in case we messed up) and new codes. Together, we explained our coding work to each other and collectively worked on teaching each other what was going on if we didn't understand it.

We split it up this way because it was the most natural and effective for us. If anyone was having trouble, other members would step in, and through a lot of trial and error, many of the sections were developed over time by several of our group members. For example, Yang worked a lot on the block and board implementation, but helped out as well with hash coding, the readme, and more. Teeranan focused on the algorithm, but was also working on the readme and changing board implementations as necessary. Quinton worked on the readme, debugging and isOK, and helped out with hash coding as well. Michelle worked on the board and block implementation, debugging, helped out with the algorithm implementation, and worked on the readme as well. However, all of the members contributed and helped each other, so no one section was solely done by a single member. Therefore, our work really reflects our cooperation and ability to communicate and work with each other, whether it was comparing code or adjusting our workloads if other members had other priorities, such as a huge essay due the next day. We worked very well together!

Design

For our block implementation, we wrote a Point class and stored the UpperLeft and LowerRight coordinates as two points in the block and used the x and y coordinates of those two points to determine the width and the height of the block. We decided to write our own Point class rather than use java.util's simply because we wanted to come up with our own hashcode method for Point. We chose to implement the hashcode for Point by turning the (x,y) of the point into a String and using the built in String hashcode method. For the hashcodes of blocks we chose to multiply the x and y coordinates of the UpperLeft and LowerRight points by different prime numbers and then add them all together to ensure a unique hashcode for every block at a different location and the same hashcode for every block at the same location.

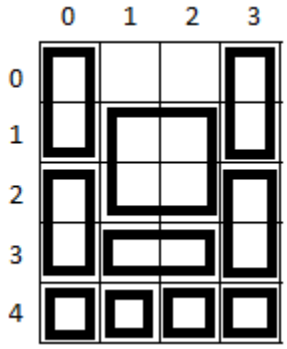
Originally we had implemented an array of arrays as our board and returned true for positions that contained blocks and false for positions that did not sort of like an adjacency matrix of sorts. However, this board design didn't keep track of which blocks were held at which position. So we implemented a tray using a HashMap of Points and their respective blocks. Every time a block is added to the board, we add every Point the block occupies on the board into the HashMap as the key, and the respective block as its value. We also add the block to a HashSet of Blocks called Blocks. For our Board hashcode we decided a good way to determine the uniqueness of boards with different blocks was to raise the UpperLeft x and y values and the LowerRight x and y values of every block in the board to a different power and then add them all together. Each board also holds an ArrayList of Strings called myMoves.

Our solver begins by searching for all of the empty spaces on our board by iterating through every possible combination of (x,y) points for a board the size of the one given to us in the initial tray file and checking which points are not located in the HashMap of Points and Blocks of our original board. Once we hit an empty space, we check to see if there are any blocks in the spaces to the left, right, below, and above it that can move to that position and if there are, then we check to see if the whole block will fit using our method validMove in Board. If it is, we make a new board with the block moved into its new location and we return the new board after adding the move we just made into the ArrayList myMoves for that particular new board. We chose an ArrayList implementation for this because we are never deleting anything from the collection and we need the items to be stored in the order that we added them so that later we can return the moves we made in the correct order. From then we check to see if our HashSet of Boards named boardSeen contains this current board. This is where our overridden "equals" method for board comes in. We wrote equals so that if the two HashMaps of points and blocks for two boards are equal then we assume they are the same board since they contain the same configuration of blocks. In this case, we do not add the blocks to our fringe structure because that would be a waste of time. We have already seen this board which means we have had it in our fringe before at some point and putting it in again would be a waste of space and time because we would repeat the same cycle of moves again. We specifically chose a HashSet for our boardSeen implementation because it contains no duplicates and by hashing our boards, it allows us to be able to determine if an item is in the HashSet already more quickly than an ArrayList where we would have to iterate through every item. If the board is not already in the HashSet of boards seen that we add it into our fringe structure and the HashSet boardSeen immediately after. For our fringe structure we chose a priority queue. We chose to implement the Manhattan distance heuristic for our program where we calculate the distance of each block from

its goal position in the board and order the priority of the boards by which one has the smallest Manhattan distance. To do this we wrote a Node class which holds a board and an integer which is the manhattan distance as well as a class NodeComparator which implements the Comparator interface and allows us to compare two objects by its manhattan distance. We use NodeComparator as the method of comparison when initializing our PriorityQueue and this allows us to pop out boards in a much more efficient manner from the fringe rather than simple breadth first traversal which is what we were doing at first with a regular Queue (implemented by a LinkedList).

For every board we pop from the fringe we first check to see if it matches the goal board and if not then we iterate through all of its empty spaces and repeat the process I just explained above where we check for valid moves and make new boards out of them and place them into the fringe based off their manhattan distance. We continue this until we have seen every configuration of movements possible in which case our fringe would be empty since we do not place boards we have already seen into the fringe, and our program exists, or we find a solution. If we find a solution we are able to print out the past moves the board took to get there because of the ArrayList myMoves we are storing inside each board. We found this implementation of the program to be the most efficient that we could think of and it worked relatively well for us. Although we are running out of memory for some of the hard problems still and a few of the puzzles take awhile to solve, we believe overall it is an effective solution. Below is a diagram of our tray so you can see how our HashMap would work.

For a board implementation that looks like this:



Coordinate view of the same puzzle

Our HashMap would look something like this:

Point	0,0	1,0	0,3	1,3	1,1	2,2	1,2	2,1	2,0	3,0	4,0	4,1	4,2	4,3	2,3	3,3
Block	B1	B1	B2	B2	B3	B3	B3	B3	B4	B4	B5	B6	B7	B8	B9	B9

Debugging

For our debug method, we wrote a class Debug and pass in the string to that class and enable certain Booleans to print out certain things later on depending on what the user chooses. To see the different debug options available the user can input –ooptions along with the intial and goal file and it will print out the debug names along with a brief description of what they do. After it will exit the program. Our debugging options include "numM" which prints out how many total moves were made at the end (does not count the moves made in boards never taken out of the fringe however), "possM" which prints out the number of possible moves each board can make depending on the empty spaces it holds (consequently we also have an "empty" option which prints out the current empty space that the board is trying to move a block to), a "printSGF" which prints the original board, the goal board, and the final ending board (which should include every block in the same location as the goal board, a "hash" returns the hashcode of the new board just made every time we make a move (so boards that we have already seen should have the same hashcode), a "mem" option which returns the amount of free memory left in the Java Virtual Machine after adding all the new possible moves a board can make to the fringe, and many more. They can all be used by inputting something like "-oempty" or "-onumBoards" as arguments before the initial tray and goal files.

For our isOK method, we started out by iterating through our blocks, which is represented as a hash set. We made sure that each block's upper left and lower right points were legal positions. To do this, we checked the positions of the points to make sure they were inside the board, which means they had to be equal to or greater than (0, 0) and less than the board's dimensions. We also checked to make sure the points were not switched around, as the lower right's point values must be greater than or equal to the upper left's point values. We compared these values to make sure that the blocks were implemented correctly. Next, we check for any overlapping blocks. Since our implementation saves blocks only by their upper left and lower right points, we converted blocks to all the points they represent. For example, a 2x2 block that occupies (0,0) and (1,1) in our implementation would be seen as four points: (0,0) (0,1) (1,0), and (1,1). When we iterate through each block, we save their points in an array list called tempPoints and then check them with an array list of all blocks currently in the board, represented by an array list called occupied. We check to see if the array list already contained the points for each block, and if it didn't, we would add the point to the occupied array list. If all goes well, it will return true, but if any problems arise, whether with the block syntax or overlaps, we throw an IllegalStateException with a descriptive error code. We thought these were all relatively useful in helping us to debug our program but the most useful tool to us in the beginning by far was printing out the boards and being able to see what they looked like after each move to ensure we were actually moving the blocks correctly.

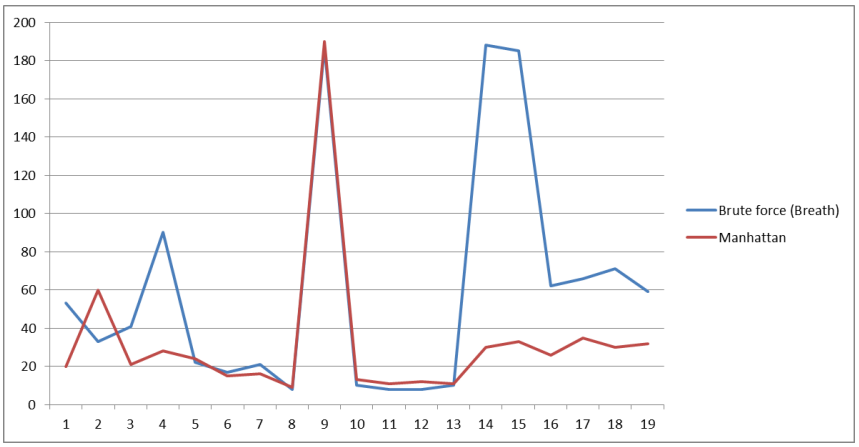
Evaluating tradeoffs

Experiment 1: Manhattan Algorithm (Current) versus Brute Force Algorithm

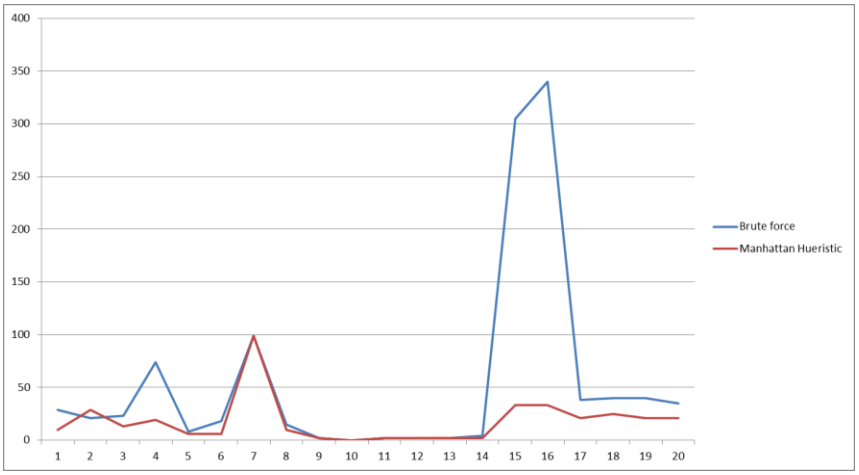
Summary: This test compares the runtime of the puzzle-solving algorithms of our current Solver and our old Solver, which used brute force. Overall, the Manhattan algorithm, our current use of heuristics, will run through the puzzles at a much faster rate than our old brute force algorithm. It also uses less space than the old algorithm.

Methods: Our methods to test this was to test the actual puzzles within each category and run through them with both algorithms and print out the time taken for each algorithm with each puzzle. We kept the majority of the code the same, only changing the algorithm to make sure that all the other variables that may change the speed/memory use of the code stayed constant. Finally, we created graphs shown in Results, that has the culmination of our data.

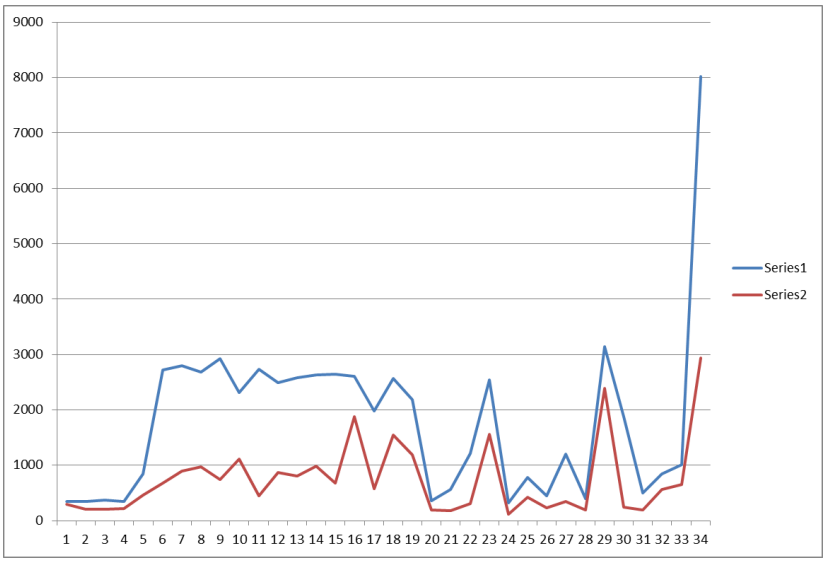
Results:



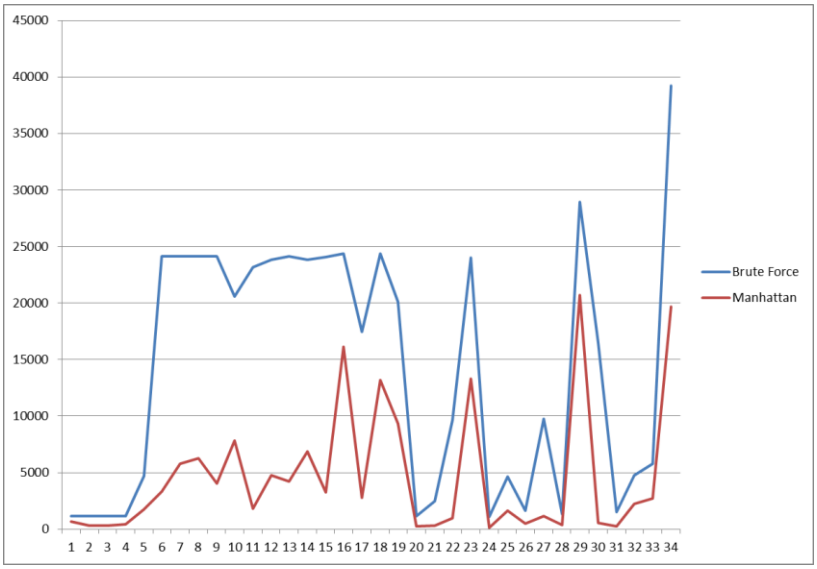
This graph shows the runtime taken for the easy puzzles. The y axis is the milliseconds taken, and the x axis just represents each different puzzle tested. The Manhattan algorithm was generally faster.



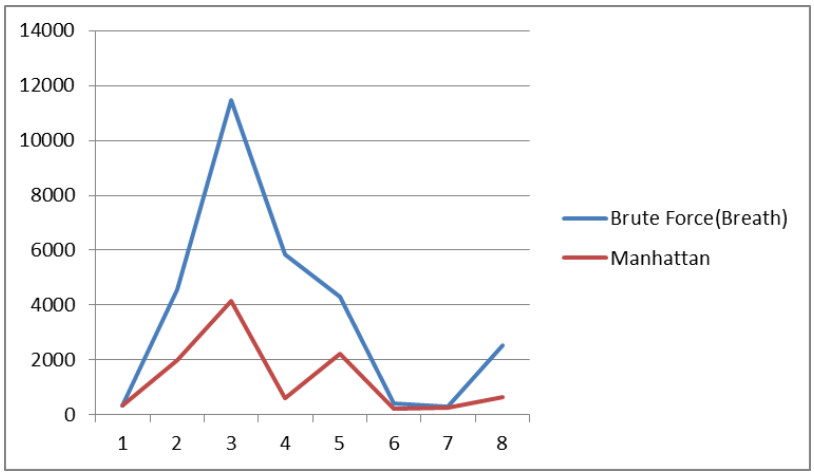
This graph shows the memory use of both algorithms for easy puzzles. The y axis shows how many boards were created, and the x axis just represents the different puzzles used. The Manhattan algorithm used less memory.



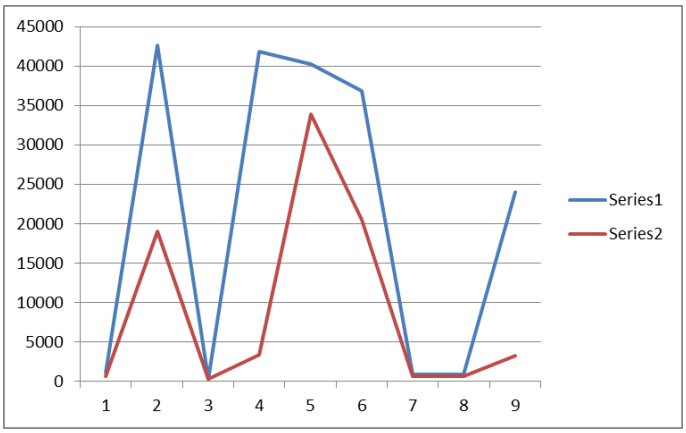
This graph compares the algorithm run times for medium puzzles. The y axis is the milliseconds taken; the x axis is the different puzzles. Series1 is the brute force and Series2 is the Manhattan algorithm. The Manhattan generally runs faster here.



This graph shows the memory use for medium puzzles between the two algorithms. The y axis is boards created, the x axis is the different puzzles, and it is shown that the Manhattan algorithm uses less memory.



This is for hard puzzles: y-axis is milliseconds taken; x-axis is different boards. The Manhattan algorithm is much faster here.



This graph shows the hard puzzles memory usage: y-axis is boards created; x-axis shows different puzzles. The Manhattan algorithm tends to use less memory.

Conclusions: As referenced in the summary, the use of the Manhattan algorithm was faster. This is because the algorithm made sure to avoid moves that would bring goal blocks farther away from the goal file and made Solver much more efficient. While on the other hand, our brute force algorithm would randomly move blocks around, whether or not it was close to or got us closer to the goal file. The brute force algorithm only wanted to make sure all the blocks matched the goal file, while our current algorithm would check the blocks and make choices that brought us closer to the goal file.

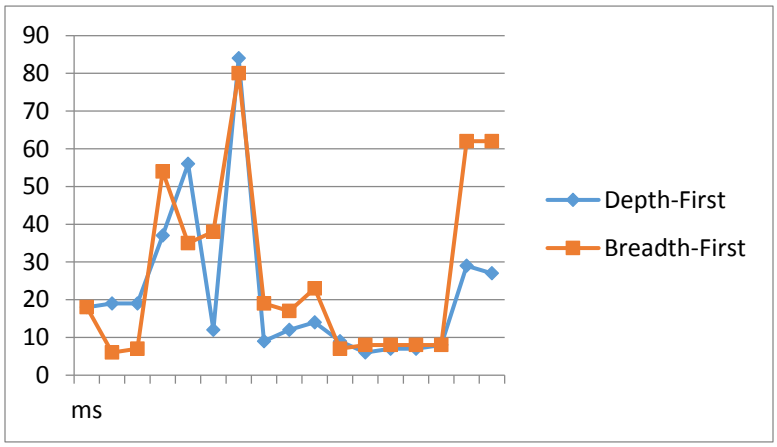
The Manhattan algorithm also used less memory while running through the puzzles. Since it would make smarter decisions, such as picking blocks closer to the goal file configuration, we saved memory by creating and storing fewer boards that we needed to keep within the fringe. In the brute force algorithm, since we haphazardly made moves, many more boards had to be created and stored, taking up much more memory. This also took more time than the Manhattan algorithm because creating more boards meant also iterating through the blocks again, storing more boards within the fringe, and checking more boards to make sure they would make valid moves. All of these extra moves the computer has to run through for the brute force algorithm causes it to run much slower than the Manhattan algorithm before finding the actual solution to the file.

Experiment 2: Breadth First Traversal (Current) versus Depth First Traversal

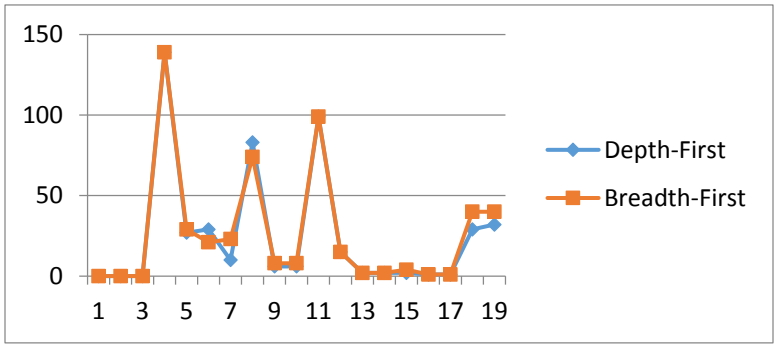
Summary: This test compares the difference in a breadth first traversal using a queue and a depth first traversal using a stack within our Solver file. Our current file uses a breadth first traversal, but here we compared it to a stack's depth first traversal. We found that, on average, the breadth first traversal was faster.

Methods: Our methods to test this difference in traversal were to create two separate Solver files that tested a depth first versus breadth first traversal. The algorithms, boards, and blocks used otherwise stayed the same. We ran through the puzzle examples given to us and kept track of the runtime and memory use for each puzzle, similarly with the last experiment. Once again, we created a graph out of our data we received, which can be seen within our Results section.

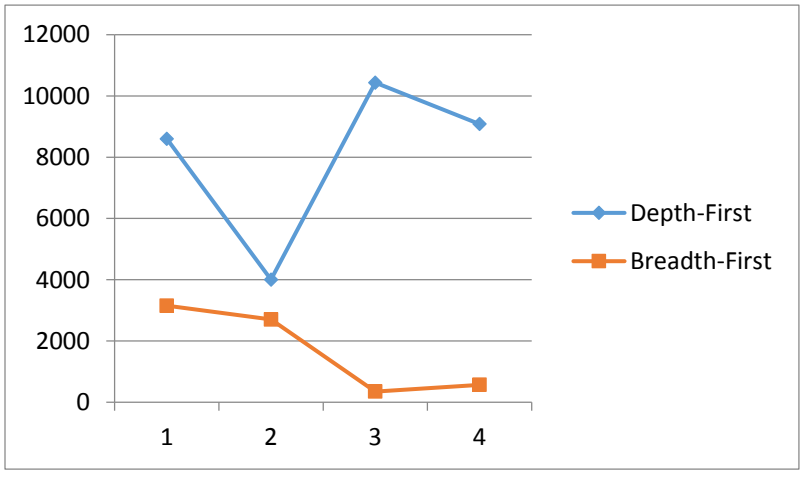
Results:



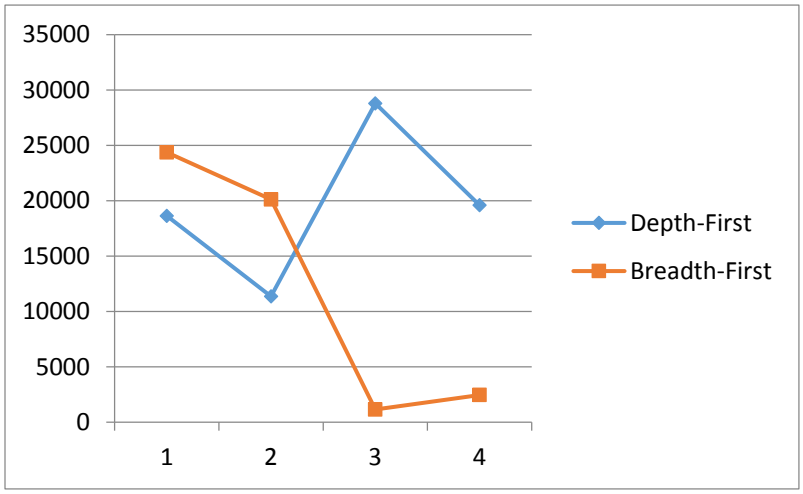
This graph shows the difference between the traversals, with the y-axis showing the time it took in milliseconds and the x-axis representing different tested puzzles in the easy category. We see that in this case, both traversals are about equal in runtime.



This graph shows the difference in memory usage through boards created (y-axis) between easy puzzles. We see that in this case, both traversals use about the same amount of memory.



This graph shows a sample size of medium puzzles, and the time it took to run through them in milliseconds. The breadth first traversal runs faster here.



This graph here shows how much memory was used (in boards created) between the two traversals in a sample size of medium boards. The y-axis is the number of boards created. On average, breadth first takes less memory than depth first.

Conclusions: Our conclusion is that our breadth first traversal was faster and more efficient overall than our depth traversal. Within our code, the breadth first traversal checks for possible moves in a board (based on our heuristics or algorithm), then goes through each recommended move and branches off the fringe with that. With each traversal, we add the seen moves to our seen board hash set, this way the boards will not move backwards in the possible move set, since the last move will be already seen. This traversal is actually a priority queue, using the heuristics we've created to choose which traversal to attend to first. The stack depth first traversal will look at which moves it should make and follow the same idea, choosing the move with the highest priority first. However, this traversal continues down the path of this board move, and puts each board in one by one as it goes further down the pathway. If it reaches a dead end, where there are no possible moves except for the ones already seen, then it will move back up the path and enter the next possible different move.

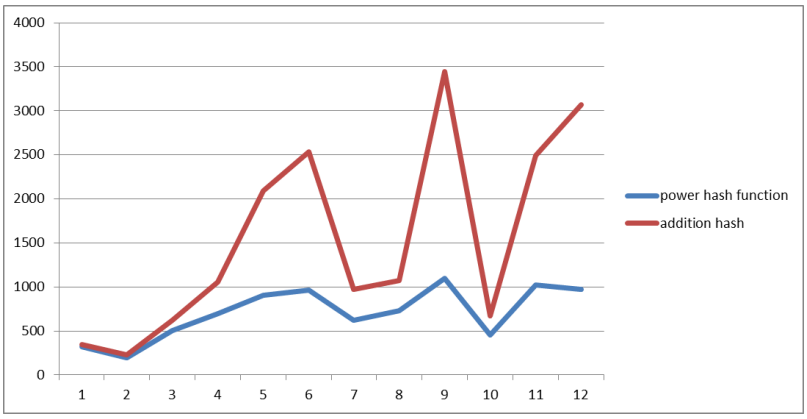
We believe the breadth first traversal is faster in average because it covers more options over time towards the goal file, while the depth first tackles one at a time. The depth first traversal, sometimes, gets "lucky", if one of the first traversals it does ends up to the goal file. This is especially more common within the easier puzzles, because the heuristics provided are more likely to hit one of the correct paths much earlier on than a harder puzzle, like the ones shown in the medium graph. This accounts for how, sometimes, the stack may sometimes return a faster result than the queue, but overall, the queue is faster because it covers more ground. In the case of the easy graphs, the stack and queue are almost indifferent in time and memory. But, since the queue is generally faster, it shows that it has to create less boards overall to get to the goal file. Due to this, it also ends up taking less memory than the stack, and is more memory efficient as well.

Experiment 3: Power Hash Code (Current) versus Addition Hash Code

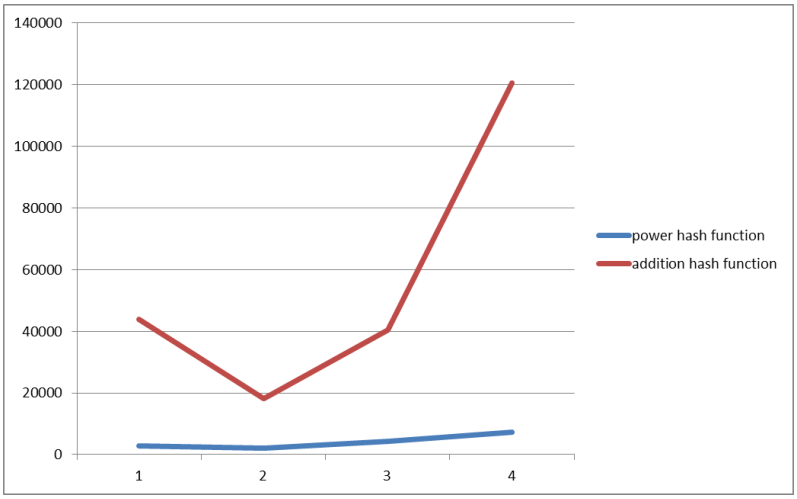
Summary: This test compares two different hash codes we implemented for each board. The power hash code, used in our current file, is compared to an old hash code we used before, called our addition hash code. How these hash codes were implemented will be in detail in our conclusions. We discovered that the power hash code generally ran faster than the addition hash code.

Methods: To test our two separate hash codes, we duplicated our Solver file and changed the second Board's hash code to our addition hash code. We then ran through the given puzzles with our two separate Solver files and timed them. We then graphed the differences in time between the two hash codes and put it onto a graph. The Results section has the graphs we've created.

Results:



This graph runs through medium level puzzles. The y axis is the time taken in milliseconds, and the x axis just represents different tested puzzles. The power hash function runs faster than the addition hash here.



This graph here runs through hard level puzzles. The y axis is time taken in milliseconds, and the x axis represents different tested puzzles. The power hash function runs faster than the addition hash here.

Conclusions: Our conclusion is that overall, the power hash code was more efficient than the addition hash code. The power hash code was implemented by iterating through the blocks and taking each point (each upper left and lower right) and setting them to a unique power. So, each of the four point values (UpperLeft's x and y and LowerRight's x and y) were set to a different power. Finally, we added these powered values to a variable, and added the values for each

iterated block to create our final hash code. In the addition hash code, we simply added all of the point values of all the blocks together to create our final hash code.

We believe the power hash code was faster because it generally had much less collisions than the addition hash code did. Since the addition hash code simply added the values, the hash codes collided much more. For example, a board of (0,1)(0,1) and a board of (1,0)(1,0) would have the same hash code despite being different boards. Due to this, when we checked the seen boards, we had to run through many more boards to make sure if a board was there or not (since the collisions would force us to look through all of the collided hash codes).

Disclaimers

Our project code can solve all of the puzzles except for 7 puzzles in the hard category. The puzzles we fail are: big.tray.1, big.tray.3, big.tray.4, c53, c64, c71, and pandemonium. For each of these puzzles, we receive an error where we run out of CPU. This shows us that we either have passed the time limit of 80 seconds, or ran out of memory (i.e. by creating too many boards, etc.). Also, although it does pass as verified within the checker, the easy puzzle 140x140 takes a long time to become verified by our checker (which means it takes a long time to solve). Enormous.3 also takes a bit longer than usual in easy.

One potential problem with our code is that our algorithm runs through every block with our heuristics and determines if it is a good move to make. However, in several of our puzzles, such as 140x140, many of the blocks shown are unable to make a move, and the only empty space is at the bottom. Our code inefficiently would run through all of these blocks before finding the blocks at the end that can actually move before deciding which move to make. This could easily contribute to the longer runtimes we might be getting.

One possible solution is to change the way we check for available moves, such as by creating some sort of data structure that will keep track of the empty spaces with blocks directly adjacent to it. This way, instead of checking each of the blocks, it can check the empty spaces and see which blocks can actually move, rather than iterating through all of the blocks. This will fix the problem of huge boards with countless blocks, such as 140x140, running for a long time. This is because it would only check the blocks around the few empty spaces, allowing it to iterate through only a handful of blocks, rather than all of the blocks on the board. However, there is also the issue of boards with many blank spaces and few blocks. Hopefully, by figuring out a way to only look at the empty spaces with blocks next to it, will decrease or eliminate this problem. Another possible alteration is to check to see if there are more blocks or more empty spaces, and work accordingly: if there are more blocks, check for empty spaces, if there are more empty spaces, iterate through the blocks!

Program Development

We first wrote our Block and Board class because we figured that those were two base structures we needed to have down regardless of what our algorithm would be because we couldn't figure out a way to traverse all the different paths without first knowing how our paths would be implemented. We wrote Junit tests for simple Board methods such as "moveBlock" and "addBlock" first because we assumed those were two necessary methods to write our Solver class and we wrote the "toString" next for our Board as a way of seeing the Board and ensuring that our add and move methods worked correctly. Next we worked on our algorithm and tried to figure out how we wanted to navigate through all of the different possible moves we could make. We came up with several different algorithms and different ways of traversing through all of the possible boards. We decided to write the solve() method last in our Solver class because we weren't sure exactly how we would implement it and we wanted the flexibility of being able to change our algorithm should we need to. Once we decided upon the Manhattan distance algorithm, that's when things began to pick up pace. We wrote our Node and NodeComparator class and chose the PriorityQueue class as our fringe and then we wrote the entire rest of the solve() method, adding methods and variables that we needed to board and block along the way. Several times we changed our mind about how we wanted to implement the hashcode method for board and block to try and make them faster as well. For our test cases for these we wrote several easy puzzles of our own that we could easily draw out on paper and used these to test our Solver class just to make sure that it was indeed making correct moves. We were happy with our decision to write the "toString" method for our Board class as one of the first things we did because it was extremely helpful in testing our Solver because after each move we could see what our board looked like after. We wrote the debugging parts of our code and the isOK method last because we figured we wouldn't know what to check for and how to check it until we knew for sure how our code would be implemented and what data structures we were going to use. By the time we wrote our debugging code, we had already debugged most of our program and we were relatively sure that it worked well. However, the constant checks to isOK and trying out our program with debug on gave us solid proof that our code was working as we had planned it to. In general we are happy with the order we chose to write and test all of our code. Although there were times we had to go back and change some of what we wrote when we changed our algorithm or decided to change our implementation of a certain data structure, such as using a HashMap as our Tray and writing our own Point class rather than using the one that comes in java.util, it was nice to have a place to start with before beginning to write our Solver class and it was relatively easy to modify the other classes later on to match whatever we came up with in Solver.