Project 3 README

GX(Joseph Jiang), FK(Jinwoo Kim)

GK(Norman Mu), FS(Kennedy Mesfun)

Division of Labor

All of our team members participated in the following:

* How to structure the fringe of our graph traversal
* How to design the State.java class, and what variables to include inside
* What sort of heuristics should be implemented to reduce runtime to an acceptable level.
* What algorithm should be used inside the Solver.java class to traverse the graph

**Joseph Jiang and Norman Mu** revised and refactored the code and streamlined methods in the Solver class

**Jinwoo Kim** coded the State class

**Kennedy Mesfun** made a set of invalid init and goal files to bulletproof the code.

Program Design

**Brief Description**

The objective of this project was to design a complete program to solve sliding block puzzles, including invalid puzzles and impossible puzzles. The central algorithm essentially brute forces all possible moves at all possible positions until the goal state is found. However, we have implemented a few heuristic improvements to help the algorithm determine which possible moves are more likely to solve the puzzle and attempt those first. If we traverse the entire graph of all possible moves and still cannot find the goal state among them, we conclude the puzzles is impossible and return nothing

The set of all possible positions is stored in a graph that is generated as we reach each node. We use a breadth first search using a priority queue to store the fringe. At each iteration, we poll the first node in the fringe, check if we’ve visited the position before, and add adjacent nodes (possible next moves) to the fringe if they’re also not visited.

**User-Defined Classes**

1. Class ‘Solver’

This class is our main class. It contains the main method to actually run the solving algorithms. It takes in the two init and goal arguments, sets up the tray, reads the files, places the appropriate blocks, performs sanity checks, solves the puzzle, and prints the solutions. Also deals with errors which print the error message.

1. Class ‘State’

State saves the status of each tray in our program with a reference to the tray before it in order to trace up its path for when we find the solution. Also saves initial and goal arrangements of blocks as static variables so all trays have access to the original trays. Block arrangements are represented by arrays of bytes in order to save memory. We store the top left position of each block, as well as the height and width of each block and a map of all sizes and the ID’s of blocks that match that size. As Java does not allow unsigned bytes, we used Byte.MIN\_VALUE to convert between our byte array and usable integers.

**Highlighted Methods**

1. Class ‘Solver’

* ‘static void work()’ method

The actual solving method. As described before, it uses a graph traversal algorithm. In this project, we use a priority queue as the fringe. We use a priority queue so we can mark different moves with a lower priority which signifies it should bring us closer to the goal tray. This allows us to implement different heuristics and optimize our runtime. After a state is polled from the fringe, we put it into a hashset that records what states we’ve visited before. Then, we determine which directions each block can move in. If the block is moveable in a specific direction and is not listed in the visited states, then we add it to the fringe.

If we find a solution, we stop the loop and look through the previous states of the current state in order to figure out the actual arrangement of blocks backwards. We find the difference between two states, store that in a stack so after we reach the init tray (who’s prev is null) we can print out the move sequence in the correct order.

1. Class ‘State’

- int isMovable(Block block)

Checks what possible moves the block can make. Return value will be 0-15, which in binary translates to 00002 to 11112. Each digit represents a cardinal direction, and a 1 indicates that we can move the block that way and a 0 indicates not. The first digit represents whether we can move west, the second east, then north and south. As movement only depends on a single block and its immediate surrounding, we search along one edge of the block to see if there are any existing blocks or if it would be the edge of the tray to determine if the block can move. For example, let’s suppose that we have 2x2 block and its top left position is (0, 0). In this case, the block will fill its tray with (0,0), (0,1), (1,0), (1,1). If the block moves to the right, it should fill (0,1), (0,2), (1,1), (1,2). We already know that we don’t have to measure (0,1), (1,1) because that position was where our block was positioned originally. So it only checks whether (0,2), (1,2) contains something already.

* public static String stringID(State input)

Makes a unique identifying string based on the position of each block, its size, etc. We first create a char array with size width\*height of the array. The char can store values up to 0xFFFF so the array can sufficiently hold information about the block. The char array will contain the information of each block in the input state and gets converted into a string for return.

* public int hashCode()

Return the hashCode for the State class. Because we have a “hashString” from the stringID method, we can just use the hashCode of that string.

* public boolean equals(Object obj)

Returns whether the Object obj is same as the current state. We can simply compare the stringID’s because they are unique for each particular combination of blocks in a tray.

Experimental Results

**Experiment 1**

1. Summary

For the first experiment, we used a normal queue data structure, and stacked each movement without priority. It worked, but very slowly and failed a lot of the tests.

1. Results

       Table 1. Experiment 1’s test results

|  |  |
| --- | --- |
| Bash Script | Results |
| run.easy | Success on all tests |
| run.medium | All tests fail (Every puzzle took over 80s) |
| run.hard | All tests fail (Every puzzle took over 80s) |

1. Conclusions

Let’s suppose that there is a block whose input is [1 1 1 1], and goal is [3 3 3 3]. No other block exists in the tray. The program will first iterate through all trays at the same depth. In this case, the program will iterate through 4 orange blocks, 6 red blocks, 4 pink blocks and then arrive at goal.

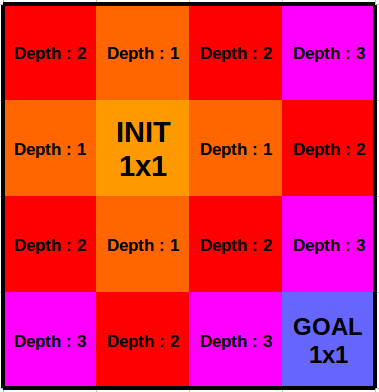


Fig 1. Experiment 1’s path finding algorithm. Same color has same level of depth.

**Experiment 2**

1. Summary

For the second experiment, we used a priority queue structure and stacked each movement into the fringe. We’ve measured priority by sum of expected shortest distance to the goal position. It succeeded on almost every puzzle except for 3 in the hard category.

1. Results

      Table 2. Experiment 2’s test result

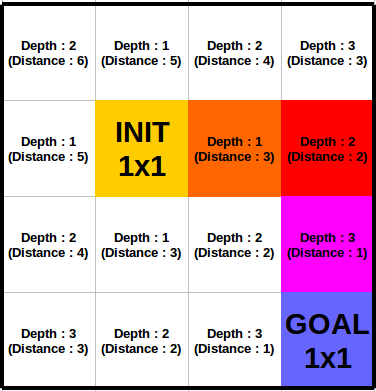
|  |  |
| --- | --- |
| Bash Script | Results |
| run.easy | Success on all tests |
| run.medium | Success on all tests |
| run.hard | 25 puzzles pass, 3 puzzles fail |

      Table 3. Experiment 2’s failed puzzles

|  |  |  |
| --- | --- | --- |
| Failed puzzle init File | Failed puzzle goal File | Time to solve |
| big.tray.4 | many.blocks.20+  180.goal | 1000s |
| many.blocks.20.goal | 120s |
| many.blocks.100.goal | 800s |

1. Conclusions

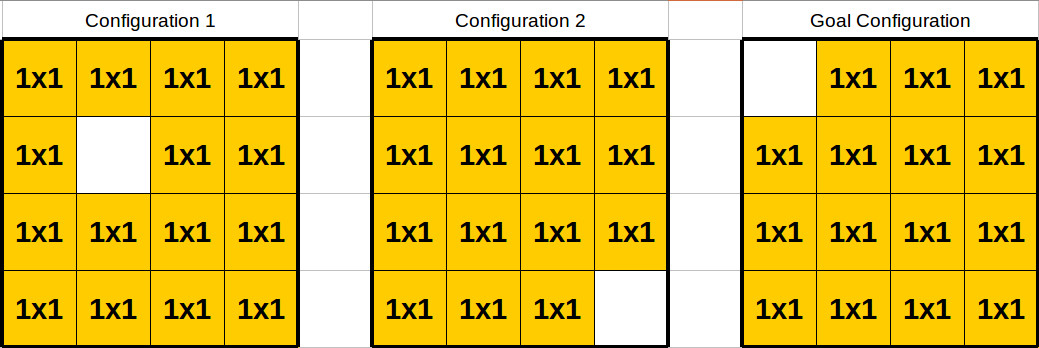
Let’s suppose that there is a block which input is [1 1 1 1], and its goal is [3 3 3 3]. No other block exist in the tray. The program will first iterate through all trays at the same depth, so supposing the priority is the same for all moves, it would first move the block east, then south, west, and north. In order to solve the puzzle, our program would eventually determine a path where it moves the block east until it hits a wall and then downward toward the goal position.



Pic 2. Experiment 2’s path finding algorithm.

Colored blocks is the path calculated by program.

From this optimization, we could pass almost every puzzle in the medium and hard categories. However, as we can see in the 3 failed puzzles, this algorithm still has some issues.



Pic 3. 2 init configurations and a goal configuration

The above situation demonstrates why our 2nd code build could not pass the final 3 tests. Given the two init configurations, one can plainly see that the first configuration requires less movement than the second one, but our heuristic calculates the “distance” of both configurations from the goal to be the same, and thus fails to appropriately prioritize the closer one and provides no optimization for our program.

**Experiment 3**

1. Summary

The general algorithm remains unchanged for our 3rd experiment. However, we’ve added a special case in which the tray consists of all 1x1 blocks with one “hole” that we want to move into a given goal configuration. In this case, we use a different heuristic that looks at the distance from the hole in the current tray to the hole in the goal tray.

1. Results

      <Table 4> Experiment 3’s test result

|  |  |
| --- | --- |
| Bash Script | Results |
| run.easy | Success on all tests |
| run.medium | Success on all tests |
| run.hard | Success on all tests |

1. Conclusions

The failed test cases in experiment 2 share the same initial configuration file, and the same number of goal configuration files. The blocks inside the init file are all 1x1 and there is only 1 space without a block. To solve these puzzles, we use our special algorithm, which is activated when we scan the tray during our sanity check to see if it matches the criteria.

Program Development

The first methods we wrote were designed to unpack the init and goal arguments and convert the files into usable data. These methods once designed should remain relatively stable, and should not impact optimization as we only run them once. Then, we discussed what methods and variables should be made static or nonstatic in the State class. Initially we used a third class, Block.java to store data about each block but realized this was very memory inefficient. Though the project specifications say we do not have to optimize memory use, our program was hitting the 8gb memory cap of the hive servers and thus cut into our runtime. In order to do away with the Block class, we saved all block data for each state in several arrays based on a Block’s ID number in the state. With our optimizations, we reduced memory usaged to ~340MB even on the hardest problems. Another cause of slow runtimes was the cumbersome .equals() method in the state object. Originally we compared lists of arrays, but we improved it by just comparing the stringID of each state which already stores information about every block in the tray. This improved our runtime by over 90%.

There are two tests that we’ve used for the program development - one using the puzzle set given from project 3 description and one using the invalid files to checkout whether it prints out the correct error message and shut down program safely. The puzzle set given from project 3 was a full set of klotski puzzles which we can’t think easily, so we’ve substituted the functionality test with these puzzles. Also we created a simple test case to verify that the error message is printed out correctly.  Since it reads from top left to bottom right, we created the test to have 0 1 and 1 0.  This would create a line instead of creating the box. We obtain our error message. This test was created for bullet proofing the project.

Improvements

There are two ways to improve this program significantly - first is using a better algorithm, and second is using a parallelization technique.

Right now, we just calculate the distance between goal configuration and current state. It doesn’t consider whether the block exists in this shortest path. Also, we pass all of the tests in experiment 3, but we’ve used separate algorithm to solve failed test case in experiment 2 - this means that the defective algorithm should be improved. It is the key factor for the performance because we’ve already optimized the other methods itself, such as isMovable or equals in State class.

Also, the path finding algorithm has large set of queue and can be easily parallelized - one example is making a master cpu to divide the states in the queue to their slave cpus. If one of the slave cpu found the solution, it will broadcast to all cpus to stop. However, this technique should be applied carefully because Java already implemented automatic multi-threading optimizations, although it is not quite good and useful.