PROJECT: **Rushhour Solver**

A DOCUMENTATION

By: Lymeng Naret

301421944

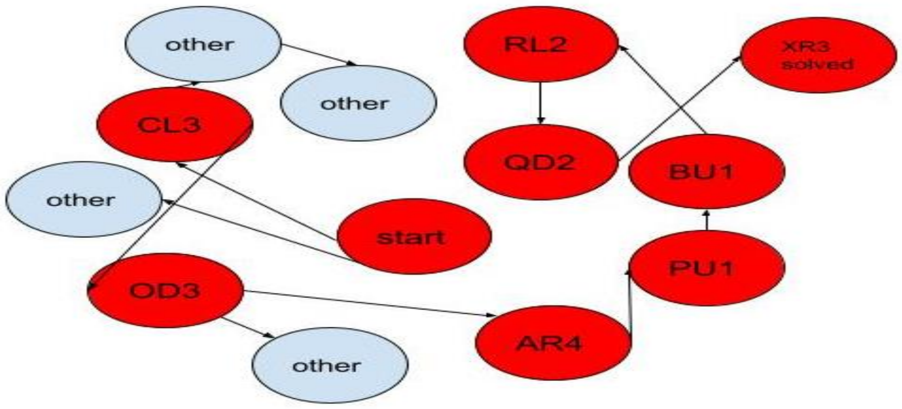
4/13/2021

Rushhour’s pathfinding algorithm

Rushhour’s pathfinding algorithm is a graph traversal algorithm that tries to find the shortest path/step to solve a 6x6 rushhour puzzle. To start, I reimplement the “RushHour” class from assignment 1. I made changes and improvements so that it can be used in other classes. Then, I create a class “state” that is similar to RushHour, but it uses less memory and has fewer functions. Afterward, I implement a graph class that holds nodes and their edges. With the classes above, I decide to use A-Star algorithm to solve the puzzles.

For RushHour, I have considered making a bitboard to save memory, but I figured it will be hard to use with other classes later, so I used a standard 2d array of char. In addition, I made a class named “VehicleList” that has a list of “Vehicle.” When a non-empty character is read from a file, Vehicle class stores the name in a char, the coordinates of each character in List of List of Integer, the type in a char (car ‘c’ or truck ‘t’), and the direction in a char (vertical ‘v’ or horizontal ‘h’). After each Vehicle is initialized, it is stored in VehicleList to be accessed and used later. With those two classes, I implemented the standard functions such as validating move, making move, and checking if a board is solved. I also added functions such as one that returns true if a move can be made and vice versa, one that reads from a 2d array of char instead of a file, and one that returns the next state of a board without changing the original board.

I implemented the class “state.” This class has a 2d array of char and a String. Like RushHour class, this class reads the state of a board, but it uses much less memory. It has a String that holds the step required to get to the current state (null for the base state). Perhaps the most important purpose of this class is the computeHash function that significantly speeds up the equals function. To assign different states to unique hashcodes, I use the XOR operator as well as the Math.pow function.

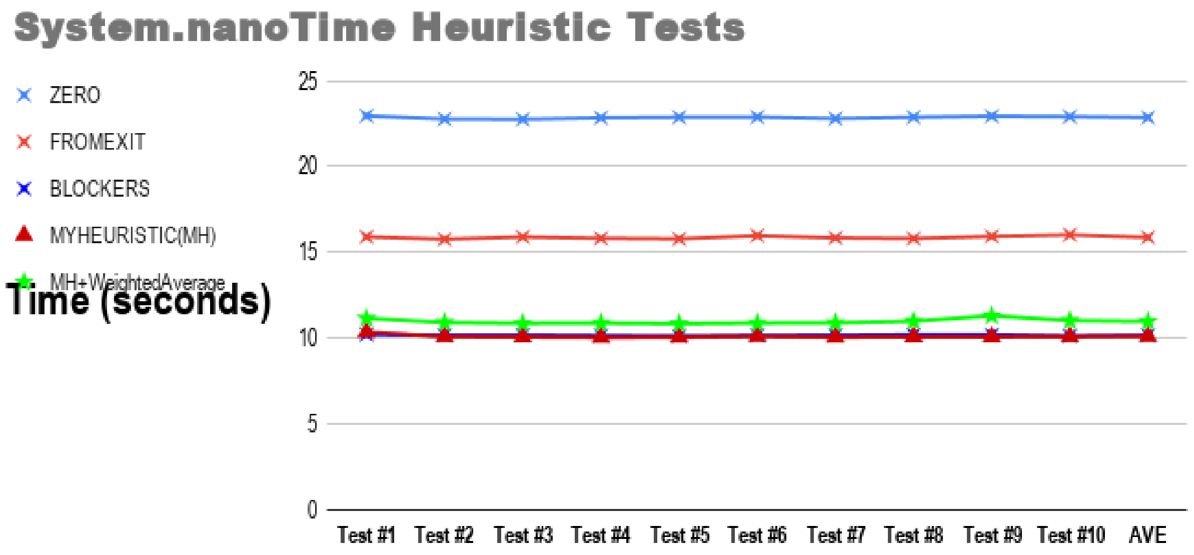


A very small and incomplete example of the usage of graph to solve puzzle A01.

To hold all the possible, unique states of the board, I built the class “graph.” It has a set of nodes (vertices) and a map with the key holding nodes of states and the value holding sets of nodes (edges or all accessible states from current states). There are standard graph functions such as getting a node using the name, getting the set of all the neighbors, adding nodes, adding edges, and removing edges. Adding and removing are configured to return boolean to avoid duplicate nodes or edges.

To represent a vertex in a graph, I made the class “node.” Each node is used to hold three int(s), a state, and a parent node. It has standard setters and getters. The setter of a state calculates a hashcode, which is 31 times the hashcode of the state, and it also has an equals function. The three int(s) are to hold value of heuristic, G, and F (G+heuristic).

I made a heuristic interface with only the unimplemented getValue function. Then I make a class called “myHeuristic” that implements that interface. In that class, there is a constructor with parameter of a class state. It analyzes the current state and give it a value, smaller is better. Using standard nested for-loops, first, it tries to find the row with the red car. Once the row is found, it checks that row. The loop is going to count the distance the red car is from the exit. At the same time, it counts the number of cars blocking the path to the exit and the number of cars blocking those cars. It gives a higher value to blockers that are closer to exit and gives a value of 2 to the blockers of the blockers (assuming it will need at least 2 to move out the way). Once the row is checked, the loop ends. The value of the heuristic is set to be the sum of the distance from exit and the value of the blockers. Whilst testing, I find that using the weighted average of the sum of the values gives shorter paths but takes slightly more time. I decided to prioritize time. I have tried different heuristics such as zero heuristic/BFS, counting number of moves that can be made, and the two heuristics above individually. Some seems to work better on a specific puzzle but is much worse on another. I run tests solving the given 35 puzzles consecutively 10 times each. Overall, the statistics show that combining the distance from exit and the value of the blockers give the best result.



Tests' data ran on a given 35 puzzles consecutively 10 times.

The algorithm I used to find the shortest paths for the puzzles is A-Star. Essentially, there is a PriorityQueue that gives priority to the states with lower value F and holds the states that are not processed. I tried using a Set but using a Map to hold closed states takes slightly less time. At the beginning, there is a starting state which holds a G value of 0 and the heuristic value is calculated. Its parent is set to null then is added to the PriorityQueue as well as the root of a graph. A normal while-loop keeps going until the PriorityQueue is empty or the puzzle is solved. To process each state, I made a generateNeighbors function. The function uses RushHour class and graph class. The RushHour class is used to try all possible moves and the graph class is used to make sure no duplicate state (stored in node) is added. After all the neighbors are generated, I followed the steps given in the Lecture 23, but I find that a simple if and else if does the job faster. Basically, a for-loop goes through all the neighbors. If one of the neighbors is a solved state, set the parent to be the state that it is generated from and the G value to be the parent's G + 1 and finally pass that state into a function called writeSolution. That function takes a node of state and an outputPath. Using the getParent function of each node, it follows the nodes using the edges from the solved state all the way back to the starting state. For each of the node, the step is stored in a String. After all of the steps are stored in the String, I use Path and Files.write to write the String into a new file named outputPath. Else if none of the neighbors is a solved state, use contains function to check if it is in PriorityQueue and Map. If a state is not in both then calculate the heuristic value, update G and parent the same way above, and add it into the PriorityQueue. After all neighbors of a state are processed, add the state into the Map to mark it as a processed/closed state. In short, the loop will try all moves possible in a state. If a move makes a new state, add it to the graph as a new node with edge connected to its parent. With PriorityQueue that makes use of Heuristic and F, it will process states that are more likely to generate the solved state first. The loop will continue till either it is solved or no new states can be generated.

In conclusion, the puzzles are stored using the class state or RushHour. RushHour has functions that make changes to a board whereas state is mainly used to store a board and has hashcode. A graph is used to store all unique states. Each node has hashcode and getParent. Hashcode is used to give uniqueness and getParent is used to backtrack from a solved state all the way back to the start. Finally, using A-Star and heuristics, loop through all possible states in a puzzle until there is either no more new states or a solved state is reached. A-Star uses PriorityQueue and a Map to try to minimize the path and time. Overall, implementing the classes above was fun and the difficulty was reasonable, but the most difficult part was optimizing the solver for shortest paths in shortest time without breaking anything.