Documentation
Project – Rush Hour Solver

Made By –

Supreet Singh Dhillon

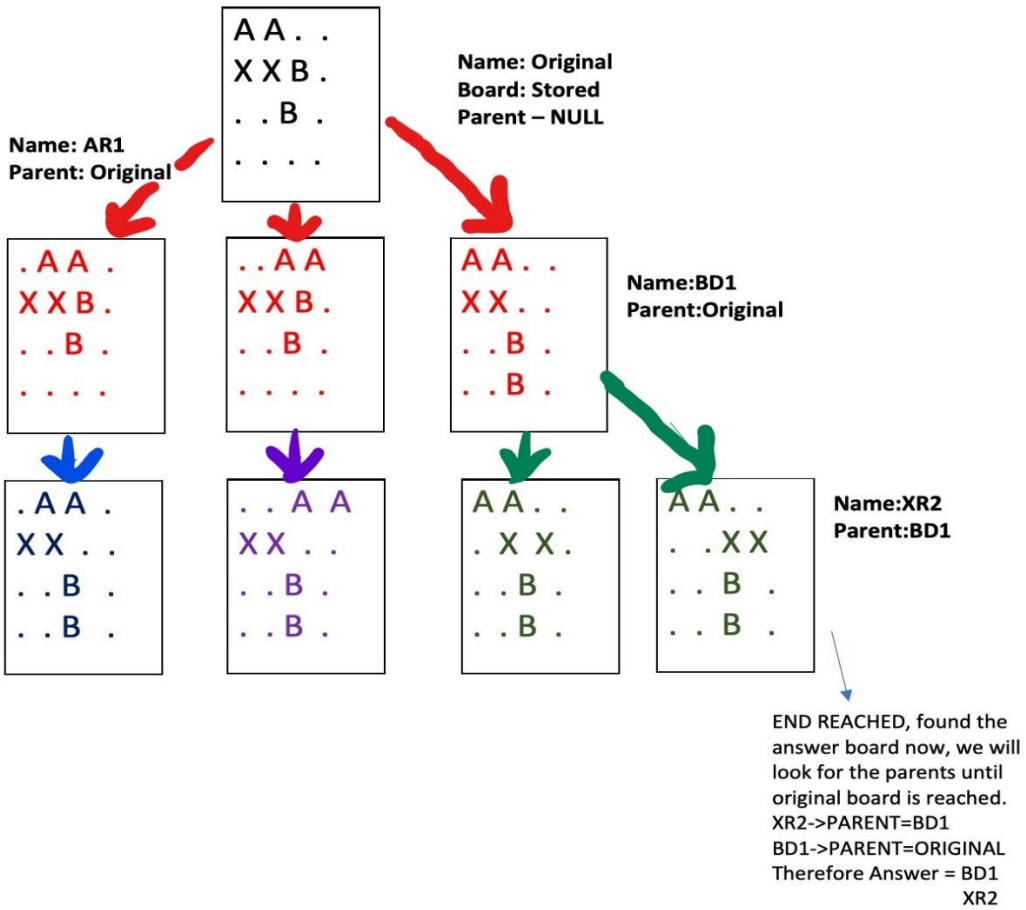
Id- 301417326

EshaanPal Singh

Id - 301416816

Approach -

We have used a tree to solve rush hour. The root of the tree is the Node which is created from the board which is input by the user and all the moves done on it become its sub Nodes and the tree goes on and on. Our stopping condition is when a Node whose board has the car X at exit position is created. To find the answer and the path we have used a recursive approach, every Node has a parent variable in it and we just find the parents of Nodes until we reach the root of the tree. (A small example is displayed below)



We have used a 2D array to store boards because it is easy to move cars on a board if it is stored as a 2D array.

We first thought about the graph approach to make the program but we were not able to link the nodes (vertices) with each other properly and the stopping condition was also becoming a problem for us in the graph method as we were becoming stuck in an infinite loop so we used trees.

Classes –

The program has 2 classes –

1) Node (Made by Supreet and Eshaan, planned this class together)

2) Solver

The class Node is used to store different boards, it has three parameters which are-

- name - stores the name of the move done on the board for example CR1 (car C has moved right by 1 step on the board).
- parent - stores the parent of the node.
- board – stores the board.

Functions -

The class Solver has 9 functions

1) same_board_check and compare_boards – all the Nodes of the tree are stored in an ArrayList called nodes_list and these functions check if the board of the new Node formed is equal to the board of any other Node in nodes_list or not. Earlier only one function was made to only compare the names of the Nodes. This was giving a problem in the function add_node as some trees were being skipped because of this as Nodes with different boards can have the same name according to the approach. So 2 functions were made to compare the boards of different Nodes to see if their boards are equal or not. (Both functions made by Supreet).
2) check_board – checks if the new board which has been formed is the answer board (checks if car X has reached the exit point or not). It checks if the y co-ordinate of the car X is 5 or not. (Made by Supreet).
3) add_node – creates and adds new nodes to the tree. It takes a Node as its input and creates its children. When the Node containing the board in which car X has reached the exit point is made, no more Nodes are created. This was the hardest function in the whole code. In the earlier approach this function was called recursively and this was causing Stack Overflow error

so then instead of calling it recursively it was called with the help of a queue next_board (was used to store all the Nodes on which the function add_node was to be used) in the function solveFromFile which removed the error. (Made by Supreet).

4) find_answer - when the Node containing the board in which car X has reached the exit point has been made it is passed on as argument to this function and this function finds the path from the original board (root) to this node by finding the parents of the nodes. These parents are pushed into a stack called answer, in order to write them in the file. (Made by Eshaan).
5) convert – the original board given by the user is read as 6 strings. This function converts those 6 strings into 6 arrays of type char and makes a 2D char array out of them called board. (Made by Supreet)
6) find_cars – takes a board as an input and finds different cars (like A, B, C) on it. (Made by Supreet)
7) find_specific_car – takes the car name and a board as an input and finds the position of the car on the board as 2D points on it and stores it in an arraylist of Points called car_holder. (Made by Eshaan).
8) find_moves_for_each_car – takes the position of the car (stored in car_holder), car name and the board and finds all the moves for the car and stores them in arraylist of integers – up_move, left_move, right_move,

down_move. The moves are stored in an array list because the size of the array list can be changed. The array lists (up_move, left_move, right_move, down_move) and the variables used to store the number of moves and the length of the moves for each car are declared outside this function so that they could also be used in the function add_node to perform the moves on the board. (Made by Eshaan)

9) solveFromFile – this uses all the functions mentioned above in one way or another and it finds the solution to the board input by the user. (Made by Eshaan)

Steps Followed by the program -

- First our program reads the board from the user and converts it into a Node with the help of the function convert and stores this board as the root of the tree and this Node is called "original" and then we store this Node in a queue called next_board (contains all the nodes which will go as an input for the function add_node) and the arraylist nodes_list (to check if a board has been repeated or not). (We have declared a variable called found whose initial value is zero but when a Node whose board has the car X at exit position is created its value changes to 1).

- Then the head elements of the queue next_board are passed into the function add_node using a while loop which terminates when found becomes equal to 1. First the Node named original is passed into the function add_node as argument. The function add_node will first find the names of all the cars in the board of original using the function find_cars and store them in an array called cars_names (A, B, C, etc.). Then we create an arraylist of Points called car_holder which stores the position of a particular car as points in 2D for instance position for car A can be stored as points (1,2) and (1,3). Then an iterator it is created which iterates through cars_names. Then a while loop is created which will stop when it.hasNext() is false. Inside this while loop a char variable temp8 stores the name of the car (temp8 = it.Next ()). Then the array car_holder is used to store the position of this car using the function find_specific_car. Then all the moves for this car are found using the function find_moves_for_each_car which stores the moves in the arraylists up_move, left_move, right_move, down_move and the number of moves in the integer variables leftcount, rightcount, upcount, downcount. Then these moves are performed one by one and when each move has been performed a new Node is formed and before this Node is put in the queue next_board it is checked if the board of this Node is equal to the board of any other Node which is already in arraylist nodes_list, all the new Nodes created are added

in both nodes_list and next_board if their boards don't match with any other boards in nodes_list. All the new Nodes created are also checked if they are the answer that is if their board has the car X at exit position. This process goes on and on until a Node is created whose board has the car X at exit position.

- When this Node is reached then no new Nodes are formed and the value of the Node find_parents (this Node is globally declared and is an input for the function find_answer) is set equal to this Node and the role of the add_node function is finished. The find_answer function takes in the Node and stores the names of its parents in the stack answer and continues to do this until the parent of a Node is original. Then the values stored on the stack answer are written on the file whose path has been provided by the user.