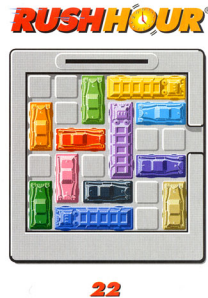


Goal

Rush Hour is a game from Thinkfun, the goal of which is to free a car from a traffic jam. Vehicles move across a 6×6 grid, either horizontally or vertically. Each vehicle is 2 or 3 squares long. Vehicles cannot exit the board, except for the red car, which can exit from the right side of the board; this is the goal of the game. Here is an example of a starting configuration:



The objective of this assignment is to write a program that finds the solution to such a problem, in a minimum number of moves.

If you want to get acquainted with this game, you can play for example [here](#).

Source recovery

Extract the contents of the `src.zip` file to the project's `src` directory.

Enable assert in your Java virtual machine

In *Visual Studio Code*, type the key combination `[Ctrl]+[,]` (the second key is 'comma'), enter `vm args` in the search bar. The item `Java>Debug>settings: Vm Args` appears with a text bar in which you must write `-ea` (for enable asserts).

As before. Submit the `HW7.java`

1 Representation of the problem

We adopt the following representation of the problem. The 6 rows are numbered from top to bottom, from 0 to 5, and the 6 columns from left to right, from 0 to 5. The `RushHour` class models a general game board described by the following five fields:

- `int nbCars` : the number of vehicles;
- `String[] color` : an array giving the color of each vehicle (for displaying the solution);
- `boolean[] horiz` : an array of booleans indicating for each vehicle whether it is moving horizontally;

- `int[] len` : an array giving the length of each vehicle (`len[i]` represents the number of spaces occupied by vehicle `i` , and is 2 or 3);
- `int[] moveOn` : an array indicating on which row (resp. column) a horizontal (resp. vertical) vehicle is moving.

By convention, the red car is the vehicle with index 0 and length 2.

The state of the board at a given moment in the game is represented by an object of the `State` class which contains the following fields:

- `RushHour plateau` : A pointer to an object of the `RushHour` class that models the invariants of the game board.
- `int[] pos` : an array indicating the position of each vehicle (the leftmost column of a horizontal vehicle or the topmost row of a vertical vehicle);
- `int c` , `int d` and `State prev` : indicate that this state was obtained from state `prev` by moving car `c` `d` spaces (`d` is `-1` or `+1` : `-1` indicates a move to the left or up, `+1` indicates a move to the right or down);

Question 1.1

- Complete the `State(RushHour plateau, int[] pos)` constructor of the `State` class which constructs an initial state (the fields `c` , `d` and `prev` are not significant in this case).
- Complete the constructor `State(State s, int c, int d)` of the `State` class which constructs a new state from `s` by moving car `c` by `d` space (`d` is `-1` or `+1`). We assume that this move is possible.
- Caution: You must remember to instantiate the `plateau` field and the `prev` field .
- Complete the `boolean success()` method of the `State` class which indicates whether this is a state corresponding to an end of the game (i.e. the red car is located immediately in front of the exit).

Test your code by running the `Test11.java` file .

Question 1.2 Complete the methods:

- `boolean equals(Object o)` of `State` which returns `true` if object `o` corresponds to the same game state as the current state (regardless of their `prev` field).
- `boolean[][] free()` which returns an array result of booleans indicating which spaces are free in the current state. We will adopt the convention that `result[i][j]` represents the space on row `i` and column `j` . We will use the board pointer to access the `nbCars` , `horiz` , `len` and `moveOn` fields of the general game configuration.

Test your code by running the `Test12.java` file .

2 Possible movements

From now on, we only work in the RushHour class .

Question 2 Complete the `LinkedList<State> moves(State s)` method , which determines the set of states that can be reached from state `s` by making a single move (a single square). This set is represented by a `LinkedList<State>` , the order in this list is not important. To write this method, use the `boolean[][] free` method of the `State` class , which indicates which squares on the board are free and which you wrote in the previous section.

Test your code by running the `Test2.java` file .

3 Search for a solution

We now tackle the search for a solution, thanks to an exploration of the game states.

3.1 DFS

We'll start by implementing a **depth-first search** , which represents the “human” way of playing. There are two ideas for this

- we need to memorize the states that we have already encountered, and for this we will use a *visited set* of type `HashSet<State>` ; to use it, we have the `contains` and `add` methods ;
- We use a stack that will initially contain the initial state. As long as it is not empty, we extract the first state from the stack. If it corresponds to a solution, we are finished. Otherwise, we add all its neighbors not already encountered to the stack and to the *visited set* .

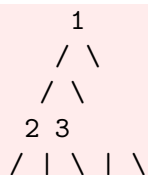
Question 3.1 Complete the `State solveDFS(State s)` method so that it performs this algorithm. It must return the state corresponding to a solution (the one where the red car is located immediately in front of the exit).

Test your code by running the `Test31.java` file .

3.2 BFS

We now want to find the shortest solution (i.e. with the least movements), for this we will use a *queue* .

If we represent the set of states by a tree whose root is the starting state and whose children each node `s` are the states that can be reached from `s` by moving, we want to traverse the nodes of this tree “by levels”, like this:



```
4 5 6 7 8
/...
```

Question 3.2 Complete the `State solveBFS(State s)` method so that it performs this algorithm. It must return the state corresponding to a solution (the one where the red car is located immediately in front of the exit).

Test your code by running the `Test32.java` file .

4 Display the solution

Question 4 Complete the `void printSolution(State s)` method that displays a solution, given the state `s` corresponding to the solution (the final state). Note that the states forming the solution are chained from `s` following the `prev` field . We will try to display the solution in the correct order.

This method should also display the total number of moves. We can add a static `nbMoves` field to the `RushHour` class for this purpose. We may use a recursive version to do this.

Test your code by running the `Test4.java` file , which should result in something like:

```
46 trips
we move the blue vehicle to the right
we move the black vehicle to the right
we move the green vehicle upwards
we move the pink vehicle down
....
```

Note: your solution may differ from this one (it depends on how the list is constructed by `moves`) but it must have the same number of moves (46).