# Selected Fun Problems of the ACM Programming Contest
# Rush Hour

Martin Pluisch
martin.pluisch@student.uni-tuebingen.de

## 1 ABSTRACT

During the seminar "Selected Fun Problems of the ACM Programming Contest" I worked on the so called "Rush Hour" problem, which was posed by the "Association for Computing Machinery" (ACM) within the context of the annual "International Collegiate Programming Contest" (ICPC), and implemented a possible solution, based on breadth-first search, programmed in C# utilizing the Unity game engine. In this paper I will describe the given problem, introduce and explain a possible solution plus its complexity, illustrate the algorithmic core of my implementation and lastly describe the quintessential parts of my GUI.

## 2 INTRODUCTION

The "Association for Computing Machinery", or ACM, was founded in New York on September 15, 1947. It is currently "the world's largest educational and scientific society, uniting computing educators, researchers and professionals to inspire dialogue, share resources and address the field's challenges" [2].

The annual "International Collegiate Programming Contest", also known as ICPC, is organized by the ACM and sponsored by IBM [4]. The ICPC is a contest between international groups of three students each; each group has to solve between eight and ten programming problems within five hours, using their preferred programming language from C, C++ or Java [4]. One such problem is "Rush Hour", which I'll now explain in detail.

## 3 PROBLEM

"Rush Hour" is based on a commercially available game, sold under the same name, in which the player has to unstick a complicated configuration of cars, so that a certain car can leave the grid [3]. The grid consist of 6 by 6 squares. Parked on this grid are up to 10 cars, each 1 square wide and either 2 or 3 squares long. Cars are placed at integer locations on said grid and can only move straight forwards or backwards - cars can not drive off the grid, can't overlap with each other and can't turn.
A puzzle configuration can be solved by moving the cars back and forth until a certain car, within this puzzle called "car 0", gets unstuck by arriving at the eastern edge of the grid.

To solve the ACM problem, one has to write a program to solve any given puzzle configuration using the minimum amount of steps; each one of those steps should be printed out afterwards. Both input and required output will be discussed next.

### 3.1 Input

A given input file can contain one or more possible puzzle configurations. A configuration begins with an integer $n$ between 0 and 10, written on a single line, declaring the number of parked cars. The next $n$ lines declare one car each, given in a specific format: $A\ B\ Y\ X$, where

- $A$ = length of the car (2 or 3),
- $B$ = horizontal or vertical orientation (H or V),
- $Y$ = row number of the initial position (0-5),
- $X$ = column number of the initial position (0-5).

"Car 0", the vehicle that has to arrive at the eastern edge of the grid, is always the first car of a configuration. The input file ends with a puzzle configuration consisting of 0 cars.

### 3.2 Output

The output produced by the program should start with the statement "Scenario $k$ requires $x$ moves.", $k$ being the number of the scenario and $x$ the minimum number of required steps. The following $x$ lines describe every single step to solve the puzzle; each line starts with an integer value, giving the "id" of the car that's moved in this step (where 0 is the id of the first car in a configuration, thus car 0), followed by a letter (either "F" or "B") indicating whether the movement of said car is forwards or backwards (based on the cars orientation).

## 4 SOLUTION

There is a multitude of possible ways to solve "Rush Hour". Since any given puzzle configuration has to be solved using the minimum amount of steps, I'm using breadth-first search, or BFS for short, which guarantees an optimal solution. I'll start by describing some of the used objects (based on C#'s OOP approach) and parsing the given input, followed by an in-depth explanation of my algorithmic core plus necessary graph-structures.

### 4.1 Objects

Based on the aforementioned input-file, we can deduct a very basic car-object, consisting of two *integers* "id" and "length", a *Vector2* "position" and an enumeration called *Orientation*.

We also need an object for the 6 by 6 grid, a "parking lot" which consists of a *List<Car>*,which stores all of the cars and an 2d-integer array *int[,] area*, which keeps track of the ids on the current grid - "area" is basically the 6 by 6 grid, storing a 0 for an empty square, or 1 and up for each car-id at a certain position; for simplicity's sake, the id of car 0 is stored as 1 etc., since the "area"-array is initialized as 6 by 6 grid filled with zeroes depicting all empty squares.

### 4.2 Parsing

Before parsing the input, we have to construct a new, empty parking lot object. Given the declared objects, we can then parse a given input file, store our cars accordingly and fill our parking lot.

Read the first line of the given input and store it as integer $n$. Next, go over each of the following $n$ lines and break the specified format $A\ B\ Y\ X$ up into molecular parts; construct a new car object $c$ for each line, specifing *c.id* as the current line-number you're on (of a given configuration), *c.length = A*, *c.orientation = B* (parsed as enumeration), *c.position.x* = *X* and *c.position.y* = *Y*. Once the car has been constructed, store it within the parking lot object.

Afterwards, the *area*-array within the parking lot object gets filled - loop over all cars and store the ids for any given car in the *area*-array at position *area[x, y]*. Since cars are 2 or 3 squares long, you also have to loop over the car's length (using int *i*) and store the car's id at position *area[x+i, y]* or *area[x, y+i]*, depending on the car's orientation.

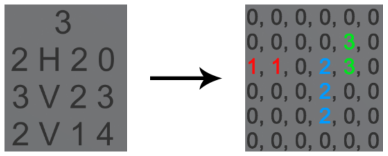


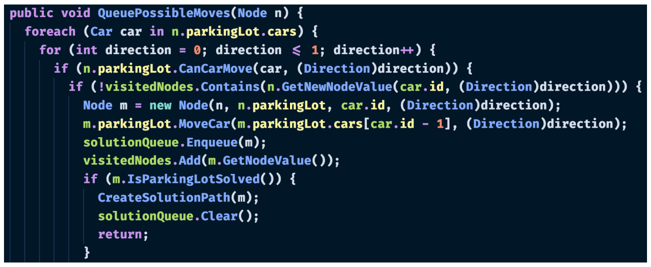
**Figure 1: Sample input and corresponding parking lot-area**

## 4.3 Nodes

My solution to the "Rush Hour" problem utilizes breadth-first search. BFS needs a data structure (tree or graph) of nodes, which it will then traverse and search in $O(|V| + |E|)$. Each node resembles one possible step - a step is simply one single car movement. If some node meets the end condition, i.e. car 0 reaching the rightmost edge, the puzzle configuration is successfully solved. Thus, a node object consists of the following variables: *Node parent*, *ParkingLot parkingLot*, *integer movedCarID* and an enumeration *Direction* called *movedDirection*. The *parent* node is used to link nodes to one another to traverse the solution path (starting from the solved node) at the end of our algorithm. Each node also keeps track of its own parking lot, stored in the *parkingLot* object. Both *movedCarID* and *movedDirection* store relevant information about the car that got moved within this node, i.e. the single step this nodes resembles. BFS needs to keep track of all visited / known nodes, so that endless loops or unnecessarily long search times won't occur. To implement this, you have to create some sort of "node value", keep track of all known node values and check for duplicates before creating a new node. All of these steps should run as fast as possible, which is why I'm using a simple hashing algorithm to construct each node value and a *HashSet<int>* to store them afterwards. To calculate a node value, I initialize an int called *hash* = 7 (or any other prime) and then loop over every single car within the node, to multiply by 17 and to add the *HashCode* of the car's position. I'm using the C#-internal function *GetHashCode()*, which returns an integer hash of a given object. The node values are stored in a *HashSet<int>* (called *visitedNodes*), since the *.Contains()* function, used to check for duplicate node values, runs in $O(1)$.

## 4.4 Algorithm

To utilize BFS, we need a *queue* to store unknown nodes for further, later examination. This *queue* is called *solutionQueue* within my implementation. To successfully solve a given parking lot setup, we start off by creating a *Node*-object for the initial configuration and add it onto our *solutionQueue*. While there are more than zero objects on our queue, dequeue a node and pass it to the BFS-algorithm as node *N*.



```
public void QueuePossibleMoves(Node n) {
  foreach (Car car in n.parkingLot.cars) {
    for (int direction = 0; direction <= 1; direction++) {
      if (n.parkingLot.CanCarMove(car, (Direction)direction)) {
        if (!visitedNodes.Contains(n.GetNewNodeValue(car.id, (Direction)direction))) {
          Node m = new Node(n, n.parkingLot, car.id, (Direction)direction);
          m.parkingLot.MoveCar(m.parkingLot.cars[car.id - 1], (Direction)direction);
          solutionQueue.Enqueue(m);
          visitedNodes.Add(m.GetNodeValue());
          if (m.IsParkingLotSolved()) {
            CreateSolutionPath(m);
            solutionQueue.Clear();
            return;
          }
```

**Figure 2: Algorithmic core - *QueuePossibleMoves(Node n)***

Within the search-algorithm, we loop over every single car of a given node's parking lot, to check if it can move backwards or forwards. If a car can move, we then construct a temporary node value (by moving the car on the given node, calculating the node value and then reverting the car's movement) to check if this step would result in an already known node. However, if the temporary node value is not known, a new node *M* gets created, using *N* as parent, a deep copy of *N*'s parking lot and the id plus direction of the possible car movement. Now, move the car on *M*'s parking lot, add *M* to the *solutionQueue* for later examination and create and store *M*'s node value to our *visitedNodes*. Lastly, check if *M* contains a solved parking lot instance, by checking if car 0 touches the right edge of *M*'s parking lot. If that's the case, you can loop over all *parent*-references, starting with the solved node *M* and ending with the starting configuration, and create the solution output accordingly. Also clear the *solutionQueue* and return our BFS-algorithm, since an optimal solution has been found.

## 4.5 Benchmarks

To measure the performance of my implementation, I solved the ACM-provided standard input ten times and measured the elapsed time. On top of that, I followed the same procedure for the hardest possible "Rush Hour"-problem for a 6 by 6 grid. The hardest possible "Rush Hour" puzzle was described and computed by Sébastien Collette, Jean-François Raskin and Frédéric Servais in 2006, and requires 93 individual steps [1]. Since the hardest configuration utilizes 13 instead of 10 cars, I've adapted my implementation accordingly.

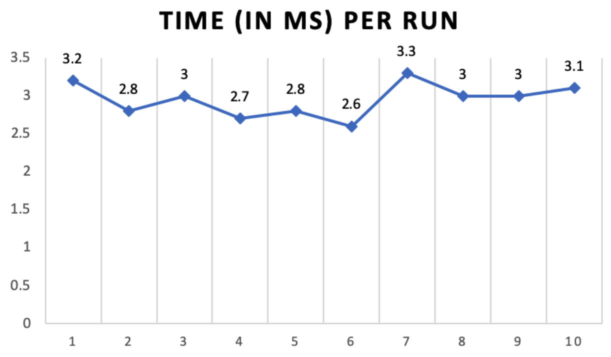


**Figure 3: Required time (milliseconds) to solve the given ACM-input configuration, split into ten runs**

Solving the given ACM-input configuration takes 2.95ms on average. This configuration is solved in 16 steps and compares and stores 1072 nodes (using node values).

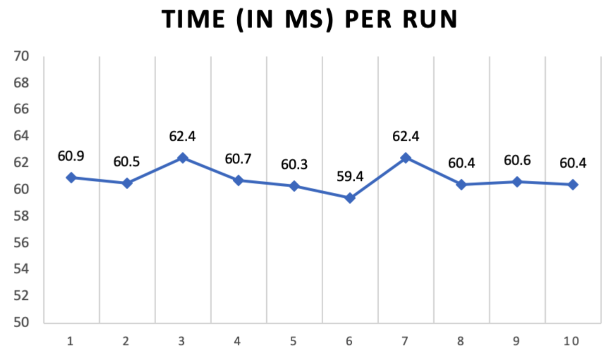


**Figure 4: Required time (milliseconds) to solve the hardest possible configuration, split into ten runs**

The hardest possible configuration is solved in 60.8ms on average. 93 steps and 19256 nodes are required. Given that this is the hardest possible configuration, one can deduce that every other solvable configuration can be solved in under 60.8ms on average.

## 5 UNITY-IMPLEMENTATION

I've used the Unity Engine to create a GUI surrounding my aforementioned search-algorithm. "Rush Hour" can be visualized quite nicely in my opinion, hence the graphical approach. Besides having a visual output, I also added a visual input-editor for faster and easier puzzle creation. In this section, I will examine my GUI and explain its essential parts.

### 5.1 Parking lot

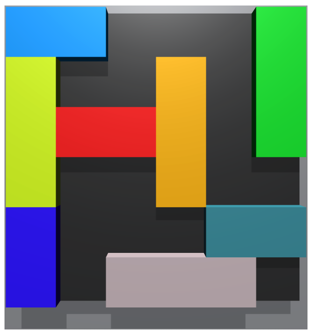
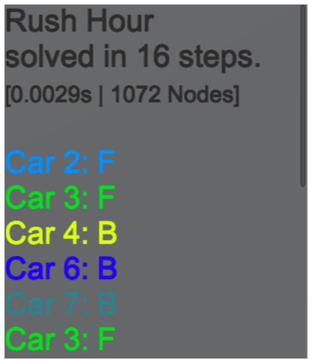


**Figure 5: Exemplary parking lot configuration, consisting of 8 cars**

The parking lot is represented by a dark-gray 6 by 6 square. On top of said square are 36 individual 1 by 1 squares, each representing a possible car-part. Internally, I call these squares *ParkingTile*. Each *ParkingTile* has a position on the 3D parking lot, which coincides with a (x,y)-coordinate on the internal parking lot 2D-integer array. Thus, each *ParkingTile* grabs its corresponding integer value from the parking lot 2D-int array (starting at coordinate (0,0) in the upper left corner, going up to (5,5) in the lower right corner). If the corresponding integer is zero, disable the connected 3D-mesh; otherwise, enable the mesh and change its material color to the connected car-color. The car-colors are globally accessible and stored as a list within the main script.



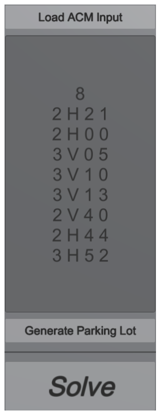
```
Rush Hour
solved in 16 steps.
[0.0029s | 1072 Nodes]

Car 2: F
Car 3: F
Car 4: B
Car 6: B
Car 7: B
Car 3: F
```

**Figure 6: Text output for a given puzzle configuration**

Besides showing each individual solution step on this 3D parking lot, I've also added a traditional text output, as required by the ACM problem description. The text output is located in the lower right corner of the GUI and meets the criteria of the given output format. Internally, it's using a scroll-view with an attached text-component.

The actual solution steps are color-coded (based on the aforementioned car-colors) for easier reading and comprehension. On top of the required output format, the text box also tells you how much time it took to solve the given configuration and how many nodes were visited.

### 5.2 Text input



```
8
2 H 2 1
2 H 0 0
3 V 0 5
3 V 1 0
3 V 1 3
2 V 4 0
2 H 4 4
3 H 5 2
```

**Figure 7: Left hand side of the GUI; used for manual text input and solving the current puzzle configuration**

On the GUI's left hand side, you'll find a text input and three buttons. Firstly, the button labeled "Load ACM Input" loads the ACM-provided puzzle configuration and creates all necessary internal objects. The text input underneath accepts the ACM-conform input format, explained in "3.1 Input", and is used to enter and define a starting puzzle configuration. The button labeled "Generate Parking Lot" generates the internal car objects and parking lot structures, plus the visual 3D-parking lot, based on the user-defined text input. Lastly the button labeled "Solve" searches and finds an optimal solution for any given puzzle configuration, utilizing the methods mentioned in "4.4 Algorithm".

### 5.3 Level editor

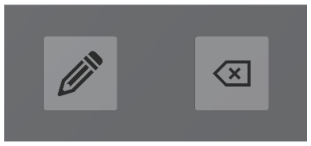


**Figure 8: Level editor panel**

Since manually typing a level configuration is somewhat tedious, I've also added a level editor. To enable the editor, simply press on the pencil-icon found in the upper right corner. Once the editor-mode is enabled, you can hover over the *ParkingTiles* and place individual car-parts by pressing the left mouse button. Switch between the car id that you want to place via the number row, with number 1 being the "escaping" car with id 1, up to number 0 being car 10 and "ß" (on a German keyboard) representing id 0 (used to clear individual *ParkingTiles*).

Next to the pencil-icon, you can find an erase-button, which simply clears the entire parking lot. Internally, a new parking lot object gets created using an empty input-text.

### 5.4 Solution playback

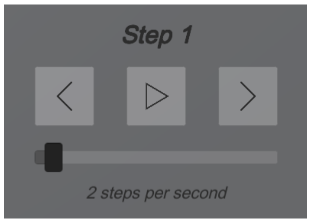


**Figure 9: Solution playback panel**

Using the playback panel, the user can either click through each individual solution step by pressing the arrow buttons on the right and left side, or start an auto-playback mode, which cycles through each step automatically. The user can also change the playback-speed using the slider underneath the three main buttons, starting at 1 step per second, up to a maximum of 20 steps per second. Stepping through a solution means stepping through the individual nodes of a solution path. After solving a puzzle configuration, a solution path (a *List<Node>*) gets created, which contains all nodes on the path from the initial node, up to the solved end node. With every step, the currently shown 3D parking lot (and its underlying *ParkingLot*-object) gets overwritten using the data from the currently selected node; internally, an integer index-pointer increases / decreases upon clicking the arrow buttons and thus the next / previous node on the created solution path gets selected for display.

### 5.5 Shortcuts

By pressing the keys $N$, $M$ and $K$, the user can quickly load some interesting puzzle configurations. Keycode $N$ loads a "medium" level, which interestingly enough visits less nodes but needs more total steps than the given ACM-input. Keycode $M$ loads a "hard" level, which requires 83 steps and 8676 nodes. Using the keycode $K$, the user can load up the aforementioned hardest possible puzzle configuration. In the background scenery, behind the UI and the 3D parking lot, you can see miniature cars "driving" around. You can enable / disable all of them by pressing the keycode $P$.

## 6 CONCLUSION

During this seminar I solved the ACM-provided "Rush Hour" problem (using breadth-first search) and expanded it by programming a graphical tool (using the Unity game engine) for creating, editing and solving puzzle scenarios. Overall, the proposed solution seems promising, especially the fact that any given puzzle scenario can be solved in approximately $\leq 60.8ms$. Optimizing the algorithmic core was the most challenging and educational part. Starting with a naive approach of storing entire *Node*-objects to keep track of visited nodes, searching and finding solutions took a couple seconds. Rethinking the act of storing nodes (switching to *HashSet*) and creating a simple hash function to create *integer* node values led to the biggest performance improvements.

This project is not finished though - one thing I'd like to add in the future is the possibility to change the parking lot size during runtime. One could add two sliders to the GUI, used for changing the height and the width of the parking lot. Thus, a user could create and solve smaller / larger puzzles than the fixed 6 by 6 ones, including rectangular grids as well. On top of that, one could remove the arbitrary limit of 10 cars (which I've already expanded to 13, due to the mentioned hardest possible puzzle) and instead allow as many cars as possible, with the only upper limit being the current grid size. Another addition I'd like to make is implementing different search algorithms to choose from, besides breadth-first search. For example, adding $A^*$ and *Dijkstra's* algorithm, and then comparing the runtime of all three for the same puzzle scenario seems like an interesting task.

## 7 ADDENDUM

My entire seminar documents (Unity-project, presentation slides, this paper) are available for download on GitHub (MIT-licensed).
https://github.com/mapluisch/Proseminar-ACM-Rush-Hour

## REFERENCES

[1] S. Collette, J.-F. Raskin, and F. Servais. On the Symbolic Computation of the Hardest Configurations of the RUSH HOUR Game. In *Proceedings of the 5th International Conference on Computers and Games (CG 2006)*, pages 220–233, 2006.
[2] ACM History Committee. ACM History. https://www.acm.org/about-acm/acm-history - visited on 22.02.2020.
[3] ACM ICPC. Rush Hour. https://db.inf.uni-tuebingen.de/staticfiles/teaching/ws0809/fun-problems/Rush-Hour.pdf - visited on 22.02.2020.
[4] Misc. Wikipedia-Authors. International Collegiate Programming Contest. https://de.wikipedia.org/wiki/International_Collegiate_Programming_Contest - visited on 22.02.2020.