# INF421 PROGRAMMING PROJECT
# SOLVER OF LOGIPIX

MARIE ALBENQUE
albenque@lix.polytechnique.fr

*The required programming language for this project is Java. All the code you submit must be commented and indented properly and must be organized logically in some classes.*

The purpose of this programming project is to write an efficient solver for *rush hour puzzles*, such as the one on the left. The goal of a rush hour puzzle is to help the red car to escape the traffic and reach the exit (on the right side). Horizontal cars can be moved left and right and vertical cars can be moved up and down. Cars are not allowed to move through other cars. One *move* is the displacement of one car to another eligible location.
A *solution* to a rush hour puzzle is a sequence of moves that allows the red car to exit. An *optimal solution* is a solution with fewest possible moves.

To get a better understanding of the game, feel free to try and play rush hour, for instance at https://www.crazygames.com/game/rush-hour-online.

This programming project is of increasing difficulty. The first part consists in creating classes that can handle rush hour puzzles, in particular instantiate and display them. The second part consists in writing a brute force algorithm that solves a rush hour puzzle. The third part designs heuristics to try to increase the efficiency of your solver.

## 1. Setting up the game

1.1. **Reading a file and instanciate a game.** A *state* of the game is a configuration of non-overlapping vehicles which can be of length 2 and 3 and which can be horizontal or vertical.

An initial state of the game will be given by a file of the following format. The first line is the size of the grid (which is square by assumption). The second line gives the number of vehicles.

Then, there is a line for each vehicle: first, we give an integral label to the vehicle, then its orientation `h` or `v` for horizontal or vertical, its length (2 or 3) and finally the absissa and the ordinate of its topleft cell. We list columns of the grid from left to right and lines from the grid from top to bottom. We always assume that the vehicle number 1 is the car that needs to exit the traffic (a.k.a the red car).

For instance, the text file encoding the initial state described above is:

```
6
8
1 h 2 2 3
2 h 2 1 1
3 h 2 5 5
4 h 3 3 6
5 v 3 6 1
6 v 3 1 2
7 v 3 4 2
8 v 2 1 5
```

**Task 1.** *Write a method that takes such a file as input, checks that it is a valid input (i.e. that no two vehicles intersect) and initializes a rush hour game.*

**Task 2.** *Write a method that displays a state of the rush hour game. This can be a very simple representation or a fancier one if you feel like it (but there won't be extra credit for fancy graphical interface).*

## 2. Solving the game: a first brute force solution

Flake and Baum [FB02] proved that deciding whether a rush hour game of size $n$ has a solution is P-SPACE complete. Hence, there is absolutely no hope to come up with a polynomial solution!

To solve this game, we are going to consider first a brute force solution and perform and exhaustive search of all the possible sequence of moves admissible from an initial configuration and deduce from that an optimal solution.

### 2.1. Computing the length of a shortest solution.

**Task 3.** *Describe the brute-force algorithm and write its pseudocode.*
*Be careful that, it is important to check whether a sequence of moves produces a state which was already explored and not to explore some sequence of moves which cannot produce a better solution that one already obtained.*
*Make explicit and justify your choice of data structures you use (in particular the one used to store the states already explored).*

Before writing more code, pay attention to the organization of your classes. Read the project until the end to see what kind of features you are going to need.

**Task 4.** *Write a method that returns all the possible moves of a given state. Be careful that for instance that three admissible moves can be performed for the green car on line 1 of the example: either moving it from 1, 2 or 3 cases to the right.*

**Task 5.** *Implement the data structure that you choose in Task 3 to store the explored states.*

Now, you can combine the methods you wrote to come up with a program that reads a file as an input and computes the number of steps of a shortest solutioin that starts from the initial configuration given in the file.

**Task 6.** *Write that program.*

You can test your code with the files available at:
http://www.lix.polytechnique.fr/~albenque/PI421/ExRushHour.tgz
Give and compare the time of executions of your algorithm in your report.

### 2.2. Reconstruction of the solution.

It is nice to know that there exists a solution with a given length but it would be actually even nicer to exhibit such a solution. A good way to produce a solution is actually to reconstruct it.

For each explored state, you have to "remember" how you obtained this state the first time. In other words what was the last move you did before getting this step, or what was the state of the game just after this last move. If you store this information, then starting from the final state you can reconstruct backwards one seuqence of moves of minimal length.

**Task 7.** *Modify the program of task 6 so that it prints the solution at the end.*

The way you print the solution can be really basic. You can display some instructions "Move the car 2 to 3 cases to the right"... or if you feel like it, produce a more animated version of the solution.

## 3. Approach based on heuristics.

### 3.1. First steps with heuristics.

To lower the execution time of your algorithm, we are going to introduce heuristics. A *heuristic* in our case is a function $h$ with associates to each state, the estimated length of a solution starting from this state. In particular, if $s_final$ denotes the state where the red car has exited traffic, then $h(s_final)$ must be equal to 0.

More precisely, we say that $h$ is an *admissible heuristic* if for any state $s$, $h(s)$ is a lower bound for the length of a solution starting from $s$. A second, slightly stronger condition is the consistency of heuristics. We say that $h$ is a *consistent heuristic* if for every pair of states $s$ and $s'$,

$$h(s) \leq h(s') + k_{s,s'},$$

where $k_{s,s'}$ is the minimal number moves to go from $s$ to $s'$. Assume that a consistent heuristic $h$ is given for rush hour.

**Task 8.** *Modify the pseudo-code of Task 3 to use the heuristic $h$ to lower the execution time of your algorithm. This heuristic can be used to stop the exploration of some sequence of moves. Prove that your algorithm is correct.*

A trivial heuristic is the function $h$ which is constant and equal to 0. What algorithm do you get if you apply the pseudocode given in Task 8 with $h = 0$ ?

A more interesting heuristic is to associate to a step $s$ the number of cars that are situated between the red car and the exit. For instance, the value of this heuristic is 2 in our example on Page 1.

**Task 9.** *Prove that $h$ is consistent.*

**Task 10.** *Implement the code described in Task 8 with the heuristic given above.*

Does the execution time improves significantly ?

3.2. **New heuristics.** Now it is your time to come up with new heuristics (at least one is expected !).

**Task 11.** *Find a new heuristic, prove that it is consistent and implement it. Compare the performance of the different algorithms on the given data-sets (or on other data sets.)*

**Evaluation and trivia.**

- You will get points for the correctness of the programs, points for the cleverness of the heuristics you designed, and points for the quality of the report and of the defense.
- The report must present concisely all the algorithms that have been implemented and explored; discuss their complexity and evaluate them on the provided data-sets. Defense will be 15 minutes long and will involve a live demo of the tool with data-sets possibly different from the ones provided and a discussion of the algorithmic choices.

## References

[FB02] Gary William Flake and Eric B. Baum. Rush hour is pspace-complete, or "why you should generously tip parking lot attendants". *Theoretical Computer Science*, 270(1):895 – 911, 2002.