1 Setting up the game

In order to implement a state of the RushHour game, we have created a **Game** class containing four fields (there is some redundancy between these fields, but we thought it was easier to do so). The first one is the size of the grid, the second one is the number of vehicles, the third one is an Array of **Car** and the last one is the grid, which is an array of Integer arrays representing the game state. The **Car** class represents a car with four fields : its number, its position(x and y), its length and its orientation (vertical or horizontal).

The **Game** class has two constructors : the first one creates an instance of the class with a *.txt* file, and the other one creates an instance of the class directly with the class fields. This second constructor will be used after in order to obtain the different admissible moves from the current state.

In the initialization of the game, we check that there are no overlapping vehicles. Otherwise, we throw an **OverlappingException**, which has been created in a separate class.

We also define a $draw()$ function in the **Game** class which prints the current state of the game. With the file $RushHour1.txt$, this function gives us the following result :

```
-------------------------
| 2 | 2 | 0 | 0 | 0 | 5 |
-------------------------
| 6 | 0 | 0 | 7 | 0 | 5 |
-------------------------
| 6 | 1 | 1 | 7 | 0 | 5 |
-------------------------
| 6 | 0 | 0 | 7 | 0 | 0 |
-------------------------
| 8 | 0 | 0 | 0 | 3 | 3 |
-------------------------
| 8 | 0 | 4 | 4 | 4 | 0 |
-------------------------
```

FIGURE 1 – Representation of an instance of the game, a 0 represents an empty space

2 Solving the game : A first brute force solution

The main idea of the brute force approach is to consider the game as a graph, and thus perform some basic graph algorithm. In the current problem, vertices represent game states. Two game states are connected if we can go from one to the other with only one move.

A first brute force solution to solve the problem is to use a Breadth-First Search (BFS) algorithm. The BFS algorithm guarantees us to find the shortest path between the starting configuration and the ending configuration (where the red car can get out of the parking) if a solution exists. If we had chosen the DFS algorithm, we would not be sure to find the

shortest solution, but only the first solution encountered. We implement the BFS algorithm in the **BreadthFirst.java** file.

Concerning data structures, we use a Queue to store vertices that need to be visited and a HashMap to store visited vertices and their distance from the starting configuration. This last choice allows us to check if a vertex has already been visited and get its distance from the source in a constant time. We also had to code a *hashcode* function and override the *equals* method in the **Game** class to be able to properly use the HashMap.

The complexity of the present algorithm is in $\mathcal{O}(n+m)$ where n is the number of vertices in our graph, and m the number of edges.

Algorithm 1: Breadth-First Search pseudocode

Data: A game configuration *source*
Result: the minimal distance between G and the final configuration if a path exists

```
1  HashMap < Game, Integer > visited
2  visited.put(source, 0)
3  Queue q
4  q.add(source)
5  while q is not empty do
6      g ← q.poll()
7      if g is solution then
8          return visited.get(g)
9      for Game next in every non visited neighbour configuration of g do
10         visited.add(next, visited.get(g) + 1)
11         q.add(next)
12 return -1                    // in case there is no solution to the problem
```

We implement the method *possibleGrids*() in the class **Game**, which returns a $LinkedList < Game >$ of all the possible game states that can be reached from the current game state with only one move.

We have then compared the different time of executions of the BFS algorithm in the following table, by calculating an average execution time over 10 executions thanks to the Java function *System.currentTimeMillis*().

File	BFS
RushHour1.txt	42
RushHour2.txt	2
RushHour3.txt	16
RushHour4.txt	9
RushHour5.txt	86
RushHour6.txt	544

TABLE 1 – Average execution time (in ms) of BFS for different starting situations

Finally, in order to print the solution at the end of the program, we have added a pointer $HashMap < Game, Game >$ to store the path of the algorithm. The value of a given key is the state of the game preceding the key state. Then, starting from the final state, we can reconstruct backwards one sequence of moves of minimal length leading to the solution.

3 Approach based on heuristics

Firstly, let us assume that we have a consistent heuristic h. Using this heuristic, we can stop the exploration of certain sequences of move to lower the execution time of our algorithm. The key idea of the heuristic-based approach is to use a priority queue instead of a simple queue in the BFS algorithm so that vertices with lower heuristic value (which are the vertices that seem closer to the solution) are visited in priority.

Algorithm 2: Modified pseudocode using heuristics

Data: A game configuration *source*
Result: the minimal distance between G and the final configuration if a path exists

```
1  visited ← HashMap < Game, Integer >
2  visited.put(source, 0)
3  q ← PriorityQueue < Game > (newHeuristicComparator())
4  q.add(source)
5  while q is not empty do
6      g ← q.poll()
7      if g is solution then
8          return visited.get(g)
9      for Game next in every non visited neighbour configuration of g do
10         visited.add(next, visited.get(g) + 1)
11         q.add(next)   // executed in O(log n) where n is the size of the queue
12 return -1                    // in case there is no solution to the problem
```

To be able to use the priority queue properly, we need to code a **HeuristicComparator** class (implementing the **Comparator** class of Java) which provides the priority queue a way to compare game states based on their heuristic values. The comparator compares the sum $h(x) + visited.get(x)$ and $h(y) + visited.get(y)$ for two visited game states x and y, where *visited* is the HashMap containing visited vertices and their distance (the number of moves) to the game starting state. It is important to note there that every vertex that is added to the queue is also added to the visited HashMap, so that there is no exception thrown when calling $visited.get(x)$ in the comparator. The vertex at the top of the priority queue is the one with the lowest heuristic value (i.e. the one that seem the closest to the solution).

Correctness of the algorithm

First of all, the algorithm terminates because each vertex is added at most once to the PriorityQueue and one vertex is polled out of the queue at each iteration.

To prove the correctness of the algorithm, it is useful to note that the present algorithm is exactly the Dijkstra algorithm, with a weight for the oriented edge (u, v) equal to $h(v)-h(u)+1$. Indeed, for a given node u, we can define $f(u) = h(u) + g(u)$ (where h is the heuristic and $g(u)$ the optimal distance known from the source s to u) the currently known optimal weight of a solution passing through u. For two adjacent nodes u and v we have :

$$\begin{aligned} f(v) &= h(v) + g(v) \\ &= h(v) + g(u) + 1 \\ &= g(u) + h(u) + h(v) - h(u) + w(u, v) \\ &= f(u) + w(u, v) \qquad \text{where } w(u, v) = h(v) - h(u) + 1 \end{aligned}$$

which gives us the desired result for the weight of the oriented edge (u, v).

Since the given heuristic h is consistent, $h(u) \leq h(v) + 1$ which implies that $0 \leq w(u, v)$. Ultimately, using the correctness of the Dijkstra algorithm for graphs with positive edge weight, we can affirm that our algorithm is correct.

In the case of the trivial heuristic $h = 0$, the algorithm acts like the BFS algorithm because the comparator used calculates the priority of each node simply based on their distance to the source, which is exactly what the queue in BFS algorithm does.

Let's show that the given heuristic counting the number of vehicles that block the way to the exit is consistent. Let s and s' be two game states. There are at least $|h(s) - h(s')|$ cars that have moved to go from s to s'. Therefore we have $|h(s) - h(s')| \leq k_{s,s'}$, where $k_{s,s'}$ is the minimal number moves to go from s to s'. This gives us the two following inequalities that prove the consistency of the heuristic.

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

We can see in the following table that the execution time is only slightly reduced. Indeed, the time complexity of this algorithm is $\mathcal{O}(m \log n)$ (not really better) because there are at most $\mathcal{O}(m)$ elements in the queue, and the *add* / *poll* operations are in logarithmic time $\mathcal{O}(\log(m))$. Then, we have the relation $m \leq n^2$ which justifies the given formula.

File	BFS	h1
RushHour1.txt	42	24
RushHour2.txt	2	0
RushHour3.txt	16	16
RushHour4.txt	9	9
RushHour5.txt	86	86
RushHour6.txt	544	551

TABLE 2 – Comparison of execution time (in ms) of the two different approaches

New heuristics !

We will now use a new, more precise heuristic. This new heuristic is similar to the previous one, but we also add the minimum number of vehicles blocking the counted vehicles on their column. The value of this heuristic is 4 in the example given in Figure 1 page 2, because two cars are blocking the red car on its way to the exit, and each blocking car are blocked by at least one car vertically.

We can convince ourselves that this heuristic is admissible because we need to move at least the number of cars blocking the red car to the exit plus the minimum of the number of cars blocking those cars. This heuristic is also consistent because in order to move from a game state s to another game state s', you need to move at least $|h(s) - h(s')|$ cars. Thus, the inequality $|h(s) - h(s')| \leq k_{s,s'}$ obtained in the previous section still stands and guarantees the consistency of the heuristic.

We get the following (barely better) performance for this new heuristic :

File	BFS	h1	h2
RushHour1.txt	42	24	12
RushHour2.txt	2	0	1
RushHour3.txt	16	16	20
RushHour4.txt	9	9	9
RushHour5.txt	86	86	73
RushHour6.txt	544	551	681

TABLE 3 – Comparison of execution time (in ms) of the three algorithms

However, if we look at the number of nodes visited during the search (Table 4), we can see that algorithms using heuristics can find the solution visiting less nodes than the simple BFS algorithm. It is particularly true for the second heuristic which allows us to skip hundreds of nodes in some cases.

File	BFS	h1	h2
RushHour1.txt	1079	1057	1036
RushHour2.txt	75	73	71
RushHour3.txt	858	813	789
RushHour4.txt	601	569	537
RushHour5.txt	3202	3013	3153
RushHour6.txt	15898	13153	11997

TABLE 4 – Comparison of the number of nodes visited by the three algorithms

We could take this new heuristic a step further, by also calculating the cars blocking the cars blocking the cars which are blocking the red car, and so on. We could implement such a heuristic recursively. However, even if this new heuristic is way more precise, each iteration would require much more time to compute. Thus, we decided to stick with our heuristic $h2$, which seems to be the right balance between a precise heuristic and a computationally cheap heuristic, as $h2$ is already struggling with *RushHour*6.*txt* in terms of execution time.

Verifying the consistency

Additionally, we can make sure that our two heuristics are consistent, by checking the length of the solution calculated with each different approach. Those results are detailed in Table 5. We can see that our solutions are well calculated by both $h1$ and $h2$, since the length of the computed solution is equal to the one computed by *BFS* which is optimal.

File	BFS	h1	h2
RushHour1.txt	8	8	8
RushHour2.txt	32	32	32
RushHour3.txt	32	32	32
RushHour4.txt	37	37	37
RushHour5.txt	51	51	51
RushHour6.txt	49	49	49

TABLE 5 – Comparison of the length of the solution calculated by the three algorithms