Ecole polytechnique

INF 421

Rush Hour Solver

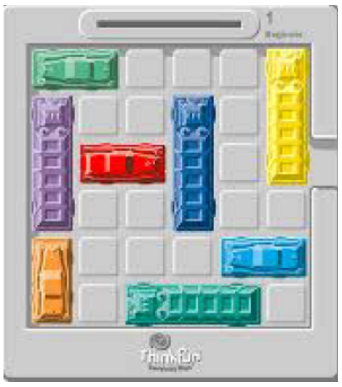
Jeremy Marquis, Franco Rabec

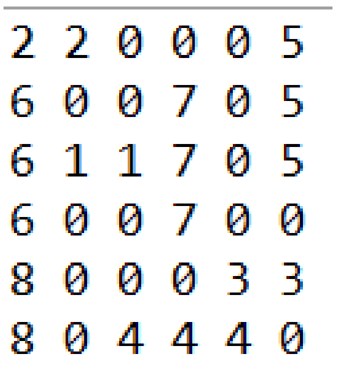
January 22, 2018

l'X
ÉCOLE
POLYTECHNIQUE
UNIVERSITÉ PARIS-SACLAY

1 Setting up the game

We first created a class called **Vehicle** in order to represent a vehicle in a Rush Hour game. The class vehicle has the fields *name, orientation, length, absissa ordinate.* We represent a state of the game thanks to class called **RushHour**. This class contains three fields. The first one called *size* represents the size of the grid. The second one is an array of **Vehicles** where we store the vehicle of the game. The third one the field *grid* is here to represent a state of the game thanks to a double array. Figure 1 explains how we represent the game thanks to the grid : the function *display* prints the double array *grid.* The class **RushHour** has two constructors the first one builds a instance of the class thanks to a txt file. The seconde one builds it with an `Array` of **Vehicle**.





```
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: Real state of the game and the outpout of the function display

2 Solving the game : a first brute-force solution

The key idea to find a solution thanks to brute force is using BFS. The vertices of the graph represent game states, vertices A and B are connected if we can go from A to B with one move (and vice versa), thanks to BFS we are sure we get the shortest path. Hence the program doesn't have to store the all graph but only to compute the neighbors of the vertex the algorithm is currently visiting. The BFS is implemented in a class **BFS**. The pseudocode is given on page 2. Concerning the data structures : we use a Queue to store the states to visit and a HashMap to store the states we've already visited, we associate a value that corresponds to the number of moves from R to the key. To store the states already explored we use a HashMap because it allows us to check if a vertex belongs to it in constant time, we can add a vertex in constant time and we can get the distance between

R and a state already visited in constant time too. In order to properly use the HashMap we had to create a function *hashcode* that associates to each state a hashcode (ideally it is injective). We had to override the methods equals in order to compare two states properly (and not by their addresses in memory as it is done by default). The complexity of this algorithm is $\mathcal{O}(m+n)$. Where n is the number of states that are as far or less far than the solution from the beginning configuration, and m the number of edges between those n states.

Algorithm 1: Bruteforce solution using DFS

```
Data: $R$ a state of a game of the class RushHour
Result: the minimal number of states to solve the problem
begin
    Hashmap $<$ RushHour, int $>$ visited
    visited.put($R, 0$)
    Queue $Q$
    $Q$.add($R$)
    while $Q$ is not empty do
        $A = Q$.poll()
        if $A$ is solution then
            return visited.getValue($A$)
        end
        for $S \in$ Neighbors($A$) do
            if $S \notin$ visited then
                visited.put($S$, visited.getValue($A$) + 1)
                Q.add($S$)
            end
        end
    end
    return -1 ; /* means there is no solution reachable from $R$ */
end
```

We wrote a method *AllPossibleMoves(R)* in the class **RushHour** this method returns a `LinkedList` containing the states that can be reached thanks to one from the state *R*. For this we implemented a third constructor for the class **RushHour**. This constructor takes an instance of the class, and creates a new instance that is like the former except that one vehicle has moved. In Table 1 we put the results an execution time of the algorithm for different start configurations.

We then use a `HashMap` to store the path between the beginning and the solution. The value is the predecessor of the key.

Configuration	Result	Execution time (ms)
RushHour1.txt	8	27
RushHour2.txt	32	3
RushHour3.txt	32	19
RushHour4.txt	37	12
RushHour5.txt	51	65
RushHour6.txt	49	338
RushHour7.txt	8	26
RushHour8.txt	49	324

Table 1: Results and execution times for brute force algorithm

3 Approach based on heuristics

We use the heuristic given to lower the execution time. The pseudocode can be seen in Algorithm 2. We will now prove that the algorithm is correct if the heuristic is consistent. Let R be the beginning state. If S is another state we note $d(R,S)$ the minimal number of moves in order to go from R to S. The algorithm 2 is exactly the Dijkstra algorithm with the weight of the edge (A,B) (now oriented) is not 1 as for brute-force but $1+(h(B)-h(A))$. The weight is never negative because if A and B are connected $h(B)-h(A)\geq -1 \Rightarrow 1+h(B)-h(A)\geq 0$. So the algorithm 2 computes the short path with the given distance, because Dijkstra algorithm is correct. Let $R=X_0\cdots X_k=S$ the path computed by algorithm 2, $R=X'_0,\cdots X'_l=S$ another path. The weight of the first path is $\sum_{i=0}^{k-1}(1+h(X_{i+1})-h(X_i))=k+h(S)-h(R)$, the weight of the second path is $l+h(S)-h(R)$ we have :

$$\begin{aligned} & k+h(S)-h(R)\leq l+h(S)-h(R) \\ \Rightarrow \quad & k\leq l \end{aligned}$$

Algorithm 2: Using the heuristic given

```
Data: $R$ a state of a game of the class RushHour
Result: the minimal number of states to solve the problem
begin
    Hashmap < RushHour, int > visited
    visited.put($(R, 0)$)
    PriorityQueue $Q$
    /* $R \leq S \quad iff \quad h(R) + Visited.get(R) \leq h(S) + Visited.get(S)$ */
    $Q$.add($(R)$)
    while $Q$ is not empty do
        $A = Q$.poll()
        if $A$ is solution then
            return visited.$getValue(A)$
        end
        for $S \in$ Neighbors($A$) do
            if $S \notin$ visited then
                visited.put($S$, visited.getValue($A$) + 1)
                Q.add($(S)$)
            end
        end
    end
    return -1 ; /* means there is no solution reachable from $R$ */
end
```

We will prove that the heuristic given is consistent. Let S, S' be two game states. If $S = S'$ we have nothing to prove because $k_{s,s'}$ is positive. If $h(S) \neq h(S')$ we note $\alpha = h(S) - h(S')$ there is at least $|\alpha|$ cars that have moved to go from S to S' or from S' to S. So $|h(S) - h(S')| = \alpha \leq k_{S,S'}$We have then :

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

The algorithm using this heuristic is implemented on a class called **Heuristic1**. It's complexity is $\mathcal{O}(mlog(n))$. It is slightly more than the complexity of BFS. But it is not impossible that it sometimes improves the running time.

We have found another heuristic for the problem that is more precise that the former. The heuristic h' will compute h(S) for a given state S as follows. We first compute the cars that are between the red one and the exit. We then compute the number h_1 for each car between the red one and the exit, we check if it fits above the path of the red car regardless the other cars. If it does we add h_1 the number of cars obstructing him to do so. If it doesn't we add h_1 the number of cars obstructing it from going below the path. If the same car obstruct several

Configuration	brute force (ms)	heuristic h (ms)	heuristic h' (ms)
RushHour1.txt	27	20	14
RushHour2.txt	3	3	2
RushHour3.txt	19	19	13
RushHour4.txt	12	12	9
RushHour5.txt	65	54	87
RushHour6.txt	338	448	485
RushHour7.txt	26	22	11
RushHour8.txt	324	464	476

cars he will be counted once. We compute the number h_2 as follows for each car between the red one and the exit, we check if it fits below the path of the red car regardless the other cars. If it does we add h_1 the number of cars obstructing him to do so. If it doesn't we add h_1 the number of cars obstructing it from going above the path. If the same car obstruct several cars he will be counted once. We define $h'(S) = h + min(h_1, h_2)$. The heuristic is consistent. Let S, S' be two different states if $h'(S) = h'(S')$ there is nothing to prove. We note $\alpha = h'(S) - h'(S')$. To go from the state S to S' we must move $|h(S) - h(S')|$ to move those cars we must at least move once the car in the way and the car blocking the car in the way from moving. Finally we give the comparison of the execution times. The heuristic h' performs slightly better than h.