Unblock Project Report

Mohamed Tnani

February 3, 2025

Abstract

This report presents a comprehensive solution to the Rush Hour puzzle, focusing on both brute-force and heuristic-based approaches. We discuss the design choices for representing the game state, implementing efficient search algorithms, and developing admissible heuristics. Experimental results demonstrate the performance of different approaches across various puzzle configurations. The implementation leverages breadth-first search and A* algorithms with carefully designed state representations and heuristic functions.

Introduction

The purpose of this project is to write an efficient solver for unblock me (also known as rush hour) puzzles, such as the one shown in Figure 1. The goal of a 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 puzzle is a sequence of moves that allows the red car to exit. An optimal solution is a solution with fewest possible moves.

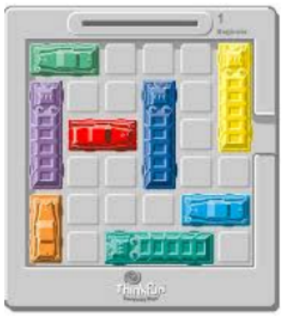

Figure 1 – Example of rush hour puzzle

1 Setting up the Game

A state of the game is a configuration of nonoverlapping 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 in Figure 1 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
```

1.1 Game Initialization

To initialize the game, we first construct the `Game` class with the following attributes :

```
1: Class Game
2:     Public:
3:         Integer nbCars                                ▷ Number of cars
4:         Integer size                                  ▷ Size of the grid
5:         Vector of Vector of Integers positions        ▷ Stores car positions
6:         Vector of Booleans horiz                      ▷ Car orientation
7:         Vector of Integers lengths                    ▷ Lengths of the cars
```

The input string is split into multiple lines. The first two lines of the input specify the grid size (`size`) and the number of cars (`nbCars`), which are then extracted to define the game environment. Empty lines are skipped, and the program extracts the index, orientation, length, row, and column of each car. The extracted data is then stored in vectors that keep track of the car orientations (`horiz`), lengths (`lengths`), and positions (`positions`).

Splitting the input string takes $O(n)$, where n is the length of the input string. Parsing the car data takes $O(m)$, where m is the number of cars. Therefore, the overall time complexity is $O(n + m)$.

1.2 Game State Tracking

To keep track of the state of the game, we add a new attribute to the `Game` class :

```
1: Vector of Vector of Booleans free          ▷ Grid occupancy status
```

We define a method `update_free()` that updates the `free` grid, setting a cell to `true` if it is unoccupied and to `false` if it is occupied.

We also create a `State` class that represents a particular game state :

```
1: Class State
2:     Public:
3:         Game G                      ▷ Current board configuration
4:         Integer moves               ▷ Number of moves done
```

2 Solving the Game : A first Brut Force Solution

Flake and Baum [1] 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, perform an exhaustive search of all the possible sequence of moves admissible from an initial configuration and deduce from that an optimal solution.

The Rush Hour puzzle can be modeled as a graph where each game state is a node and each legal move is an edge. Breadth-first search (BFS) efficiently finds the shortest solution by exploring all possible moves in order of increasing depth, guaranteeing the first solution found uses the minimal number of moves. Hence, we need first a method to search for adjacent states.

A simple approach to find next possible states would be to move each car i in two directions :

- **Right (or Down)** when `right = true`
- **Left (or Up)** when `right = false`

Once we did that, we can search for all next states by moving each car right (down) and left (up) to find the solution. Below, is the pseudocode for both methods.

2.1 BFS Algorithm and Choice of data structure

Now that we can search for next possible states given an initial state of the game, we are ready to implement the BFS Algorithm to solve the problem. However, we need to use an efficient data structure to save explored states.

```
function StatesInDirection($NextStates$, $i$, $right$)    ▷ right is equivalent to down when the car i is vertical
    $col, row \leftarrow$ positions of the car i
    $\delta \leftarrow 1$ if $right$ else $-1$
    $limit \leftarrow G.size$
    while true do
        if $G.horiz[i]$ then
            $col \leftarrow col + \delta$
        else $row \leftarrow row + \delta$

        if the car can't fit the Board or crashes with a car then return
        create a game g similar to G except with (col, row) for the car i
        $NextStates.add($State(g,moves+1)$)$
        $\delta \leftarrow \delta + 1$ if $right$ else $\delta - 1$
end function
```

```
function AllNextStates
    NextStates $\leftarrow \{\}$    ▷ Empty set
    for i in range(G.nbcars) do
        StatesInDirection(Next States, i, true)
        StatesInDirection(Next States, i, false)
    return NextStates
end function
```

We chose an **unordered set** in our implementation because we need a data structure that can check if a state is visited in $O(1)$ and that has an $O(1)$ method to add a state. These two traits are possible using an `unordered_set`.

To maintain a set of visited states efficiently, it was necessary to define a proper hash structure and implement the equality operator ('==') for state comparison for both classes `Game` and `State`.
However, we shouldn't involve `moves` in these new methods, otherwise two states that have the same `Game` and not the `moves` aren't equivalent, so we can visit them both which increases the complexity without benefit.

Algorithm 1 Brut Force BFS Algorithm to solve RushHour

```
function BFS_RushHour($initial\_state$)
    $queue \leftarrow [initial\_state]$
    $visited \leftarrow \{\}$                                ▷ Empty set
    while $queue$ is not empty then
        $current\_state \leftarrow queue.dequeue()$
        if $current\_state.succeed()$ then
            return $current\_state.moves$
        $visited.add(current\_state)$
        for $next\_state$ in $current\_state.next\_steps()$ do
            if $next\_state \notin visited$ then
                $queue.enqueue(next\_state)$
    return -1
end function
```

2.2 Experimental results

We tested the BFS algorithm on the problem instances provided in the database available via the following link :
https ://marceaucoupechoux.wp.imt.fr/files/2024/11/ExRushHour.zip.
To analyze the influence of each parameter (number of cars, depth of the shortest path) on execution time, we plotted the results in a 3D graph. Results are shown in Figure 2.

The experimental results show that the execution time generally increases with the shortest path length. The number of cars impacts execution time, since adding more cars increases the number of the next states we need to explore in each level of the BFS. However, its effect appears to be less significant than that of the path depth as it is shown in Figure 2. A few outliers suggest cases where BFS took significantly longer, possibly due to increased branching.

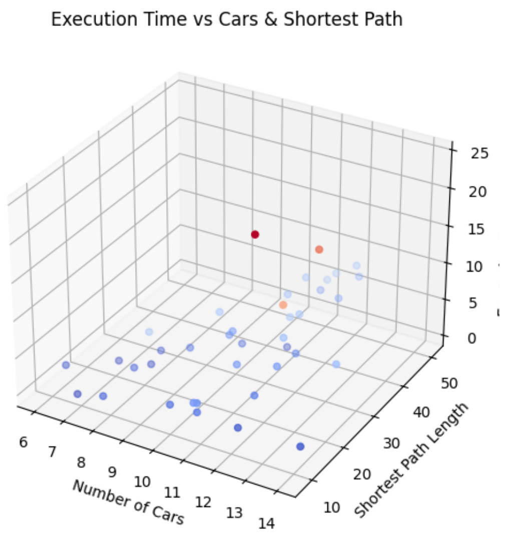

FIGURE 2 – Execution time in seconds of Algorithm 1 a function of the number of cars and shortest path depth.

2.3 Solution Output

We can modify the program to output the solution as a sequence of States, a useful approach is to replace the current integer variable `moves` in the `State` class with a vector called `all_moves`.
The vector `all_moves` stores pairs of the form $(car\_id, position)$, where :
-*car_ id* : represents the ID of the car being moved.
-*position* : specifies the target position after the move.

By keeping track of all moves leading to the current state, this structure allows the program to reconstruct and display the sequence of the solution.

However, we need to update the function `StatesInDirection` so that it inserts a new `State` with the original vector `all_moves` and the new move done by the car `i`. In the BFS code, we need to change ***return*** *current_ state.moves* by ***return*** *current_ states.all_ moves.*

3 Approach based on Heuristics

In the standard BFS algorithm, the main limitation is that it explores all possible next states without any prioritization. The key improvement proposed here is to prioritize states that have a higher likelihood of being part of the optimal solution, based on observable patterns in the current board configuration. In this case, we are going to introduce *heuristics.*

A heuristic in our case is a function h that associates to each state s the estimated length of a solution starting from this state. In particular, if s_f denotes the state where the red car has exited traffic, then $h(s_f) = 0$.

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',

$$h(s) \leq h(s') + k_{s,s'} \tag{1}$$

where $k_{s,s'}$ is the minimal number of moves to go from s to s'.

3.1 Choosing Heuristics :

A trivial heuristic in our case would be $h \equiv 0$. However, this would not modify the algorithm's behavior, effectively reducing it to standard Djikstra's Algorithm.

More interesting heuristic functions include :

- $h_1(s)$: The number of vehicles blocking the red car's path to the exit
- $h_2(s)$: The number of vehicles in the right section of the board (if we partition the grid into left and right regions relative to the red car's position)

These heuristics provide meaningful guidance to the search algorithm while maintaining admissibility.

3.2 Heuristic Analysis

The blocking cars heuristic h_1 satisfies consistency since :
$\forall s \to s', h_1(s) \leq h_1(s') + 1$, as each move affects at most one blocking car.

However, the right-section heuristic h_2 may violate $h_2(s) \leq h_2(s') + 1$ when vehicles move between partitions without reducing congestion.

Table 1 – Heuristic Properties

Heuristic	Admissible	Consistent
$h \equiv 0$	✓	✓
h_1 (Blocking cars)	✓	✓
h_2 (Right-section vehicles)	✓	×

3.3 Implementation

Given our selected heuristics, we implement an adapted BFS algorithm that prioritizes states using heuristic guidance. For each state s, we define :

$$\text{priority}(s) = g(s) + h(s) \tag{2}$$

where :

— $g(s)$: The number of moves taken to reach state s (exact cost)

— $h(s)$: The heuristic estimate of remaining moves to the solution

The state comparison operator is defined solely based on this priority value :

$$s_1 \leq s_2 \iff \text{priority}(s_1) \leq \text{priority}(s_2) \tag{3}$$

This prioritized exploration leads to Algorithm 0, which efficiently finds optimal solutions by focusing on the most promising paths first.

In the worst case, the time complexity remains exponential. However, a well-designed heuristic significantly reduces the number of visited states.

Algorithm 2 A^* Algorithm using a heuristic h to solve RushHour

```
1:  function AStar(start, h)
2:      $Q \leftarrow$ PriorityQueue(start, h(start))
3:      $V \leftarrow \emptyset$
4:      while $Q \neq \emptyset$ do
5:          $s \leftarrow Q$.PopMin()
6:          if $s$ is solution then return $s$.GetPath()
7:              if $s \notin V$ then
8:                  $V$.Add($s$)
9:                  for $s'$ in $s$.Neighbors() do
10:                     $s'$.SetPriority($s'$.Moves + $h(s')$)
11:                     $Q$.Push($s'$)
       return $\emptyset$
12:
```

3.4 Experimental Results

To test our heuristics, we analyzed the execution time both with and without heuristics, using the dataset provided in Section 2. The results are shown in Figure 3.4. As demonstrated, we have drastically improved the algorithm's execution time.

The comparative evaluation reveals that the Distance heuristic (h_{dist}) demonstrates consistently superior performance to the Blocking Cars heuristic (h_{block})

The significant improvement in execution time suggests promising opportunities for future work. It may suggests the potential for developing a composite heuristic function using different heuristics.

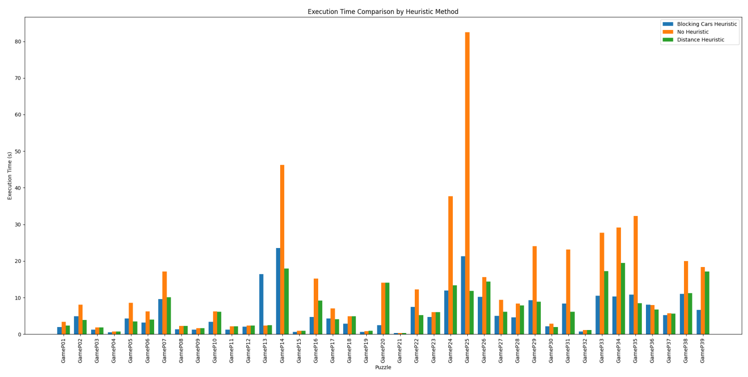

FIGURE 3 – Execution Times for different Heuristics using Algorithm 0

Conclusion

This project successfully developed a complete solution to the Rush Hour puzzle, implementing an optimal pathfinder through systematic state-space exploration. Our work produced several key contributions :

First, we established an efficient base implementation using optimized data structures that reduced memory overhead while enabling fast state comparisons. The core solver employed breadth-first search to guarantee shortest-path solutions, with careful attention to move generation and state representation.

To address the computational limitations of the naive approach, we introduced and evaluated two admissible heuristics : the Blocking Cars heuristic and the more effective Distance heuristic. Benchmarking demonstrated that heuristic guidance reduced execution time drastically.

The project's outcomes suggest several promising directions for future work. A particularly compelling avenue would be developing machine learning techniques to automatically generate and combine heuristic functions. Reinforcement learning could train an adaptive heuristic that dynamically adjusts its evaluation based on board configuration patterns.

Références

[1] Flake, G. W., & Baum, E. B. (2002). Rush Hour is PSPACE-complete, or "Why you should generously tip parking lot attendants". *Theoretical Computer Science*, 270(1), 895-911.