Rush Hour Puzzle Solver

Karan Tibrewal

Abstract

The **Rush Hour Puzzle** presents a 6x6 grid with some configuration of cars and trucks. The objective of the game is to get the red car through the end gate, by moving a vehicles in each turn to make the passage possible. Typically, at each state, there are several cars that can be moved, and to several spots. Then, the branching factor for the search problem is large; we attempt to tackle this by developing two heuristics: the **Obstacle Heuristic** and the **Dependency Heuristic**. We then perform an A* search with these heuristics which perform substantially better than uninformed variants.

Introduction

This paper puts forward a Python package which implements an agent to solve the **Rush Hour Puzzle**. The rules are simple: there is a nxn grid with some configuration of cars and trucks. At each turn, we can move a car up or down its row or column (depending on its orientation; cars cannot "turn" their orientation). The objective is to get the red car through the end gate by making as little moves as possible.

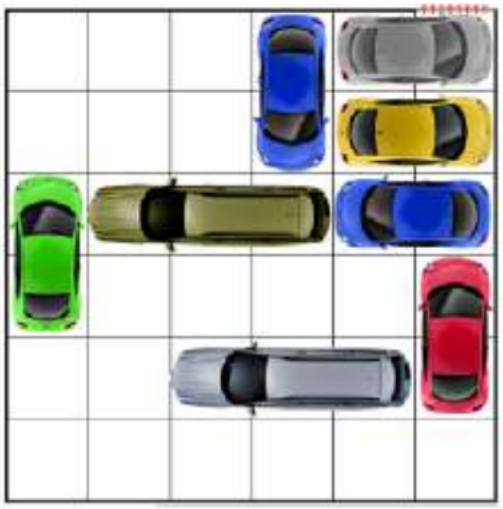


Figure 1: Example Problem

In the above example, we want to get the red car through the red door (cell $(0, 5)$). Of course, we can't go over/bump into other cars. Nor can cars turn: they can just move up or down in their respective row or column (depending on their orientation).

Let's consider our first move: there are 7 possibilities (move green up by 1, move green up by 2, move down up by 1, move green down by 2, move grey left by 1, move grey left by 2, move red down by 1). Going forward, for our second move, we have anywhere between 5 - 8 possibilities. Thus, we can see that as a search problem, the Rush Hour Puzzle has a large branching factor. If we estimate $b = 4$ and $d = 20$, then an uninformed search would traverse through more that 10 billion states!

Fortunately, we can do better: we develop an agent that makes use of efficient heuristics (**"Obstacle Heuristic"** and **"Dependency Heuristic"**) to traverse the search space intelligently.

The Rush Hour Puzzle as a Search Problem

Here, we look at the Rush Hour Puzzle as an informed search problem. Let's begin by formulating the search space:

States: A state is described by the configuration of vehicles on the board.
Initial State: The grid configuration given to us.
Actions: For a given state, we can move a car with row-wise orientation n spaces to the left or right, or a car with column-wise orientation n spaces up or down, such that it doesn't intersect with another car in its path, and stays in the boundary of the grid.
Transition Model: Given an action, we move the car in question accordingly to its new position. This is now a new state.
Goal Test: The red car has safe passage through the gate – that is, the red car is in the square with the exit.
Path Cost: Each moves counts as 1 unit of cost.

Navigating the Search Space Efficiently

We use an A* search to solve the puzzle. The search picks nodes in the frontier with the lowest $f(n) = g(n) + h(n)$, where $g(n)$ is the actual path cost incurred to get to this node (i.e., the depth of the node), while $h(n)$ is the heuristic function of choice. We describe our heuristic functions below:

The Obstactle Heuristic
We define the Obstacle Heuristic as follows:

$$h(n) = \#\ cars\ in\ the\ path\ from\ red\ car\ to\ exit\ gate$$

For example, consider:

Figure 2: Example Problem for Obstacle Heuristic

For this node, the number of cars in the way of the red car and the exit is 2 (that is, the yellow car and the red car). Hence $h(n) = 2$.

It is easy to see that this heuristic is consistent: if we make a move removing a car from the red car's path then $h(n)$ goes down by one, but $g(n)$ goes up by one. Hence, $h(n) = c(n, a, n') + h(n')$.

If we do not make a move, then $h(n)$ remains the same, while $g(n)$ goes up by one. Hence, $h(n) < c(n, a, n') + h(n')$. **Hence, this heuristic is consistent.** It follows that it is also **admissible**.

The Dependency Heuristic
We define the Dependency Heuristic as follows:

$$h(n) = \#\ cars\ that\ must\ be\ moved\ to\ get\ red\ car\ to\ exit\ gate$$

This captures some more information than the Obstace heuristic. For example, consider the configuration from Figure 2. In order to get the red car to the exit gate, we must move the cars that are in its path: namely, the yellow and green car. However, notice that in order to move the yellow or green car, we must move other cars! For example, we must move the horizontal green car to move the first green car. We continue finding such cars who restrict the red cars movement. We can think of this as a dependency map. Then, in this case, $h(n) = 6$.

This heuristic is consistent as well: for a given action, if we move a car out of the dependency map, although the $h(n)$ estimate for the resulting node decreses, it sees a corresponding rise in $g(n)$. If not, then, $g(n)$ alone. Hence, $h(n) \leq c(n, a, n') + h(n')$, and our heuristic is **consistent**. It follows that it is **admissible**.

Analysis of Search Performance

The following table summarizes the performance of our algorithm and heuristics:

Traffic Puzzle	Heuristic Used	# Goal Tests	Time Taken
Puzzle 1	Obstacle	169	real 0m6.921s user 0m4.603s sys 0m0.040s
Puzzle 1	Dependency	8	real 0m2.067s user 0m0.522s sys 0m0.043s
Puzzle 2	Obstacle	1565	real 4m7.591s user 4m0.614s sys 0m1.567s
Puzzle 2	Dependency	1565	real 4m8.196s user 3m59.943s sys 0m1.763s
Puzzle 3	Obstacle	1036	real 0m26.257s user 0m16.996s sys 0m0.201s
Puzzle 3	Dependency	151	real 0m6.480s user 0m2.312s sys 0m0.028s

Figure 3: Performance Analysis

The **Dependency** heuristic substantially out performs the **Obstacle** Heurisitic. This is expected: the Dependency **dominates** the Obstacle heuristic, and therefore, it was a better choice from the very get go. Essentially, the obstacle heuristic is able to give us some information about the obstacles in the way of the red car, but gives us no indication about **how we should move these obstaces**. The Dependency heuristic tries to build on this weekness of the former, and **tries to give us some information about how we might go about moving these obstaces**.

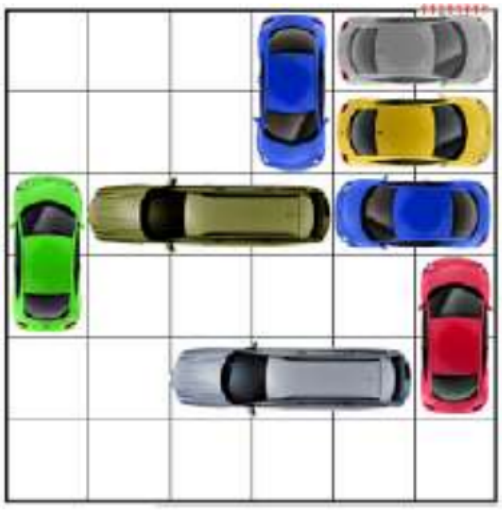
Conclusion

Hence, the heuristic functions substantially increase the capabilities of our agent. The Dependency Heuristic builds a map of dependencies of cars that must be moved to free up the path of the red car, and as of such, proves to be an efficient way to navigate the search space.

We could improve on our strategies in a number of ways. Firstly, we can implement some sort of a reward system: for example, suppose a car can be moved to a location that no other car can access; then, if we end up moving the car, it is best to move it to this location as it will never interfere with other pieces again! Further, we can train our agent on random board choices so that if it finds itself in a familiar position, it can find its way (just by memory!).

Appendix

Puzzle 1



Initial State A

Figure 4: Example Puzzle 1

Solution using Dependency Heuristic:

```
karantibrewal$ time python TrafficPuzzleBase.py
1. Test Puzzle One
2. Test Puzzle Two
3. Test Puzzle Three
4. Input new puzzle from keyboard
Choose an option [1-4]: 1
1. Obstacles Heuristic
2. Dependency Heuristic
Choose an option [1-2]: 2
0   0   0   D   A   A
0   0   0   D   B   B
E   F   F   F   C   C
E   0   0   0   0   **
0   0   G   G   G   **
0   0   0   0   0   0
Solution:
Moving E UP by 2
Moving F LEFT by 1
Moving G LEFT by 2
Moving D DOWN by 4
Moving C LEFT by 1
Moving B LEFT by 1
Moving A LEFT by 2
Moving ** UP by 3
Number of times Goal Test Function is called: 8
```

Puzzle 2

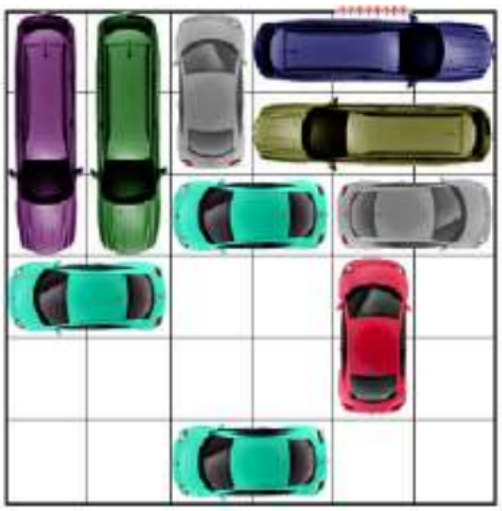


Figure 5: Example Puzzle 2

```
karantibrewal$ time python TrafficPuzzleBase.py
1. Test Puzzle One
2. Test Puzzle Two
3. Test Puzzle Three
4. Input new puzzle from keyboard
Choose an option [1-4]: 2
1. Obstacles Heuristic
2. Dependency Heuristic
Choose an option [1-2]: 2
H   I   A   F   F   F
H   I   A   G   G   G
H   I   B   B   D   D
C   C   0   0   **  0
0   0   0   0   **  0
0   0   E   E   0   0
Solution:
Moving ** DOWN by 1
Moving C RIGHT by 3
Moving H DOWN by 3
Moving I DOWN by 3
Moving B LEFT by 2
Moving A DOWN by 3
Moving B RIGHT by 2
Moving H UP by 3
Moving I UP by 3
Moving E LEFT by 2
Moving A DOWN by 1
Moving C LEFT by 3
Moving A UP by 1
Moving ** UP by 1
Moving E RIGHT by 3
Moving A DOWN by 1
```

```
Moving C RIGHT by 2
Moving I DOWN by 3
Moving F LEFT by 2
Moving G LEFT by 2
Moving H DOWN by 3
Moving B LEFT by 2
Moving D LEFT by 2
Moving ** UP by 3
Number of times Goal Test Function is called: 1565
```

Puzzle 3

Figure 6: Example Puzzle 3

```
karantibrewal$ time python TrafficPuzzleBase.py
1. Test Puzzle One
2. Test Puzzle Two
3. Test Puzzle Three
4. Input new puzzle from keyboard
Choose an option [1-4]: 3
1. Obstacles Heuristic
2. Dependency Heuristic
Choose an option [1-2]: 2
0   0   D   C   C   C
0   0   D   A   B   B
0   E   D   A   0   **
F   E   G   G   G   **
F   E   0   H   H   H
F   0   0   0   0   0
Solution:
Moving E UP by 2
Moving F UP by 3
Moving G LEFT by 2
Moving H LEFT by 3
Moving A DOWN by 3
Moving B LEFT by 1
```

```
Moving ** UP by 1
Moving G RIGHT by 3
Moving D DOWN by 1
Moving C LEFT by 1
Moving ** UP by 1
Number of times Goal Test Function is called: 151
```