Unblocking-Me: Efficient Solver for Rush Hour Puzzles

Hok SENG & THANH DIEP Phoung

February 2025

Contents

1 Introduction

The "Unblock Me" puzzle consists of a grid with horizontal and vertical vehicles that can move along their respective axes. The objective is to find the shortest sequence of moves to allow the red car to exit. This problem is known to be PSPACE-complete, making it computationally challenging for large grid sizes.

2 Setting Up the Game

2.1 Reading a File and Instantiating a Game

2.1.1 Checking if a File is a Valid Input

Question 1: The first step in solving the Rush Hour puzzle is to ensure that the input file describing the initial game state is correctly formatted and valid. This includes:

- Ensuring no two vehicles overlap.
- Checking that vehicles fit within the grid dimensions.

To achieve this, the `Check` class provides a method `check_file()` that reads the file, parses its contents, and validates the constraints before initializing a `RushHour` game instance.

2.2 Implementation Choices

The method follows these steps:

1. Read and preprocess the file: Remove empty lines and extract grid size and vehicle count.
2. Store validated vehicles in a list and return an initialized game by `check_file()` which calls `build_grid()` inside the `RushHour` class to check for overlapping.

Algorithm 1 Detect Overlapping

```
grid ← 2D array of size grid_size × grid_size initialized with all entries "."
for each vehicle in vehicles do
    for each (x, y) in vehicle's occupied positions do
        if grid[y − 1][x − 1] ≠ "." then
            raise OverlapError
            grid[y − 1][x − 1] ← vehicle.label
```

2.3 Displaying the Rush Hour Game State

Question 2: Once a valid game configuration is initialized, it is essential to visualize the state of the puzzle. The `RushHour` class provides a method to construct and display the game grid.

2.4 Implementation Choices

- The `build_grid()` method initializes an empty grid and places vehicles based on their attributes.
- The `display_grid()` method visually represents the grid with row and column labels.

2.5 Example Output

The displayed grid:

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

3 Solving the Game: A Brute Force Approach

Question 3: A brute-force algorithm explores all possible move sequences starting from the initial configuration until a solution is found. The state space is represented using a graph, with nodes corresponding to unique game states and edges representing valid moves.

Algorithm 2 Brute-Force Search for Solving Rush Hour

Ensure: A path leading to a solution, or `NO_SOLUTION` if no solution exists

```
1:  visited ← empty set
2:  return Solve(S_0, visited)
3:  Solve(S, visited)
4:  if IsWinning(S) then
5:      return path
6:      if S is in visited then
7:          return NO_SOLUTION
8:          Add S to visited
9:          for all move in GetPossibleMoves(S) do
10:             S' ← ApplyMove(S, move)
11:             path ← Solve(S', visited)
12:             return NO_SOLUTION
```

Choice of data structure: we use `hash_table` to store the unvisited states of the game by avoiding re-exploring boards. Its time complexity is $O(1)$ amortized (in average).
Complexity Analysis of the Brute-Force Algorithm:

- **Exploring All Moves (Line 13):** The function iterates over all possible moves in the current state, which defines the *branching factor* b. This means that from each state, up to b new states are generated.
- **Recursive Calls (Line 14):** The algorithm applies each move and recursively calls itself on the new state, increasing the *depth* d of the search.

Let $T(d)$ denote the worst-case number of game states explored at depth d. The recursive nature of the brute-force search implies:

$$T(d) = b \cdot T(d-1) \tag{1}$$

By expanding the recurrence, we obtain:

$$T(0) = 1 \tag{2}$$

Thus, our recurrence simplifies to:

$$T(d) = b^d \tag{3}$$

Question 4: The function `get_possible_moves()` inside the class `RushHour` find all the possible moves of a given state of a game.
Question 5: In this type of problem can be regarded as shortest path problem, we tend to use Dept-first Search(DFS) as when testing with the existing dataset, the average time complexity of them are much smaller compared to Breadth-first search. Nevertheless, in general, the Dept-first Search(DFS) is not complete and can be improved using **IDA*** [2]. Since, the nature of **IDA*** doesn't use dynamic programming, we decide to use Breadth-first Search(BFS) algorithm for this problem.

Algorithm 3 Breadth-First Search (BFS) for Solving Rush Hour with Hash Set for Visited States

```
1: visited ← empty Hash Set
2: Queue ← empty FIFO List
3: Push (S_0, [ ]) onto Queue
4: while Queue is not empty do
5:     (S, path) ← Dequeue from Queue
6:     if IsWinning(S) then
7:         return path
8:         if S is not in visited then
9:             Add S to visited
10:            for all move in GetPossibleMoves(S) do
11:                S' ← ApplyMove(S, move)
12:                Enqueue (S', path + [move]) into Queue
13:                return NO_SOLUTION
```

Question 6: The average time of execution of the BFS is : 0.759s.

3.1 Reconstruction of the Solution

Question 7: To reconstruct the solution path in the Rush Hour problem, we need to store information about how each state was reached. The modification involves the following steps:

We use a dictionary (Hash Map) called `predecessors` to store the parent state of each explored state since the time complexity of finding an element is $O(1)$ in average:

```
predecessors[state] = previous_state
```

4 Approach Based on Heuristics

4.1 General Formulation

Before delving into the algorithm, we must first reformulate our problem in a more suitable framework: an *implicit graph*. Let $G = (V, \Gamma, w_{ij})$ be an implicit graph, where V represents the set of

all nodes, Γ is an operator that defines the transitions between nodes, and w_{ij} denotes the weight of the edge connecting node n_i to node n_j. The operator Γ acts on each node n_i such that:

$$\Gamma(n_i) = (n_j, w_{ij})$$

This means that applying Γ to node n_i yields the next node n_j along with the transition cost w_{ij}.

The foundational work of Peter E. Hart et al. [1] established that, for any implicit graph, if the computed heuristic $\hat{h}(s)$ is *consistent*, then the termination and the optimality is always guaranteed in the absence of tie-breaking (i.e., when no two nodes share the same cost $\hat{f}(s)$. However, in the presence of tie-breaking, there exists a tie-breaking rule such that the optimality will be achieved (Theorem 3 in [1]).

In our problem setting, V corresponds to all possible states, Γ defines the available transitions from one state to another, and w_{ij} represents the cost of moving from the current state to the next. For simplicity, we assume a uniform cost model where $w_{ij} = 1$.

Question 8: The following is the pseudo code version of provid heuristic by the definition

Algorithm 4 BFS Search for Solving Rush Hour using Heuristics(A*)

visited ← empty **Hash Set**
Priority Queue ← empty **Min-Heap**
predecessors ← empty **Hash Map**
Compute $h(S_0)$ (heuristic value for the initial state)
Push $(h(S_0), 0, S_0, [\,])$ onto *Priority Queue* ▷ $(f, g, \text{state}, \text{path})$
while *Priority Queue* is not empty **do**
    $(f, g, S, path)$ ← **Pop** from *Priority Queue* (lowest f-value first)
    if IsWinning(S) **then**
        return *path*
    else
        if S **is not in** *visited* **then**
            Add S to *visited*
            for all *move* in GetPossibleMoves(S) **do**
                S' ← ApplyMove(S, *move*)
                g' ← $g + 1$ ▷ Cost of reaching new state
                if S' **is not in** *visited* **or** $g' < cost(S')$ **then**
                    Update cost of S' in *visited* to g'
                    h' ← Heuristic(S')
                    f' ← $g' + h'$
                    Push $(f', g', S', path + [S'])$ into *Priority Queue*
return `NO_SOLUTION`

We have chosen this approach based on Theorem 3 of [1]. Since we cannot select a "smart" tie-breaking rule in advance, we just choose the FIFO as tie-breaking rule. The FIFO is manipulated by the Min-Queue. The operator Γ is implemented by the method GetPossibleMoves() and ApplyMove(). The correctness of the algorithm is confirmed by the work of [1] for any $\delta-$graph with consistent heuristic.

Remark 4.1 : If we set the heuristic function to a constant value of $h(s) \equiv 0$, the A* algorithm reduces to **Uniform Cost Search (UCS)**, which is equivalent to Breadth-First Search (BFS) when all moves have the same cost and:

$$f(s) = g(s)$$

This means that the algorithm prioritizes states based only on $g(s)$, making it identical to UCS. Since every move in the Rush Hour problem has the same cost (typically 1), UCS explores all states at the same depth level before moving deeper, which is precisely the behavior of BFS.

4.2 First Steps with Heuristic

To improve efficiency, heuristics are introduced to prioritize certain move sequences. An admissible heuristic is one that never overestimates the number of moves required to reach the solution. A simple heuristic is counting the number of blocking vehicles between the red car and the exit provided in **Question 9:**

Proof. Consider these following cases:

1. A blocking car is removed. In this case, the new state have $h(s') = h(s) - 1$ and:

$$h(s) \leq (h(s) - 1) + k_{s,s'} \iff h(s) \leq h(s).$$

2. A non-blocking car moves which does not affect the red car's path moves. The number of blocking cars remains the same. Therefore

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

Therefore $h(s)$ is consistent. □

4.3 New Heuristic

In a new heuristic function $h(s)$, we define:

- $h(s) = 0$ if the red car is not blocked.
- If a vehicle v blocks the red car, then $h(s) = 1 +$ the number of vehicles in the chain of blockages.
- The heuristic recursively penalizes each vehicle that is blocking the movement of another blocking vehicle.

Question 11: Proof of consistency for the chain blockage heuristic:

Proof. Let s be a state and s' be a successor state where one move is applied. We consider two possible cases:
Case 1: Moving a vehicle that does not affect the blockage chain. If a vehicle is moved but does not change the number of blocking vehicles in the chain, then:

$$h(s') = h(s)$$

Since $k_{s,s'} = 1$, we get:

$$h(s) - h(s') = 0 \leq 1 = k_{s,s'}$$

Thus, the consistency condition holds.
Case 2: moving a vehicle that reduces the blockage chain. If a move reduces the blockage chain, then $h(s') < h(s)$. Since the heuristic directly counts blocking vehicles and their dependencies, the decrease in $h(s)$ is at most 1 per move:

$$h(s) - h(s') \leq 1$$

Since $k_{s,s'} = 1$, this gives:

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

Thus, the consistency condition holds. □

5 Implementation and Results

Question 10: The Performance comparisons between brute-force, heuristic-based methods and improved heuristics are presented below. Execution times and the number of explored states are analyzed for 40 puzzles of different configurations given during the project.

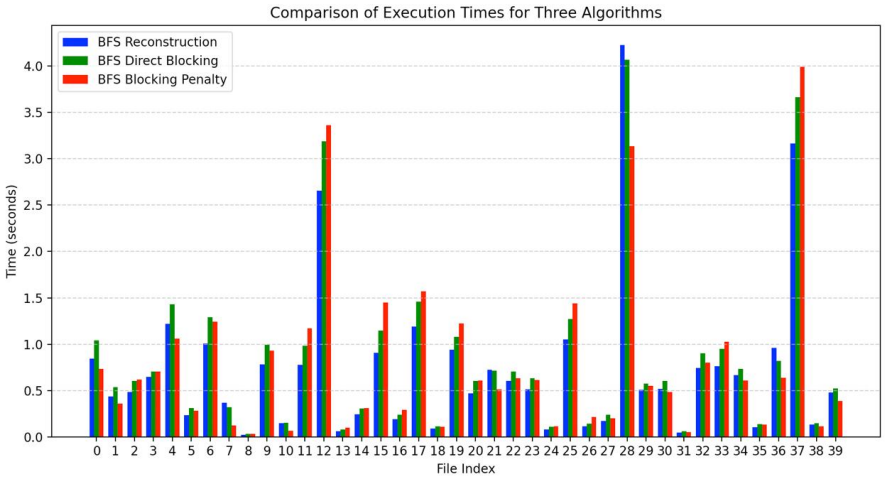

Figure 1: Comparision of performance (execution time (s)) of different heuristic approach. BFS Reconstruction: Question 7, BFS Direct Blocking: Question 10, BFS Blocking Penalty: Question 11.

Q	Algorithm	Mean of explored states	Mean Time Execution(seconds)
Q7	BFS Reconstruction	3070.775	0.7372
Q10	BFS Direct Blocking	2866.925	0.8297
Q11	BFS Blocking Penalty	2640.7	0.7961

Table 1: Comparison of average explored states and time execution of different BFS algorithms

We can observe that using a more sophisticated heuristic does not necessarily reduce the time complexity. On the contrary, in some cases, it may increase the execution time. This "anomaly" can be explained by the following reasons:

1. The dataset may favor certain heuristics over others.
2. In our formulation of the implicit graph, the cost of every edge is equal. As a result, our tie-breaking rule (FIFO) may not always perform optimally.

3. The computational overhead of certain sophisticated heuristics may outweigh the cost reduction they aim to achieve, particularly those involving recursive calculations.

4. Additional overhead introduced by the implementation of the priority queue, along with unnecessary deep-copying mechanisms and data abstraction, can negatively impact performance.

The last issue can be mitigated by refactoring the code to employ optimization techniques such as cache locality. However, the first three reasons cannot be resolved at runtime. Instead, a more sophisticated decision-making process based on big data and artificial intelligence could be applied to analyze various heuristics and tie-breaking rules. Unfortunately, the scope of this project does not allow us to explore such an approach.

6 Conclusion

In this project, we explored various algorithmic approaches to solving the "Unblock Me" puzzle, transitioning from brute-force search methods to heuristic-driven optimizations. Through rigorous experimentation and comparative analysis, we demonstrated the impact of different heuristics on execution time and computational efficiency.

Our findings indicate that while heuristics can significantly reduce the search space, their effectiveness depends on multiple factors, including the structure of the puzzle instances, the computational overhead introduced by heuristic evaluations, and the tie-breaking mechanisms used within the search algorithm. In some cases, more sophisticated heuristics did not yield expected improvements in execution time due to additional computational costs, reinforcing the importance of balancing heuristic complexity with practical performance considerations.

Future research could explore adaptive heuristic selection, leveraging machine learning techniques to dynamically determine the most efficient heuristic for a given puzzle configuration. Additionally, optimizing the implementation to minimize unnecessary computations, enhance cache locality, and refine priority queue management could further improve solver efficiency. By integrating such advancements, heuristic-based solvers could achieve greater scalability and robustness, making them more effective for complex problem instances.

References

[1] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (July 1968), pp. 100–107. ISSN: 2168-2887. DOI: 10.1109/TSSC.1968.300136.

[2] Ami Hauptman et al. "GP-rush: Using genetic programming to evolve solvers for the rush hour puzzle". In: *Proceedings of the 11th Annual Genetic and Evolutionary Computation Conference, GECCO-2009* (July 2009), pp. 955–962. DOI: 10.1145/1569901.1570032.