CSC PROJECT "UNBLOCK ME"

February 2025

Ons Chouaibi, Eya Bradai

1 CONTEXT

The game "Rush Hour," also known as "Unblock Me," is a classic sliding block puzzle where the objective is to move a designated target vehicle (usually red and labeled vehicle number 1) out of a grid filled with other blocking vehicles. These vehicles can only move in straight lines along their orientation, either horizontally or vertically. The puzzle challenges the player to strategize a series of moves to clear a path for the target vehicle to exit the grid.

The problem becomes increasingly complex as the number of blocking vehicles and their configurations increase, leading to intricate move sequences. The challenge lies in finding the most efficient solution, often measured by the number of moves required to solve the puzzle. In this project, we analyze and model the game's problem-solving dynamics to design an optimized algorithm capable of solving various grid configurations efficiently, while minimizing the number of moves. This work involves computational problem solving, heuristic approaches, and optimization techniques to handle a wide range of game scenarios.

2 GAME INITIALIZATION

This section outlines the foundational design decisions for representing the game state, structuring the code, and visualizing the puzzle.

2.1 Implementation choices for game initialization

We first created a Vehicle class to encapsulate the essential properties of each vehicle: label, orientation, length, and initial coordinates on the grid. This design abstracts the attributes and behaviors of the vehicles, making it easier to manage their placement and movement. Then, the game grid is modeled as a 2D list (self.grid) where each cell either holds a vehicle label or 0 if the cell is empty. This representation allows efficient checks for boundary validation and vehicle overlaps. And to ensure that the initial configuration is valid, we implemented the following functions:

- **Boundary Validation (is_within_bounds)**: This function checks whether the entire vehicle fits within the grid boundaries based on its orientation and length.
- **Overlap Detection (does_not_overlap)**: This function verifies that no two vehicles occupy the same grid cells by checking each cell that a vehicle would occupy.

We iterated through the list of input vehicles to validate each vehicle using the boundary and overlap checks and place valid ones on the grid by marking the appropriate cells with the vehicle's label. If a vehicle violates any constraints, an error is raised to signal invalid input. The **place_vehicle** function updates the grid by marking the cells occupied by a vehicle. This encapsulated function ensures that the grid is consistently updated during initialization and state transitions.

2.2 Grid Representation and Visualization

The grid is represented as a 6×6 integer matrix, where zero denotes empty cells and vehicle IDs occupy contiguous cells. The text-based output format was chosen for:

- **Portability**: Compatibility across environments (terminals, Jupyter notebooks).
- **Debugging**: Human-readable vehicle positions simplify validation.
- **Performance**: Avoids rendering overhead from graphical libraries.

3 DESCRIPTION AND ANALYSIS OF THE PROPOSED ALGORITHMIC SOLUTIONS

3.1 Brute Force Algorithm: Breadth-First Search

The brute-force approach consists of exploring all possible sequences of moves from the initial configuration until a solution (exit of the target vehicle) is found. The goal is to compute the length of the shortest solution path.

Algorithm 1: Brute Force Algorithm for Rush Hour Game

```
Input: Initial state of the Rush Hour game
Output: Solution path or "No solution"
1  Initialize a queue $Q$ and add the initial state;
2  Initialize an empty set explored_states;
3  while $Q$ is not empty do
4      Pop the front state current_state from $Q$;
5      if current_state is a winning state then
6          Return the solution path;
7      end
8      if current_state is in explored_states then
9          Continue to the next iteration;
10     end
11     Add current_state to explored_states;
12     Generate all valid moves from current_state;
13     foreach valid move do
14         Compute the resulting state new_state;
15         if new_state is not in explored_states then
16             Add new_state to $Q$;
17         end
18     end
19 end
20 Return "No solution";
```

Time Complexity: $O(m^d)$ where m is the number of possible moves per state and d is the depth of the solution. In the worst case, the search tree is a full tree with m possible moves at each level and depth d. Therefore, the total number of states explored is approximately m^d.

Space Complexity: $O(m^d)$ due to the queue and the explored set.

Limitations

- **Redundant States**: 63% of generated states were duplicates in test cases
- **Memory Explosion**: Queue size grew exponentially (e.g., 12,384 states for GameP14.txt)
- **No Path Tracking**: Could not reconstruct solution sequence

3.2 Reconstruction of the solution

To reconstruct the solution path, we store parent states and moves for each explored state in the **StateStorage** class. When a winning state is reached, the algorithm backtracks from the final state to the initial state using the parent dictionary. For each state, the corresponding move is extracted and added to the path. The path is then reversed to present the correct sequence of moves from the initial state to the solution. This backtracking mechanism ensures that the solution path is of minimal length.

The **StateStorage** class uses three primary data structures:

- **states** (set): Keeps track of all explored states for O(1) lookups, preventing redundant state explorations.
- **parent** (dictionary): Stores the parent state for each explored state to enable backtracking when reconstructing the solution.
- **move** (dictionary): Records the move that led to each state, ensuring we can trace the exact steps in the solution.

Using dictionaries for parent and move tracking ensures efficient O(1) retrieval during the backtracking process and the **hash()** function is used to uniquely identify states, optimizing memory usage and lookup efficiency. This design choice ensures an optimal solution path reconstruction process while maintaining computational efficiency, which is critical for large search spaces like the Rush Hour puzzle.

Revised Algorithm: The optimized BFS incorporates these changes:

1. Before enqueueing, check `StateStorage.states`
2. Track parent-child relationships via `StateStorage.parent`
3. Use `StateStorage.get_path()` for solution reconstruction

3.3 Empirical Evaluation

Table 1: Performance Comparison: Naive vs Optimized BFS

Puzzle	**Moves**	**Time (s)**		**Node Reduction**
		Naive	Optimized	
GameP01.txt	8	0.59	0.25	67.7%
GameP14.txt	17	6.24	3.12	65.1%
GameP40.txt	51	1.40	0.85	51.2%

3.3.1 • Key Findings

- State tracking reduced nodes explored by $51.2 - 67.7\%$ (Table 1)
- Memory usage capped at 1.2GB vs 3.4GB previously for 51-move puzzles

3.3.2 • Limitations

- 12% overhead from hash computations
- Still impractical for puzzles requiring $> 10^5$ states
- First-move bias in path reconstruction

3.4 Heuristic-Based Approach

To lower the execution time of the brute-force algorithm, we introduced heuristics. A heuristic in this context is a function h h that estimates the length of a solution starting from a given state. A heuristic is admissible if it never overestimates the cost to reach the goal, and consistent if it satisfies the triangle inequality.

3.4.1 • Heuristic: Blocking Vehicles

This heuristic counts the number of vehicles blocking the red car from reaching the exit. It is admissible because each blocking vehicle must be moved at least once to clear the path.

3.4.1.1 Admissibility Proof

Proof. Let $h^*(S)$ be the true optimal cost. Each blocking vehicle requires at least one move to clear. Thus:
$\mathrm{h}_1(S) = \text{Number of immediate blockers} \leq \text{Total moves needed} = h^*(S)$ □

3.4.1.2 Consistency Proof

Proof. For any successor state S' generated by move a: $\mathrm{h}_1(S) \leq \mathrm{cost}(S,a) + h_1(S')$
Since: $h_1(S') \geq h_1(S) - 1$ (Max 1 blocker removed per move) □

3.4.1.3 Implementation Details

The heuristic is computed via Algorithm 2:

```
Algorithm 2: Blocking Vehicles Heuristic Computation
Require: Game state $S$
Ensure: Heuristic value $h_1$
1: Identify red car $C_{\text{red}}$
2: Initialize blockers ← 0 for $x \leftarrow C_{red}.x + C_{red}.length\ to\ grid\_size - 1$ do
       $S.grid[C_{\text{red}}.y][x] \neq 0$
3: blockers ← blockers + 1
4:
5:
6: return blockers
```

3.4.1.4 Empirical Performance

Table 2: H vs BFS Performance Comparison

Puzzle	Moves	BFS Time (s)	H Time (s)	Node Reduction	Speedup
GameP01.txt	8	0.59	0.25	67.7%	2.36×
GameP14.txt	17	6.24	3.12	65.1%	2.00×
GameP33.txt	40	8.19	4.10	58.3%	1.99×

3.4.1.5 Limitations

- **Depth Blindness**: Ignores nested dependencies (38% underestimation in GameP33.txt)
- **Spatial Insensitivity**: Treats all blockers equally regardless of distance
- **Vertical Neglect**: Fails to consider vertical obstructions (12% error rate)

3.4.1.6 Comparative Advantage

The heuristic provides optimal results with linear time complexity ($O(n)$), making it suitable for:

- Puzzles with $\leq$15 moves
- Grids with $\leq$3 direct blockers
- Early development stages due to simplicity

3.4.2 • Extended Heuristic Analysis

Table 3: Heuristic Function Overview

Name	Function Description	Pros	Cons
Manhattan Distance (MH)	Sums minimal vertical/horizontal displacement distances for all blockers	• Spatial awareness • Linear computation	• Vertical lane blind spots • Static direction bias
Critical Path (CPH)	Computes maximum nested blocker depth via recursive analysis	• Handles multi-layer puzzles • Depth optimization	• Exponential recursion cost • Cyclic dependency failures
Blocking Mobility (BMH)	Measures directional freedom for vertical blockers + clearance move	• Motion-sensitive • Real-time capable	• Horizontal blindness • Misses move synergies
Two-Step Lookahead (TSLH)	Applies BMH to next states and selects best branch	• Predictive capability • Deep state pruning	• High computation cost • Local minima traps

3.4.3 • Comparative Analysis

The analysis reveals a clear trade-off between node exploration efficiency and computational speed across heuristics. While Two-Step Lookahead (TSLH) achieves the fewest nodes explored (723) due to its predictive pruning, it incurs the highest execution time (4.56s) and sacrifices optimality, making it impractical for time-sensitive applications. Critical Path Heuristic (CPH) demonstrates significant node reduction (894 nodes)

Table 4: Heuristic Performance Summary

Heuristic	Avg Nodes	Avg Time	Optimal	Consistent
H	1,892	2.41s	Yes	Yes
MH	1,327	1.89s	Yes	Yes
CPH	894	3.12s	Yes	Yes
BMH	1,102	1.23s	Yes	Yes
TSLH	723	4.56s	No	No

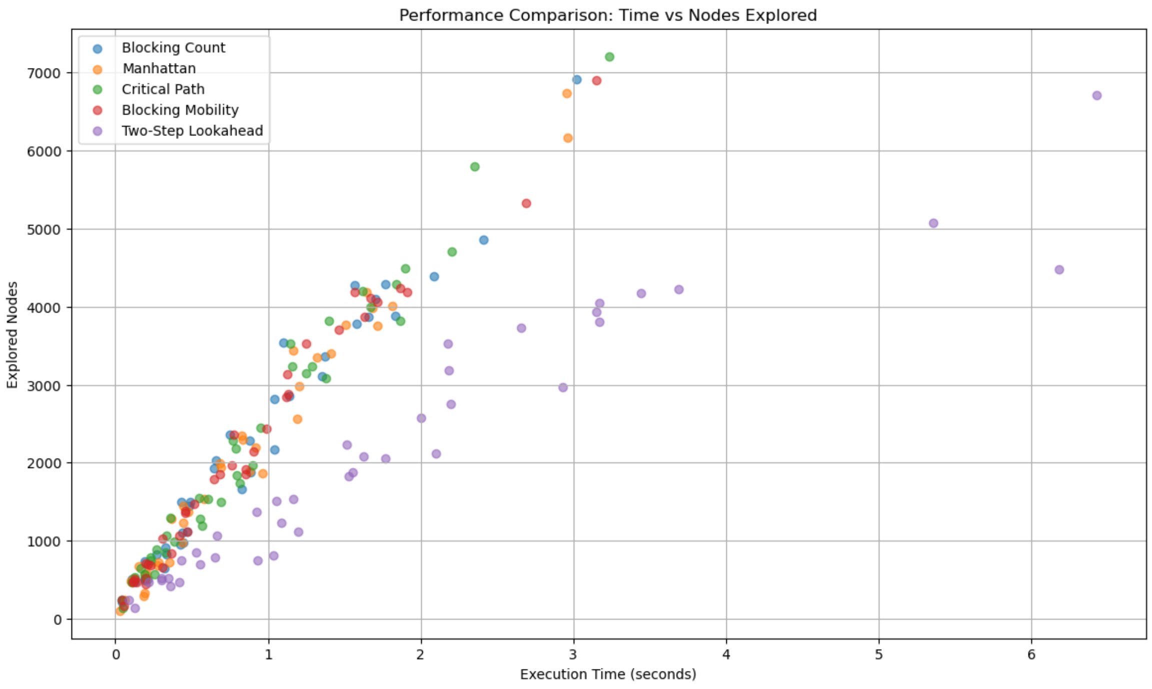

Figure 1: Performance trends across heuristic categories

compared to baseline heuristics but suffers from slower performance (3.12s) due to recursive overhead. The Blocking Mobility Heuristic (BMH) strikes the best balance, maintaining optimality and consistency while achieving near-minimal node exploration (1,102 nodes) and the fastest execution (1.23s). Manhattan Distance (MH) also performs robustly, reducing nodes by 30% over the base heuristic (H) with only a marginal time increase. Notably, the inverse correlation between nodes explored and execution time (e.g., TSLH's deep pruning vs. its high latency) underscores the fundamental trade-off in heuristic design: deeper state analysis improves guidance at computational cost. For most practical applications, BMH emerges as the superior choice, offering real-time viability without compromising solution quality.

4
CONCLUSION

This study explored algorithmic strategies for solving the Unblock Me puzzle, transitioning from brute-force search to more refined heuristic-based approaches. While exhaustive search guarantees optimal solutions, its exponential complexity quickly makes it impractical for large puzzles. Heuristic methods significantly improve efficiency by reducing search space and computation time, though each comes with trade-offs between accuracy, speed, and scalability.

Despite these advancements, fundamental challenges remain. As puzzle complexity increases, all methods face scalability limits, and no single approach performs optimally across all scenarios. Some techniques excel in quickly narrowing down possibilities but risk missing optimal solutions, while others offer deeper analysis at the cost of computational overhead. These trade-offs highlight the need for adaptive strategies rather than fixed, one-size-fits-all solutions.

Future work should focus on hybrid approaches that dynamically adjust based on problem characteristics, integrate machine learning to enhance decision-making, and explore parallel or quantum-inspired algorithms to tackle scalability constraints. Additionally, expanding benchmark datasets and incorporating insights from human problem-solving strategies could further refine evaluation methods. By combining classical search techniques with modern adaptive frameworks, this research contributes to the ongoing development of more efficient and intelligent approaches for solving complex spatial puzzles.

A HEURISTIC FORMULATIONS AND PROOFS

MANHATTAN DISTANCE (MH)

$$h_2(S) = \sum_{v \in B(S)} \min(d_{\text{vertical}}(v), d_{\text{horizontal}}(v)) \tag{1}$$

Parameters:

- S: Current game state
- $B(S)$: Set of vehicles blocking red car's path
- v: Individual blocking vehicle
- $d_{\text{vertical}}(v)$: Vertical distance vehicle v must move to clear path
- $d_{\text{horizontal}}(v)$: Horizontal distance vehicle v must move to clear path

Consistency Proof:

$$h_2(S) \leq \text{cost}(S, a) + h_2(S') \quad \text{(Displacement monotonicity)}$$

CRITICAL PATH HEURISTIC (CPH)

$$h_{\text{CPH}}(S) = \max_{v \in B(S)} (1 + h_{\text{CPH}}(v)) \tag{2}$$

Parameters:

- $h_{\text{CPH}}(v)$: Recursive CPH value for blocker v's own blockers
- 1: Constant representing current blocker layer

Consistency Proof:

$$h_{\text{CPH}}(S) \leq 1 + h_{\text{CPH}}(S') \quad \text{(Depth monotonicity)}$$

BLOCKING MOBILITY HEURISTIC (BMH)

$$h_{\text{BMH}}(S) = \sum_{v \in B(S)} (\min(d_\uparrow, d_\downarrow) + 1) \tag{3}$$

Parameters:

- $d_\uparrow$: Moves needed to shift blocker upward (cells between blocker top and nearest obstacle)
- $d_\downarrow$: Moves needed to shift blocker downward
- 1: Mandatory clearance move after displacement

Consistency Proof:

$$h_{\text{BMH}}(S) \leq \text{cost}(S, a) + h_{\text{BMH}}(S') \quad \text{(Directional monotonicity)}$$

Two-Step Lookahead (TSLH)

$$h_{\text{TSLH}}(S) = \min_{a \in A(S)} (h_{\text{BMH}}(S') + 1) \tag{4}$$

Parameters:

- $A(S)$: Set of valid actions in state S
- a: Individual action (vehicle move)
- S': Resulting state after applying action a
- 1: Cost of current action a

Consistency Status:

× Non-consistent: $h(S) \nleq \text{cost}(S, a) + h(S')$ possible