Simarpal Singh
SID: 862264880
Email: ssing226@ucr.edu
May 8, 2025

CS 205: Assignment 1 - Railway Shunting

Integrity Statement

To complete this assignment, the following resources were consulted:

1) Lecture Slides by Professor Keogh
    a) 2_Blind Search_part1
    b) 2_Blind Search_part2
    c) 3_Heuristic Search
2) Python 3.13 documentation
3) Project description
4) https://www.transum.org/software/sw/starter_of_the_day/starter_January9.asp
5) https://en.wikipedia.org/wiki/Train_shunting_puzzle

The following libraries and their respective documentation were consulted when appropriate:

1) Numpy
2) Seaborn
3) Matplotlib
4) Dataclasses
5) Queue
6) Typing
7) Os
8) Datetime
9) Pytest

All major code was originally written.

Report Outline

Introduction

Algorithm Implementation

Experimental Setup

Results and Analysis

Conclusion

Works Cited

Introduction

Railway Shunting Problem

The Railway Shunting Problem is a combinatorial optimization challenge involving the rearrangement of train cars into a specified goal configuration, subject to a constrained set of legal operations. This problem is grounded in real-world logistics, where physical limitations, such as siding capacity, directionality of movement, and limited maneuvering space add layers of complexity. Inspired by the 8-Puzzle, the Railway Shunting Problem extends the core ideas of search and reordering to a domain that mirrors practical transportation scenarios.

To solve this problem, we apply three different search strategies:

1) Uniform Cost Search (UCS)
2) A* with Misplaced Train Heuristic
3) A* with Manhattan Distance Heuristic

The following sections provide detailed descriptions of each search algorithm. The objective of this project is to compare these methods in terms of efficiency, optimality, and computational costs.

Background

Rail shunting, the process of sorting and assembling railcars, has evolved significantly since the birth of rail transport. In the early days of railways, shunting was performed manually or with the assistance of horses, which proved to be efficient for maneuvering wagons in yards. It wasn't until the mid-19th century that dedicated shunting locomotives were introduced, such as the Stroudley 0-6-0Ts in the 1860s, which were adaptations of mainline engines for yard work (*"When were locomotives first used for shunting? What were the first Shunting Engines?"*).

Steam-powered shunters were introduced in the 19th century, yet the last British Railways shunting horse retired in 1967. The transition to diesel and electric shunting locomotives in the 20th century marked a significant technological advancement, enhancing efficiency and safety in rail yards. Today, shunting operations are integral to railway logistics, utilizing sophisticated locomotives and automation technologies to manage the complex task of assembling and disassembling trains.

Interestingly enough, Dijkstra came up with the shunting yard algorithm, designed to convert an infix expression into a postfix expression, using the practicality of shunting in train cars (*"The Shunting Yard Algorithm"*). Just as railroad yards can temporarily hold cars on side tracks while rearranging, he realized you can sort mathematical operations by moving numbers and operators between data structures such as stacks (*"How in the world did dijkstra come up with the shunting yards algorithm : r/computerscience"*).

Problem Description

In this version of the Railroad Shunting problem:

- The main track allows bidirectional movement of trains
- There are multiple sidings, each with a limited capacity
- The goal is to rearrange an initial sequence of train cars into a target sequence using the fewest possible moves
- Moves include pushing/pulling a train into or out of a siding or rearranging trains on the main track

Algorithm Implementation

Uniform Cost Search (UCS)

As noted in the project description, Uniform Cost Search is a simplified version of A*, where h(n) (the heuristic) is hardcoded to be 0. This means that our equation in A*,

$$f_i(n) = g_i(n) + h_i(n) \quad (path\,cost + heuristic)$$

is simplified to,

$$f_i(n) = g_i(n)$$

where we expand solely based on the lowest path cost. This often results in the expansion of many irrelevant nodes, particularly in deeper search spaces.

A* with Misplaced Heuristic

This algorithm is a variant of A*, where our heuristic counts the number of train cars not in their goal position. It is simple and admissible, but not always informative, as it doesn't consider how far off a train car is from its intended destination.

$$h_{misplaced}(n) = \sum_i [car_i \neq goal_i]$$

A* with Manhattan Distance Heuristic

This algorithm calculates its heuristic by summing up the number of moves each car is away from its goal position. This heuristic is more informed than the misplaced heuristic and typically results in fewer nodes expanded.

$$h_{manhattan}(n) = \sum_i [pos_{current}(i) - pos_{goal}(i)]$$

This assumes a linear track.

Experimental Setup

Benchmarks were run across multiple puzzles of increasing difficulty:

- **Easy**: 2-3 cars, minimal rearrangement
- **Medium**: 3-4 cars, moderate sidings usage
- **Hard**: 4+ cars, significant reordering

Metrics tracked:

- Path length (solution depth)
- Number of nodes expanded
- Execution time
- Maximum queue size

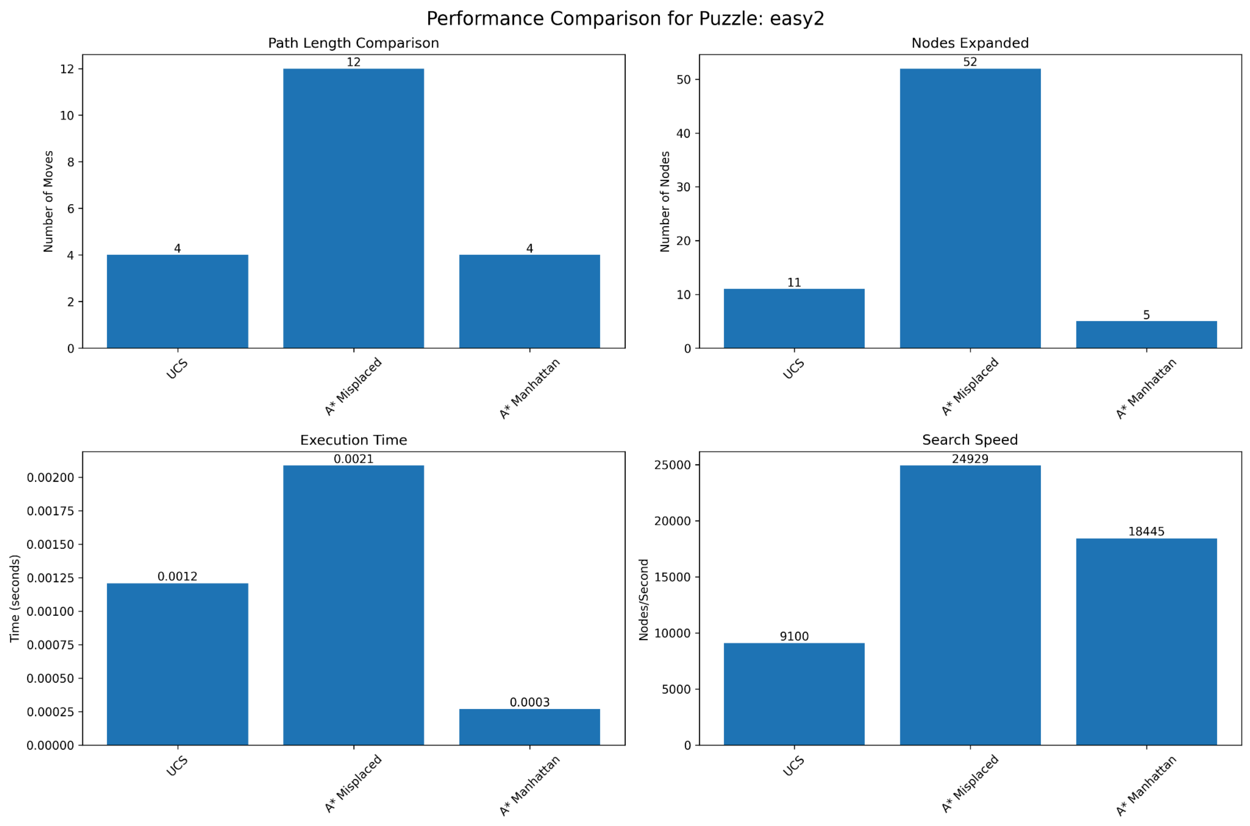
Results and Analysis

The following tables summarize the performance metrics for puzzles of increasing complexity: easy, medium, and hard.

Easy

Summary Statistics for Puzzle: easy2

```
    Initial State: 2 → 1 → 3
    Goal State: 1 → 2 → 3
    Expected Depth: 2 moves
    Sidings: 2

Solution path:
Step 0:
Main Track: 2 → 1 → 3
Siding 1:
Siding 2:

Step 1:
Main Track: 1 → 3
Siding 1:
Siding 2: 2

Step 2:
Main Track: 3
Siding 1: 1
Siding 2: 2

Step 3:
Main Track: 2 → 3
Siding 1: 1
Siding 2:

Step 4:
Main Track: 1 → 2 → 3
Siding 1:
Siding 2:
==================================================
Algorithm: UCS
------------------------------
Path Length: 4
Nodes Expanded: 11
Max Queue Size: 10
Execution Time: 0.0012 seconds
Nodes per Second: 9100.07

Algorithm: A* Misplaced
------------------------------
Path Length: 12
Nodes Expanded: 52
Max Queue Size: 19
Execution Time: 0.0021 seconds
Nodes per Second: 24929.00

Algorithm: A* Manhattan
------------------------------
Path Length: 4
Nodes Expanded: 5
Max Queue Size: 7
Execution Time: 0.0003 seconds
Nodes per Second: 18444.61
```



A* with Manhattan heuristic clearly outperforms the other searches by finding the optimal path with the fewest nodes expanded and fastest execution time, while A* with Misplaced heuristic drastically overestimated the solution cost.

Medium

Summary Statistics for Puzzle: medium2

Initial State: 2 → 3 → 1
Goal State: 1 → 2 → 3
Expected Depth: 6 moves
Sidings: 3

Solution path:
Step 0:
Main Track: 2 → 3 → 1
Siding 1:
Siding 2:
Siding 3:

Step 1:
Main Track: 3 → 1
Siding 1: 2
Siding 2:
Siding 3:

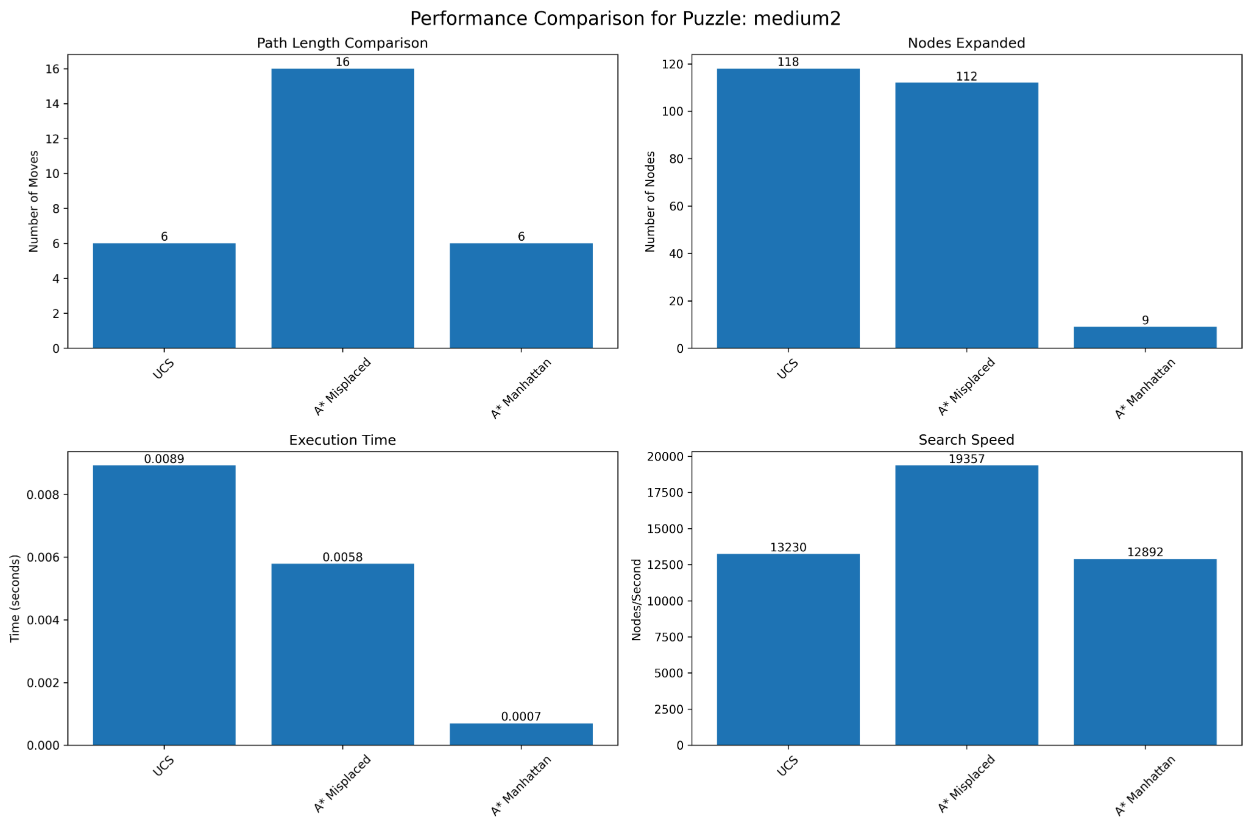
Step 2:
Main Track: 1
Siding 1: 2 → 3
Siding 2:
Siding 3:

Step 3:
Main Track:
Siding 1: 2 → 3
Siding 2: 1
Siding 3:

Step 4:
Main Track: 3
Siding 1: 2
Siding 2: 1
Siding 3:

Step 5:

```
Main Track: 2 → 3
Siding 1:
Siding 2: 1
Siding 3:

Step 6:
Main Track: 1 → 2 → 3
Siding 1:
Siding 2:
Siding 3:
==================================================

Algorithm: UCS
------------------------------
Path Length: 6
Nodes Expanded: 118
Max Queue Size: 58
Execution Time: 0.0089 seconds
Nodes per Second: 13230.18

Algorithm: A* Misplaced
------------------------------
Path Length: 16
Nodes Expanded: 112
Max Queue Size: 56
Execution Time: 0.0058 seconds
Nodes per Second: 19357.26

Algorithm: A* Manhattan
------------------------------
Path Length: 6
Nodes Expanded: 9
Max Queue Size: 17
Execution Time: 0.0007 seconds
Nodes per Second: 12892.33
```



A* with Manhattan heuristic again delivered the optimal solution with maximal node expansion, whereas the misplaced heuristic led A* astray, resulting in an overly long solution path, despite a fast search speed.

Hard

Summary Statistics for Puzzle: hard2

Initial State: 2 → 4 → 1 → 3
Goal State: 1 → 2 → 3 → 4
Expected Depth: 10 moves
Sidings: 3

Solution path:
Step 0:
Main Track: 2 → 4 → 1 → 3
Siding 1:
Siding 2:
Siding 3:

Step 1:
Main Track: 4 → 1 → 3
Siding 1: 2
Siding 2:
Siding 3:

Step 2:
Main Track: 1 → 3
Siding 1: 2 → 4
Siding 2:
Siding 3:

Step 3:
Main Track: 3
Siding 1: 2 → 4
Siding 2: 1
Siding 3:

Step 4:
Main Track: 4 → 3
Siding 1: 2
Siding 2: 1
Siding 3:

Step 5:

Main Track: 2 → 4 → 3
Siding 1:
Siding 2: 1
Siding 3:

Step 6:
Main Track: 1 → 2 → 4 → 3
Siding 1:
Siding 2:
Siding 3:

Step 7:
Main Track: 2 → 4 → 3
Siding 1: 1
Siding 2:
Siding 3:

Step 8:
Main Track: 4 → 3
Siding 1: 1 → 2
Siding 2:
Siding 3:

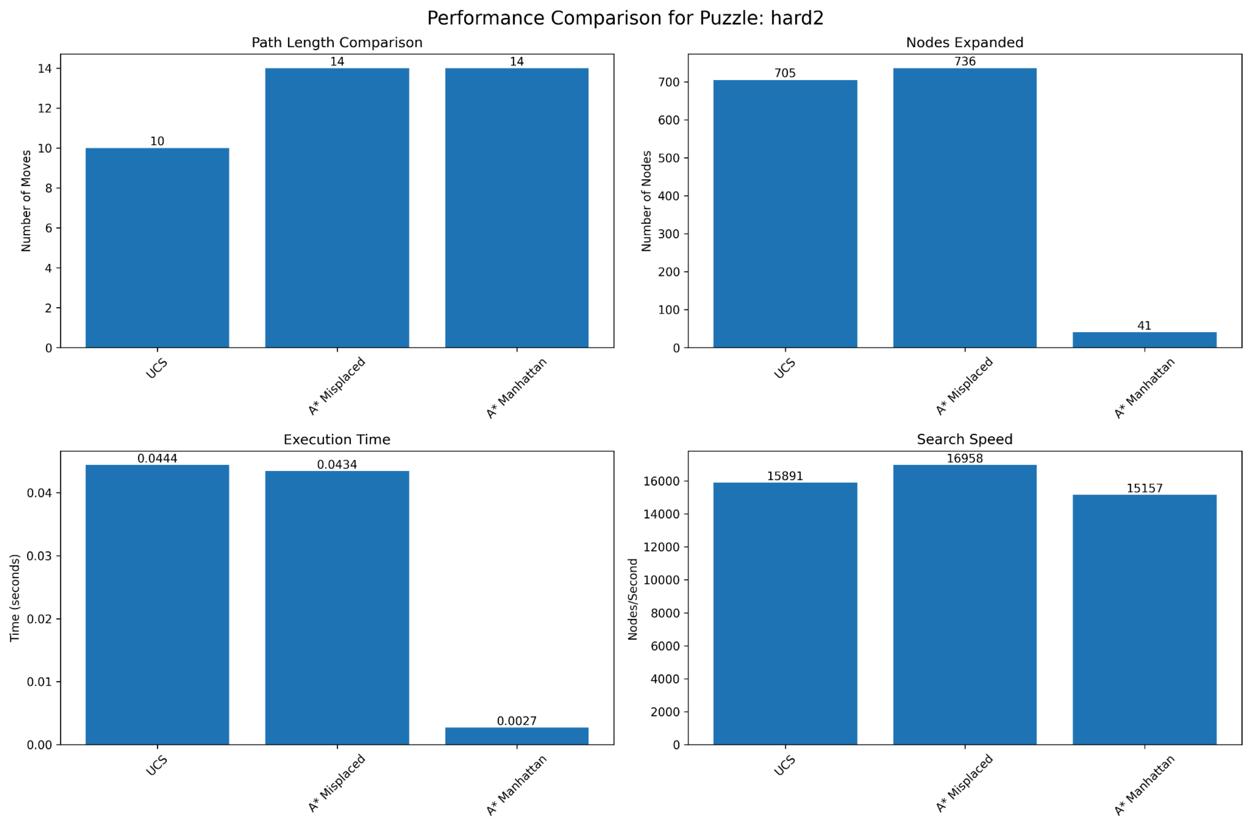
Step 9:
Main Track: 3
Siding 1: 1 → 2 → 4
Siding 2:
Siding 3:

Step 10:
Main Track:
Siding 1: 1 → 2 → 4
Siding 2: 3
Siding 3:

Step 11:
Main Track: 4
Siding 1: 1 → 2
Siding 2: 3
Siding 3:

```
Step 12:
Main Track: 3 → 4
Siding 1: 1 → 2
Siding 2:
Siding 3:

Step 13:
Main Track: 2 → 3 → 4
Siding 1: 1
Siding 2:
Siding 3:

Step 14:
Main Track: 1 → 2 → 3 → 4
Siding 1:
Siding 2:
Siding 3:
==================================================

Algorithm: UCS
------------------------------
Path Length: 10
Nodes Expanded: 705
Max Queue Size: 398
Execution Time: 0.0444 seconds
Nodes per Second: 15890.93

Algorithm: A* Misplaced
------------------------------
Path Length: 14
Nodes Expanded: 736
Max Queue Size: 451
Execution Time: 0.0434 seconds
Nodes per Second: 16957.76

Algorithm: A* Manhattan
------------------------------
Path Length: 14
Nodes Expanded: 41
```

Max Queue Size: 73
Execution Time: 0.0027 seconds
Nodes per Second: 15156.57



In this more complex puzzle, A* with Manhattan heuristic maintained its advantage by achieving near-optimal results with dramatically fewer node expansions, while UCS and the Misplaced heuristic expanded significantly more nodes and produced less efficient paths.

Conclusion

In this project, we explored the Railroad Shunting Problem by comparing three different search algorithms: Uniform Cost Search, A* with a Misplaced Train Heuristic, and A* with a Manhattan Distance Heuristic. Each algorithm was evaluated across puzzles of varying complexity, and performance was measured using significant metrics such as path length, nodes expanded, execution time, and maximum queue size.

Uniform Cost Search, thought optimal by design, quickly became impractical as puzzle complexity increased. Its lack of heuristic guidance caused it to expand an unnecessary number of nodes, resulting in long runtimes and large memory usage. While UCS succeeded in finding the shortest possible path, the computational cost incurred underscores its inefficiency for scalable, real-world applications.

A* search dramatically improved performance by leveraging an informed design. The Misplaced Train Heuristic, despite its simplicity and admissibility, often led the search astray. It significantly overestimated solution depth and frequently expanded nearly as many or more nodes than UCS. This proves that even an admissible heuristic can be ineffective if it fails to provide meaningful state differentiation.

The Manhattan Distance Heuristic, on the other hand, consistently delivered near-optimal or optimal solutions with minimal computational overhead. It outperformed both UCS and the Misplaced Train Heuristic across all puzzle complexity levels by expanding fewer nodes and requiring less time. Its superior performance illustrates the value of using domain-informed heuristics that not only accurately estimate goal proximity, but also reflect the underlying structure of the problem space.

This project showcases the critical role of heuristics in shaping efficiency and scalability of search algorithms. In practical domains like railway operations, where time, space, and resource constraints must be taken into account, algorithms like A* with well-crafted heuristics provide a viable pathway toward intelligently designed systems.

Works Cited

“How in the world did dijkstra come up with the shunting yards algorithm : r/computerscience.” *Reddit*, 24 November 2024, https://www.reddit.com/r/computerscience/comments/1gyjlno/how_in_the_world_did_dijkstra_come_up_with_the/. Accessed 5 May 2025.

“The Shunting Yard Algorithm.” *The Shunting Yard Algorithm*, https://mathcenter.oxford.emory.edu/site/cs171/shuntingYardAlgorithm/. Accessed 5 May 2025.

“When were locomotives first used for shunting? What were the first Shunting Engines?” *RMweb*, https://www.rmweb.co.uk/forums/topic/176556-when-were-locomotives-first-used-for-shunting-what-were-the-first-shunting-engines/?utm_source=chatgpt.com.

Code found at: https://github.com/Simar0108/RailroadShuntingSolver