Sekai CTF 2024 - Railgun Writeup

toxicpie

10 Sep. 2024

Abstract

In this article we detail a novel approach to the programming problem proposed by null_awe et al. (SekaiCTF 2024). Previous work requires convoluted reasoning processes that often result in non-intuitive constructions. We address this issue by presenting a technique that simplifies the solution through a reduction to a more difficult version of the problem, which enables an elegant and straightforward algorithm.

1. Introduction

The problem in interest was first introduced by null_awe in 2024 [1]. A simplified statement of it is as follows:

A positive integer $2 \leq n \leq 2000$ and two permutations p and q of $1, 2, \ldots, n$ are given.

There is an $n \times n$ grid. On the left side of the i-th row, a railgun with power p_i is fired at a 45-degree angle to the top-right. We may place reflectors along any horizontal edge in the grid. Railguns travel in straight lines and bounce off reflectors as well as the top and bottom sides of the grid. The objective is to position the reflectors such that the right side of the i-th row is hit by the railgun with power q_i.

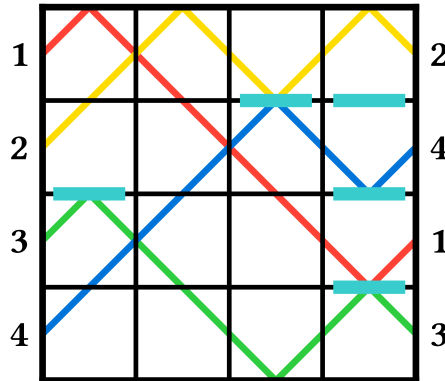
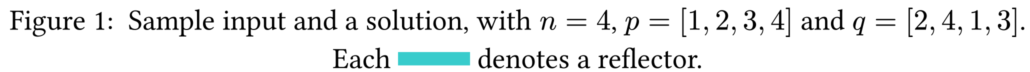


Figure 1: Sample input and a solution, with $n = 4$, $p = [1, 2, 3, 4]$ and $q = [2, 4, 1, 3]$.
Each  denotes a reflector.

The current best known solution is due to the problem author [2], using a very sophisticated strategy with multiple steps to reflect railguns into desired positions. In this paper we present an alternative construction with significantly lower algorithmic complexity, resulting in a clean implementation that is easier to understand.

2. Preliminaries

2.1. Reflectors and Swapping

First we consider the case where no reflectors are placed (see Figure 2). We follow the vertical order of railguns and move from left to right. Each time two railguns intersect, their relative order is reversed. This sequence of swapping determines the final result, i.e., which row railgun i would hit on the right side.

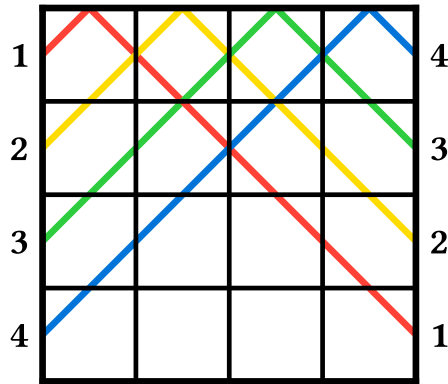


Figure 2: No reflectors.

When a reflector is placed on an edge where two railguns intersect (see Figure 3), their relative vertical positions are preserved instead of exchanged, so the final row they will hit at the end are swapped.

When a reflector is placed on an edge that only one railgun touches (see Figure 4), then it changes the path of that railgun, resulting in various complex situations.

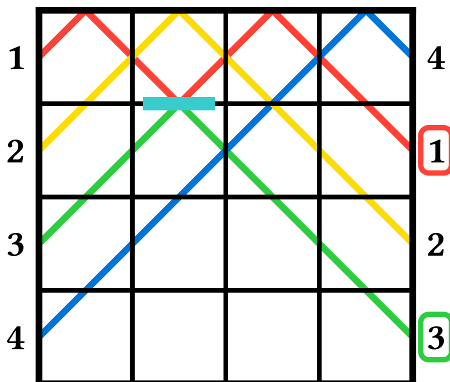


Figure 3: 1 and 3 are swapped at the end.

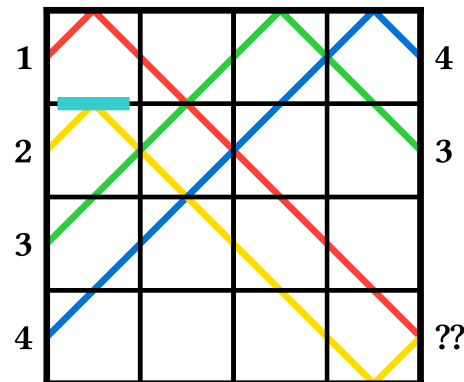


Figure 4: what

We may observe the following important property.

Theorem 1: Placing a reflector at a railgun intersection does not change the overall "shape" of railgun trajectories, i.e., the set of all paths that railguns pass through remains the same.

Proof: Obvious. □

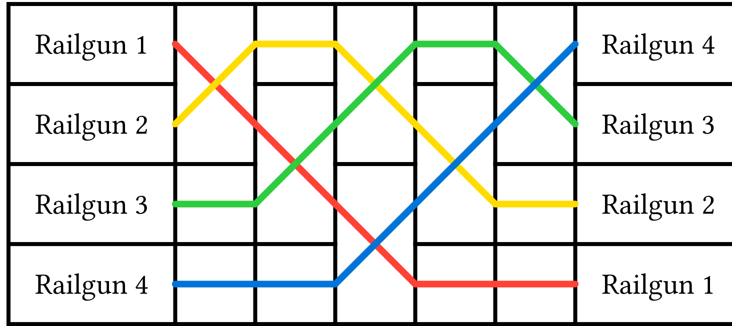
The core idea of our solution is to only consider reflectors placed on intersections. This way only swaps will occur, and we can avoid the complexity of reasoning about individual trajectories of railguns.

2.2. Sorting Network

As we have observed, if every reflector is placed at the intersection of two railguns, then the choice of "whether or not to place a reflector" is equivalent to "whether or not to swap two railguns". The problem therefore can be thought of as obtaining the permutation q from p using some sequence of swaps.

In addition, by Theorem 1, we know that the locations at which swaps can occur are fixed, and that each swap affects the vertical order of railguns at two fixed indices.

We can observe the structure of swaps from the 4×4 example shown in Figure 2. The following table shows the vertical positions of railguns as swaps happen:



Note that the order at which swaps occur resembles a bubble sort network [3], and the same pattern can be found in all $n \times n$ grids.

2.3. Railgun Parity

If we could control every swap, then the problem is solved as we can sort any permutation p into q using the sorting network. However, not every swap correspond to a reflector placement, as some of them occur on vertical edges and are therefore inevitable.

This unfortunate fact is due to parities. Two railguns with the same parity always meet on a horizontal edge, while two railguns with different parities always meet on vertical edge.

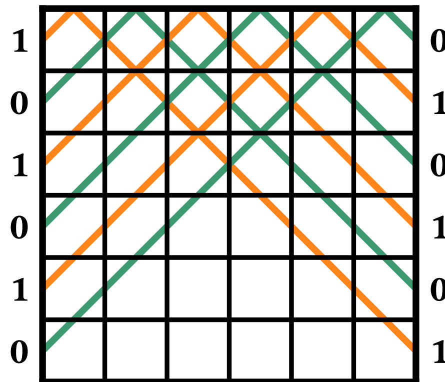


Figure 5: Railguns with two colors to denote different parities.

If we only place reflectors on horizontal edges where railguns intersect, then it is sufficient to sort railguns of both parities, but not enough to sort all railguns. Our main result is to devise a change that resolves this issue.

3. Main Results

3.1. Modifying The Problem

We introduce the following modification to the problem.

On the left of the i-th row, instead of firing one railgun, two railguns with power p_i are fired to the top-right and bottom-right respectively. The goal is to make *both* railguns with power q_i to hit the right side of the i-th row.

Theorem 2: Every valid solution to the above problem is also a solution to the original one.

Proof: Trivial. □

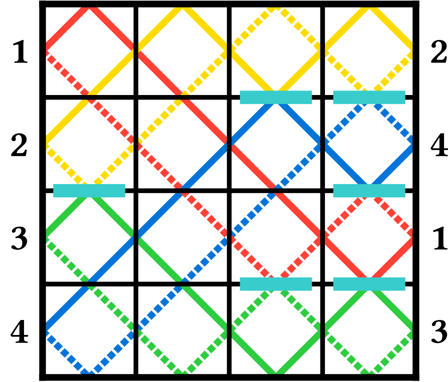


Figure 6: The same example with $n = 4$, $p = [1, 2, 3, 4]$ and $q = [2, 4, 1, 3]$, along with a solution.

We claim that there is always a solution to the modified problem, and the rest of this paper will be focused on solving it instead. While this version may seem more difficult at first, it actually enables some strategies that are much harder to see in the original.

3.2. Railgun Parity Revisited

Railguns in this version also come in parities. Figure 7 shows the parities of railguns in an example 5×5 grid, where railguns of the same parity intersect at horizontal edges, and railguns of different parities intersect at vertical edges.

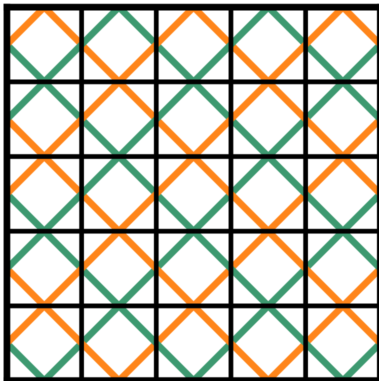


Figure 7: Railguns with two colors to denote different parities.

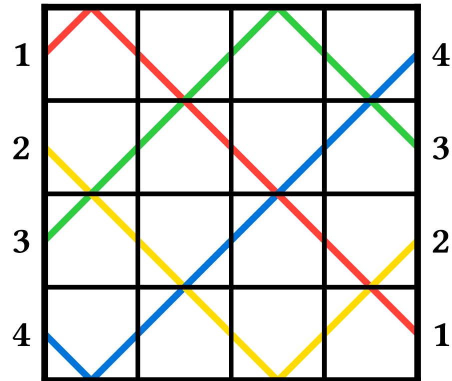
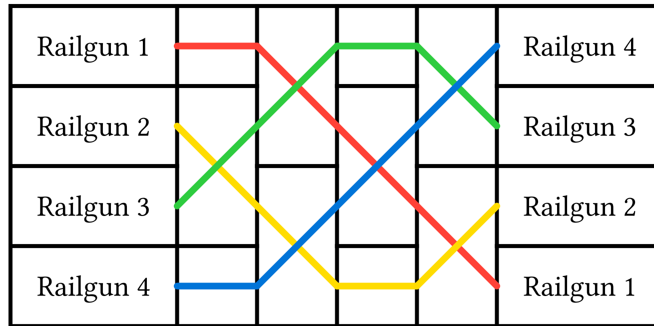


Figure 8: All railguns of one parity.

Since placing a reflector at a horizontal edge only affects two railguns of the same parity, by Theorem 1, the set of all paths followed by all railguns of a single parity doesn't change. In particular, notice that each row intially contains exactly one railgun of each parity, and so does each row at the end (see Figure 8). Therefore, we may solve the problem for each parity independently.

3.3. Sorting Network Revisited

Similar to an earlier section, we may observe the structure of swaps from the 4×4 example shown in Figure 8. The following table shows the vertical positions of railguns:



Notice that the indices of swaps resemble an odd-even sort network [3], and the same pattern can also be found in both parities, and in all $n \times n$ grids.

Since every swap occurs on an horizontal edge, we are able to control all of them by placing reflectors. Therefore, it is possible sort railguns of one parity from any permutation p into q using the sorting network.

3.4. The Algorithm

In conclusion, we can use all the above observations to solve our version of the problem:

1. Fire two railguns from each row.
2. For railguns of one parity, use the odd-even sort network to sort them into place.
3. Repeat for the other parity.

Appendix A. shows an optimized version that processes both parities at once.

Bibliography

[1] null_awe, "Railgun," Project SEKAI Programming Contest 2022, Aug. 2024.

[2] null_awe, "solve.cpp," Official source code and writeups for SekaiCTF 2024. [Online]. Available: https://github.com/project-sekai-ctf/sekaictf-2024/blob/main/ppc/railgun/solution/solve.cpp

[3] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching (Second Edition)*. 1997, pp. 219–247.

Appendix A. Reference Implementation

We provide a simple implementation for our solution with the following input format.

n

$p_1 \; p_2 \; \cdots \; p_n$

$q_1 \; q_2 \; \cdots \; q_n$

```
use std::collections::HashMap;
use io_utils::{read, reads};

fn main() {
    let n: usize = read();                               // read input
    let p: Vec<i32> = reads();
    let q: Vec<i32> = reads();

    let mut current: Vec<i32> =                          // fire two railguns from each row
        p.into_iter().flat_map(|x| [x, x]).collect();
    let target_idx: HashMap<i32, usize> =                // indices of elements in q
                                                         // the goal is to sort by it
        q.into_iter().enumerate().map(|(i, x)| (x, i)).collect();

    let mut answer = vec![vec![b'0'; n]; n - 1];
    for step in 0..n {                                   // iterate over columns
```

```rust
        for i in 1..n {                                     // perform swaps at horizontal edges
            let a = current[i * 2 - 1];
            let b = current[i * 2];
            if target_idx[&a] < target_idx[&b] {            // put reflector if no swap needed
                answer[i - 1][step] = b'1';
            } else {                                        // otherwise let them swap
                current.swap(i * 2 - 1, i * 2);
            }
        }
        for i in 0..n {                                     // perform swaps at vertical edges
            current.swap(i * 2, i * 2 + 1);
        }
    }
    for row in answer {
        println!("{}", std::str::from_utf8(&row).unwrap());
    }
}

mod io_utils {                                              // module for input helper functions
    fn read_line() -> String {
        let mut buf = String::new();
        std::io::stdin()
            .read_line(&mut buf)
            .expect("could not read line");
        buf
    }
    pub fn read<T>() -> T
    where
        T: std::str::FromStr,
        <T as std::str::FromStr>::Err: std::error::Error,
    {
        read_line().trim().parse().expect("parse error")
    }
    pub fn reads<T>() -> Vec<T>
    where
        T: std::str::FromStr,
        <T as std::str::FromStr>::Err: std::error::Error,
    {
        read_line()
            .split_whitespace()
            .map(|x| x.parse().expect("parse error"))
            .collect()
    }
}
```