# Internship Report

Murat Bolu

September 5, 2024

My internship at TÜBİTAK BİLGEM İLTAREN was from 1–29 July 2024 at the Ümitköy campus. My department was system administration software, and I completed the tasks in C++. My progress was monitored primarily by Emre Balcı and Murat Özdemiray.

## 1 Static Data Structures

I started the internship writing statically allocated data structures from scratch. Even though the C++ standard library provides some statically allocated data structures like `std::array`, I was requested to write them by myself and avoid using the standard library. Statically-allocated data structures are preferred in low-level applications for several reasons. First, they do not use `malloc` or `new`. These methods require an operating system or an equivalent program that will provide the necessary memory. These kinds of memory managers are not present in embedded applications and must be avoided. Second, they do not utilize the stack. While the stack may feel like an infinite memory source, stack overflow poses a significant risk for such applications. Statically allocated data structures allocate their memory at once while the program starts, and the stack is not utilized except for small memory allocations like `int`'s and pointers.

## 2 Traveling cities problem

### 2.1 Province distance data

In the first internship problem, I was provided an XLS file containing the distances between each pair of provinces in Turkey. I exported that XLS file to a CSV file and did some very light preprocessing by hand. I removed the first two lines containing the title text and junk information. Only 81 lines containing the names of the provinces and the distances from that province, separated by semicolons, were left. I wrote a parser that imported that text file into a data structure in memory. Finally, I had an array of arrays, with size 81 times 81, containing the distances as a number.

## 2.2 Problem description

With this data, my task was to visit as many provinces as possible, given some constraints. The distances between provinces were not considered except when applying constraints. After the constraints were applied, the problem transformed into the longest simple path problem in an undirected, unweighted graph, with the goal of visiting most vertices.

The longest simple path problem is finding the longest path in a graph without visiting any vertex more than once. It is an NP-Hard problem, and even the approximations are computationally complex. The best way to approach this problem is through heuristics. I have experimented with several heuristics and chosen to implement some of them.

In our graph, all pairs of provinces are connected through an edge. These edges have a value corresponding to the distance. Naturally, some of these edges must be invalid since the longest simple path in a connected graph is trivial. Such edges are invalidated through three constraints. The first is the starting province, and the second and the third are X and Y values. X value represents the distance, and Y value is the error margin of that distance. For example, if X is 200 and Y is 50, only edges with values between 150 and 250 are valid.

As mentioned, when the constraints are applied, the problem turns into the longest simple path in an undirected, unweighted graph, starting from a particular vertex. To tackle this problem, I first implemented the brute-force algorithm. It tried to solve for the longest route, naively searching in the space of routes. I added printing so that when it found a longer route than the previous maximum, it printed the length of the route and the route itself. The brute-force algorithm wasn't efficient because there were many redundancies while searching. I looked for heuristics to improve my search algorithm.

## 2.3 Heuristics research

I researched longest path heuristics and started reading the "Solving the Longest Simple Path Problem with Heuristic Search" paper by Yossi Cohen, Roni Stern, and Ariel Felner. It gave me some ideas, like caching the previous states and utilizing bi-connected components.

I implemented caching within the context of my problem's requirements. The algorithm searches the space of states and caches the best state seen so far. States hold the visited provinces in order, in a stack structure. States also keep a boolean array of visited provinces and the number of visited provinces for efficiency purposes. The best state is the state with the most visited provinces. The caching works as follows. When the algorithm encounters a new state, it compares its visited boolean array with the best state's visited boolean array. The state is skipped if the former is a subset of the latter. The reason is that if the best state contains all of the provinces visited by the current state, that branch is explored and can be skipped.

I tried implementing the bi-connected components heuristic. However, it didn't work as well as I expected. Even though I decided to scrap it, I'll explain it briefly here. The main idea is that a graph can be divided into bi-connected components, called block-cutpoint-tree. These components are only connected by a single edge, meaning that if a simple path is being searched, one cannot "go back" to a bi-connected component again. So, fully exploring a bi-connected component is the best idea before going to the next component. Block-cutpoint-tree can be constructed in linear time, so it isn't computationally costly.

I also tried to implement other algorithms, like Kahn's Algorithm for topological sorting and Tarjan's Strongly Connected Components Algorithm. These didn't work either because we could not consider our graph to be directed. In hindsight, it was intuitive that these approaches didn't work. If they worked, they would have been working in linear time, directly contradicting the NP-Hardness of the problem, assuming $P \neq NP$.

### 2.4 Longest route with best X and Y values

My last task for this problem was finding the longest route with the best X and Y values. The best can be defined arbitrarily, and I was requested to minimize the $X + Y^2$ value. I also fixed the starting province to Ankara for this task. The algorithm searches starting from the smallest $X + Y^2$ values, trying to find the longest path starting in Ankara. Since my algorithm takes too long to find the longest route, I applied a heuristic for early termination. If after a set number of iterations, say, ten thousand, the length of the best route does not change, the algorithm exits. Early termination allows the algorithm to exit if it cannot find a better route.

## 3 Bus scheduling problem

In the second internship task, I had to generate and analyze the data myself. Data generation is trivial; it is the timetable of a bus stop. Buses arrive at the bus stop intermittently, with fixed periods. The timetable includes the hours and minutes, and the number of buses that came at said times. Some buses have a single period and arrive at the stop at every period. Some buses have varying periods and arrive at the stop every other period. For example, if a bus has a period of 10 minutes, it will arrive at the stop every 10 minutes and will increase the counter for that timestamp by one. If a bus has alternating periods of 5 and 7 minutes, it will arrive at 5th, 12th, and 17th minute marks until the ending time. It should be noted that only the number of buses that arrived at a specific time is known.

The analysis of data is the trickier part. It aims to detangle the buses and identify them correctly. The problem would have been significantly more straightforward if the buses only had fixed periods. The algorithm would identify the bus with the highest frequency by observing the first data point and "subtract" its effect from the timetable. Then, it would identify the bus with the subsequent highest frequency using the same approach. Indeed, I applied this approach first, and it worked sufficiently for the trivial case. However, alternating periods pose a problem in this case. Even though two alternating periods have a proper period, which is their sum, the algorithm cannot work by trying to find the highest frequency by observing the first data point. I had to have a different approach.

My approach identifies the periods from the smallest to the largest and can discern between alternating and constant periods. If the period is alternating, the algorithm determines the sum of two oscillating periods and tries to identify the two different components. If the period is constant, it finds two identical "oscillating" components and reports one of them as the constant period instead.

There are two caveats to my approach. The first is that the data generator only generates periods smaller than half of the total time, i.e., it only identifies periods with at least two instances. This is known in signal processing theory as the Nyquist theorem. If a bus with a large period only arrives at the stop once, its period and type cannot be determined correctly. Second, while my algorithm can identify all buses correctly for almost all test cases, it can only identify some buses for extensive test cases. This issue may be because some buses have the same alternating periods, which cannot be detangled correctly. I tried to augment my algorithm to detect these cases individually. However, I couldn't do it correctly.

I also tried to utilize some of the techniques I learned in my signal processing course at university. I implemented the fast Fourier transform algorithm and analyzed the Fourier transform of the time series data. Theoretically, Fourier transform could identify the data's periodicity and yield the buses' periods. However, it was unnecessary for constant periods because the simple approach worked fine. For the alternating periods, I couldn't manage to analyze the FT data to extract the periods. Additionally, with the FT approach, the second caveat I mentioned earlier, identifying the buses with the same total alternating periods, would have persisted. That's why I abandoned this idea and implemented an ad-hoc algorithm.