1 Introduction

1.1 Problem Overview

The travelling salesman problem (TSP) asks a fundamental graph theory question.

"Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?"

The problem is a classical example of an NP-Complete problem, and there exist several algorithms which provide speedups over a naive, brute-force solution.

**Naive**, $O(n!)$ Examines all possible paths and returns the best.

**Held-Karp**, $O(n^2 2^n)$ A DP algorithm which prunes based on shortest paths.

There also exist many heuristic algorithms—those which find an approximation to the actual solution in a smaller time complexity. We will not be focusing on these methods; our implementation will strictly find the optimal solution.

We will first implement and optimize the naive TSP solution, followed by Held-Karp if time allows.

1.2 Test Set Sourcing

Our testing will consist of test sets taken from the FSU 'Travelling Salesman Problem Dataset'. Sample cases will include:

**FIVE**, a test-set with (5) cities.

**GR17**, a test-set with (17) cities.

**FRI26**, a test-set with (26) cities.

**ATT48**, a test-set with (48) cities.

---

1Wikipedia, 'Travelling Salesman Problem'

2https://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html
2 Proposal

Let us consider the brute force solution to the TSP and look at its implementation.

Naive Solution:
1. Consider city 1 as the starting and ending point.
2. Generate all (n-1)! Permutations of cities.
3. Calculate the cost of every permutation and keep track of the minimum cost permutation.
4. Return the permutation with minimum cost.

Time Complexity: \( O(n!) \)

The Haskell implementation of this solution involves generating all permutations of cities, each permutation being an element of a list. Each permutation is then evaluated sequentially and the permutation with min cost is returned.

Similar to Marlow’s approach, we aim to use the Par Monad and spawn sparks to evaluate a group of permutations in parallel.

A further optimization includes using RePA instead of lists to speed up computation.

If time allows, we will try to implement the dynamic programming (Held-Karp) solution using Accelerate for our arrays.