# Parallel Functional Programming Project Proposal

Solving the Traveling Salesman Problem with Genetic Algorithms

Timothy Johns
Uni:  tcj2114

## Project Overview

In this project, we'll use a Genetic Algorithm to find an approximate solution to the NP-hard Traveling Salesman Problem.

### Traveling Salesman Problem

In the Traveling Salesman Problem (TSP), we're given `N` cities, and the goal is to find the shortest path that starts and ends at the same city and reaches each of the other cities exactly once along the way.

### Genetic Algorithms

A Genetic Algorithm (GA) aims to solve an optimization problem through a process that mimics evolution.  In a typical GA, there is a population of `N` solution Candidates.  The "fitness score" for each candidate is a numeric representation of how well that candidate solves the optimization problem.  The GA will then follow this pseudo code:

```
Unset
1) Create initial population of N random candidates

2) Calculate fitness for each candidate

3) If the candidate with the best fitness so far is good enough, return it

4) Create the next population as an evolution of the current population

5) Repeat steps 2 through 4 until step 3 eventually breaks out of the loop
```

There is a wide variety in the ways that GAs implement the evolution part of step 4.  Typically, this will include selecting two "parent" candidates from the current population (with a preference for candidates with higher fitness) and producing a new "child" candidate as a combination of the parents' traits (this is known as "Crossover").  Then some of the traits are further randomized (which is known as "Mutation").

### Reference Implementation

The Coding Train on YouTube has a series of programming tutorials that cover solutions to TSP in JavaScript, culminating in a GA-based solution. For this project, we'll model a Haskell solution to TSP on the solution presented in those videos. The videos are intentionally beginner-friendly, so we'll likely optimize some of the core algorithm before adding parallelism (e.g. storing the distances between cities in a lookup table, instead of calculating the distances over and over).

### Parallelism

The pseudo code above describes a process that is inherently serial. We need to process each population in order, since the current population (and its Fitness scores) are used to generate the next population.

However, we can calculate the Fitness scores for the Candidates of a population in parallel and we can generate the Candidates for the next population in parallel. Doing so should allow us to speed up each iteration of the main loop, thus speeding up the whole program significantly.

For TSP, we will likely have a population size in the neighborhood of 100 to 1000, so generating 1 spark per Candidate should be reasonable. We will presumably use the `parList` strategy, or something similar. We can experiment with different population sizes and with grouping together sets of Candidates to process within a spark to see if this has any effect on the speedup. Similarly, we can generate a Candidate and calculate its Fitness in a single task, to see if cutting the number of sparks in half has an effect on the performance.

Note that the structure of this algorithm is quite similar to the structure of K-Means Clustering. In the pseudo code above, replace "population" with "clusters", replace "candidates" with "points", and replace "calculate fitness score" with "find nearest cluster", and we basically have the K-Means Clustering algorithm. So, we should be able to use what we learned in class for K-Means Clustering as inspiration for parallelizing TSP as well.

## Implementation Details

### City Generation

In the reference implementation, the number of cities is hard-coded, and then the city locations are randomly generated upon startup. In our implementation, we will either:

1. Pass in the number of cities as a command-line argument and let the program generate the city locations using a hard-coded random seed, so that the same set of cities is generated each time.
2. Or the city locations will be stored in a file that the program reads at startup.

We'll run this program for varying numbers of cities, so that we can investigate how the performance (both for single-threaded and parallel implementations) scales as the number of cities grows.

## Candidate Representation

In this implementation, the cities are given an ID from `0` to `N-1` and they have 2-D floating point coordinates. A Candidate solution is a path through the cities, which is represented by an `N`-element array of city IDs, covering each of `0` through `N-1` exactly once.

## Fitness Scores

A Candidate's total `path_length` is calculated by summing the Euclidean distance between cities along the path. Then, the Candidate's Fitness score is calculated as `1 / (path_length + 1)`, so that smaller path lengths have higher scores, and the extra 1 in the denominator protects against division by 0.

## Parent Selection

Parents are selected for reproduction based on their Fitness score. In particular, if a Candidate has score `X` and the total score for the whole population is `Y`, then the Candidate will be selected as the next parent with likelihood `X / Y`.

## Crossover

A child Candidate is produced from parent Candidates A and B by starting with some random contiguous subset of A's path, and then filling in the rest of the cities in the order that they appear in B's path. This guarantees that the child Candidate's path still covers all of the cities exactly once and inherits some similarity from both parents.

## Mutation

After creating a child Candidate via Crossover, the child Candidate's path can be Mutated by randomly swapping neighboring cities in the path.

## Stop Condition

The reference implementation does not actually implement step 3 in the above pseudo code. Instead, it just runs forever and shows the algorithm's progress visually in a browser. Our implementation will need to have a stop condition so that we can evaluate performance improvements. We'll go with one of the following approaches:

1. For a given set of cities, evaluate the actual optimal path using a different algorithm (e.g. brute force). Then, the GA can return the first Candidate solution whose path length is less than `X%` longer than the optimal solution.

2. Or run until `N` generations pass without finding any improvement in the best solution.
3. Or run for `N` generations, regardless of the results that have been found so far.

Option 1 isn't very practical, since we'd need to already have the answer to find the answer. Option 2 would make sense if our main goal was to solve TSP. But, since our main goal is really to optimize the parallel performance of this algorithm, we might go with option 3 so that our performance comparisons are comparing apples to apples.

## Handling Randomness

This project will involve a lot of random number generation. We should ensure that from one run to the next (for the same set of cities) we see the exact same sequence of random values. This way, we can rule out random number generation as a factor in performance differences between runs. This will also allow us to check that the final path that is returned is consistent for different parallel strategies and numbers of threads. To achieve this, we'll start the program with a hard-coded random seed (or perhaps pass it in as a command-line argument).

Generating random numbers in parallel tasks is also error prone. So, we will either:

1. Generate all of the random values needed by a task in the main thread and pass it as an input to parallel tasks.
2. Or in the main thread, generate a separate random seed to be used by each parallel task, so that they are not re-using the same random values.