3201 Project Report

George Neonakis

Scott Jennings

Elliot Moors

**1. Introduction**

The travelling salesman problem (TSP) is infamous in the fields of computer science and combinatorics. Given a set of points, or cities, the goal is to find the shortest possible route that passes through all of the points and ends at its starting position. Though it is conceptually very simple, the search space of the problem is enormous, with a complexity of the order n factorial – in fact the exact complexity is (n-1)!/2, as a given tour can be reversed or can start and end at any of the n points to yield the same result. Such a rapidly growing function very quickly becomes far too large to solve with brute force, and so must be approached in a more elegant, heuristic-driven manner.

Evolutionary algorithms (EAs) are one commonly used stochastic method for attempting to solve the TSP. EAs simulate the process of natural selection by generating a population of candidate solutions, combining and mutating their attributes to form “offspring”, and selecting the best individuals to continue into the next generation. This process is repeated until an acceptable solution is found, or until the population ceases to improve significantly. In this paper we will detail the design of an algorithm that we used to model the TSP, and will discuss the results we achieved in attempting to solve three instances of the problem: Western Sahara, Uruguay, and Canada.

**2. Methods**

2.1 Basic Design

In this section we will outline the design of the initial version of the algorithm, before advanced techniques were implemented.

*2.1.1 Implementation*

This project was implemented in Python 3. The majority of modules were written from scratch, but the additional libraries used were:

* numpy $
* matplotlib for visualization
* Python profiler for data collection $
* Other libraries? $

*2.1.2 Representation*

The most natural and obvious representation for a solution to the TSP is a permutation of all of the points in a given instance. $ We chose to represent our solutions by reading the points for a given data set from a text file and creating a dictionary that maps each point’s coordinates to an index. We then created Route objects that store a permutation of those indices in an array; this permutation then represents the path travelled through the points for that solution.

*2.1.3* *Initialization*

The population was initialized randomly by generating array permutations using Python’s random module.

*2.1.4 Evaluation*

For each individual, fitness was determined by calculating and summing the Euclidean distance between each pair of neighboring points. The TSP is thus a fitness minimization problem, i.e. lower fitness levels are better. We calculated the fitness for each Route individual upon creation, and stored it as a class variable for later access to avoid redundant calculations.

*2.1.5 Parent Selection*

We used a deterministic tournament selection mechanism to select the mating pool for each generation. Participants were chosen randomly from the population, with the winner being the individual with the lowest fitness in the tournament. This process was repeated until the mating pool reached a desired size.

*2.1.6 Recombination*

Our original recombination method was a standard order crossover algorithm; for each pair of parents, one offspring was generated by taking a randomly chosen sub-path from one of the parents, and then adding the remaining points in the order in which they occurred in the other parent; for the second offspring, the same process was used with the parents switched. Early in the project it became apparent that this was a significant bottleneck in our algorithm, due to the sheer number of iterations it required, and so recombination ended up being one of the main targets of our later optimizations.

*2.1.7 Mutation*

Our original mutation method was a basic swap mutation function, which simply chose two random points within a Route and switched their positions.

*2.1.8 Survivor Selection*

We used μ + λ selection with a fixed population size to determine which individuals would continue into the next generation. The original population was combined with the offspring and sorted by increasing fitness; the first μ individuals (μ = population size) in the combined list were then kept for the next generation, and the rest were discarded.

*2.1.9 Staling*

To prevent our algorithm from continuing to run unnecessarily while making negligible progress, we introduced a “staling” (or stagnation) mechanism. After each generation, the average fitness of the population was compared with the previous average to determine whether there has been a significant change; if several generations passed successively without any significant change, the program terminated. The minimum required ratio of change and the number of generations before staling occurred were set as modifiable parameters (discussed in section 2.3).

*2.1.10 Profiler*

$ We used a profiler to collect data on the efficiency of our algorithm. At the end of each run, the profiler displayed the total runtime of the program, as well as the number of calls to each function and the cumulative amount of time each function took to complete. This helped us determine what the main weaknesses of our algorithm were when we set out to make optimizations.

*2.1.11 Graphing*

Using the matplotlib library, we added functionality to display two graphs at the end of each program run. The first graph plots the average fitness and best fitness trend lines over time (measured in generations), and the second graph visualizes the route represented by the individual with the best fitness.

2.2 Advanced Design & Optimizations

After the base implementation was complete, we set out to make a number of optimizations to its efficiency and effectiveness. This section will describe the major changes we made, not necessarily in chronological order.

*2.2.1 Improved Crossover*

As a result of the sheer computational complexity involved with the Travelling Salesmen Problem, we knew that operations which involved iteration through individuals needed to be well optimized. Knowing this, we considered the properties of a tour and took advantage of the fact that any circular tour where you see every city exactly once need not have any specific starting location. That is, shifting every element in the individual an equal amount would not have any effect on the fitness of that individual. Using this property, we generate two random points *r­­1*, *r2*, and copy the region [*r1*, *r2*­] from both parents into each offspring, but shifting the elements to the start of the offspring. From here we refill the remainder of the offspring without the need to insert any elements to the front of the list, and without the need to wraparound the individual. As a result, we were able to check for duplicate elements much quicker (O (1) down from O (*m*) where *m* is the number of elements in the interval [*r1*, *r2*]) and bring the runtime of the function to O (*n*) where *n* is the number of elements in an individual.

*2.2.2 Dynamic Mutation*

In the early phases of the project, we had intended to utilize a fitness sharing or “niching” mechanism – which would involve preserving low-fitness sub-paths in mutation and recombination – as one of our main advanced techniques. However, we found that it was difficult to implement this feature without adding a great deal of complexity to the indexing and iteration of the routes. Ultimately, this lead to the development of a feature we call “heuristic swap”; rather than attempting to identify desirable sub-paths and maintain them, we opted to instead systematically target abnormally long sub-paths with our mutation mechanism, thus achieving a similar effect.

We then continued to develop this feature into a three-step dynamic mutation mechanism that changes at given checkpoints in the program. It begins by using scramble mutation (randomly choosing a subset of a Route and shuffling it), as it is a very destructive form of mutation that can cause significant drops in fitness in the early stages of the algorithm, when most of the Routes are chaotic and very few sub-paths are anywhere close to optimal. It then transitions to the heuristic swap mutation described above in order to more specifically target individual sub-paths that are contributing relatively large shares of the overall fitness. Finally, it switches to basic swap mutation in the later stages before terminating, as heuristic swap is unable to make smaller-scale optimizations to the Route, which are necessary to minimize fitness as much as possible.

*2.2.3 Precomputing Distances*

One simple way to increase the speed of our evaluation method was to precompute the distances between every city in the tour and storing the data in a dictionary for quicker lookup. By precomputing a dictionary of dictionaries, one for each city, we were able to pass over the data once, which shortened the computation time significantly, the largest example Canada taking only 10 seconds. After all the distances were computed our fitness function was changed to simply lookup the distances between each city in the tour, and as a result gave better performance.

2.3 Parameters

The parameters used to refine the behavior of our algorithm are as follows:

* Population size
* Generation limit
* Mating pool size
* Tournament size
* Crossover/recombination rate
* Mutation rate
* Staling thresholds for dynamic mutation and termination
* Staling limit (# of successive generations without significant change required for staling)
* Heuristic swap fitness threshold (minimum relative fitness required for heuristic swap to target a sub-path)