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 O(n!) – 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 solving the TSP (Eiben, 2008). EAs simulate 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 cycle repeats 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, and the Python profiler.

2.1.2 Representation & Initialization

The most obvious representation for a solution is a permutation of all of the points in a given instance. We first read the points from a text file and created a dictionary that maps each point's coordinates to an index. We created Route objects that store a permutation of those indices in an array; this permutation represents the path travelled through the points for that solution. The population was initialized by randomly generating a number of these permutations.

2.1.3 Evaluation

Fitness was determined by summing the Euclidean distance between each pair of neighboring points; the TSP is thus a fitness minimization problem. We calculated the fitness for each Route upon creation, and stored it as a class variable for later access to avoid redundant calculations.

2.1.4 Parent Selection

We used deterministic tournament selection to pick the mating pool for each generation. Participants were chosen randomly, 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.

2.1.5 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, and so recombination ended up being one of the main targets of our later optimizations.

2.1.6 Mutation

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

2.1.7 Survivor Selection

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

2.1.8 Staling

To prevent our algorithm from continuing to run while making negligible progress, we added 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).

2.1.9 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 parts of our algorithm needed to be improved.

2.1.10 Graphing

Using matplotlib, we added functionality to display two graphs at the end of each 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 completing the base implementation, we made a number of optimizations to its efficiency and effectiveness. This section will describe the major changes we made.

2.2.1 Improved Crossover

Due to the sheer complexity of the TSP, operations involving iteration through individuals needed to be well optimized. We considered the properties of a tour and took advantage of the fact that any circular tour need not have a specific starting point, i.e. shifting every element in the individual an equal amount would not have any effect on its fitness (Cicirello, 2006). We

generated two random points r_1, r_2, and copied the region [r_1, r_2] from both parents to the start of each offspring. We then filled the remainder of the offspring without the need to insert any elements to the front of the list, or to wraparound the individual. Thus, 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 [r_1, r_2]) 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

Early in the project, we had intended to utilize a mechanism which would involve preserving low-fitness sub-paths as one of our advanced techniques (Peng, 2016). 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 maintain desirable sub-paths, we opted to systematically target abnormally long sub-paths via mutation, thus achieving a similar effect. We continued to develop this feature into a dynamic mutation mechanism that changes at given checkpoints. 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 few sub-paths are optimal. It then transitions to our heuristic swap mutation in order to specifically target 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 was to precompute the distances between every city in the tour, and store 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, with 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 (500, 1000, or 2000)
- Generation limit (10000)
- Mating pool size (0.5 * population size)
- Tournament size (4)
- Crossover/recombination rate (0.9)
- Mutation rate (0.1, value suggested by Rexhepi, 2013)
- Staling thresholds for dynamic mutation and termination (population size * 0.25, * 5, and * 10 respectively for heuristic swap, basic swap, and termination)
- Staling limit (# of generations without significant change required for staling, set to 10)
- Heuristic swap threshold (min. relative fitness for heuristic swap to target a sub-path, set to either f / 2 or f / 0.75, where f = mean length of all sub-paths in an individual)

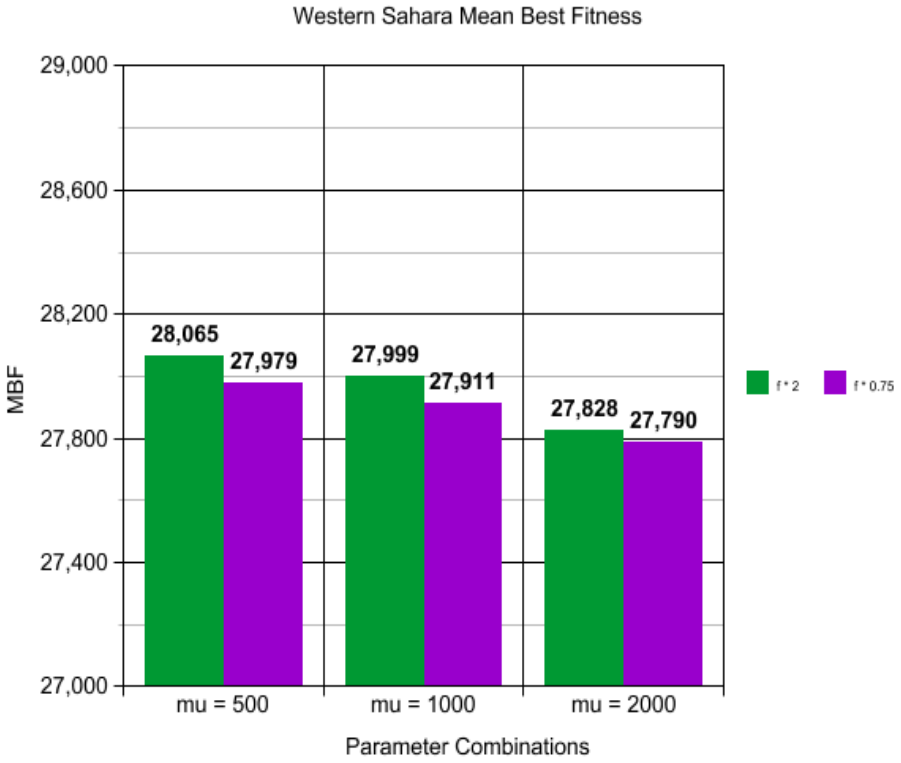
In our data collection, we tested 6 different conditions using different combinations of population size (500, 1000, 2000) and heuristic swap threshold (f / 2, f / 0.75).

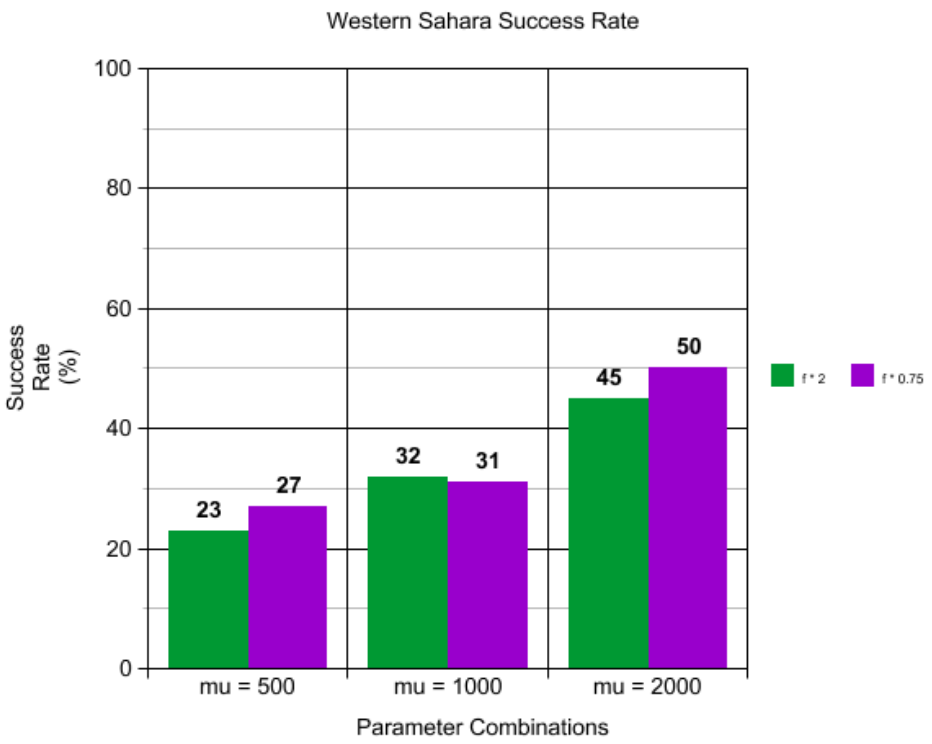
3. Results

3.1 Western Sahara

The following figures illustrate the results for Western Sahara. Note that success rate was determined by the number of runs that reached a fitness of 27601 (actual best fitness is 27603, slightly lower due to rounding errors).

SAHARA (n = 100)	Mean Best Fitness	Best Fitness SD	Mean Generations	Generations SD	Mean Time (seconds)	Time SD	Success Rate (%)
500 x 2	28065.08	435.48	166.61	18.54	2.00	0.22	23
500 x 0.75	27979.24	365.35	165.04	16.04	2.07	0.22	27
1000 x 2	27999.37	387.58	161.15	13.89	4.13	0.40	32
1000 x 0.75	27911.39	330.00	164.10	13.31	3.99	0.35	31
2000 x 2	27828.96	292.72	162.61	12.53	8.01	0.70	45
2000 x 0.75	27790.73	271.78	162.18	12.42	7.89	0.59	50

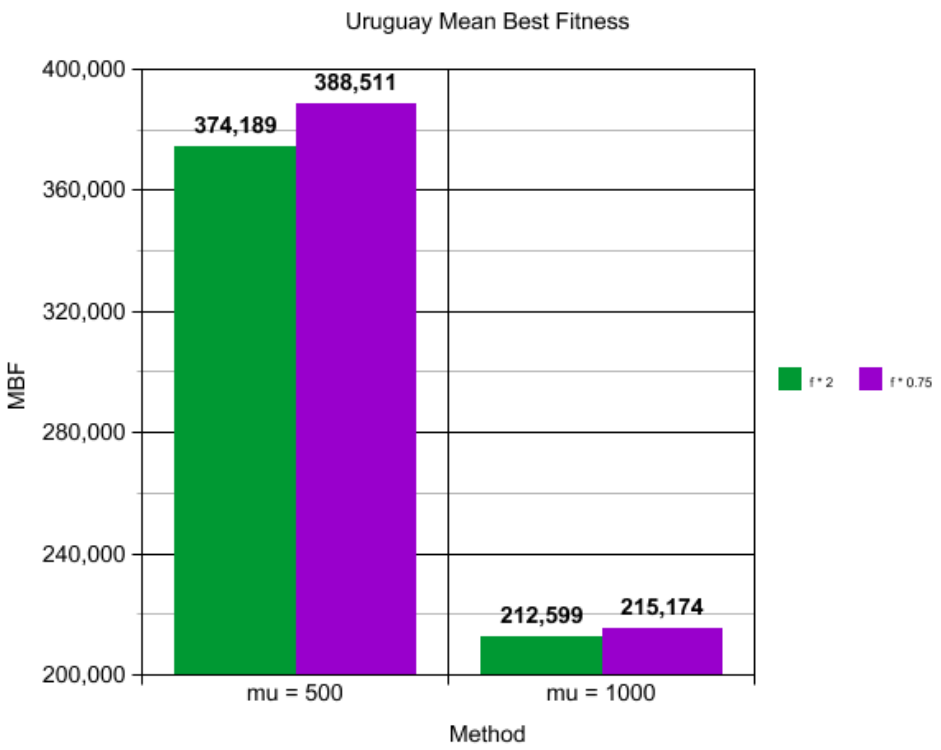




3.2 Uruguay

The following figures illustrate the results for Uruguay. Note that the 2000 population size was not used for these tests due to the excessive runtime of the algorithm; some tests were also run simultaneously on different processors, which may have interfered with runtimes.

URUGUAY (n = 10)	Mean Best Fitness	Best Fitness SD	Mean Generations	Generations SD	Mean Time (SD)	Time SD
500 x 2	374189.10	25899.01	2676.00	257.78	569.67	67.92
500 x 0.75	388511.35	31182.62	2370.50	303.47	794.14	169.30
1000 x 2	212599.30	13970.54	5593.00	455.74	3306.84	659.70
1000 x 0.75	215174.50	15962.65	5278.10	523.94	3105.63	838.66



4.3 Canada

Ultimately, the algorithm was unable to make any significant progress in the Canada instance of the problem due to its size, and the runtime was too long to gather enough data for statistical analyses. Of the few Canada runs that were completed throughout the development of the algorithm, the best fitness achieved was 52565739 in 1928 generations using the 1000 x 2 parameter settings.

4. Discussion

The results show that the algorithm is able to solve the Western Sahara instance about 50% of the time using the best combination of the given settings (2000 x 0.75). The effect of population size is clear, with the worst-performing combination scoring a MBF within 1.7% of the solution, more than double the proximity of the best-performing combination (0.7%), and less than half of the best success rate (23% vs 50%). The effect of the mutation threshold is also noticeable, with

the higher threshold consistently yielding lower MBFs at no significant cost to runtime. Interestingly enough, the generation count appears to be entirely unaffected by any of the parameters in this instance.

As for the Uruguay instance, population size is even more significant of a factor, with the higher population size nearly cutting the MBF in half. However, the heuristic threshold value seems to have the opposite effect here, converging faster and causing marginally higher MBF values. It is possible that the sample size was too low to yield representative results, though it may also be the case that the heuristic swap criteria are too simplistic to have any major impact on larger instances of the problem. While the theory behind heuristic swap seems to be logically sound, perhaps its implementation requires more refinement to accurately and consistently eliminate large sub-paths from the routes.

Overall, the algorithm achieved its primary goal of solving the TSP with reasonable consistency for the Western Sahara instance, and came somewhat close to the optimal fitness of 79114 for the Uruguay instance. There are certainly more optimizations that can be made to allow the algorithm to maintain the higher population sizes that are recommended in the literature without reaching overwhelmingly long runtimes; with a few more upgrades to its efficiency, and possibly also to the mechanics of the dynamic mutation mechanism, it could very likely solve the Uruguay instance, and maybe even the Canada instance as well.

5. References

Cicirello, Vincent A. (2006) 'Non-wrapping order crossover: An order preserving crossover operator that respects absolute position', *Genetic and Evolutionary Computation Conference.*

Eiben, A.E. and Smith, J.E. (2008) *Introduction to Evolutionary Computing (Natural Computing Series)*, 2 edn., Springer.

Peng, Xingguang, Liu, Kun and Jin, Yaochu (2016) 'A dynamic optimization approach to the design of cooperative co-evolutionary algorithms', *Knowledge-Based Systems*, 109(C), pp. 174-186.

Rexhepi, Avni, Maxhuni, Adnan and Dika, Agni (2013) 'Analysis of the impact of parameters values on the Genetic Algorithm for TSP', *IJCSI International Journal of Computer Science Issues*, 10(1), pp. 158-164.