AM5080 PROJECT

Parallelized Genetic Algorithm for the Travelling Salesman problem

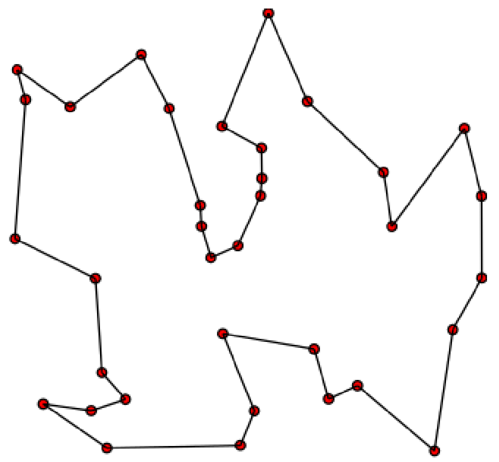
---

ADVAIT V DHOPESHWARKAR - CE17B024

Table of Content:

Introduction:

The Travelling salesman problem (TSP) is a very popular problem in computer science and network flows. In the TSP we ask the following 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?**”. For example, in the figure below, the dots represent the city locations and the closed path indicates the shortest possible tour (cycle) that connects every city.



It is a NP-hard problem in combinatorial optimization Which means that there are no known polynomial time algorithms to solve the TSP. There are various other versions of this problem like the travelling purchaser problem, vehicle routing problem etc.
The TSP was formulated in the year 1930 and is one of the most extensively studied problems in optimization. There are many known heuristics as well as exact algorithms to solve the TSP all of which have an exponential time complexity.

In this project, I have made an attempt to solve (get good acceptable upper bounds on the optimal tour for) the TSP problem using the parallelized version of the genetic algorithm, which is a heuristic algorithm. My goal was to get reasonable upper bounds to a problem size of 200-300 which is a fairly large problem size considering the complexity of the problem.

Brief introduction to the Genetic Algorithm and its use case in the TSP:

Genetic algorithm is a popular metaheuristic algorithm inspired by the process of natural selection that belongs to the class of evolutionary algorithms. Genetic algorithms are generally used for generating high quality solutions to optimization and search problems by relying on biologically inspired operators such as mutation, crossover and selection.
In the genetic algorithm (GA henceforth) a population of candidate solutions is evolved to a better solution. Each individual or candidate solution has a set of properties (chromosomes or

genotype) associated with it which can be mutated or altered. The evolution starts with an initial population of individuals and an iterative process with the population in each iteration referred to as a generation. In each generation, the fitness of every individual in the population is calculated. Fitness is the value of the associated objective function. The fitter individuals are stochastically selected from the current population and each individual's genotype is modified (recombined through crossover and/or mutated) to form the next generation (population for the next iteration).

In our case, candidate solutions (individual/phenotypes) are city tours. For this we use an order of cities in the form of an array. This order decides the order in which the cities are visited by the salesman. The chromosomes of a phenotype are the cities and their relative order in which they appear within each phenotype. The objective function for the TSP is the tour length.

The following are the steps involved in the Genetic algorithm:
Initialization: We start with an initial population of individuals (candidate solutions).
Selection: During each successive generation, a portion of the existing population is selected to breed a new generation. Individual solutions are selected through a fitness-based process, where fitter solutions (as measured by a fitness function) are typically more likely to be selected. Certain selection methods rate the fitness of each solution and preferentially select the best solutions. Other methods rate only a random sample of the population, as the former process may be very time-consuming.
Genetic Operators: The next step is to generate a second generation population of solutions from those selected through a combination of genetic operators: **crossover** (also called recombination), and **mutation**.

1. ***Crossover :*** A pair of “Parent” solutions are selected from the previous generation and are crossed over in a certain way so that the “child” solution gets its characteristics partly from either of its parents. This process is the crossover process.
2. ***Mutation :*** A number of changes (random) are introduced to the genotype of an individual. This simulates the mutation process in nature.

Need for parallelization:

As discussed earlier, the number of possible solutions to check increases exponentially with problem size in the case of the TSP problem thus a need for parallelization for speed up arises. Genetic algorithms require large population sizes and often the genotype of individuals can become very long. For example, in TSP problems where the size of genotype, which is represented by the order of the cities in which they are visited, depends on the number of cities involved. Thus, making the problem computationally expensive to solve using the GA. Thus, parallelized implementation of the algorithm can be used so that the computational tasks can be distributed to different CPUs making the algorithm faster.

Parallelization variants:

Now, I shall discuss the variants of the parallel genetic algorithm and the implementation that I have used in this project for solving the TSP problem.
Parallel implementations of genetic algorithms come in two flavors. Coarse-grained parallel genetic algorithms assume a population on each of the computer nodes and migration of individuals among the nodes. Fine-grained parallel genetic algorithms assume an individual on each processor node which acts with neighboring individuals for selection and reproduction. Other variants, like genetic algorithms for online optimization problems, introduce time-dependence or noise in the fitness function.
I have used the coarse-grained parallel genetic algorithm version for the project. This variant can also be viewed in terms of the “Island model”. In this we use multiple genetic algorithms with different starting populations on each node to solve a single task . All these algorithms try to solve the same task and after they’ve completed their job, the best individual of every algorithm is selected, then the best of them is selected, and this is the solution to a problem. This is one of the most popular approaches to parallel genetic algorithms, even though there are others. This approach is called the ‘island model’ because populations are isolated from each other, like real-life creature populations may be isolated living on different islands. Additionally the phenotypes on each of the islands may occasionally migrate to another island resulting in crossover among the populations on different islands.

The following diagram illustrates the generic “Island model” :

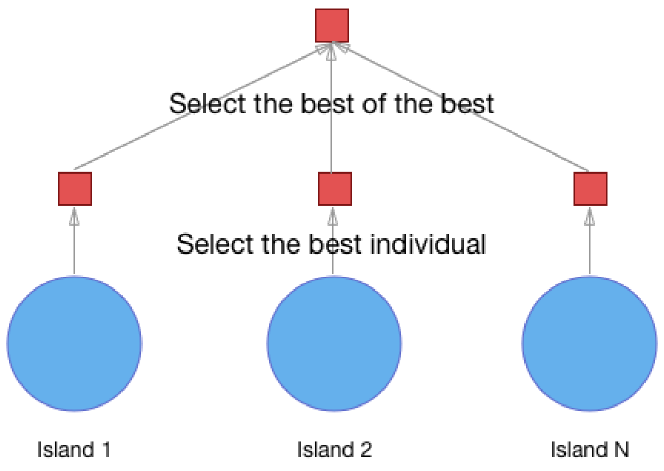


Image 1. Parallel genetic algorithm

The implementation I have used is based on the “Island model” where each of the processes represents an island. We solve the problem by starting with different sets of populations on each of these processes (islands) and run the genetic algorithm. After a few fixed number of iterations, there is a migration of individuals from one island to the neighbouring ((rank+1)%size)th process (island). After the required number of iterations, the solution from each of the processes is compared and the best is the solution(current best upper bound) to the problem.

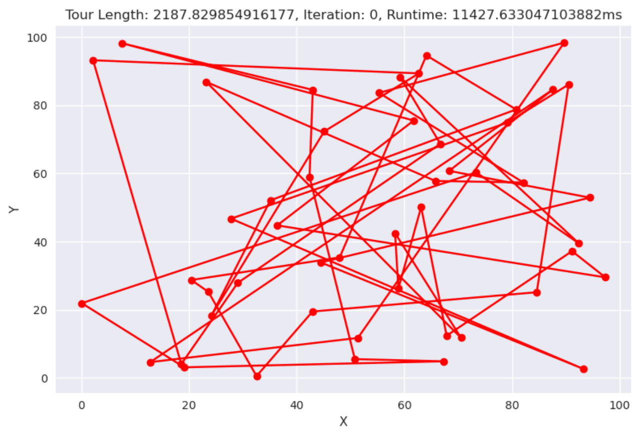
Observation and test runs:

The following are the test runs for different problem sizes, number of processes, and population size per process. For ease of comparison, total population has been kept constant ie. number of processes x population size per process = roughly constant.

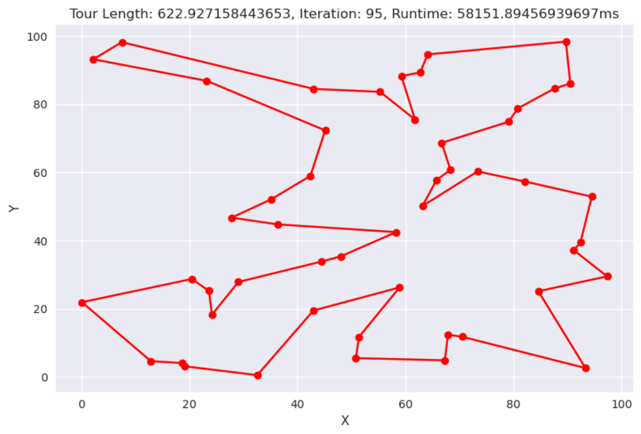
For the following, the test case for 50 cities is used from the testCase50.csv file which is generated using the generate_test_case.py program. Thus same city locations are used for 4 runs below for 1, 2, 3 and 4 processes respectively. This example is used to test for the quality of solution obtained and the time of execution of the program for different numbers of processes. I have used only upto 4 processes due to my hardware availability constraints at my end.

1)Number of cities=50, number of processes=1, population=50, number of iterations=100

Starting best solution:

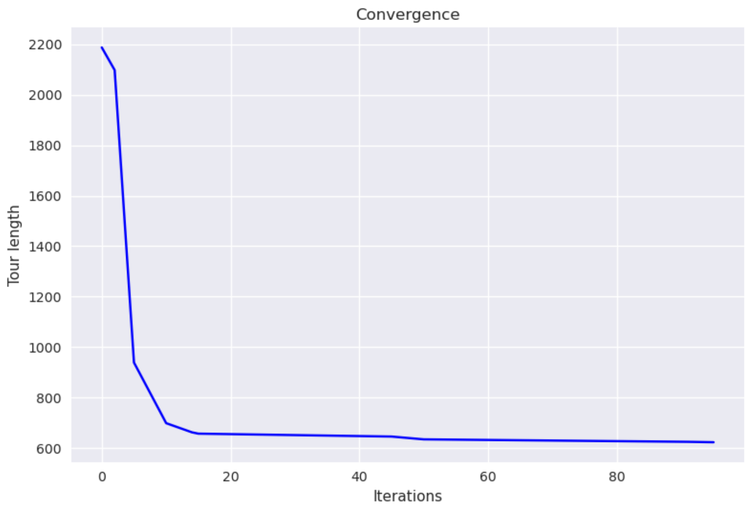


Best solution obtained:



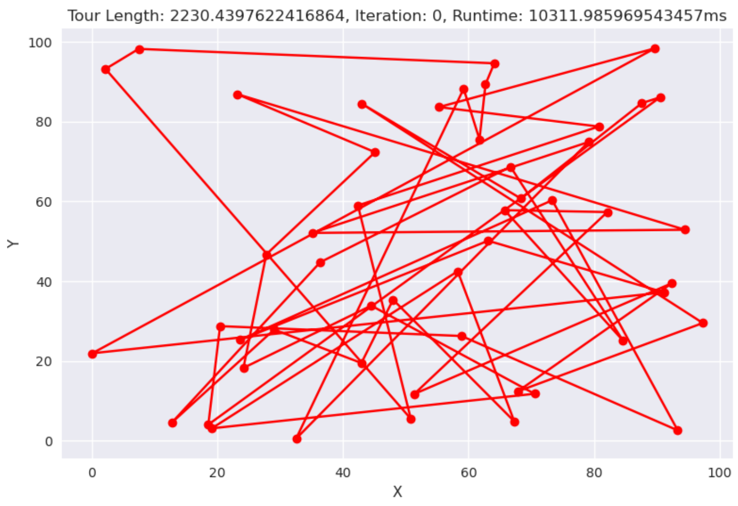
Tour Length=**622.92**, Runtime=**58151** ms

Convergence plot (tour length vs iteration):

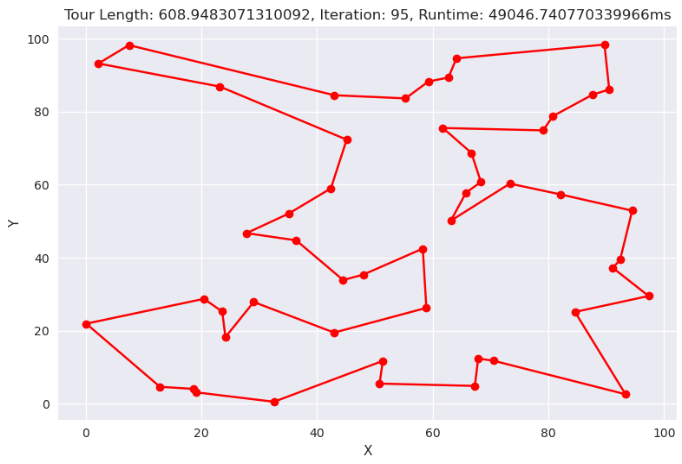


2) Number of cities=50, number of processes=2, population=25, number of iterations=100

Starting best solution:

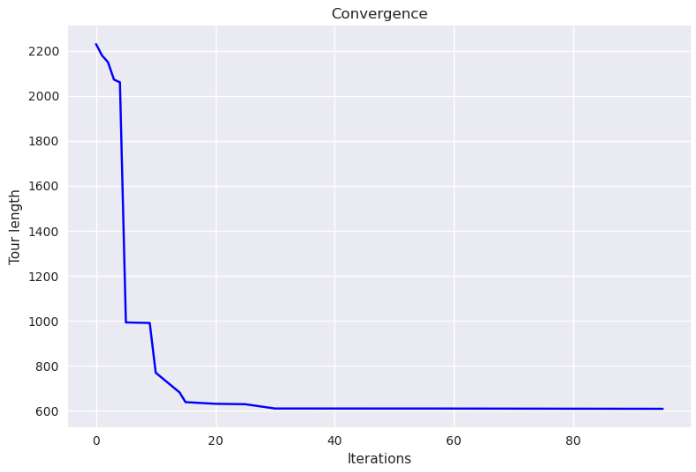


Best solution obtained:



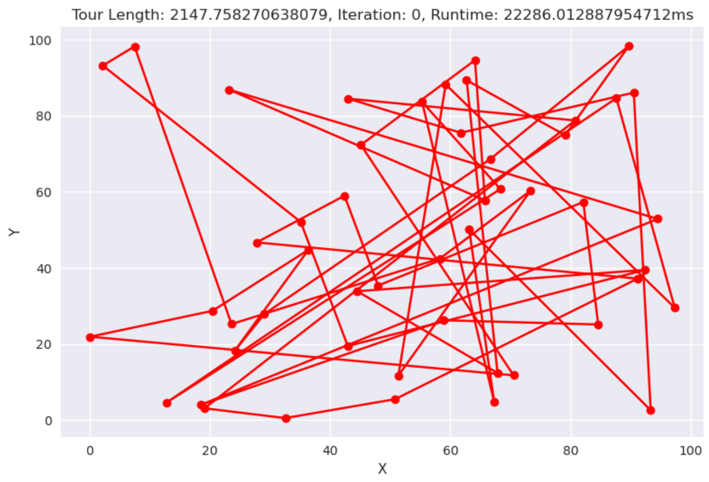
Tour Length=**608.94**, Runtime=**49046**

Convergence plot (tour length vs iteration):

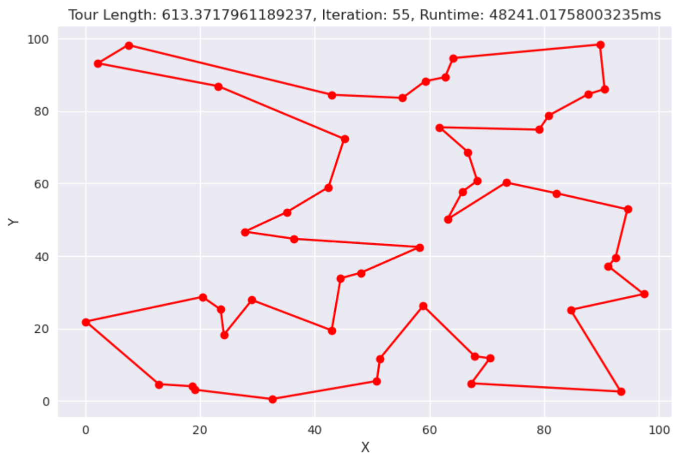


3)Number of cities=50, number of processes=3, population=17, number of iterations=100

Starting best solution:

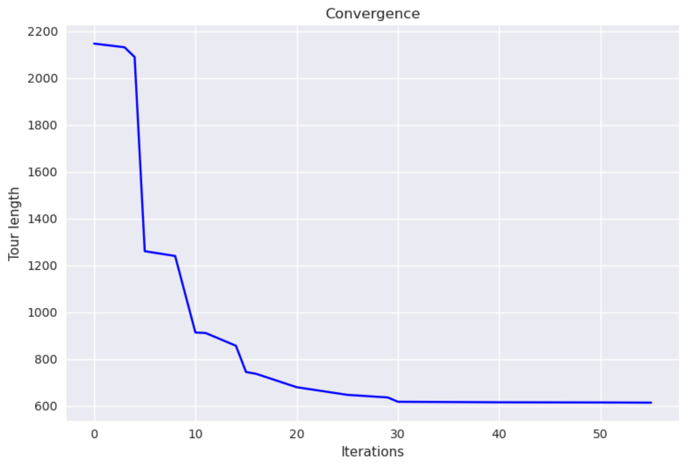


Best solution obtained:



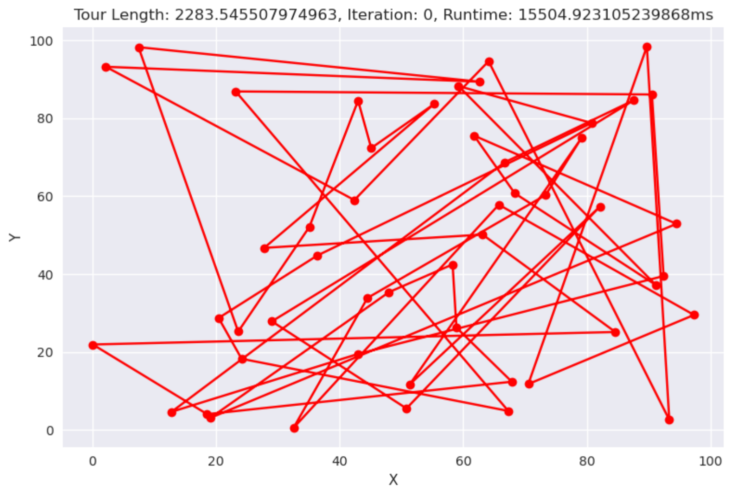
Tour length=**613.37**, Runtime=**48241** ms

Convergence plot (tour length vs iteration):

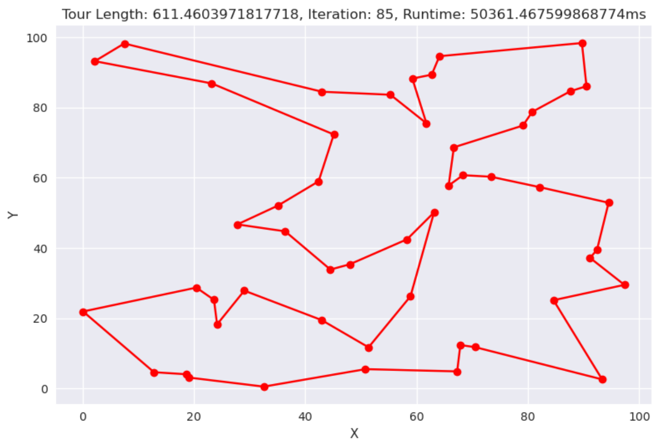


4)Number of cities=50, number of processes=4, population=12, number of iterations=100

Starting best solution:

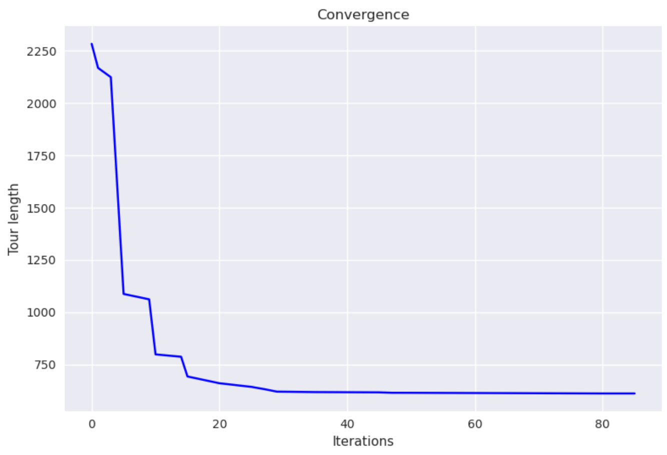


Best solution obtained:

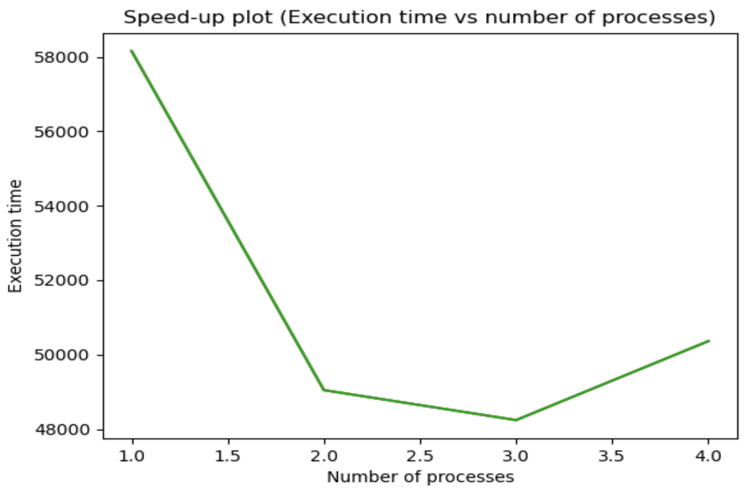


Tour Length=**611.46**, Runtime=**50361**

Convergence plot (tour length vs iteration):



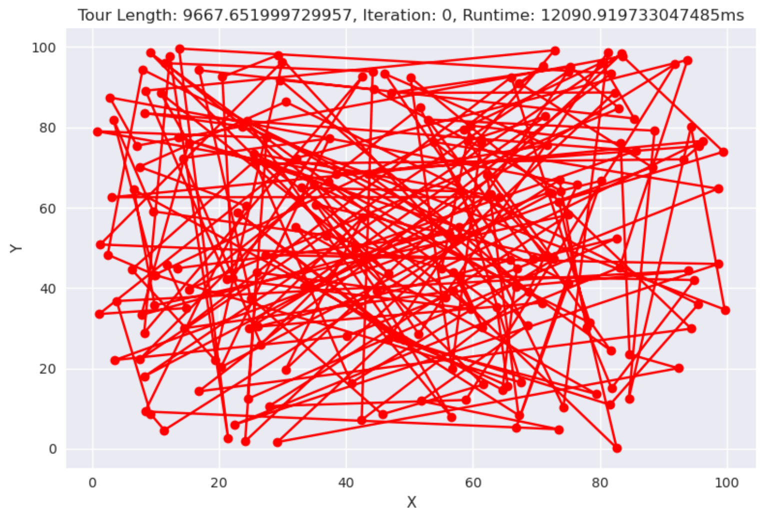
Speed up plot: (Runtime vs number of processes)



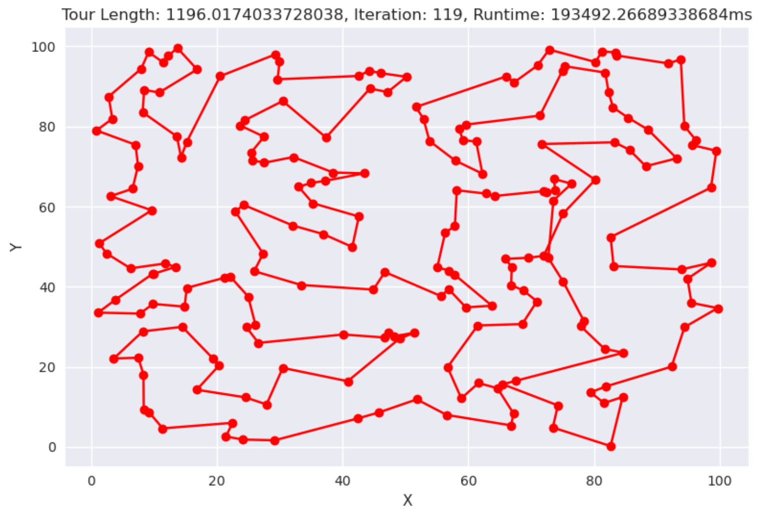
- As can be observed from the test case of 50 cities, a reasonable upper bound is obtained within 100 iterations. With a larger number of iterations exact solutions can also be obtained for problem sizes of 50-100 cities.
- We can also observe that we get a speedup in the algorithm when a larger number of processes are used. I obtained the best result for 3 processes in the above example.
- We can see that with genetic algorithms, we can get a reasonable upper bound to the TSP very quickly. This may be used as a starting solution in other algorithms.
- Genetic algorithm performs much better than brute force algorithm for solving TSP which tests every solution naively.

Some of the results obtained for different problem sizes and different parameters:
Results of a sample run for a problem size of 200, using 3 processes, a population size of 20(per process) and maximum of 200 iterations :

Starting best:

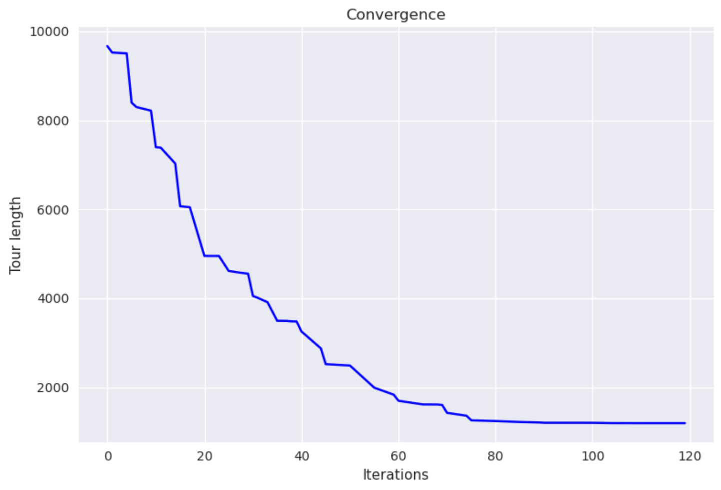


Best solution obtained:

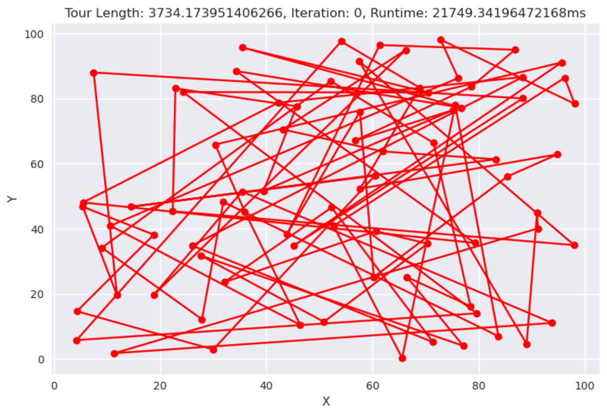


Tour Length=**1196.01**, Runtime=**193.5 seconds**

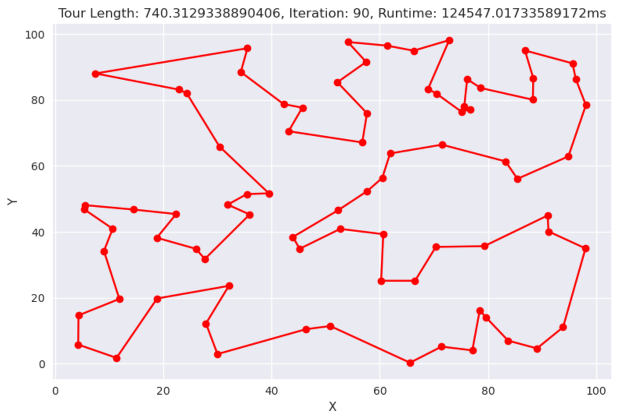
Convergence plot (tour length vs iteration):



Results of a sample run for a problem size of 80, using 2 processes, a population size (per process) of 40 and maximum of 100 iterations :
Starting best solution:

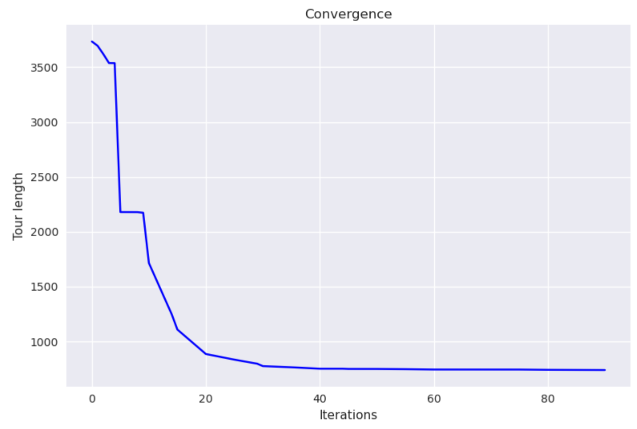


Best solution obtained:



Tour Length=**740.31**, Runtime=**124.5 seconds**

Convergence plot (tour length vs iteration):



Conclusion and remarks

From the observations we can make the following conclusions:

- Genetic algorithms are a very good way to get a good upper bound for TSP problems, even with a problem size of 200-300 (which is a fairly complex problem as there are O(n!) Number of possible solutions) relatively quickly.
- Once we have a good upper bound, we may use it to check for and validate results from other better algorithms or even use it as a starting point for other algorithms.
- We can achieve a speedup in the algorithm by using a parallel implementation of the algorithm.
- There are multiple ways of parallelizing the algorithm. These can be broadly seen as Coarse-grained and Fine-grained types. Coarse-grained implementation is easier to implement and gives decent results. It uses the "Island model" approach.
- The genetic operators used in the algorithm can be customized based on the application of the algorithm. There is no single concrete way of implementing the mutation or cross over operators while implementing the algorithm.
- Using a larger population typically yields better end results in terms of tour length but there are more computations involved and thus may require larger runtime for a fixed number of iterations of the algorithm.

---

*Note on code file, and submission files:

The submission includes two code files: **"generate_test_case.py"** and **"parallel_tsp_island_migration.py".** The **"generate_test_case.py"** is used to generate the common test case to be tested across multiple program runs. It is however not mandatory to

generate any test case as the main **“parallel_tsp_island_migration.py”** has an option to generate the points(randomly), but these will not be stored and therefore will not be available for use across other runs. The **“generate_test_case.py”** generates a csv file called **“testCase.csc”** in which the test case will be stored. This file will be available in the folder after it is generated.
Once **“parallel_tsp_island_migration.py”** is executed, the results are stored in the **Results** folder in the current working directory. But the old results are lost therefore make sure to change the name of the folder after each execution.
If you want to use any of the already available test cases in the program, you will have to rename the corresponding csv file as **“testCase.csv”** and run the program.