Solving Travelling Salesman Problem using SA and GA (48 US Cities)

Student ID: 2351856, Module Leader: Jens Christian Claussen,
School of Computer Science, University of Birmingham, UK, B15 2TT

***Abstract*—Metaheuristic algorithms have proved that they could be great solvers for the Travelling Salesman Problem (TSP). However, due to their unique functionality, it is very difficult for the developers to say which one of them is the most efficient. For that reason, the present paper focuses on the comparison between two metaheuristic algorithms, the Simulated Annealing (SA) and the Genetic Algorithm (GA), for solving the Travelling Salesman Problem with the 48 capitals of the US. The performance of the two algorithms was investigated and compared through several computational experiments. Through these experiments, in terms of solution quality, the Simulated Annealing seems to be better than the Genetic Algorithm.**

I. Keywords

Travelling Salesman Problem, optimization problem, 48 US capitals, Simulated Annealing, Genetic Algorithm, metaheuristic algorithms, MATLAB, Wilcoxon signed-rank test.

II. Introduction

IN computer science, metaheuristic refers to a method of computation that optimizes a problem by trying to maximise the solution quality and minimise at the same time the number of resources that we need. Unfortunately, metaheuristics do not guarantee that the optimal solution will be found but they can help to produce near-optimal results.

In this case, the focus is on the Travelling Salesman Problem (TSP), which for a wide range of industrial applications, is one of the most well-known and researched combinatorial problems in terms of the efficiency of its solution approach. It is an NP-hard problem and asks 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?” [1].

The purpose of this paper is to examine the above circumstances in practice and efficiently obtain the good solutions by improving the search process. So, considering the generic structure (pseudocode) of the two evolutionary algorithms [2], the solution to TSP was implemented by writing on MATLAB my own version of the Simulated Annealing (SA) and the Genetic Algorithm (GA) code.

In the following sections, the mechanism of each algorithm will be described, as well as some experimental tests regarding the parameter tuning and the cost (or so-called objective or fitness) function. Finally, a statistical comparison, which was implemented by the Wilcoxon signed-rank test, will be analyzed in detail.

III. Simulating Annealing and Genetic Algorithm

A. Simulated Annealing and Genetic Algorithm within the concept of the Travelling Salesman Problem

In the current TSP scenario, SA and GA work in a search space consisting of a set of permutations of 48 US cities. The goal is to find the tour with the least cost by taking into account some constraints. For example, the node is required to visit every other node only once in a way that the total covered distance will be minimized [3].

Considering the above, an important factor regarding the algorithms' implementation is the balance (trade-off) between exploration and exploitation. On the one hand, the proposed algorithms should achieve a good exploration that corresponds to a great diversity of tours (search large unknown regions of the search space without getting stuck in the local optima). On the other hand, they should also achieve good exploitation which means to be capable to find local optimum by searching the local areas (exploit the current good solutions to get even better solutions).

To achieve that trade-off, SA and GA use a combination of the randomised search (for good exploration) and the local search technique (for good exploitation). In this way, they can explore the good region of the search space and find a global solution.

In general, SA and GA have many features in common. For example, both are stochastic search algorithms and their procedure is similar.

First, the algorithms calculate the (Euclidean) distance between the cities through a cost function by passing an initial 'random' solution (e.g., cities' coordinates). Next, based on the result, they generate a subsequent candidate solution and evaluate each one by passing it as an argument in the cost function. Finally, they repeat the above steps until either some convergence criterion is satisfied or the maximum number of iterations has been reached.

However, the main difference between SA and GA is the number of candidate solutions that they maintain for evaluation. In particular, GA maintains a population of candidate solutions whereas SA maintains only one best solution.

Lastly, one more important note is that the TSP can be formulated as a linear programming problem, which means that the cost function and constraints are linear. In other words, the variables are required to have integer values.

B. Implementation of Simulated Annealing

Simulated Annealing was one of the first generic heuristic algorithms for solving combinatorial problems such as TSP scenarios. It is a probabilistic technique used for approximating the overall minimum (global optima) of a function that may possess several local minima [4]. It is inspired by annealing in metallurgy. The process starts by heating the metal to a high temperature and slowly cools it. The notion of slow cooling refers to the probability of accepting worse solutions which automatically allows for a more extensive search for the global optimal solution.

The procedure starts with a large number of possible solutions where the SA's responsibility is to keep both good and bad solutions until eventually yields to a small set of optimal solutions and then keep the best one of them. Precisely, in every iteration, SA decides if need to replace or stay with the existing solution while the pool becomes more and more strict, mimicking the slow cooling of metallic annealing. It does not always reject changes that decrease or increase the cost function. This means that SA avoids sticking to local optima (good exploitation) by accepting worse solutions according to the probability function:

$$P = exp(\frac{-\Delta f}{T})$$

where Δf is the variation in the objective function and T is the temperature's control parameter.

The probability calculation is proportional to the quality of the solution. To be more specific, P is determined by how far or close the new solution (e_new) is to the existing one (e), thus, it can be represented according to the following circumstances:

• If $e\_new < e$ (e_new better than e), accepting better solutions with $P := 1$

• Otherwise, accepting worse solutions with a certain probability $P := exp(\frac{enew-e}{T})$

Furthermore, it was mentioned above that SA randomly generates a large number of possible solutions. Every new possible solution is characterised as the neighbour solution of the existing one. To generate a new solution, each time, we randomly apply one of the following strategies (Figure 1):

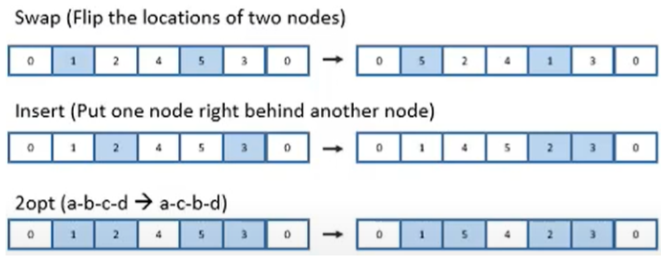


Figure 1 - Strategies of generating a new solution

1. SwapSearch: Given two cities with index i and j respectively, swap the location of $city[i]$ with $city[j]$.

2. InsertionSearch: Given two cities with index i and j respectively, put one city right behind the other one.

3. 2-opt [5]: Take a route that crosses over itself and reorder it so that it does not happen. In other words, the algorithm randomly picks two indexes (two cities for swapping) and splits the route (initial solution) into three parts. The first part starts from the beginning of the route until before the city with the first index. The second part takes the two selected cities including the ones between them and reverses their order. Lastly, the third part starts after the second index until the end of the route. Finally, the algorithm merges these three parts back together creating in that way a new route.

$$Xnew = \begin{cases} swap(X), & if\ r \geq 0.33 \\ insert(X), & if\ 0.66 \geq r > 0.33 \\ 2opt(X), & if\ r > 0.66 \end{cases}$$

Figure 2 - Strategies of generating a new solution (pseudocode)

Looking at the figure above (Figure 2), we can see that the choice of strategy to be implemented in each iteration is based on the parameter r. Here, the ability of good exploration relates to the larger values of the probability of accepting an inferior neighbor solution.

The idea comes from the randomized search approach, wherein each iteration, it creates a random number r from a range between 0 and 1. Next, for the neighbour selection we divide the probabilities evenly and based on which range of probabilities r is larger, SA executes the corresponding strategy.

Similarly, the same random number is compared with the probability parameter P mentioned above. In this way, SA decides if it should replace the existing solution or not (see pseudocode in Appendix B).

Based on the following flowchart (Appendix A) and pseudocode (Appendix B) which I designed, the main idea behind SA can be summarized in the following steps:

1. Initialise the necessary parameters (initial solution, temperature control parameter and an iteration counter). Note here that our initial solution is a random permutation of 48 cities.

2. Define the initial solution as the best/optimal solution and calculate its cost function. This is the sum of the (Euclidean) distances between all the cities of the solution for a particular order.

3. Next, a while-loop starts and it is controlled by the temperature parameter. It stops when reaching the temperature limit (specifically, when it will be decreased enough to be less or equal to 1 because there is no annealing after that point).

4. For each iteration, we pick a fit neighbour of the existing solution by passing the coordinates of this existing solution to the $neighbour()$ function. As explained earlier, this part of the code provides a good exploration capability as the $neighbour()$ function creates a new solution by randomly selecting one of the previous-mentioned three strategies.

5. We calculate the new (Euclidean) distance/cost-function using the coordinates of the new solution.

6. Once the probability is calculated. The SA decides at random whether it is going to replace the existing solution with the new one or not. If it decides to replace, remember that SA can accept this new solution even if it is worse than the existing one. If the SA accepts a better or worse solution is based on the above-mention conditions related to the phenomenon that the probability is proportional to the quality of the solution.

Also, this characterizes the ability of the algorithm for good exploitation, since the algorithm can accept worse solutions, it does not stop at the local optima and can continue the search in the rest of the space (exploit good/worse solutions to get even better solutions).

7. Next, the SA checks whether the new solution is better than the current best one by comparing the results of the objective function of the new solution and the objective function of the best solution. If the new solution is better, replace it.

8. Increase the number of iterations. If SA reaches the maximum iterations, it decreases the temperature by multiplying it with a cooling factor and then setting the number of iterations to 0. Decreasing the temperature means that decrease the chance of accepting a worse solution. The lower the temperature, the closer we are to the optimal solution. So, the pool becomes more and more strict and eventually yields to a smaller set of optimal solutions.

9. After all these comparisons and the while-loop is broken, output the final best solution.

Appendix C shows the results of the SA algorithm running with the best parameters. The result with 10648 units can be described as one of the best of this SA algorithm.

The implementation of all that was explained in this subsection can be found in the code files provided in the same folder as this report. (e.g., simulated-annealing.m, swapSearch.m, insertionSearch.m, twoOptSearch.m, etc.).

C. Genetic Algorithm

In the field of evolutionary computation, a Genetic Algorithm is a technique for solving optimization models (such as TSP), and it is inspired by approaches adopted from the field of genetics in biology.

In this technique, the algorithm starts by taking a finite set of individuals called population and each solution of the search space is represented by one individual/chromosome. In the next phase, each individual is evaluated based on the fitness function which measures the quality of the chromosomes in the population. Then, GA selects the two best fittest chromosomes from the current generation (exploitation). Then, it allows these two models/parents to mate and breed new individuals/offsprings by applying crossovers and/or mutations (exploration). Once it recalculates the cost of the new individuals, the existing population is discarded as the new offspring forms the next generation by replacing the least-fit individuals. Thereby, individuals with better fitness are more likely to survive in the population's next generation. Then, the winners might mate to produce the next generation.

In this research, a local searcher named 2-opt (same as it was applied to SA) is used as an extension of the Genetic Algorithm. In this way, the Genetic Algorithm provides new search areas, while 2-opt improves convergence.

The above procedure can be described with the following flowchart (Appendix G) and pseudocode (Appendix H and I) which I designed. In connection with TSP, the steps are:

1. Chromosome representation: There are several representation methods that can be used for the TSP (e.g., path representation, binary representation, adjacency representation, ordinal representation etc.). Considering the most natural way to present a tour, in this project, I use the "path representation" method where the cities are listed in the order in which they are visited. For example, consider a tour with four cities, if a salesman goes from city 1 to 2, to 3, to 4 sequentially and then comes back to city 1, the path representation of the tour will be as shown in Figure 3.

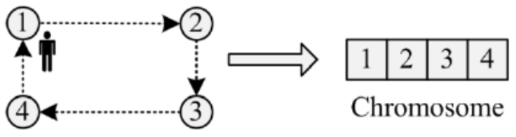


Figure 3 - Path representation [7]

2. Population initialization: The goal in the population construction phase is to return a population so that each row contains a set of individuals (genome). The initial population is randomly generated and there are several common techniques that can be used (e.g., random binary, random integer, random permutations, etc.). In this case, I use random permutation because TSP is a permutation problem. Note, an efficient production of an initial population of solutions is very important for achieving good results because starting with a good set of solutions, next generations can end up with even better ones.

3. Evaluation of fitness function: Once the population is generated, we calculate the Euclidean distance of each individual. Again, because TSP is a minimization problem, we always want to have the shortest distance.

4. Selection operation: Through the selection step, two parents are selected for crossover. Some selection methods evaluate the fitness of each solution and preferably select the best solutions. Other methods choose a random sample of the population, as the first-mentioned procedure can be very time-consuming. In this scenario, working with the 48 cities is not that much time-consuming, so the two parents with the best fitness scores are always selected (no randomly). Besides, at this point, the selection step is used to exploit the 'good' genetic material in the current set. There are also other selection methods such as roulette wheel selection, tournament selection, ranking selection, etc.

5. Crossover operation: The main purpose is to create two offspring (children) coming from the parents chosen through the selection phase. For combinatorial problems, there are different methods (e.g, uniform, one-point, two-point crossover etc.). Because for TSP the horizontal genome transcription is acceptable, the two-point crossover is appropriate. Compared to single-point crossover, the endpoints of the genome are not forced to be the endpoints of crossover, thus, making the procedure easier to implement.

Take into account that the "1-cross point (CP)" bit in $offspringl$ and $offspring2$ is the 1-CP bit in $parent1$ and $parent2$, respectively. Moreoever, if the remaining bits in $parent2$ do not exist in $offspring1$, the remaining bits in offspring1 will be filled with the remaining values of the $parent2$ (similarly, for $offspring2$). Finally, a crossover rate (probability) is defined so that randomly choose if the GA will apply the crossover technique for the currently obtained chromosome to form the children.

6. Mutation operation: If the crossover was successful, the next step is mutation. The genetic diversity of populations is increasing by the mutation operation. In this way, it prevents the GA to be trapped in a local minimum. For the purpose of this project, "Swap Mutation" was chosen as it is the easiest one to implement. In swap mutation, we select two positions on the chromosome at random and we swap them (Figure 4). This method is common in permutation-based encodings.

Finally, a low mutation rate (probability) is defined so that randomly choose if the GA will apply the mutation technique for the currently obtained chromosome to form the children.

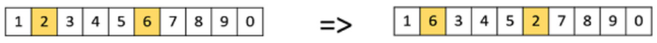


Figure 4 - Swap Mutation representation

8. Local search: In order to improve the quality of the solution, a 2-Opt local searcher is applied to make modifications to a genome and create better genomes at a higher rate. Two edges of the current tour are replaced by two other edges in order to achieve a shorter tour. However, care must be taken, because it can also reduce diversity in the population and make the algorithm converge to some local optimal. The difference is noticeable by comparing the results of the $OutputTestGAwithout2OptSwap.m$ file with the $OutputTestGAwith2OptSwap.m$ file (runs GA 30 times with/without the 2-Opt function).

9. Evaluation and Reproduction: Calculates the fitness of the two new offspring and creates a new population by find the worse individuals and replacing them with these two new offspring. Again, individuals with better fitness are more likely to survive in the population's next generation.

10. Elitism: This is the last step where we collect all the best individuals from each iteration. We do not want to destroy the best solutions which we got so far, otherwise, those individuals might being lost during this selection scheme as even the best individuals can have a probability of not be selected.

11. Termination Criteria: In this paper, the maximum number of iterations is used as the termination criterion.

For more information, please read also the paper "Analyzing the Performance of Mutation Operators to Solve the Travelling" [9].

IV. Experimental testing

A. The two phases of the experiment

Parameters are the configurable elements of every algorithm, thus, the choice of what values these parameters will take, can have a significant impact on the performance of the algorithm.

The experimental procedure was split into two phases. The first phase executes about of 30 trial runs for each algorithm to tune the parameters. The second phase (after obtaining good parameters from the first phase) executes 30 independent runs with 10000 fitness (objective function) evaluations for each algorithm. Also, it records the average (Euclidean) distance and standard deviation from the results over that 30 runs for each algorithm, respectively.

All the results/records of the following experiments can also be found altogether in the excel file provided with that report. Additionally, you can find the log files of SA and GA in their project folders.

B. Phase 1 - Parameter tuning with SA

The parameter tuning for the SA algorithm was based on the research of Fischetti and Stringher [6]. There are three parameters that needed to be tuned.

First, consider that the update procedure of the temperature parameter T is based on a cooling factor $\in (0, 1)$ and its typical values are $0.95\ to\ 0.99$ (for when cooling is applied to each SA iteration).

Besides, the cooling process is applied only when the algorithm reaches the maximum number of iterations. Thus, the maximum number of iterations can be considered as the second parameter for tuning. During the experiment, the typical values were 100, 500, 1000. Higher values were also tested (e.g., 10000), but in the end, there was not much difference.

Fischetti and Stringher emphasize that the temperature control parameter is one of the most crucial parameters. The temperature T iteratively decreases during the SA execution so as to make worsening moves less and less likely in the final iterations. In other words:

• If T gets larger, the probability P of accepting more worse solutions increases as the SA search space becomes wider.

• After each iteration, T decreases, the probability P of accepting worse solutions also decreases because the SA search space becomes too narrow so that the algorithm be able to converge at some point.

Therefore, T is the third parameter for tuning and we set the initial temperature high, because we want a high probability of accepting worse solutions. During the experiment, the typical values were 100, 1000, 10000.

Examining the results of the parameter tuning test, in Appendix D, we notice that there are two parameter combinations that give us the same shorter Euclidean distance (10648 units). Comparing the time elapsed (or the number of repetitions performed by SA) of both combinations, the best parameter combination is:

$$[T = 1000, CoolingRate = 0.99, MaximumIterations = 500]$$

Of course, if we choose the other combination, SA will work with more repetitions, therefore, it will be more secure that we will have the best possible results. However, the goal is to strike a balance between the time complexity and the quality of the algorithm results. Also, the number of repetitions performed by the first combination is good enough to achieve the goal.

In this experiment, the range of SA results is between 10600 and 11700 units. All these results are very good, which means that all these combinations could be used to solve the current TSP. Although, to distinguish the importance of parameter tuning and that the selected parameter combinations are appropriate, in another experiment (see Appendix E), I run the SA algorithm with smaller parameter values. We can see that the results are much worse than before (the range of results ranges from 14000 to 41000 units).

For a real-time experience, you can try to execute the file $'testSAparametertuningreq4.m'$ provided in the same folder as this report.

C. Phase 1 - Parameter tuning with GA

The parameter tuning for the GA algorithm was based on Mosayebi and Sodhi's research [8]. There were four parameters that needed to be tuned and about 30 iterations were executed. The following candidate values were chosen:

1. Population size [70, 100]
2. Maximum iterations [100, 500, 1000]
3. Mutation rate (probability) [0.8, 0.9]
4. Crossover rate (probability) [0.02, 0.1, 0.8]

Looking at the results in Appendix J, we can see that each parameter combination gives us completely different numbers (the distance values are not such close to each other as in the SA experiment). Thus, it is easier here to choose the best parameters which are:

$$Populationsize = 70$$

$$Maximumiterations = 1000$$

$$Mutationrate = 0.1$$

$$Crossoverrate = 0.9$$

The positive side is that using the above parameter combination, we have a small population size (thus, less nested iterations) and a low mutation rate. Otherwise, the GA would do faster steps, as a result, to not be able to maintain diversity within the population and prevent premature convergence.

Regarding the maximum iterations, on the one hand, its value can be considered as high if we think that this occurs a high number of objective functions evaluations, thus more comparisons. On the other hand, with more iterations, we have a higher probability to achieve an approximate optima which on this experiment (Appendix J), we got a good result wherewith less iteration might not get it.

Lastly, the crossover and mutation rate determines the percentage of the population that will change. Unlike for crossover, for mutation we want it to occur in a small portion of the population. Thus, the mutation rate must be very low and the crossover rate high. In our case, both within the appropriate standards.

For a real-time experience, you can try to execute the file $'testGAparametertuningreq4.m'$ provided in the same folder as this report.

D. Phase 2 - Average and Standard deviation of SA Objective function

The SA's objective function is the distance calculation function, thus, for the objective function testing process, for getting 10000 evaluations need to setup the temperature's T value to 10000. The experiments results after 30 iterations with T = 10000 are shown in Appendix F. The overall results are great as the given distances are very low and do not differ significantly from each other (a range between 10628 and 11089 units).

For a real-time experience, you can try to execute the file $'testSA10000fitnessreq5'$ provided in the same folder as this report.

E. Phase 2 - Average and Standard deviation of GA Objective function

The GA's objective function is also the distance calculation function, thus, for the objective function testing process, for getting 10000 evaluations need to setup 100 population size and 100 maximum iterations (100x100 = 10000). The experiments results after 30 iterations with $populationSize$ = $maximumIterations$ = 100 are shown in the Appendix K.

We can agree that the results are not ideal and this is because the maximum number of iterations is not high enough, which means that GA requires more time/iterations in order to find a good optimal solution.

For a real-time experience, you can try to execute the file $'testSA10000fitnessreq5'$ provided in the same folder as this report.

V. Wilcoxon signed-rank test

The last part of the project was to compare the results for SA and GA statistically using a Wilcoxon signed-rank test [9]. In the second phase of the experiment part, SA and GA were run 30 times and executed 10000 evaluations. To compare the performance, for each algorithm, we are taking the best distances from the second phase and passing them as arrays to the MATLAB functions called $signrank()$ to run the Wilcoxon signed-rank test.

Comparing the results of Appendix K and Appendix F, in each iteration, the distance SA is always smaller/better than the distance obtained from GA. Regarding the Wilcoxon signature classification test, from the results (Appendices L and M), the value $h = 1$ indicates that the test rejects the null hypothesis which proves that there is a sufficient difference between the SA and the GA algorithm.

VI. Conclusion

The aim of this paper was to compare the Simulated Annealing and the Genetic Algorithm by solving the Travelling Salesman problem of the 48 US Cities. Flowcharts and Pseudoces were provided in order to represent the different combinations of strategies that each algorithm implemented. However, for both algorithms, the main idea was based on the trade-off between exploitation and exploration in order to reach the global optima.

To a great extent, both algorithms are very good solvers and can provide optimal solutions if the right set of parameters are set. For Simulated Annealing, if the priority is the solution quality, the cooling rate should be set close to one but this increases the number of iterations that the algorithm will perform. For Genetic Algorithm, setting a larger population size, then the higher the possibility of getting an optimal solution but again, the execution time increases exponentially. Finally, based on the experiments of phase 2 (10000 evaluations) and looking at the overall results through the Wilcoxon test, SA implementation got better results than GA.

References

[1] En.wikipedia.org. 2022. Travelling salesman problem - Wikipedia. [online] Available at: ⟨https://en.wikipedia.org/wiki/Travelling_salesman_problem⟩ [Accessed 3 March 2022].

[2] De Jong, K. and Fogel, L., 1997. Handbook of Evolutionary Computation, pp. C2.5:3 and C6.3:7.

[3] Fan Yang 2010. Solving Traveling Salesman Problem Using Parallel Genetic Algorithm and Simulated Annealing.

[4] Anthony R. and Edwin D. Reilly 1993. Encyclopedia of Computer Science, Chapman and Hall.

[5] En.wikipedia.org. 2021. 2-opt - Wikipedia. [online] Available at: ⟨https://en.wikipedia.org/wiki/2-opt⟩ [Accessed 3 March 2022].

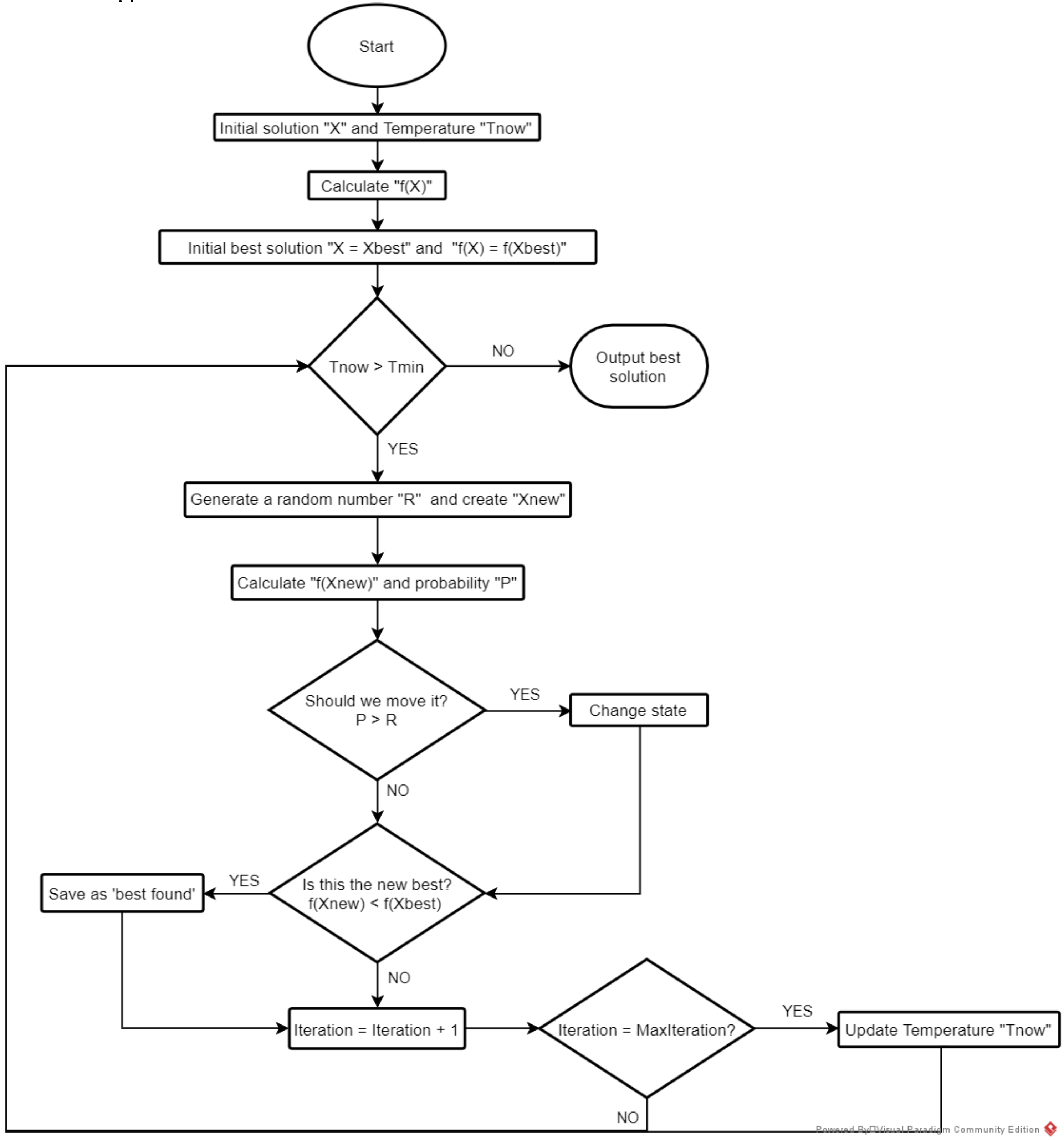
[6] Fischetti, M. and Stringher, M., 2022. Embedded hyper-parameter tuning by Simulated Annealing. pp.2 [online] Arxiv.org. Available at: ⟨https://arxiv.org/pdf/1906.01504.pdf⟩ [Accessed 4 March 2022].

[7] Shariat, Afshin and Babaei, Mohsen. (2012). AN EFFICIENT CROSSOVER OPERATOR FOR TRAVELING SALESMAN PROBLEM. Int. J. Optim. Civil Eng. 2. 607-619.

[8] Mosayebi, M. and Sodhi, M., 2020. Tuning genetic algorithm parameters using design of experiments. GECCO, pp.1937–1944.

[9] uk.mathworks.com. 2022. Wilcoxon signed-rank test - MathWorks. [online] Available at: ⟨https://uk.mathworks.com/help/stats/signrank.html⟩ [Accessed 11 March 2022].

Appendix A - SA Flowchart


Start
Initial solution "X" and Temperature "Tnow"
Calculate "f(X)"
Initial best solution "X = Xbest" and "f(X) = f(Xbest)"
Tnow > Tmin
NO
Output best solution
YES
Generate a random number "R" and create "Xnew"
Calculate "f(Xnew)" and probability "P"
Should we move it? P > R
YES
Change state
NO
Is this the new best? f(Xnew) < f(Xbest)
YES
Save as 'best found'
NO
Iteration = Iteration + 1
Iteration = MaxIteration?
YES
Update Temperature "Tnow"
NO
Powered By Visual Paradigm Community Edition

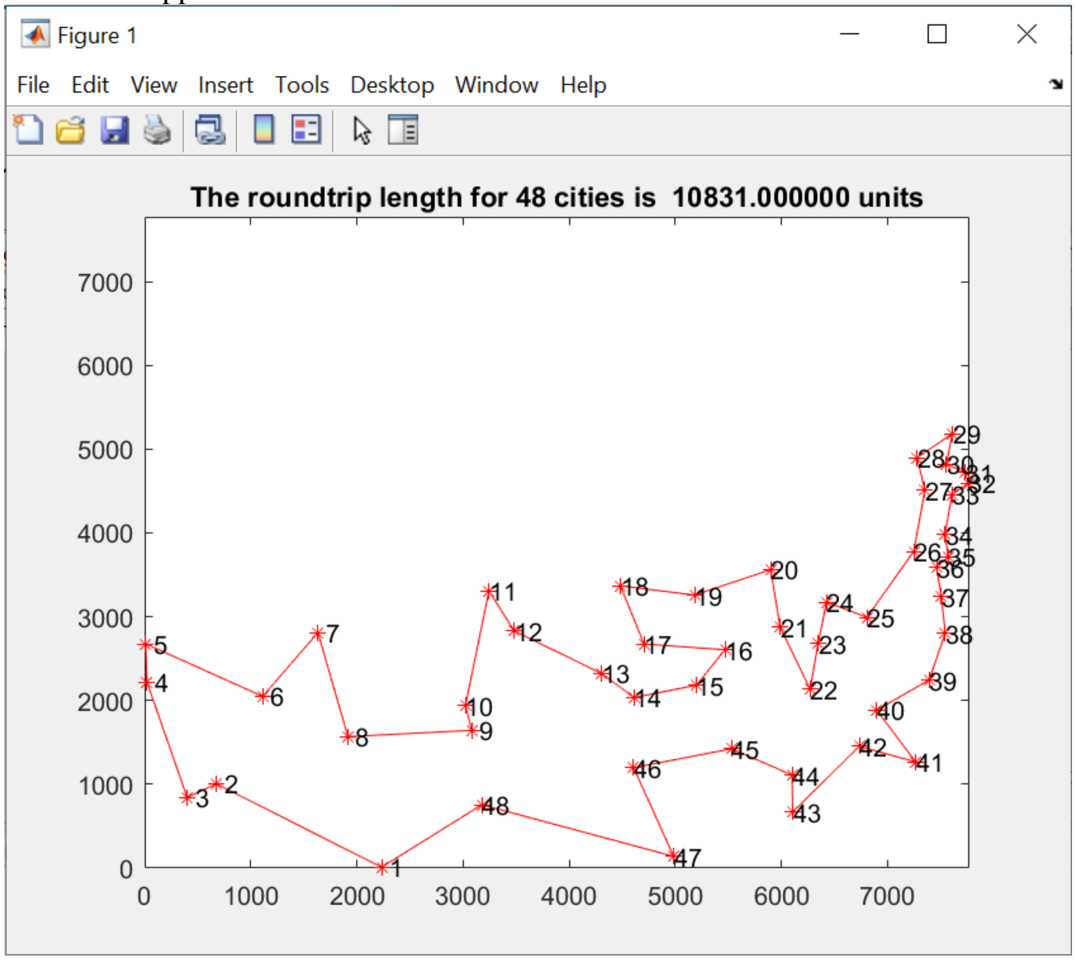
Appendix B - SA Pseudocode

Generic Simulated Annealing algorithm for minimisation

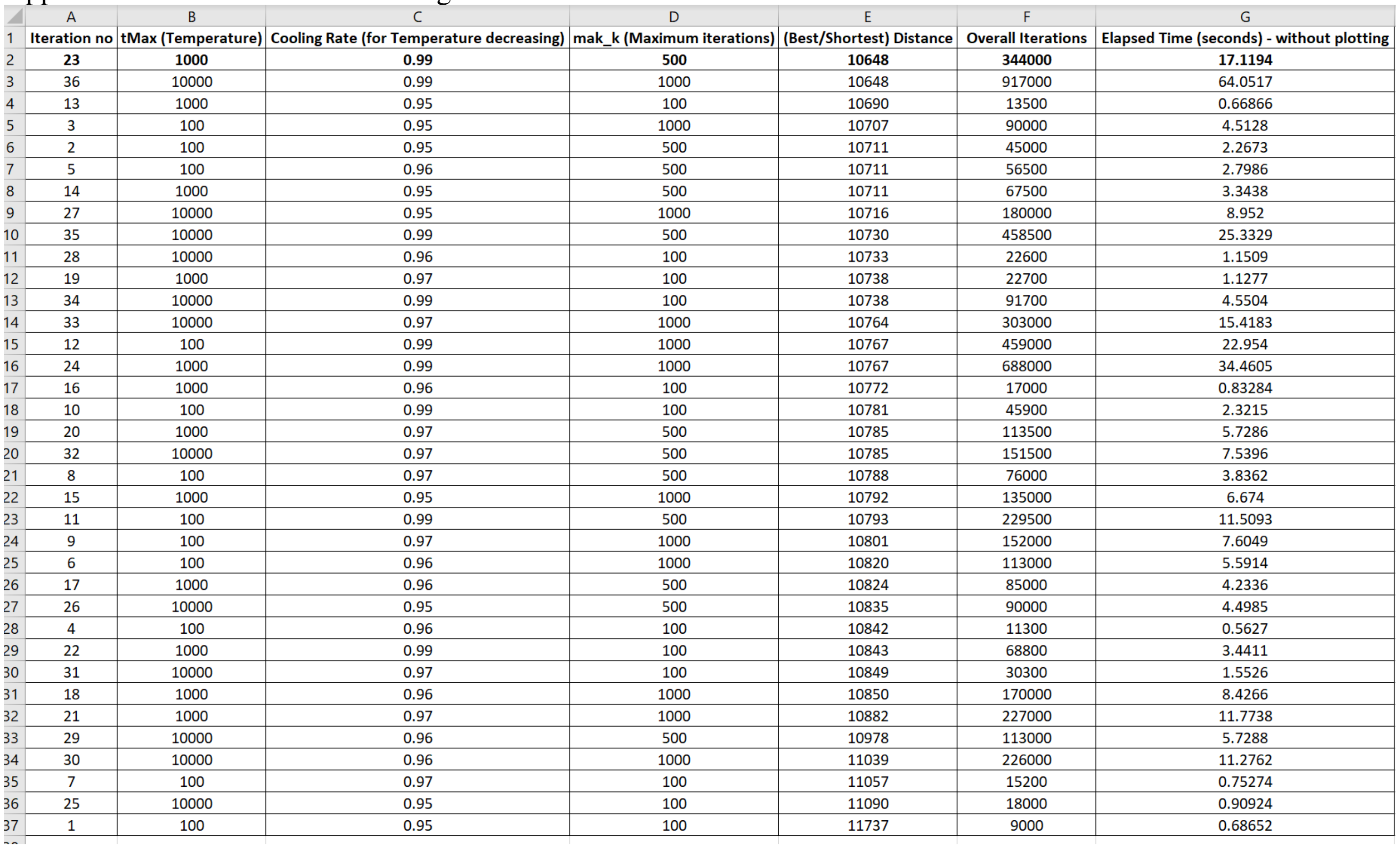
```
x:=x0, e:= f(x)          // Initial solution and objective function value (energy)
xbest:=x; ebest:= e;     // Initial "best" solution and "best" objective function value
Tnow = Tmax              // Set the maximum temperature value
while (Tnow > Tmin)      // Loop until to reach the temperature limit
      r = R(0, 1)                  // Generate a random number "r" (between 0 and 1)
      xnew := neighbour(x, r)      // Pick some neighbour randomly
      enew := f(xnew)              // Compute its objective function value
      if P(e, enew, Tnow) > r then // Should we move to it?
            x:= xnew; e:= enew     // Yes, change state
      if enew < ebest then
            xbest:= xnew; ebest:= enew    // Save as 'best found'
      Iteration := Iteration + 1          // Increase evaluation/iteration
      if Iteration = MaxIteration then    // Reached the maximum iterations?
            Tnow = Tnow * coolingRate     // Update/decrease the temperature
            Iteration = 0                 // Set the iterations back to 0
Output xbest, ebest                // Return best solution and best objective function value
```

- If "enew < e" (enew better than e), accepting better solutions with " P: = 1 "
- Otherwise, accepting worse solutions with a certain probability "P:= exp((e-enew)/Tnow"

Appendix C - SA Plot Results

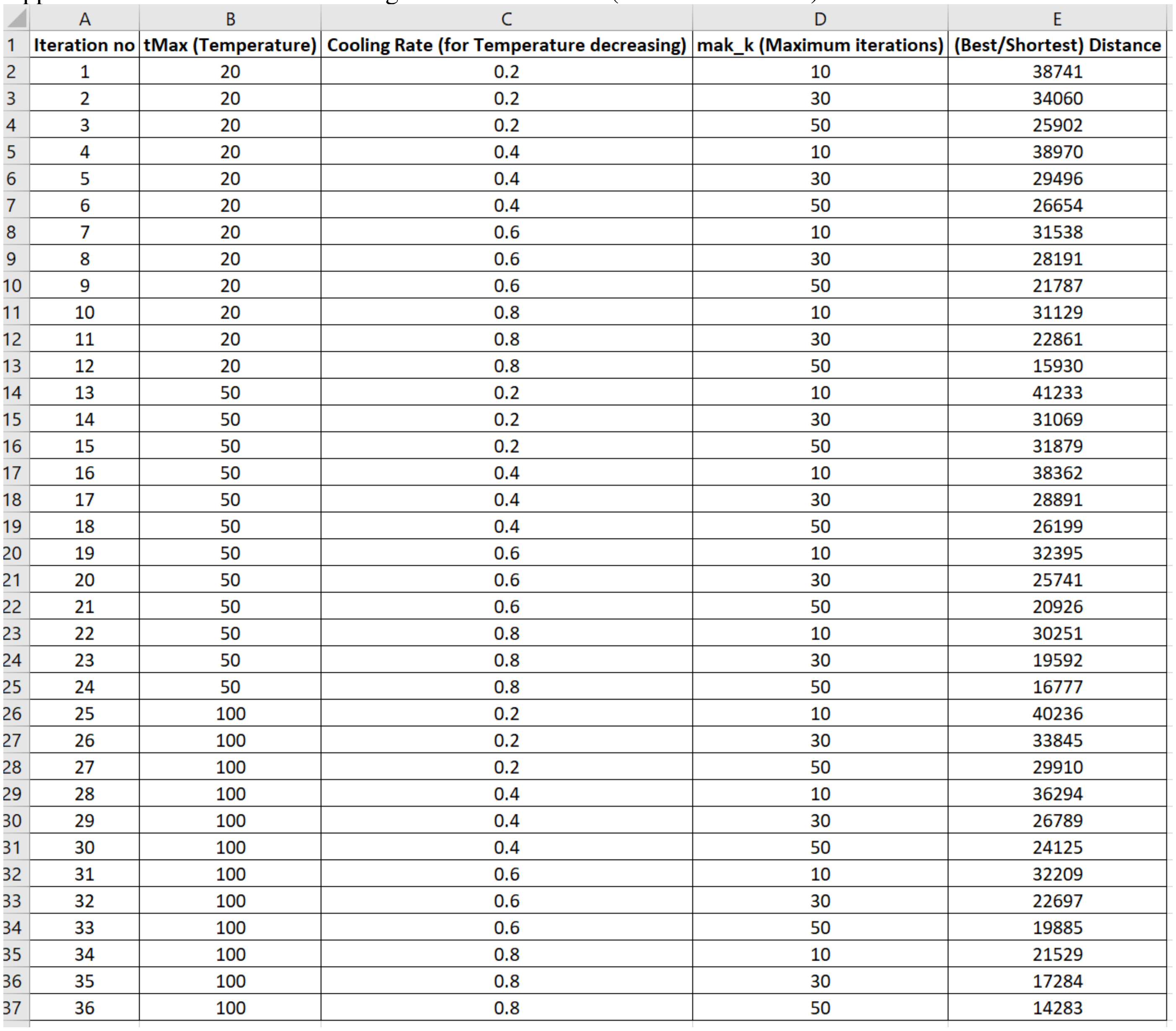


Appendix D - SA Parameter Tuning with 30 iterations



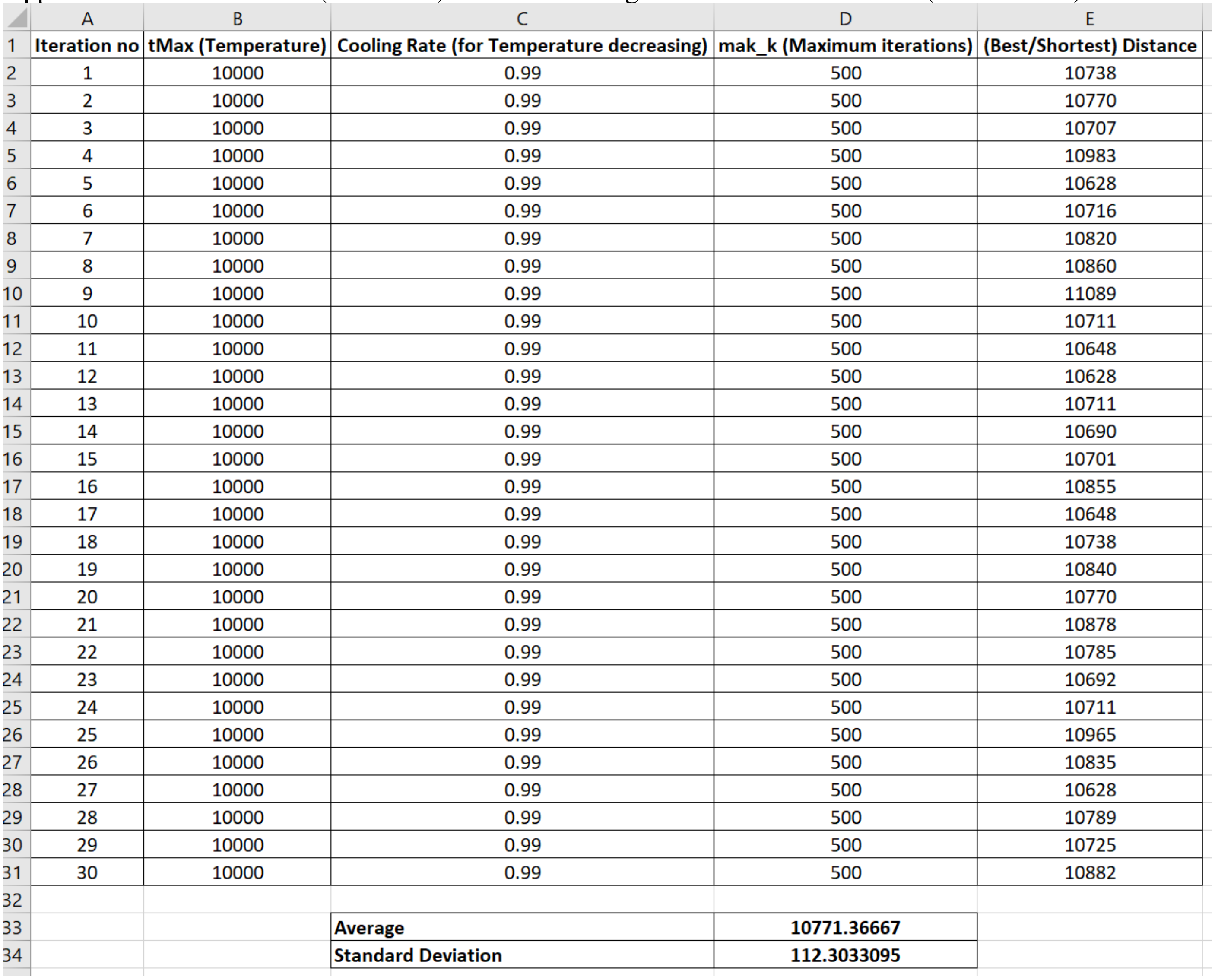
	A	B	C	D	E	F	G
1	**Iteration no**	**tMax (Temperature)**	**Cooling Rate (for Temperature decreasing)**	**mak_k (Maximum iterations)**	**(Best/Shortest) Distance**	**Overall Iterations**	**Elapsed Time (seconds) - without plotting**
2	**23**	**1000**	**0.99**	**500**	**10648**	**344000**	**17.1194**
3	36	10000	0.99	1000	10648	917000	64.0517
4	13	1000	0.95	100	10690	13500	0.66866
5	3	100	0.95	1000	10707	90000	4.5128
6	2	100	0.95	500	10711	45000	2.2673
7	5	100	0.96	500	10711	56500	2.7986
8	14	1000	0.95	500	10711	67500	3.3438
9	27	10000	0.95	1000	10716	180000	8.952
10	35	10000	0.99	500	10730	458500	25.3329
11	28	10000	0.96	100	10733	22600	1.1509
12	19	1000	0.97	100	10738	22700	1.1277
13	34	10000	0.99	100	10738	91700	4.5504
14	33	10000	0.97	1000	10764	303000	15.4183
15	12	100	0.99	1000	10767	459000	22.954
16	24	1000	0.99	1000	10767	688000	34.4605
17	16	1000	0.96	100	10772	17000	0.83284
18	10	100	0.99	100	10781	45900	2.3215
19	20	1000	0.97	500	10785	113500	5.7286
20	32	10000	0.97	500	10785	151500	7.5396
21	8	100	0.97	500	10788	76000	3.8362
22	15	1000	0.95	1000	10792	135000	6.674
23	11	100	0.99	500	10793	229500	11.5093
24	9	100	0.97	1000	10801	152000	7.6049
25	6	100	0.96	1000	10820	113000	5.5914
26	17	1000	0.96	500	10824	85000	4.2336
27	26	10000	0.95	500	10835	90000	4.4985
28	4	100	0.96	100	10842	11300	0.5627
29	22	1000	0.99	100	10843	68800	3.4411
30	31	10000	0.97	100	10849	30300	1.5526
31	18	1000	0.96	1000	10850	170000	8.4266
32	21	1000	0.97	1000	10882	227000	11.7738
33	29	10000	0.96	500	10978	113000	5.7288
34	30	10000	0.96	1000	11039	226000	11.2762
35	7	100	0.97	100	11057	15200	0.75274
36	25	10000	0.95	100	11090	18000	0.90924
37	1	100	0.95	100	11737	9000	0.68652

Appendix E - SA Parameter Tuning with 30 iterations (with small values)



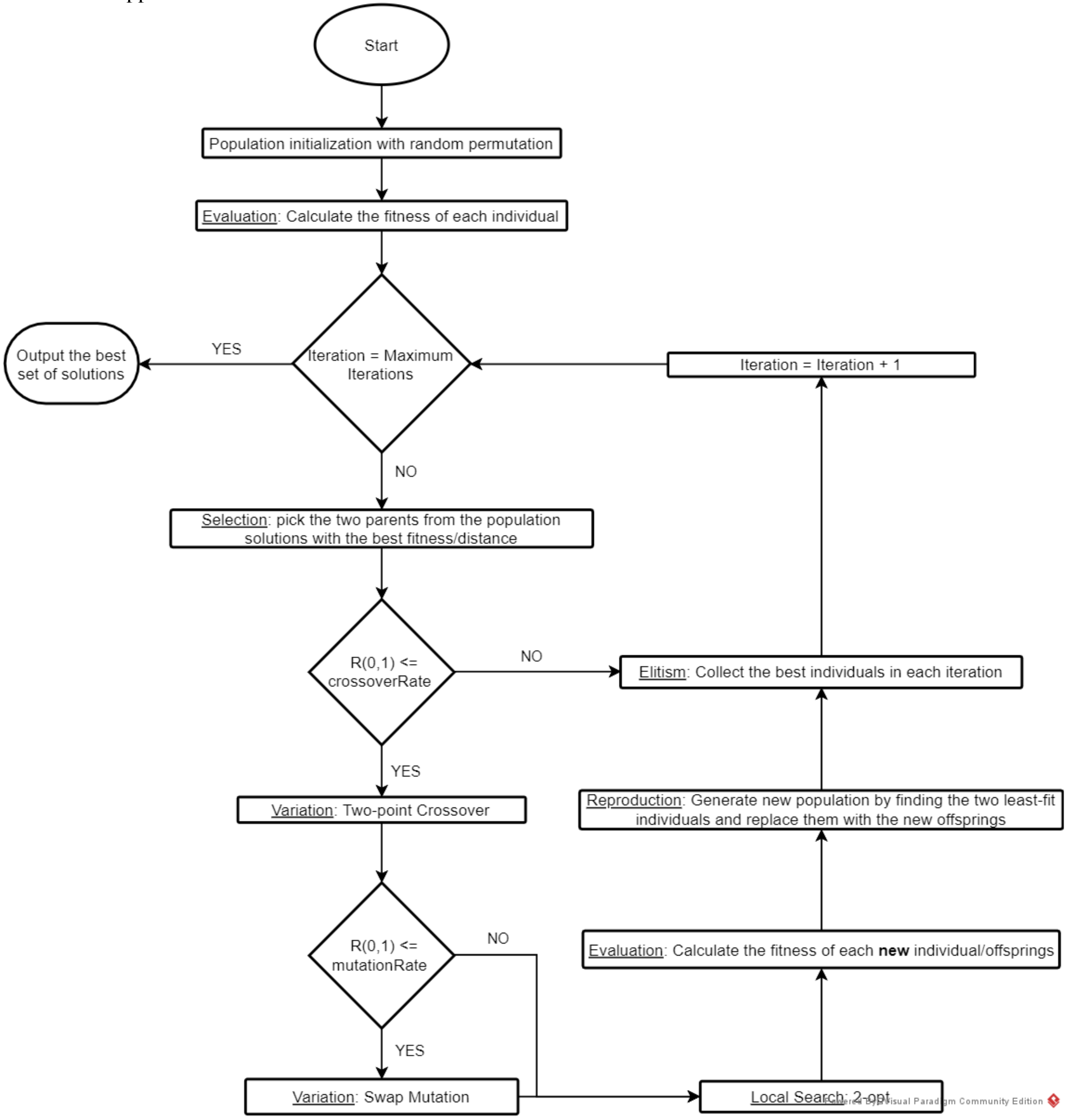
	A	B	C	D	E
1	Iteration no	tMax (Temperature)	Cooling Rate (for Temperature decreasing)	mak_k (Maximum iterations)	(Best/Shortest) Distance
2	1	20	0.2	10	38741
3	2	20	0.2	30	34060
4	3	20	0.2	50	25902
5	4	20	0.4	10	38970
6	5	20	0.4	30	29496
7	6	20	0.4	50	26654
8	7	20	0.6	10	31538
9	8	20	0.6	30	28191
10	9	20	0.6	50	21787
11	10	20	0.8	10	31129
12	11	20	0.8	30	22861
13	12	20	0.8	50	15930
14	13	50	0.2	10	41233
15	14	50	0.2	30	31069
16	15	50	0.2	50	31879
17	16	50	0.4	10	38362
18	17	50	0.4	30	28891
19	18	50	0.4	50	26199
20	19	50	0.6	10	32395
21	20	50	0.6	30	25741
22	21	50	0.6	50	20926
23	22	50	0.8	10	30251
24	23	50	0.8	30	19592
25	24	50	0.8	50	16777
26	25	100	0.2	10	40236
27	26	100	0.2	30	33845
28	27	100	0.2	50	29910
29	28	100	0.4	10	36294
30	29	100	0.4	30	26789
31	30	100	0.4	50	24125
32	31	100	0.6	10	32209
33	32	100	0.6	30	22697
34	33	100	0.6	50	19885
35	34	100	0.8	10	21529
36	35	100	0.8	30	17284
37	36	100	0.8	50	14283

Appendix F - SA calculate (Euclidean) distances average and standard deviation (30 iterations)



	A	B	C	D	E
1	**Iteration no**	**tMax (Temperature)**	**Cooling Rate (for Temperature decreasing)**	**mak_k (Maximum iterations)**	**(Best/Shortest) Distance**
2	1	10000	0.99	500	10738
3	2	10000	0.99	500	10770
4	3	10000	0.99	500	10707
5	4	10000	0.99	500	10983
6	5	10000	0.99	500	10628
7	6	10000	0.99	500	10716
8	7	10000	0.99	500	10820
9	8	10000	0.99	500	10860
10	9	10000	0.99	500	11089
11	10	10000	0.99	500	10711
12	11	10000	0.99	500	10648
13	12	10000	0.99	500	10628
14	13	10000	0.99	500	10711
15	14	10000	0.99	500	10690
16	15	10000	0.99	500	10701
17	16	10000	0.99	500	10855
18	17	10000	0.99	500	10648
19	18	10000	0.99	500	10738
20	19	10000	0.99	500	10840
21	20	10000	0.99	500	10770
22	21	10000	0.99	500	10878
23	22	10000	0.99	500	10785
24	23	10000	0.99	500	10692
25	24	10000	0.99	500	10711
26	25	10000	0.99	500	10965
27	26	10000	0.99	500	10835
28	27	10000	0.99	500	10628
29	28	10000	0.99	500	10789
30	29	10000	0.99	500	10725
31	30	10000	0.99	500	10882
32					
33			**Average**	**10771.36667**	
34			**Standard Deviation**	**112.3033095**	

Appendix G - GA Flowchart


Start
Population initialization with random permutation
Evaluation: Calculate the fitness of each individual
Iteration = Maximum Iterations
YES
Output the best set of solutions
NO
Iteration = Iteration + 1
Selection: pick the two parents from the population solutions with the best fitness/distance
R(0,1) <= crossoverRate
NO
Elitism: Collect the best individuals in each iteration
YES
Variation: Two-point Crossover
Reproduction: Generate new population by finding the two least-fit individuals and replace them with the new offsprings
R(0,1) <= mutationRate
NO
Evaluation: Calculate the fitness of each new individual/offsprings
YES
Variation: Swap Mutation
Local Search: 2-opt
Powered By Visual Paradigm Community Edition

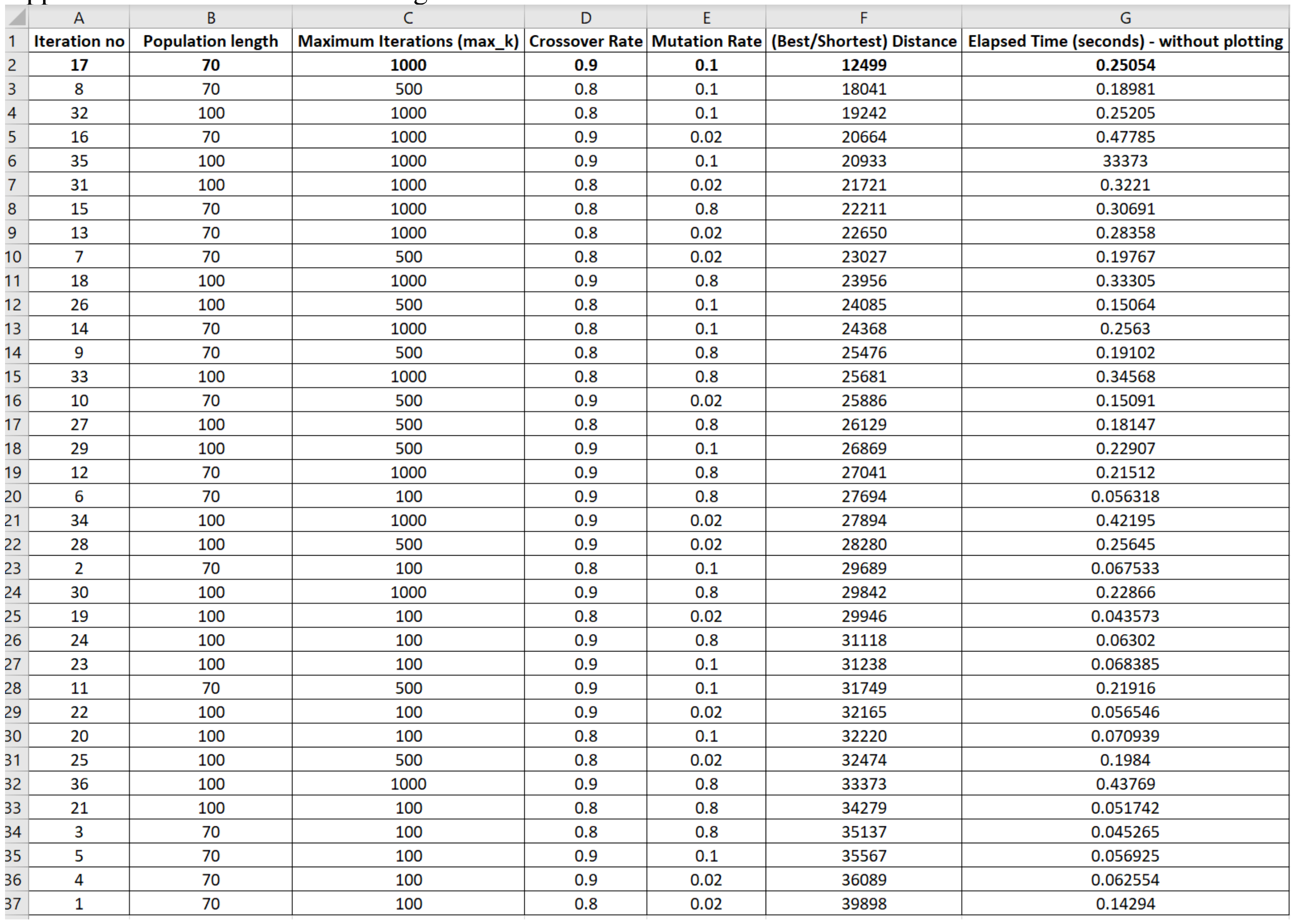
Appendix H - GA Pseudocode Part 1

```
Pseudocode - Genetic Algorithm for minimisation (TSP)

Input: list_cities, population_length, max_iterations, crossover_rate, mutation_rate

Output: best_distance

for i:=0 to population_length

      //Generate initial population of solutions
      population[] = randperm(list_cities_length)

      // Evaluate the fitness of each individual
      fitness[] = distance(individual_coordinates)
end

iteration = 1
while it has not reach the max_iterations

 Selection: Select the two parents from the population solutions based on their fitness
      1. Find the individual with the shortest distance (parent1)
         parent1_fitness = distance(parent1_coordinates)
      2. Remove parent1 from the population
         Population[parent1] = []
      3. Find the individual with (next) shortest distance (parent2)
         parent2_fitness = distance(parent2_coordinates)
      4. Restore parent1
         population[] <-- parent1

 Variation:
      if crossover_rate >= R(0, 1)
            Two-point Crossover: Given two parents we breed two offspring/children
            cross_point = list_cities_length/2
            // The 1-Cross-Point bit in offspring1 and offspring2
               is the 1-Cross-Point bit in parent1 and parent2, respectively
            // If the remaining bits in parent2/parent1 do not exist in
               offspring1/offspring2 the remaining bits in offspring1
               will be filled with the values of the parent2/parent1 remaining bits
               offspring1[1 to cross_point] <-- parent1[1 to cross_point]
               offspring2[1 to cross_point] <-- parent2[1 to cross_point]

            if mutation_rate >= R(0, 1)
                  Swap Mutation: Select two positions at random, and swap the values
                  // Select two indexes at random
                     chrom1 := rand(list_cities)
                     chrom2 := rand(list_cities)

                  // Swap the selected two chromosomes on each offspring
                     offspring1[chrom1 chrom2] = offspring1[chrom2 chrom1]
                     offspring2[chrom1 chrom2] = offspring2[chrom2 chrom1]
            end
```

Appendix I - GA Pseudocode Part 2

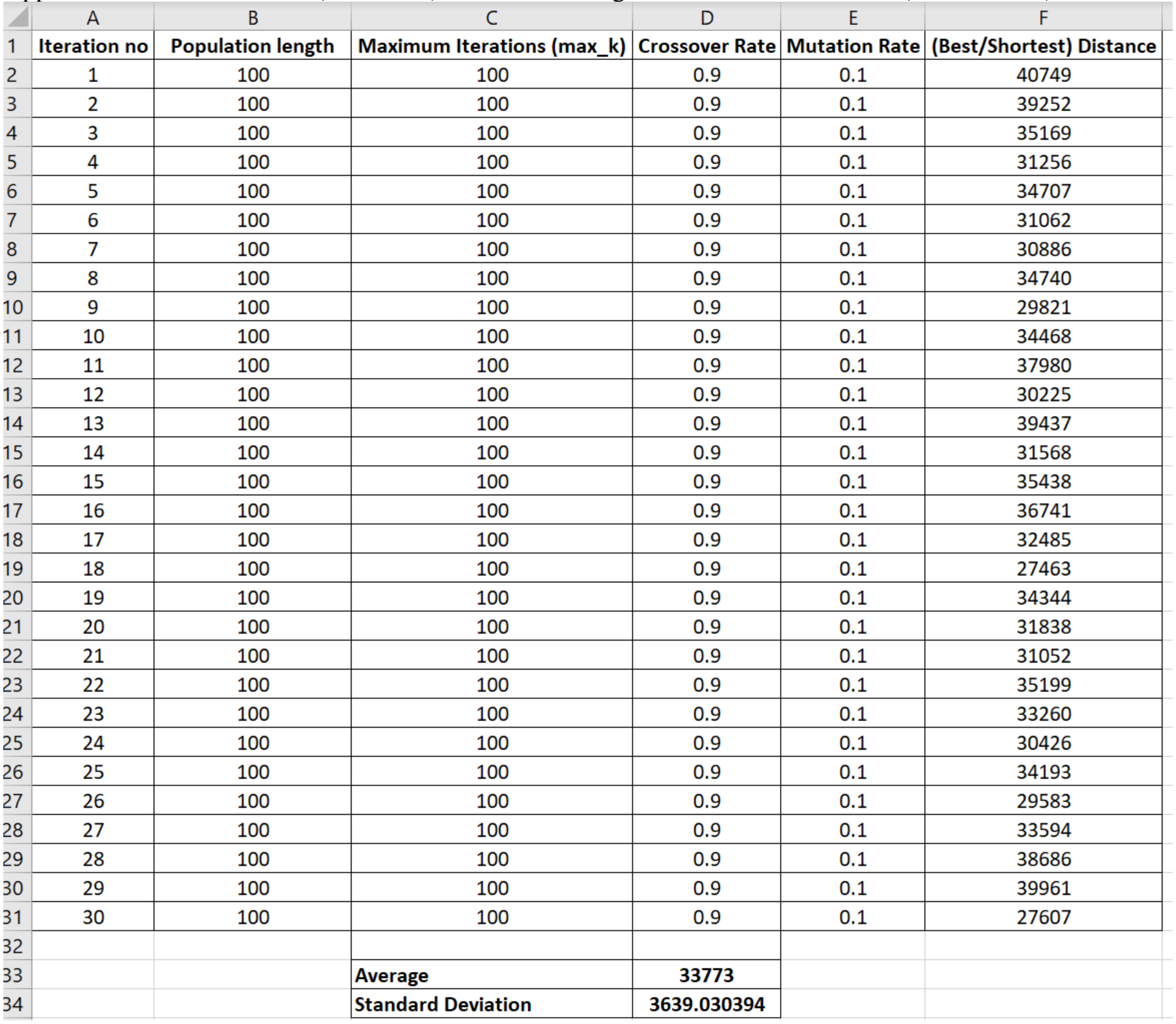
```
          //Local Searcher: Apply 2-Opt Local search for each offspring
            offspring1 = two_opt(offspring1)
            offspring2 = two_opt(offspring2)

          //Evaluate the fitness of new individuals
            offspring1_fitness = distance(offspring1_coordinates)

          //Reproduction: Generate new population by replacing least-fit individuals
            1st_worse_individual = max(fitness) // Find the index of the slowest
            offspring1 = 1st_worse_individual   // Replace with offspring
            2nd_worse_individual = max(fitness)
            offspring3 = 2nd_worse_individual
    end
    // Elitism - a collection of the best individuals in each iteration
       best_individual_index = min(fitness) // find the index of fastest individual
       elitism[] <-- population[best_individual_index] // store it to elitism array
       total_distances[] <-- fitness[best_individual_index] // store its distance

    iteration = iteration + 1

end
Output best_distance := min(total_distances)
```

Appendix J - GA Parameter tuning results for 30 iterations



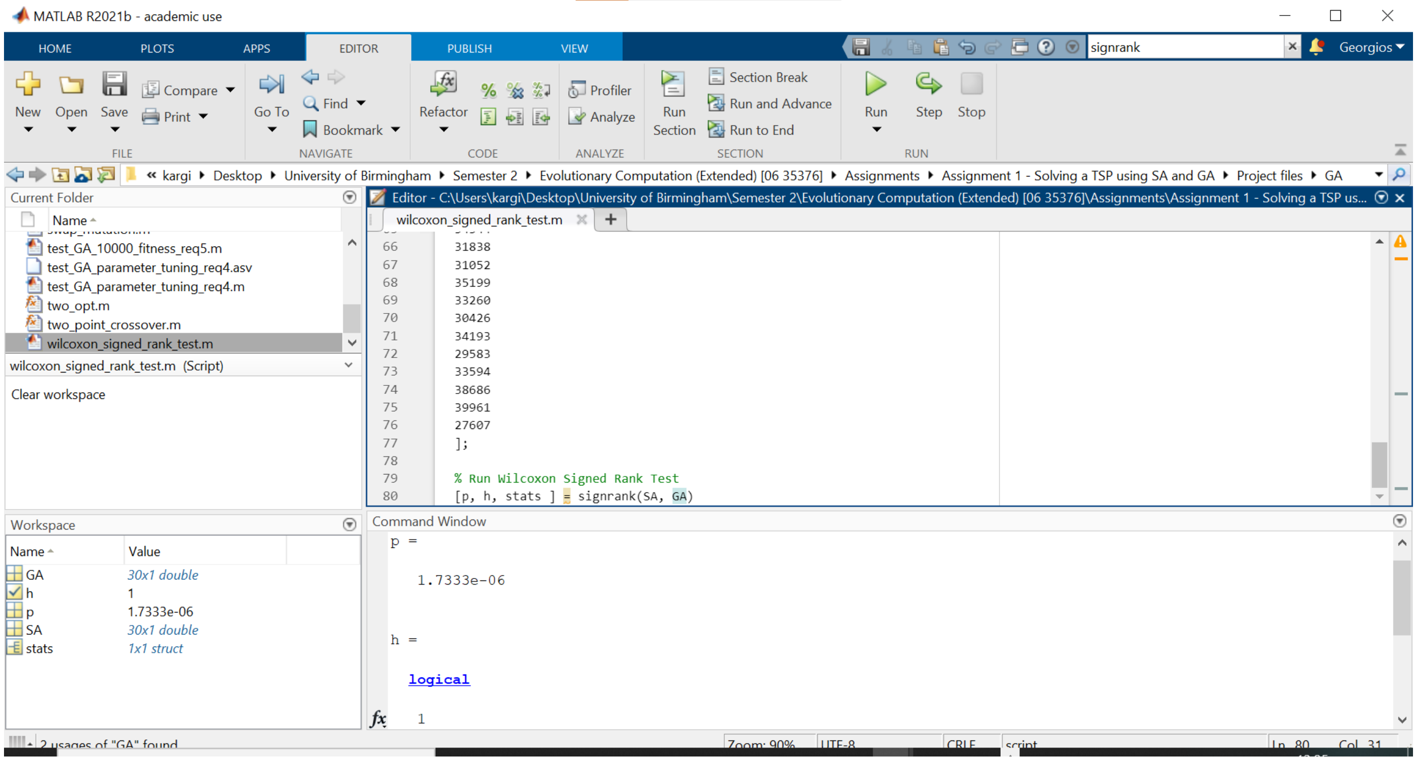
	A	B	C	D	E	F	G
1	**Iteration no**	**Population length**	**Maximum Iterations (max_k)**	**Crossover Rate**	**Mutation Rate**	**(Best/Shortest) Distance**	**Elapsed Time (seconds) - without plotting**
2	**17**	**70**	**1000**	**0.9**	**0.1**	**12499**	**0.25054**
3	8	70	500	0.8	0.1	18041	0.18981
4	32	100	1000	0.8	0.1	19242	0.25205
5	16	70	1000	0.9	0.02	20664	0.47785
6	35	100	1000	0.9	0.1	20933	33373
7	31	100	1000	0.8	0.02	21721	0.3221
8	15	70	1000	0.8	0.8	22211	0.30691
9	13	70	1000	0.8	0.02	22650	0.28358
10	7	70	500	0.8	0.02	23027	0.19767
11	18	100	1000	0.9	0.8	23956	0.33305
12	26	100	500	0.8	0.1	24085	0.15064
13	14	70	1000	0.8	0.1	24368	0.2563
14	9	70	500	0.8	0.8	25476	0.19102
15	33	100	1000	0.8	0.8	25681	0.34568
16	10	70	500	0.9	0.02	25886	0.15091
17	27	100	500	0.8	0.8	26129	0.18147
18	29	100	500	0.9	0.1	26869	0.22907
19	12	70	1000	0.9	0.8	27041	0.21512
20	6	70	100	0.9	0.8	27694	0.056318
21	34	100	1000	0.9	0.02	27894	0.42195
22	28	100	500	0.9	0.02	28280	0.25645
23	2	70	100	0.8	0.1	29689	0.067533
24	30	100	1000	0.9	0.8	29842	0.22866
25	19	100	100	0.8	0.02	29946	0.043573
26	24	100	100	0.9	0.8	31118	0.06302
27	23	100	100	0.9	0.1	31238	0.068385
28	11	70	500	0.9	0.1	31749	0.21916
29	22	100	100	0.9	0.02	32165	0.056546
30	20	100	100	0.8	0.1	32220	0.070939
31	25	100	500	0.8	0.02	32474	0.1984
32	36	100	1000	0.9	0.8	33373	0.43769
33	21	100	100	0.8	0.8	34279	0.051742
34	3	70	100	0.8	0.8	35137	0.045265
35	5	70	100	0.9	0.1	35567	0.056925
36	4	70	100	0.9	0.02	36089	0.062554
37	1	70	100	0.8	0.02	39898	0.14294

Appendix K - GA calculate (Euclidean) distances average and standard deviation (30 iterations)



	A	B	C	D	E	F
1	**Iteration no**	**Population length**	**Maximum Iterations (max_k)**	**Crossover Rate**	**Mutation Rate**	**(Best/Shortest) Distance**
2	1	100	100	0.9	0.1	40749
3	2	100	100	0.9	0.1	39252
4	3	100	100	0.9	0.1	35169
5	4	100	100	0.9	0.1	31256
6	5	100	100	0.9	0.1	34707
7	6	100	100	0.9	0.1	31062
8	7	100	100	0.9	0.1	30886
9	8	100	100	0.9	0.1	34740
10	9	100	100	0.9	0.1	29821
11	10	100	100	0.9	0.1	34468
12	11	100	100	0.9	0.1	37980
13	12	100	100	0.9	0.1	30225
14	13	100	100	0.9	0.1	39437
15	14	100	100	0.9	0.1	31568
16	15	100	100	0.9	0.1	35438
17	16	100	100	0.9	0.1	36741
18	17	100	100	0.9	0.1	32485
19	18	100	100	0.9	0.1	27463
20	19	100	100	0.9	0.1	34344
21	20	100	100	0.9	0.1	31838
22	21	100	100	0.9	0.1	31052
23	22	100	100	0.9	0.1	35199
24	23	100	100	0.9	0.1	33260
25	24	100	100	0.9	0.1	30426
26	25	100	100	0.9	0.1	34193
27	26	100	100	0.9	0.1	29583
28	27	100	100	0.9	0.1	33594
29	28	100	100	0.9	0.1	38686
30	29	100	100	0.9	0.1	39961
31	30	100	100	0.9	0.1	27607
32						
33			**Average**	**33773**		
34			**Standard Deviation**	**3639.030394**		

Appendix L - Wilcoxon signed rank test results part 1

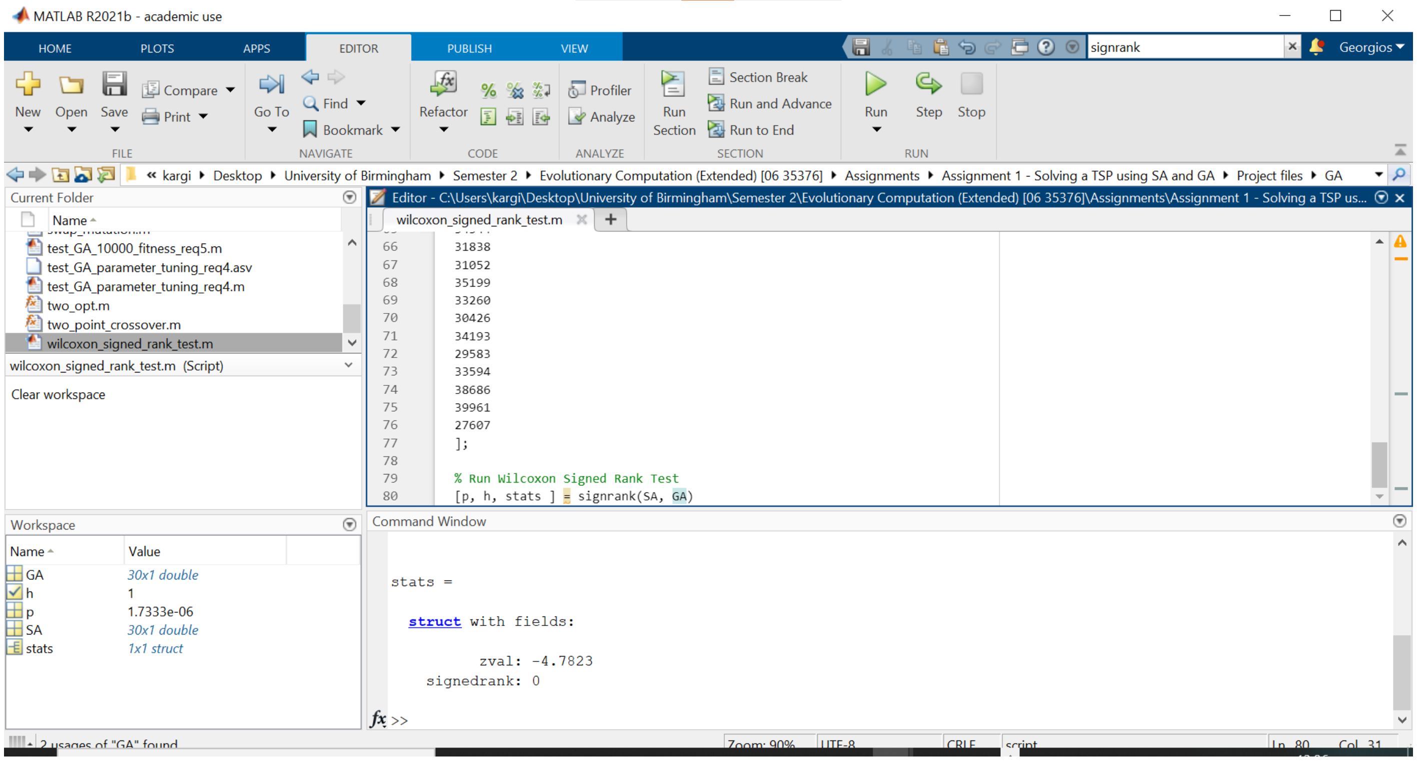

MATLAB R2021b - academic use
HOME
PLOTS
APPS
EDITOR
PUBLISH
VIEW
signrank
Georgios
New
Open
Save
Compare
Print
FILE
Go To
Find
Bookmark
NAVIGATE
Refactor
CODE
Profiler
Analyze
ANALYZE
Run Section
Section Break
Run and Advance
Run to End
SECTION
Run
Step
Stop
RUN
kargi ▸ Desktop ▸ University of Birmingham ▸ Semester 2 ▸ Evolutionary Computation (Extended) [06 35376] ▸ Assignments ▸ Assignment 1 - Solving a TSP using SA and GA ▸ Project files ▸ GA
Current Folder
Name
test_GA_10000_fitness_req5.m
test_GA_parameter_tuning_req4.asv
test_GA_parameter_tuning_req4.m
two_opt.m
two_point_crossover.m
wilcoxon_signed_rank_test.m
wilcoxon_signed_rank_test.m (Script)
Clear workspace
Editor - C:\Users\kargi\Desktop\University of Birmingham\Semester 2\Evolutionary Computation (Extended) [06 35376]\Assignments\Assignment 1 - Solving a TSP us...
wilcoxon_signed_rank_test.m
Workspace
Command Window
2 usages of "GA" found
Zoom: 90%
UTF-8
CRLF
script
Ln 80
Col 31

```
    31838
    31052
    35199
    33260
    30426
    34193
    29583
    33594
    38686
    39961
    27607
    ];

    % Run Wilcoxon Signed Rank Test
    [p, h, stats ] = signrank(SA, GA)
```

Name	Value
GA	30x1 double
h	1
p	1.7333e-06
SA	30x1 double
stats	1x1 struct

```
p =

   1.7333e-06


h =

  logical

   1
```

Appendix M - Wilcoxon signed rank test results part 2


MATLAB R2021b - academic use

Editor - C:\Users\kargi\Desktop\University of Birmingham\Semester 2\Evolutionary Computation (Extended) [06 35376]\Assignments\Assignment 1 - Solving a TSP us...

wilcoxon_signed_rank_test.m

Current Folder:
- test_GA_10000_fitness_req5.m
- test_GA_parameter_tuning_req4.asv
- test_GA_parameter_tuning_req4.m
- two_opt.m
- two_point_crossover.m
- wilcoxon_signed_rank_test.m

wilcoxon_signed_rank_test.m (Script)

Clear workspace

```
    31838
    31052
    35199
    33260
    30426
    34193
    29583
    33594
    38686
    39961
    27607
    ];

    % Run Wilcoxon Signed Rank Test
    [p, h, stats ] = signrank(SA, GA)
```

Workspace

Name	Value
GA	30x1 double
h	1
p	1.7333e-06
SA	30x1 double
stats	1x1 struct

Command Window

```
stats =

  struct with fields:

        zval: -4.7823
  signedrank: 0

fx >>
```

2 usages of "GA" found