L-Università ta' Malta

ICS2207 - Assignment
Machine Learning 1

Daniel Magro

Bachelor of Science in
Information Technology (Honours)
(Artificial Intelligence)

May 2018

Table of Contents

Statement of Completion

Item	Completed (Yes/No/Partial)
Implemented GA	Yes
Implemented ACO	Yes
Evaluated GA vs ACO for selected Instances	Yes
Described GA architecture in report	Yes
Described ACO architecture in report	Yes

The first step before either the GA or the ACO start running is that the TSP instance, obtained from TSPLIB[14], is converted into a list of cities, i.e. a list of objects, each storing the city's ID, x coordinate and y coordinate. This operation is accomplished by the *InputReader*, *City* and *Location* classes. Once this is done, the GA and the ACO can be called, given this list of cities as a parameter.

Description of GA Implementation

The first step before the GA starts executing is the initialization of the distance matrix. This step basically creates a square 2D array, of *number of cities x number of cities* size, where each cell in the matrix represents the distance between the 2 cities of its index. So for example, the distance between city 5 and city 17 is stored in distanceMatrix[5-1][17-1]. The distance of each cell is calculated by the initialiseDistanceMatrix method. Since this is a symmetric TSP, only one of distanceMatrix[i][j] and distanceMatrix[j][i] is computed, since they will obviously store the same value. The distance metric used is Euclidean Distance. The distance matrix was implemented as an optimization feature, so that the distance between 2 cities could be simply retrieved from an array in constant time, rather than get calculated each time it is required.

The representation chosen to encode a path was the Path Representation. The Path Representation is said to be the most natural way of representing a path, as it simply lists the order of the cities visited, for example: [1, 4, 2, 3] means that first city 1 was visited, then 4, then 2, then 3 and finally back to 1. Apart from being a very natural representation, the Path Representation allows for the most powerful Crossover Operators to be applied to chromosomes with relative ease and efficiency. Some of these crossover operators are: PMX (Partially Mapped Crossover), POS (Position Based Crossover) and SCX (Sequential Constructive Crossover). SCX is the crossover operator which was applied in this implementation.
Another alternative is storing the edges between two cities, instead of the cities themselves. For this implementation such a representation did not make much sense, given that the cities are all connected to each other.
One other alternative representation is the Ordinal Representation, which stores how a 'Canonic tour' is to be followed. This representation is useful since One-Point Crossover always yields a feasible chromosome, however given one-point crossover's relatively poor performance, this representation is not often used. [1]
One interesting representation is the Temporal Representation. This representation was not used as I. Mitchell and P. Pocknell suggest that there is no definitive performance gain over the Path Representation, and that this representation is virtually impossible to visualise without the aid of another program. [2]

A Nearest Neighbour Algorithm was used for part of the Initial Population Generation. By using the NN to generate some of the Chromosomes in the Initial Population, the GA converges much quicker. The number of chromosomes generated using the NN Algorithm was set to 10% of the total population, and the rest of the population was generated randomly. 10% was chosen as it was shown that using a higher number does not make the GA converge faster, and if anything, diversity is lost from randomly generated chromosomes, and the GA would get stuck on local maxima. [3]
An alternative initial population generation strategy, which could be used instead of the Nearest Neighbours Algorithm, is the k-means algorithm, called KIP in this context. When applying KIP,

certain cities are grouped using the k-means algorithm. Then, TSP is performed on each of the 'sub-graphs', and between all the centroids of the subgraph to re-connect the graph. [4]

Rank-Based Roulette Wheel Selection was used to select chromosomes for Crossover. This method was chosen as N. Mohd Razali and J. Geraghty state that as the number of cities increases, i.e. for any instance with more than 20 cities, Rank-Based selection does not converge prematurely, and shows a significantly smaller percentage of deviation from the optimal solution, when compared to Tournament Selection. [5]

The Crossover Operator chosen was the Sequential Constructive Crossover (SCX). In his paper, Z. H. Ahmed benchmarks ERX and GNX, two of the most effective operators, and SCX, and shows that SCX outperforms both crossover operators in both quality of solutions and solution times. [6]
I. H. Khan also reports that SCX beats all other crossover operators in terms of the quality of the solutions produced, beaten in computational costs only by TPX and SX. [7]

After all the children of a generation are generated from crossover, mutation is carried out on the children, and added to the population. The chromosomes chosen for mutation are based on a Mutation Rate. Every chromosome has a 5% chance of being mutated. 5% is considered a very high mutation rate for GAs in general, however it is explained in the last paragraph why in this implementation such a high number does not have any negative effects, and only reintroduces diversity into the population.
The chosen Mutation Operator was the Reverse Sequence Mutation (RSM), which selects two random cities in the chromosome's path representation and switches every pair of cities in between those two cities. According to O. Abdoun et al., RSM is the most efficient mutation operator, and is the operator which yields the best results when dealing with the TSP. [8]

After all the children of a generation are generated, and mutation is carried out on those chromosomes, the evolution of the current generation to the next one occurs. In this implementation, all the generated chromosomes are added to the population, such that the population will store the current generation, the children of the current generation, the mutated chromosomes and a new initial population. Then, the fitness of the entire population is calculated, and the population is sorted such that the fittest chromosomes are at the start of the ArrayList. Then, chromosomes are added one by one from the beginning of the list until the Set storing the next generation is of the required size, i.e. the initial population size. Chromosomes are stored in a set so that duplicates are automatically ignored.
Given the way that mutated chromosomes are added to the population, instead of overwriting their original chromosome, it is safe to have a relatively high mutation rate as this will not negatively affect the population. In fact, the fitness of the population is monotonically increasing with each passing generation, since if in the worst case, a generation makes absolutely no improvement, the same chromosomes from the population from the last generation can be chosen again for the next generation as they are not overwritten.

The parameters for the GA, such as population size, crossover rate, mutation rate and number of generations were all either set because they were the values which proved to provide the best results for a number of instances, or because they were the values suggested by the paper presenting the crossover operator used for this implementation. A. Rexhepi, A. Maxhuni and A. Dika, in their paper, suggest that increasing the mutation rate any higher than 5% provides no added benefit, however mention that further work needs to be done to determine an optimal value for the initial population size. [9]
An alternative method to find the optimal values for TSP GAs, is creating and running a ML algorithm to find the best values for the parameters of the GA.
It is important to understand that it is very hard to find universally optimal parameters, such that they will offer the optimal performance for every instance, as most often one set of parameters will work best for a small range of instance sizes. This is explained further in the Evaluation section.

Description of ACO Implementation

Similarly to what was done for the GA, the distanceMatrix was again initialised (the distance matrix is initialised by each algorithm because it makes each algorithm 'standalone' and so that initialisation is included in the running time of both algorithms.

An *Ant* object was created, which stores the path taken by the ant, the length of that path, on which city the ant is on, as well as which cities it has already visited.

A 2D matrix for pheromone levels is also initialised, which is of the same size as the distance matrix, however instead stores the pheromone level between every 2 cities.

The parameters of the ACO are then initialised, in my implementation:

- The number of ants has been set to 10, as suggested in the original Dorigo paper.
- *q_0* has been set to 0.9, meaning that 90% of the time, the next city an ant will visit is decided based on the path with the highest pheromone level and shortest distance, and the other 10% of the time the next city is chosen probabilistically, where each path is given a probability based on its pheromone level and distance.
- *alpha* was set to 0.1, which means the evaporation rate of the pheromones is of 10% (i.e. 90% of the pheromone remains)
- *tau_0* is the amount of pheromone deposited on every edge when performing local updates. This was set to $\frac{1}{number\ of\ cities * l_{nn}}$, where l_{nn} is the inverse of the length of a path calculated by the nearest neighbour algorithm.
- *beta* was set to 2, meaning how much more important is given to the distance between the 2 cities over the pheromone level on the path between them.

Then, each ant is assigned a random starting city. After that, each ant moves one city, and when all ants have moved one city, all the ants move to another city and so on. Ants obviously cannot go to a city that they have already visited.

The city that the ant will go to next is decided by the nextCity method, which first scores all the ant's unvisited cities. Then, depending on whether the randomly initialised q falls within the range of q_0, the next city is chosen either according to the best score, or probabilistically.

Once the ant's next city has been decided, a local pheromone update needs to be performed on the path between the ant's current city and the next chosen city. The pheromone level between on the path is set to: $(1 - alpha) * currentPheromoneLevel + alpha * tau_0$, i.e. evaporation is performed on the current pheromones, such that alpha*100% of the pheromone level is removed, and pheromone is deposited by the ant equal to $alpha\ (the\ percentage\ removed) *\ tau_0$. When All the cities have been visited, each ant updates the pheromone levels between their current (last) city back to their start city.

Finally, global pheromone update is performed. In this step, the best ant, i.e. the one with the shortest path, is chosen to deposit significantly more pheromone, as a reinforcement mechanism, and the shorter the path is, the more pheromone gets deposited. The algorithm traverses each ant's path, applying evaporation, however the best ant is made to deposit even more pheromone, equal to $alpha * \Delta\tau(r,s)$, i.e. $alpha * \frac{1}{total\ length\ of\ ant's\ tour}$.

The implementation for ACO was inspired by the papers referenced [10] to [13], but mostly M. Dorigo and L. M. Gambardella’s paper [12].

Evaluated GA vs ACO for selected Instances

A folder named **TSPinstances** was created, in which TSP instances from TSPLIB[14] can be saved. When the program is run, it will run both GA and ACO on every instance in that folder.
A folder **TSPinstanceslibrary** was also created, in which several instances are stored, but not run by the program.

The program works such that GA and then ACO is run on every instance in the 'TSPinstances' folder, each time outputting the technique used (GA or ACO), the name of the instance, the best path found, the length of that path and the total time taken by the GA or ACO to run.

7 TSP instances were chosen to be evaluated by both algorithms. Each instance was run 3 times, so that an average could be taken to eliminate any possible outliers. The results of all the runs are included in the following table:

Instance Name and Algorithm Used:	Length of Best Path	Runtime (ms)	Length of Best Path	Runtime (ms)	Length of Best Path	Runtime (ms)
berlin52 GA	7598.4	4398	7713	4832	7713	5252
berlin52 ACO	7544.4	553	7544.4	602	7544.4	550
ch130 GA	6622.8	12384	6477.5	13366	6524.5	14374
ch130 ACO	6478.5	2307	6370	2626	6383	2715
kroB200 GA	32874	24932	33099	26259	33223	27524
kroB200 ACO	30862	5774	31510	6306	31859	7947
linhp318 GA	47926	47962	48153	52743	47937	58214
linhp318 ACO	45013	14947	44937	16628	45091	17223
pcb442 GA	57315	89851	57529	102498	57144	103768
pcb442 ACO	74917	31873	70661	35501	72998	33403
pr1002 GA	309289	422346	309557	462518	310249	377359
pr1002 ACO	432328	163527	414807	175593	410937	151278
rl1323 GA*	310531	712970				
rl1323 ACO*	444366	398380				

* - Algorithm was only run once on this instance given how long it takes to run

The results from the previous table were averaged and shown in the *Average Result* and *Average Runtime (ms)* columns. The average result was compared with the *Optimal Result* for that instance, so that the *Accuracy* of the algorithm could be shown. *Accuracy gain per Runtime* aims to measure how close the algorithm gets to the optimal solution per unit of time that it runs, in other terms it measures the 'efficiency' of the algorithm, it gives a score to the algorithm for how close it gets to the optimal for every unit of time that it runs for. This metric is only meant to serve as a comparison with the score of the same instance run in different algorithms. The formulae used are shown below:

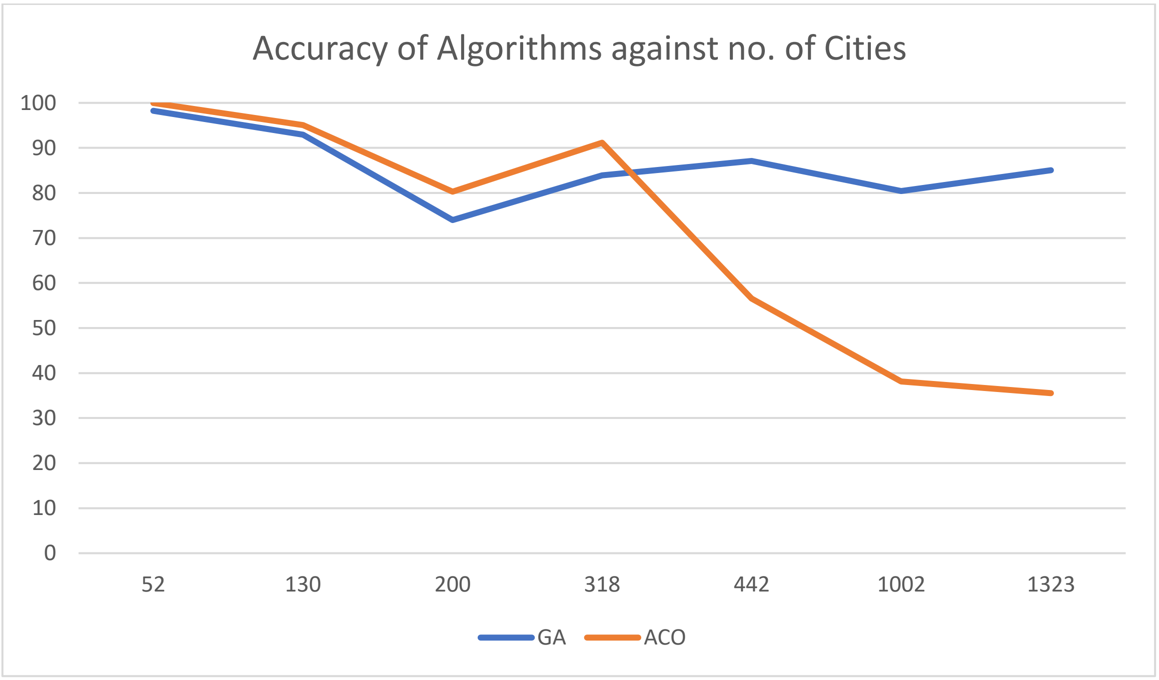
$$Accuracy\ (\%) = 100 - \left(\frac{Average\ Result - Optimal}{Optimal}\right) * 100$$

An Accuracy of **100** means that the algorithm's output was the optimal solution, whereas an Accuracy of **0** means that the algorithm's output was twice the optimal output.

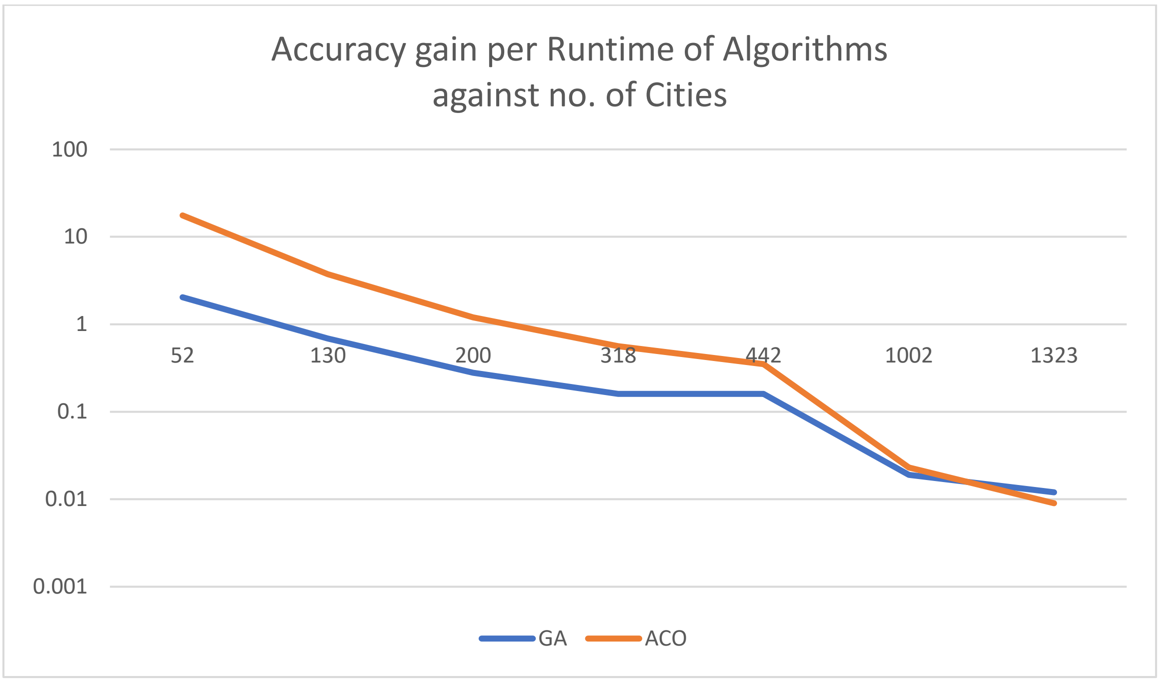
$$Accuracy\ gain\ per\ Runtime = \left(\left(\frac{Accuracy + Average\ Runtime}{Average\ Runtime}\right) * 100\right) - 100$$
$$= \left(\frac{Accuracy}{Average\ Runtime}\right) * 100$$

Instance Name:	Average Result	Average Runtime (ms)	Optimal Result[15]	Accuracy (%)	Accuracy gain per Runtime
berlin52 GA	7674.8	4827.3	7542	98.24	2.04
berlin52 ACO	7544.4	568.3	7542	99.97	17.59
ch130 GA	6541.6	13374.6	6110	92.94	0.69
ch130 ACO	6410.5	2549.3	6110	95.08	3.73
kroB200 GA	33065	26238	29437	73.98	0.28
kroB200 ACO	31410	6676	29437	80.29	1.2
linhp318 GA	48005	52973	41345	83.89	0.16
linhp318 ACO	45014	16266	41345	91.13	0.56
pcb442 GA	57329	52973	50778	87.1	0.16
pcb442 ACO	72859	16266	50778	56.51	0.35
pr1002 GA	309698	420741	259045	80.45	0.019
pr1002 ACO	419357	163466	259045	38.11	0.023
rl1323 GA*	310531	712970	270199	85.07	0.012
rl1323 ACO*	444366	398380	270199	35.54	0.009

The Accuracy values for every instance were plotted on a line chart, and grouped into two different lines, based on whether the value was obtained by the GA or ACO, as can be seen in the following line chart:



Similarly to the previous chart, the Accuracy gain per Runtime values for every instance were plotted on a line chart, and grouped into two different lines, based on whether the value was obtained by the GA or ACO, as can be seen in the following line chart:



It is very evident from the first graph that for smaller instances, the ACO algorithm produce slightly better solutions than the GA. However, when the number of cities becomes greater than ~350 cities, the quality of results of the ACO algorithm plummet very quickly, as low as 35% (meaning producing results up to 1.65 times as large as the optimal).

The above suggests that the ACO algorithm should be applied to instances with less than ~350 cities, and GAs on instances with more than 350 cities. However, it must be kept in mind that each algorithm has different running times, so apart from the quality of the solution, it is important to consider how long the algorithms take to produce an answer. One attribute of both GAs and ACO is that they provide a solution from as early as the first iteration. Thus, for measuring the quality of the result produced with respect to the runtime of the algorithm, the *Accuracy gain per Runtime* metric was calculated.
From the second graph, it is evident that for instances of even up to 1000 cities, ACO offers the best performance per unit of time running. That being said, the time taken by the GA is still reasonable, especially when compared to the intractable brute force algorithm, taking just 7 minutes for pr1002, and giving a x1.2 solution. The ACO algorithm, in 3 minutes, gives a x1.62 solution.

Even though for instances of up to and around 1000 cities, ACO does offer the best result per unit time, it is very clear that for instances of more than 350 cities or so, GAs are the preferred choice, as their solutions never go higher than x1.3 of the optimal.

My implementation of GAs for TSP always runs for 5000 generations, as opposed to having a termination condition. Similarly, my implementation of the ACO algorithm always runs for 1000 iterations, as opposed to stopping because of a set termination condition.

The termination condition was not included due to time constraints; however it could have been implemented very easily, such that new generations stop being produced either if the maximum number of generations would have been carried out, or if for the last, say, 500 generations a better solution has not been found, and thus it assumes it has already found the optimal solution and exits.

The number of generations the GA is carried out, or the number of times the ants are simulated explains the results observed earlier. Since the ACO algorithm is only run for 1000 iterations, it always finishes significantly faster than the GA, on average 3-4 times as fast for instances of 52-1002 cities.

For smaller instances the ACO just so happens to provide, on average, better results than GAs, paired with the fact that they terminate 8 times as fast on very small instances (such as berlin52). This is achieved given the fact that 1000 iterations are enough to converge to an almost optimal path on such small instances. The GA on the other hand still must run for 5000 iterations, and usually does not find a more optimal path.
When the number of cities in the instance grows, ACO starts to struggle to compare to the performance of GAs. The runtime of ACO is still significantly faster than that for GAs (even twice as fast for these large instances), however the smaller number of iterations hinders its ability to find a very close approximation to the optimal solution. This is where GAs' 5000 generations prove useful, because in those generations it has the ability to explore so much more of the search space and evolve to find better solutions.

Having read the previous paragraph, one might assume that the ACO, given as many iterations as the GA, might achieve comparable results, this is however definitely not the case. Empirically, repeated testing of the performance of both the GA and the ACO with 5000 iterations revealed that not only did the ACO start to take 3 times longer than the GA, but it ended up producing even worse results than it produced with just 1000 algorithms.

In conclusion, the problems described above can be solved by including a termination condition for both algorithms, this would most probably eliminate most of the time discrepancy between GAs and ACO for smaller instances. Furthermore, in ACOs, the best path found from all iterations should also be kept, such that if further iterations produce worse results than the prior, the best path is not lost. In theory this should never be the case since the best path gets constantly reinforced by new pheromones, however in my implementation, most probably due to the value of the evaporation rate, this is not the case.

D. Gong and X. Ruan suggest that the best algorithm exists in having a Hybrid approach of GA and ACO for TSP, where every chromosome of the GA is also an ant in the ACO. The crossover of the GA is then also based on the pheromone levels in the ACO for the involved paths. [16] [17]

How to Run the Program

In order to select which instances the program will find solutions for, simply download any Symmetric instances from TSPLIB[14], and place them inside the **TSPinstances** folder. Some instances have already been provided inside the folder, and a few more inside **TSPinstanceslibrary**. Instances inside TSPinstances will be evaluated by the algorithms, whereas those in TSPinstanceslibrary will not.
Simply double-click on **run.bat** and the program will perform both GA and ACO on every instance in the TSPinstances folder. It will output the name of the algorithm (GA or ACO), the instance name, the best path found, the length of the best path, and the time taken to find that path.

References

GA:

[1] Jean-Yves Potvin, "Genetic algorithms for the traveling salesman problem", Annals of Operations Research, 1996
https://link.springer.com/article/10.1007%2FBF02125403
https://iccl.inf.tu-dresden.de/w/images/b/b7/GA_for_TSP.pdf

[2] I. Mitchell and P. Pocknell, "A Temporal Representation for GA and TSP", Parallel Problem Solving from Nature PPSN VI, 2000
https://link.springer.com/chapter/10.1007%2F3-540-45356-3_64
https://link-springer-com.ejournals.um.edu.mt/content/pdf/10.1007%2F3-540-45356-3.pdf

[3] D. Kaur and M. M. Murugappan, "Performance enhancement in solving Traveling Salesman Problem using hybrid genetic algorithm," IEEE, 2008.
http://ieeexplore.ieee.org.ejournals.um.edu.mt/document/4531202/?part=1

[4] Y. Deng, Y. Liu and D. Zhou, "An Improved Genetic Algorithm with Initial Population Strategy for Symmetric TSP", Mathematical Problems in Engineering, 2015
https://www.hindawi.com/journals/mpe/2015/212794/

[5] N. Mohd Razali and J. Geraghty, "Genetic Algorithm Performance with Different Selection Strategies in Solving TSP", 2011
https://www.researchgate.net/publication/236179245_Genetic_Algorithm_Performance_with_Different_Selection_Strategies_in_Solving_TSP
https://pdfs.semanticscholar.org/010b/545848cfd29fe6e83987d494fdd00b486229.pdf

[6] Z. H. Ahmed, "Genetic Algorithm for the Traveling Salesman Problem using Sequential Constructive Crossover Operator", International Journal of Biometrics and Bioinformatics (IJBB), 2010
http://www.cscjournals.org/library/manuscriptinfo.php?mc=IJBB-41
http://www.ppgia.pucpr.br/~alceu/mestrado/aula3/IJBB-41.pdf or
http://www.cscjournals.org/manuscript/Journals/IJBB/Volume3/Issue6/IJBB-41.pdf

[7] I. H. Khan, "Assessing Different Crossover Operators for Travelling Salesman Problem", International Journal of Intelligent Systems Technologies and Applications, 2015
https://www.researchgate.net/publication/288854638_Assessing_Different_Crossover_Operators_for_Travelling_Salesman_Problem
http://www.mecs-press.org/ijisa/ijisa-v7-n11/IJISA-V7-N11-3.pdf

[8] O. Abdoun, J. Abouchabaka and C. Tajani, “Analyzing the Performance of Mutation Operators to Solve the Travelling Salesman Problem”, International Journal of Emerging Sciences (IJES), 2012
https://arxiv.org/abs/1203.3099
https://arxiv.org/ftp/arxiv/papers/1203/1203.3099.pdf

[9] A. Rexhepi, A. Maxhuni and A. Dika, “Analysis of the impact of parameters values on the Genetic Algorithm for TSP”, International Journal of Computer Science Issues (IJCSI), 2013
https://pdfs.semanticscholar.org/76fe/335e709579fc5c636a2d4cd99c5627561cd0.pdf

ACO:

[10] M. Dorigo, M. Birattari and T. Stützle, “Ant Colony Optimization – Artificial Ants as a Computational Intelligence Technique”, IEEE Computational Intelligence Magazine, 2016
https://courses.cs.ut.ee/all/MTAT.03.238/2011K/uploads/Main/04129846.pdf

[11] J. Lettman, “Ant Colony Optimization”, YouTube, https://youtu.be/xpyKmjJuqhk

[12] M. Dorigo and L. M. Gambardella, “Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem”, IEEE Transactions on Evolutionary Computation, 1997
http://www-igm.univ-mlv.fr/~lombardy/ens/JavaTTT0708/fourmis.pdf

[13] M. Dorigo and L. M. Gambardella, “Ant colonies for the traveling salesman problem”, BioSystems, 1997
http://people.idsia.ch/~luca/acs-bio97.pdf

Evaluation:

[14] – “MP-TESTDATA - The TSPLIB Symmetric Traveling Salesman Problem Instances”, Available:
http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html

[15] – “Best known solutions for symmetric TSPs”, Available:
http://elib.zib.de/pub/mp-testdata/tsp/tsplib/stsp-sol.html

[16] – D. Gong and X. Ruan – “A hybrid approach of GA and ACO for TSP” – IEEE 2004 -
https://ieeexplore-ieee-org.ejournals.um.edu.mt/document/1341948/

[17] – H. H. Mukhairez and A. Y. A. Maghari, “Performance Comparison of Simulated Annealing, GA and ACO Applied to TSP”, International Journal of Intelligent Computing Research (IJICR), 2015
https://doi.org/10.20533/ijicr.2042.4655.2015.0080