ISSN 1846-6168 (Print), ISSN 1848-5588 (Online)
https://doi.org/10.31803/tg-20220407095736

Preliminary communication
Received: 2022-04-07, Accepted: 2022-04-07

Influence of Programming Language on the Execution Time of Ant Colony Optimization Algorithm

Luka Olivari*, Luca Olivari

Abstract: Supply chains can be accelerated by route optimization, a computationally intensive process for a large number of instances. Traveling Salesmen Problem, as the representative example of routing problems, is NP-hard combinatorial problem. It means that the time needed for solving the problem with exact methods increases exponentially with the increased dataset. Using metaheuristic methods, like Ant Colony Optimization, reduces the time needed for solving the problem drastically but finding a solution still takes a considerable amount of time for large datasets. In today's dynamic environment finding the solution as fast as possible is important as finding a quality solution. The programming language used for finding the solution also influences execution time. In this paper, the execution time of Ant Colony Optimization to solve Traveling Salesman Problems of different sizes was measured. The algorithm was programmed in several programming languages, execution time was measured to rank programming languages.

Keywords: ACO; Ant Colony Optimization; execution time; programming language; Traveling Salesmen Problem; TSP

1 INTRODUCTION

In order to reduce inefficiencies and considerable economic waste, any company with a distribution network needs to accelerate their supply chains. One aspect of supply chain optimization is finding the route with minimal cost. For a large number of instances, route optimization is a computationally very intensive process. For some time now, due to the dynamic and ever-changing environment, requirements for route optimizers have not just been finding the quality solution but finding it as fast as possible. [1] Although there is a metaphorical deluge of heuristic and metaheuristic solvers for route optimization in the scientific literature [2], Ant Colony Optimization (ACO) algorithm is chosen because it has proven to be an efficient and reliable algorithm time and time again over many years. Also, ACO has been historically often used as a solver for a variety of supply chain and vehicle routing problems. [1] Regarding computational intensity, it can represent lots of other algorithms such as Particle Swarm Optimization, Firefly, Bee Colony, Artificial Bee Colony, etc., as they are very similar in implementation to the ACO.

For those interested in creating software applications for solving route optimization problems, an important decision is which programming language to use, as it may considerably impact computational time. The purpose of this paper is to give an answer to the question of which are the fastest and slowest programming languages for this purpose. Python, C, C#, R, and MATLAB as some of the most popular programming languages in science and the industry, have been compared and ranked according to the execution time of the ACO algorithm.

Traveling Salesman Problem (TSP) is chosen to represent routing problems as one of the simplest among them, and yet very complex to solve. TSP is considered the simpler version of the Vehicle Routing Problem (VRP). While VRP looks for the shortest path for m vehicles, visiting n customers, TSP is reduced to one vehicle. A solution to TSP is the shortest Hamiltonian cycle in a complete graph.

Both problems are well known to the scientific community, with TSP being introduced in 1930, [3] and VRP being introduced in 1959. [4] Both problems have their dynamic variant, where information, such as customer location, is subject to change after the vehicle has already started the route. Such variants of problems are named Dynamic Traveling Salesman Problem (DTSP), and Dynamic Vehicle Routing Problem (DVRP). In those dynamic variants of the problems, it is especially important to make quick decisions, as they are made as vehicles are already on the route. Making quick decisions often compromises with decision quality. That means, that more time is used for calculating better solutions, reactiveness to the dynamic changes is decreased.

TSP is an NP-hard computational class of problem, which means that optimum cannot be found by exact algorithms in a time suitable for practical use. Computational problems are divided into complexity classes which determine how much resources are needed to solve a given problem, and how needed resources scale with an increased size of the problem. The computational complexity of a problem is denoted as O and it refers to the worst-case scenario. The size of the problem is denoted as n and it refers to the number of nodes in a graph. Some of the complexity classes are P class, NP class, NP-complete class, and NP-hard class. Solution to the P class of problems can be found in polynomial time $O(n^2)$. For the NP class of problems, solution can be found in nondeterministic polynomial time $O(2^n)$. It is obvious that the computational time of NP problems increases significantly faster with size than the computational time of P problems. NP-complete problems are the hardest problems in NP class, and NP-hard class problems are at minimum as hard as NP-complete problems. [5]

The maximum number of possible tours in symmetric TSP is $(n - 1)!/2$. [6] Brute force algorithms take factorial time $O(n!)$ to solve TSP, which is the worst possible computational complexity class (excluding the theoretical infinite computational complexity class). With dynamic programming, by using the Held-Karp algorithm in this case, the computational complexity of the problem is reduced to exponential time $O(n^2 2^n)$ [7] which means that calculation for large-scale problems takes too much time for practical use.

That is the main reason why heuristics and metaheuristics are often used for finding a near-optimal solution.

Ant Colony Optimization (ACO) algorithm is a metaheuristic method used for solving complex combinatorial problems. ACO is biologically inspired algorithm. It mimics behaviour of ant colonies in search of food. The first ACO algorithm, called Ant System (AS) was created by M. Dorigo in 1996, [8] it was applied to the classical Traveling Salesman Problem. ACO mimics a pheromone trail which real ants use as a communication method while searching for food. High-quality solutions to the complex combinatorial problems can be obtained with ACO algorithm. [9]

To the authors' knowledge, no such comparison of programming language influence on the execution time of the ACO algorithm has been made. Although, other programming language comparisons exist.

In [10] authors compare the execution time of the same algorithm programmed in different programming languages on Windows and Mac operating systems. Chosen programming languages are often used for numerical analysis in macroeconomics. The authors confirmed that compiled programming languages are considerably faster than scripted programming languages.

In [11] authors compare syntax, Lines of Code, Machine Dependency, Compilation Time, and Execution Time of most common high-level programming languages. The authors concluded that there isn't the best overall programming language for all-purpose. Each programming language has its own strengths and weaknesses which potential programmer needs to evaluate according to their own requirements.

In [12] authors compare execution time, memory consumption, and energy efficiency of 27 programming languages. The authors concluded that even though there is a connection between performance and energy consumption, more time-efficient languages are not always ones that consume the least amount of energy. Also, the authors state that it is possible to find the best programming language for execution speed and energy consumption, but it is not the case if memory usage is also taken into consideration.

In [13] authors compare Julia programming language execution speed with several other popular programming languages. Authors increase the complexity of algorithms and measure execution time in comparison with problem size. Julia performed well compared with other languages.

In chapter 2 of this paper Traveling Salesman Problem and Ant Colony Optimization algorithm are explained in detail. Chapter 3 describes the experiment, and a discussion about results is presented in chapter 4.

2 TRAVELING SALESMAN PROBLEM AND ANT COLONY OPTIMIZATION ALGORITHM

2.1 Traveling Salesman Problem

Traveling Salesman Problem (TSP) looks for the shortest path in a complete graph between a given set of nodes, which are often called locations, cities, or customers, that goes through every existing node exactly once and returns to the origin. Only the first/last node is visited twice to close the path. The number of nodes in the problem is denoted with n. If i and j are nodes of a graph, and node i is adjacent to node j, which means that two nodes are connected with an edge ij, Solution to TSP is the shortest Hamiltonian cycle in a complete graph with weighted edges. All graphs can be assumed to be complete by assigning positive infinity (or very large constant) weight to the non-existing edges. [14]

If the cost of traveling between node i and node j is the same as traveling in another direction, from node j to node i, the problem may be represented with an undirected graph. In this case, the problem is called *Symmetric TSP*. If traveling cost is different when traveling from node i and node j than traveling from node j to node i, then is called *Asymmetric TSP*. [15]

The mathematical structure of TSP is a complete weighted graph, where each city salesmen need to visit is represented as a node in the graph, roads between cities are represented with weighted edges between nodes, while edge weight value represents distance or cost of traveling between two cities.

The distance from node i to node j is denoted d_{ij}, where $i, j = 1,\ldots, n$. As a convention, the value of d_{ii} is set to positive infinity to disallow connecting the node to itself. Variable x_{ij} is needed to formulate asymmetric TSP.

$$x_{ij} = \begin{cases} 1, & \text{if node } j \text{ is reach from node } i \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

Variable $x_{i,j}$ is a binary variable, and equal to 1 if node i is reached from node j, and otherwise it is 0 (zero). As stated above in the text, the total number of cities or nodes is denoted n.

Traveling salesman problem can be formulated as:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij} x_{ij} \tag{2}$$

The length of the tour is defined as the sum of edge weights of all the edges included in the tour. To select all edge weights included in the tour, their edge weight is multiplied by $x_{ij} = 1$, to exclude edges that are not part of the solution, their edge weight is multiplied by $x_{ij} = 0$.

Constraints in the TSP are that all nodes in the tour must have exactly one edge pointing to the node, and one edge pointing away from it. As there can be only one node from which node j is visited, called entry node, and entry node can be any node except node j, the constraint can be formulated as:

$$\sum_{\substack{i=1 \\ i \neq j}}^{n} x_{ij} = 1 \quad \forall j = 1, 2, ..., n \tag{3}$$

Also, there can be only one node that is visited after node i, called *exit node*, and that node can be any node except node i, the constraint can be formulated as:

$$\sum_{\substack{j=1 \\ j\neq i}}^{n} x_{ij} = 1 \quad \forall i = 1, 2, ..., n \tag{4}$$

Binary restrictions for x_{ij} can be written as:

$$x_{ij} \in \{0,1\} \quad \forall i = 1, 2, ..., n \quad \forall j = 1, 2, ..., n \tag{5}$$

With this formulation, it is possible that the solution is an unconnected graph, where every connected component is considered subtour. Those solutions may be optimal, but not feasible, as the tour is not closed. To exclude subtours i.e., tours that are not connecting all the cities, additional subtour braking constraints are needed. If S is set of all edges in subtour, the constraint can be written as:

$$\sum_{(i,j)\in S} x_{ij} \leq |S| - 1 \tag{6}$$

This reads as, if subtour exists, not all edges in that subtour can be selected, or from all edges in subtour, at least 1 edge needs to be removed. [15]

Traveling Salesman Problem is represented in the computer as $n \times n$ square matrix, called adjacency matrix where n represents the number of nodes. Value of cell a_{ij} represents the distance between node i and node j. There are no loops in the graph, as it makes no sense that the node connects to itself, so diagonal values of the matrix are always 0.

2.2 Ant Colony Optimization Algorithm

Ant Colony Optimization (ACO) was created in 1996. by M. Dorigo [8]. At the time it was called Ant System (AS). It is a metaheuristic inspired by the stigmergy of the ant colonies. Stigmergy is the mechanism of interaction and coordination between agents. In this case, ants, modify the environment. Agents can be entirely unaware of each other, there is no hierarchy, and there is no direct communication between them. This type of self-organization can produce seemingly intelligent solutions, without any planning or management, by simple agents who are not even aware of each other and have no memory. [16] Ants modify their environment by laying pheromone trails for other ants to "read". Pheromone is a chemical left on the ground by ants for various purposes, upon death ant will release warning pheromone, when searching for food and will lay pheromone to mark its path to find a way back, when an ant finds a food source it will release pheromone for other ants to follow, and when the food source is depleted, it will release yet another kind of pheromone to inform other ants that the food is gone.

Ant Colony Optimization algorithm uses only one kind of artificial pheromone, inspired by the one that ants leave on the ground when they find a source of food. When a single ant finds the food source, it will mark its way back to the colony. The goal of an ant colony is to collect the maximum amount of food by spending the minimum amount of energy. In order to do so, they need to find the shortest path from the colony to the food source. Often there are multiple ways to reach the food. At first, ants will randomly choose a path to reach the food, but in time all ants will travel the shortest one.

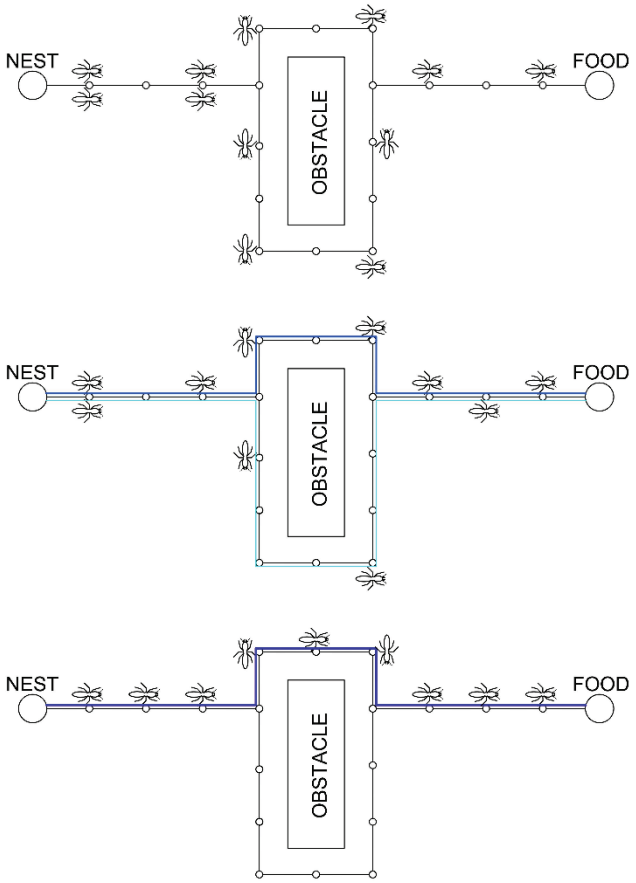

Figure 1 Ant searching for food [authors]

If there are two possible paths to reach the food source, ants have a 50% chance to choose either of them. Those ants that travel shorter path will reach food source faster as is shown in the top picture in Fig. 1. On the way back, all ants leave the same amount of pheromone on each trail, but on the longer trails, a pheromone has more time to evaporate until the next ant is confronted with the choice of path. Now ants have a higher chance to choose the shorter path because it has a stronger pheromone trail. As it is shown in the middle picture in Fig. 1, more ants will travel the shorter path, leaving an even stronger pheromone trail, while pheromone on a longer trail evaporates.

Eventually, as it is shown in the bottom picture in Fig. 1, all the ants will use the shorter path, as pheromone on the longer path is completely evaporated. [17]

Mathematical models of ACO algorithms may slightly variate, like in the case of different methods of laying pheromone. The algorithm can be modeled to release the same amount of the pheromone on the path regardless of path quality or to release a higher amount of the pheromone on paths with higher quality. Or pheromone evaporation may occur before or after laying new pheromone levels in each iteration.

2.2.1 ACO Mathematical Model

There are three important concepts to be modeled, laying pheromone trail, pheromone evaporation, and decision-making. A mathematical model for depositing a higher amount of pheromone on a better-quality path will be used, and it can be formulated as:

$$\Delta\tau_{ij}^{k} = \begin{cases} \dfrac{1}{L_k}, & k^{\text{th}} \text{ ant travels on the edge } ij \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

Where $\Delta\tau_{ij}^{k}$ is the quantity of pheromone deposited by k^{th} ant when traveling from node i to node j; L_k is the path length. The longer the path length, the less pheromone is deposited on every edge that makes that path. If ant did not travel over the edge, it leaves 0 (zero) amount of pheromone on that edge. After every ant has deposited the pheromone, pheromone levels on each edge need to be calculated by summation of all pheromones laid by every ant, which can be formulated as:

$$\tau_{ij}^{k} = \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \tag{8}$$

Where m is the total number of ants. This mathematical expression simulates pheromone deposition without evaporation. If evaporation needs to be modeled, expression is formulated like this:

$$\tau_{ij}^{k} = (1-\rho)\tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \tag{9}$$

Where ρ is the evaporation rate and can be any number between 0 and 1. If the evaporation rate ρ is set to 1, all previous pheromone levels will be evaporated, and expression will be the same as one above. If the evaporation rate ρ is set to zero, no evaporation occurs. If the evaporation rate ρ is set to, let's say 0,4 it means that 40% of previous pheromone levels will evaporate, or more accurately 60% of previous pheromone levels will remain and new pheromone levels will be added. A higher evaporation rate will lead to higher randomization of the new solution.

To use pheromone levels to influence the decision-making of an artificial ant, the probabilities equation will be used:

$$P_{ij} = \frac{(\tau_{ij})^{\alpha}(\eta_{ij})^{\beta}}{\sum\left[(\tau_{ij})^{\alpha}(\eta_{ij})^{\beta}\right]} \tag{10}$$

Where P_{ij} is a probability that an ant will choose a certain path. Quality of edge ij is indicated by η_{ij}, and is calculated as:

$$\eta_{ij} = \frac{1}{L_{ij}} \tag{11}$$

where L_{ij} is the length of the edge ij.

Parameters α and β are used to increase the impact of τ or η on path choosing process of artificial ant. If β is set to 0 (zero) quality of edge ij is not considered in the decision-making process. If α is set to 0 (zero) then the pheromone level of edge ij is not considered in the decision-making process, but that beats the purpose of the Ant Colony Algorithm.

2.2.2 ACO Pseudocode and Initial Parameters

Traveling Salesman Problem will be solved by using Ant Colony Optimization algorithm programmed in several programming languages. ACO algorithm code is based on [18] and [19].

Pseudo code for ACO:

procedure *ACO algorithm for TSP*
    Initialize ACO parameters, initialize pheromone trails
    while (stopping condition not met) **do**
        Construct solutions
        Update pheromone levels
    end
    Display results
end [18]

After initializing the parameters of the ACO algorithm, each ant is placed on a random node. Each ant randomly creates a path in the graph according to the probability rules of an algorithm. After each ant has exhausted all unvisited nodes, it returns to the initial node. The last visited node must be one step away from the initial node. All path lengths are calculated, and the best solution is stored. Pheromones for each path are updated according to path fitness. After the pheromone matrix is updated, evaporation occurs. The algorithm loops until the stopping criteria is met, in this case, the maximum number of iterations. The final step is to display the best path and total path length.

The Ant Colony Optimization algorithm's initial parameters include the maximum number of iterations, number of ants, initial pheromone level, the desirability of an edge, alpha and beta parameters.

The maximum number of iterations determines how many times ACO will run in order to find the solution. The more time algorithm runs the higher chance of finding a better solution, but it takes more time to run the algorithm. In simple problems, solutions may be found in a low number of iterations so there is no need for the algorithm to run any longer as an optimal solution is already found.

The number of ants determines how many artificial ants will be looking for the solution in each iteration i.e., how many solutions will be in each iteration. Again, the higher the number of ants higher the chance of finding a better solution but the algorithm takes more computational resources.

The initial pheromone level is the amount of pheromone on each edge before the first iteration. The initial pheromone level is inversely proportional to an average distance of all the edges multiplied by the number of nodes.

The desirability of an edge is value inversely proportional to edge length, the shorter the edge the more desirable it is for artificial ants to use.

The evaporation rate is set to a constant value, it determines the percentage of pheromone that will evaporate at end of each iteration. The higher the evaporation rate, the more randomized solutions will be.

α and β parameters increase or decrease the influence of pheromone level and path desirability when randomly choosing a path.

Parameter values are shown in table 1.

Table 1 ACO parameters

Parameter	Value
Maximum number of iterations	50
Number of ants	10
Pheromone evaporation rate	80%
α	1
β	2

3 EXPERIMENT

The goal of this experiment is to measure the Execution time of five popular programming languages when solving the Traveling Salesman Problem with the Ant Colony Optimisation algorithm. **Execution time** is "wall-clock time" needed to run the program i.e., processing input given by the user, in this case, node coordinates, and generate a solution, in this case, shortest path on a graph. It should be noted that compiling time is not included in Execution time.

Chosen programming languages are Python, C, C#, R, and MATLAB. These programming languages are very popular in science and industry. According to https://www.tiobe.com/tiobe-index/, all programming languages are ranked in the top 15 most popular programming languages in February 2022. Phyton being ranked no.1, C being ranked no.2, C# being ranked no.5, R being ranked no.13, and MATLAB being ranked no.14. Another important factor for choosing these languages is the authors' familiarity with them.

Python is an open-source, object-oriented interpreted scripting language, used mostly for system administration, CGI programming, and other small computing tasks. It is available for most computing platforms. [20] Visual Studio with Python extension and Python version 3.9.5 was used. Execution time was measured using the *time* module.

C is a flexible, general-purpose programming language, it was initially designed in 1972 for system programming, but it can be used in a wide range of application areas. C gained in popularity in the 1980s because the widely used UNIX operating system provided a compiler for it on different computers. [20] Visual Studio with C/C++ extension and gcc version 11.2.0 was used. Execution time with *clock* function from library *time*.

C# is the simple object-oriented programming language for general purposes. The main concepts of C# are borrowed from Java and C++. [20] Visual Studio with C# extension v1.24.0 and .NET SDK 6.0.102 was used. Execution time was measured using *watch* class.

R programming language was primarily conceived to be used for statistical computing. In the 90s almost hundreds of computer scientists and mathematicians were improving, at that time, very popular programming language. The main goals were to offer free, easy, and versatile programming language. [13] R v4.1.2 version was used in the experiment. R Studio with and R version v4.1.2 was used. Execution time was measured using the *system.time* function. *Elapsed time* was measured and reported because it refers to elapsed wall-clock time.

MATLAB, short for Matrix Laboratory, is a programming platform often used for numeric and scientific computation. MATLAB is a scripted programming language. MATLAB R2015a was used, and Execution time was measured using *tic* and *toc* functions.

The Execution time of the ACO algorithm in each programming language was measured as the number of nodes in TSP was incrementally increased. Execution time was measured for each data set i.e., number of nodes *n.* Problems were divided into two groups. In the first group problems scale form size $n = 20$ to $n = 200$ with an increment of 20. This means that execution time was measured for problem sizes $n = 20, 40, 60, \ldots, 200$. In the second group problems scale from $n=200$ to $n=2000$ with an increment of 200. This means that execution time was measured for problem sizes $n = 200, 400, 600, \ldots, 2000$. Nodes were randomly generated, but the time needed for random number generation was not measured. It should be noted that the quality of the solution was not considered, only the Execution time for a limited number of iterations. Although a very important characteristic of any solver, convergence rate was not measured for two reasons. First, it is considered that, if the algorithm were to run enough times in any programming language, statistically, the convergence rate should be the same in all cases because they all run the same algorithm with the same parameters. Another reason why convergence rate was not considered is that benchmark datasets consist of irregular increases between them. For example, as found on TSPLIB (http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/) instances sizes are $n = 14, 29, 52, 58, 130, 150, \ldots, 1000, 1291, 1577$, etc. For our purpose, we wanted a regular increase in instance size, so we used randomly generated coordinates for each instance. Because node coordinates were randomly generated, optimal solutions were not available, and convergence rate was not possible to determine. Also worth mentioning is that coordinates of the nodes do not influence execution time at all.

Original code was created in MATLAB, and then "translated" into other programming languages. In C, C# and R used arrays were static. In Python, even though lists were used their size wasn't changed during the execution of the algorithm. In MATLAB new elements were added to the list. ACO algorithm was created to be time-efficient, but authors cannot guarantee that the algorithm in all languages was made as time-efficient as possible. Experienced

programmers in optimization problems and languages used would most likely find room for improvement. One of the possible improvements would be to preallocate memory i.e., that a program allocates all the required memory blocks once after start-up, rather than allocate memory multiple times during execution and leave a memory that is no longer needed for the garbage collector to free.

The experiment was conducted on Windows 10 Enterprise operating system. Computer configuration is Intel(R) Core™ i7-8750H CPU 2.20 GHz, Installed RAM 16 GB, 64-bit operating system, x64-based processor. Source code used in the experiment can be found on the GitHub repository available at https://github.com/l-olivari/influence-of-programing-language-on-the-execution-time-of-ant-colony-optimization-algorithm.git.

As stated in Tab. 1, the algorithm stopping criteria was 50 iterations, and 10 ants were used in each iteration. If execution time was less than 20 seconds, the algorithm was run 5 times and results were averaged to reduce the influence of background processes of the operating system. Similarly, if execution time was between 20 and 60 seconds, the algorithm was run 3 times and the result was averaged. If execution time was above 60 seconds, the algorithm was run only once as time variations for multiple runs were insignificant relative to total time.

Results of absolute Execution time for problems size n = 20 - 200 are shown in Tab. 2. As it can be seen, the fastest results are obtained with the C programming language.

Table 2 Absolute Execution time in seconds for problems size n = 20 - 200

n	Python	C	C#	R	MATLAB
20	0,194	0,007	0,018	0,111	0,229
40	0,560	0,030	0,058	0,256	0,437
60	1,079	0,068	0,122	0,426	0,654
80	1,795	0,121	0,211	0,661	0,871
100	2,672	0,190	0,326	0,907	1,106
120	3,676	0,273	0,462	1,255	1,325
140	4,926	0,372	0,626	1,517	1,579
160	6,350	0,484	0,809	1,921	1,821
180	7,989	0,615	1,022	2,425	2,051
200	9,624	0,758	1,253	2,862	2,324

In Tab. 3 relative time is shown. The fastest performing programming language is given value 1,0. Other values are calculated by dividing their respective absolute Execution time by the fastest absolute Execution time for each problem size. The table gives a clear indication of how many times other languages are slower than a best-performing programming language.

Table 3 Relative Execution time for problems size n = 20 - 200

n	Python	C	C#	R	MATLAB
20	27,0	1,0	2,5	15,5	31,8
40	18,8	1,0	1,9	8,6	14,7
60	16,0	1,0	1,8	6,3	9,7
80	14,9	1,0	1,7	5,5	7,2
100	14,1	1,0	1,7	4,8	5,8
120	13,5	1,0	1,7	4,6	4,9
140	13,2	1,0	1,7	4,1	4,2
160	13,1	1,0	1,7	4,0	3,8
180	13,0	1,0	1,7	3,9	3,3
200	12,7	1,0	1,7	3,8	3,1

Graphical representation of data from Tab. 2 i.e., the absolute execution time for problem sizes n = 20 - 200 is shown in Fig. 1.

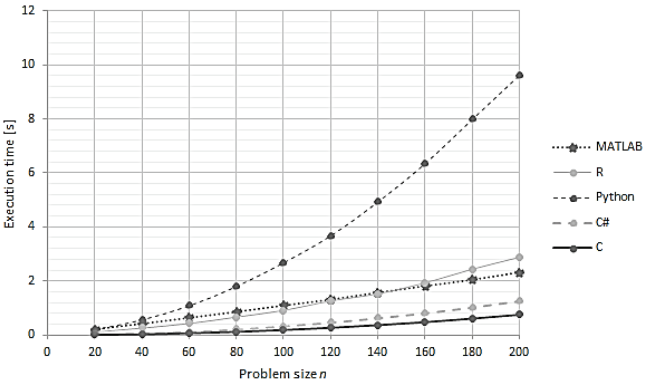

Figure 2 Absolute Execution time (n = 20 – 200) [authors]

As absolute Execution time for the Python programming language is much larger compared to other programming languages, it was excluded from graphical representation in Fig. 2 to offer a clearer comparison between other programming languages.

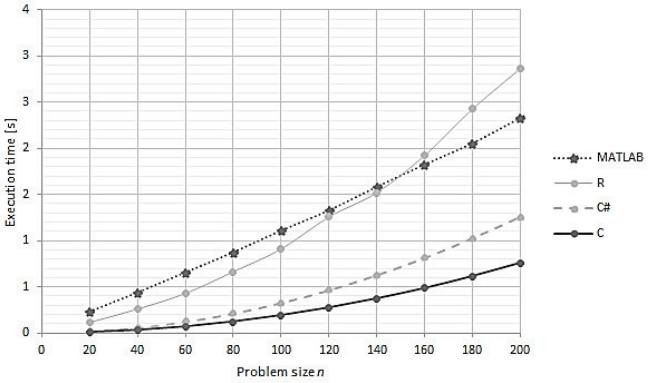

Figure 3 Absolute Execution time (n = 20 – 200) without Python [authors]

Results of absolute Execution time for problems size n = 200 - 2000 are shown in Tab. 4. As it can be seen, the fastest results are again obtained by the C programming language, while Python has the worst results.

Table 4 Absolute Execution time in seconds for problems size n = 200 - 2000

n	Python	C	C#	R	MATLAB
200	9,62	0,758	1,25	2,86	2,32
400	36,87	3,034	5,01	9,81	5,35
600	80,93	6,809	11,13	23,36	10,12
800	147,42	12,162	19,70	39,95	16,76
1000	223,36	18,928	30,61	62,40	24,00
1200	335,51	27,048	44,26	91,78	34,40
1400	452,51	37,031	59,64	124,15	44,96
1600	605,57	48,349	77,46	160,44	55,40
1800	797,30	62,341	98,03	204,53	69,02
2000	919,46	76,958	120,03	253,70	82,98

In Tab. 5 relative time is shown. The fastest performing programming language is given a value of 1,0. As in Tab. 3, other values are calculated by dividing their respective absolute Execution time by the fastest absolute Execution

time. The table gives a clear indication of how many times other languages are slower than a best-performing programming language.

Table 5 Relative Execution time for problems size n = 200 - 2000

n	Python	C	C#	R	MATLAB
200	12,7	1	1,7	3,8	3,1
400	12,2	1	1,6	3,2	1,8
600	11,9	1	1,6	3,4	1,5
800	12,1	1	1,6	3,3	1,4
1000	11,8	1	1,6	3,3	1,3
1200	12,4	1	1,6	3,4	1,3
1400	12,2	1	1,6	3,4	1,2
1600	12,5	1	1,6	3,3	1,1
1800	12,8	1	1,6	3,3	1,1
2000	11,9	1	1,6	3,3	1,1

Graphical representation of data from Tab. 4 is shown in Fig. 3 i.e., the absolute execution time for problem sizes n = 200 - 2000.

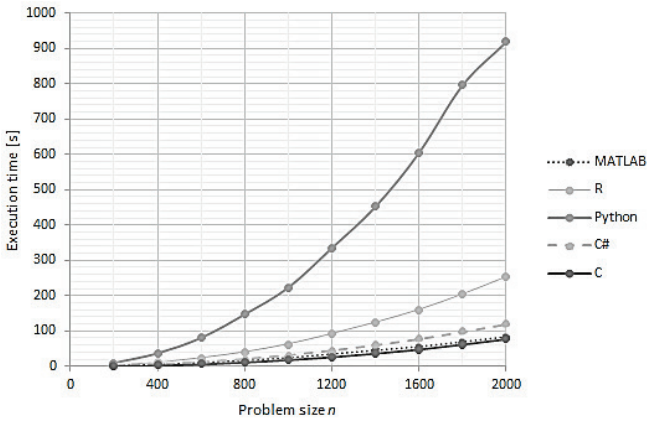

Figure 4 Absolute Execution time (n = 20 – 200) [authors]

As was the case with Fig. 2, results obtained by Python programming language were excluded from graphical representation in Fig. 4 to offer a clearer comparison between other programming languages.

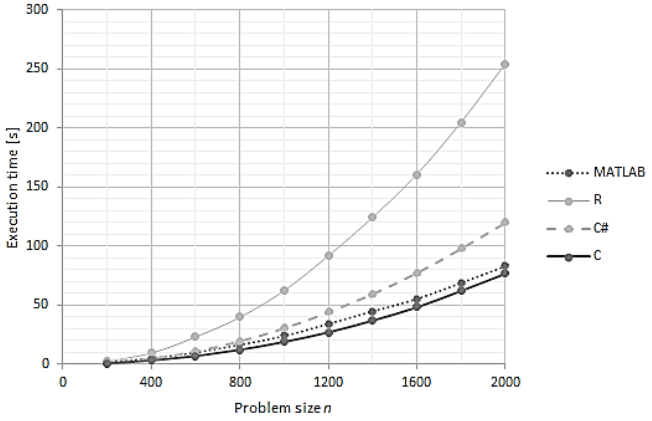

Figure 5 Absolute Execution time (n = 200 – 2000) without Python [authors]

4 DISCUSSION

For problem sizes n = 20 - 200 fastest programming language turns out to be C, closely followed by C#. Relative differences in Execution time between programming languages are higher the smaller problem it is. For example, MATLAB is almost 32 times slower than C for problem size n = 20, but only 3,1 times slower for problems size n = 200. Similarly, R is 15,5 times slower than C for problem size n = 20, to be only 3,8 times slower than C for problem size n = 200. For problem sizes n = 20, programming languages ranked from fastest to the slowest are:

1. C
2. C#
3. R
4. Python
5. MATLAB

For problems sizes n = 40 - 140 situation is different as Python falls to the last place, and languages are ranked:

1. C
2. C#
3. R
4. MATLAB
5. Python

For problems size n = 160 - 200, programming languages are ranked:

1. C
2. C#
3. MATLAB
4. R
5. Python

For problem sizes n = 200 - 2000, C is again being fastest, with C# in the second place up to the problem size n = *400*, but then is taken over by MATLAB. Python remained the slowest one. So, for problems size n = 200 - 400 ranking is not changed.

Finally, for problems size n = 400 - 2000, programming languages are ranked:

1. C
2. MATLAB
3. C#
4. R
5. Python

C programming language is undisputed winner of this race, which is not surprising because it is "lowest" of "high" programming languages, also it is compiled language, while others, on this list, except C#, are scripted languages. C has proven to be the fastest programming language for both, small and large problem sizes.

C# is somewhat slower than C, but it performs consistently just above 1,5 times slower than C except for small problem sizes (n = 20 and 40). C# hold second place for all problem sizes up to n = 400.

MATLAB Execution time was very interesting to follow, as it started as the slowest programming language, to overtake Python at problem size n = 40, and R at problem size n = 160, only to end up in the second place at problem size n = 600. It turns out to be just 1,1 times slower than C for problem size n = 2000, while C# was 1,6 times slower for the same problem size. Authors tested will MATLAB overtake C for even larger problem sizes, but as it is not the case, results are not reported in the tables.

Python is convincingly the slowest programming language overall. It was slower than other candidates to such an extent it had to be removed from graphical representations to allow a clearer representation of other programming languages. The biggest relative difference from C was 27 times slower at problem size n = 20, and the smallest relative difference from C is 11,9 times slower at problem size n = 2000. As results were unexpected, to make sure that mistake isn't made on our part and syntax is correct, we used an independent algorithm (which can be found on: https://pypi.org/project/ACO-Pants/) and compared results. Results from our code and independent code can be found side-by-side in Tab. 6.

Table 6 Relative Execution time for two different algorithms in Python

n	Our code	Independent code	n	Our code	Independent code
20	0,194	0,193	200	9,624	9,398
40	0,560	0,699	400	36,869	37,984
60	1,079	0,880	600	80,933	85,574
80	1,795	1,499	800	147,421	158,731
100	2,672	2,283	1000	223,356	247,655
120	3,676	3,240	1200	335,513	355,658
140	4,926	4,560	1400	452,507	491,909
160	6,350	5,949	1600	605,573	653,725
180	7,989	7,562	1800	797,296	808,770
200	9,624	9,398	2000	919,456	1017,300

As it can be seen, the results are roughly the same. Our algorithm was somewhat slower for smaller instances (n = 20 - 200) but was somewhat faster for large instances (n = 200 - 2000).

R is performing consistently well, as with the other programming languages, the relative difference is reducing with the size of the problem, to be about 4 times slower than C for problem sizes n = 140 – 200, and about 3,3 times slower than C for problem sizes above n = 400.

5 CONCLUSION

The fastest programming language for solving Traveling Salesman Problem with Ant Colony Optimization algorithm, unsurprisingly, turns out to be C, closely followed by C# for problem sizes less than n = 400, at which point it is overtaken by MATLAB. Python turns out to be the slowest programming language for solving this kind of problem for all problem sizes, except for n = 20, in which case MATLAB is to be the slowest one.

It was interesting to see MATLAB, which was the slowest programming language for problem size n = 20, taking third place for problem size n = 160 - 200, and then climbing to second place for problem sizes n = 400 - 2000.

A possible explanation why MATLAB was the slowest programming language in the beginning and got at the second place, in the end, is because original code was created in MATLAB and then "translated" into other programming languages. As stated above in the text, code is based on [19], whose author is internationally recognized for his innovations in optimization algorithms. It is possible that small in-language optimization techniques add up for large instances in the case of complex algorithms such as Ant Colony Optimization and result in lower Execution time.

Although relative differences for small problem sizes are largest, the programming language does not make a considerable difference as results are calculated within a fraction of the second. For large problem sizes, Python should be avoided, while C, C#, and MATLAB are all good choices.

In future research, it would be important to find out what makes Python, in this group, the slowest programming language for large instances, and can something be done to alleviate slow execution times. Also, it would be interesting to see how the Execution time of the ACO algorithm scales when it is used for solving VRP and DVRP problems and to include even more programming languages such as C++, Java, JavaScript, Julia, and others. Future analysis can be expanded to include other metrics such as lines of code, memory management, and energy consumption.

Notice

The paper will be presented at MOTSP 2022 – 13th International Conference Management of Technology – Step to Sustainable Production, which will take place in Primošten/Dalmatia (Croatia) on June 8–10, 2022. The paper will not be published anywhere else.

6 REFERENCES

[1] Dzalbs, I. & Kalganova, T. (2020). Accelerating supply chains with Ant Colony Optimization across a range of hardware solutions. *Comput. Ind. Eng., 147*, 106610. https://doi.org/10.1016/j.cie.2020.106610

[2] Osaba, E., Yang, X. S., & Del Ser, J. (2020) Is the Vehicle Routing Problem Dead? An Overview through Bioinspired Perspective and a Prospect of Opportunities. In: Yang X. S. & Zhao, Y. X. (eds) *Nature-Inspired Computation in Navigation and Routing Problems*. Springer Tracts in Nature-Inspired Computing. Springer, Singapore. https://doi.org/10.1007/978-981-15-1842-3_3

[3] Applegate, D. L., Bixby, R. E., Chvátal, V., & Cook, W. J. (2011). *The Traveling Salesman Problem: A Computational Problem*. Princeton: Princeton University Press. https://doi.org/10.1515/9781400841103

[4] Dantzig, G. B. & Ramser, J. H. (1959). The Truck Dispatching Problem. *Manage. Sci., 6*(1), 80-91. https://doi.org/10.1287/mnsc.6.1.80

[5] Arora, S. & Barak, B. (2009). *Computational Complexity: A Modern Approach*. Cambridge University Press. https://doi.org/10.1017/CBO9780511804090

[6] Taha, H. A. (2017). *Operations Research an Introduction*, 10th edition. Fayetteville: University of Arkansas.

[7] Held, M. & Karp, R. M. (1962). A Dynamic Programming Approach to Sequencing Problems. *Journal of the Society for Industrial and Applied Mathematics, 10*(1), 196-210. https://doi.org/10.1137/0110015
[8] Dorigo, M., Maniezzo, V., & Colorni, A. (1996). Ant system: Optimization by a colony of cooperating agents. *IEEE Trans. Syst. Man, Cybern. Part B Cybern., 26*(1), 29-41. https://doi.org/10.1109/3477.484436
[9] Mavrovouniotis, M., Yang, S., Van, M., Li, C., & Polycarpou, M. (2020). Ant colony optimization algorithms for dynamic optimization: A case study of the dynamic travelling salesperson problem (Research Frontier). *IEEE Comput. Intell. Mag., 15*(1), 52-63. https://doi.org/10.1109/MCI.2019.2954644
[10] Aruoba, S. B. & Fernández-Villaverde, J. (2015). A comparison of programming languages in macroeconomics. *J. Econ. Dyn. Control, 58*, 265-273. https://doi.org/10.1016/j.jedc.2015.05.009
[11] Fatima, N. & Parveen, Z. (2016). Performance Comparison of Most Common High Level Programming Languages. *Int. J. Comput. Acad. Res., 5*(5). 246-258. Available: http://www.meacse.org/ijcar
[12] Pereira, R. et al. (2017). Energy Efficiency across Programming Languages. *Int. Conf. Softw. Lang. Eng.*, 256-257.
[13] Januszek, T. & Pleszczyński, M. (2018). Comparative Analysis of the Efficiency of Julia Language against the Other Classic programming languages. *Silesian J. Pure Appl. Math., 8*(1), 49-56.
[14] Korte, B. & Vygen, J. (2020). The Traveling Salesman Problem. In: *Combinatorial Optimization. Algorithms and Combinatorics, 21*, 473-505. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-21708-5_21
[15] Hoffman, K. L. & Padberg, M. (2001). Traveling Salesman Problem. *Encyclopedia of Operations Research and Management Science, 3*(1), 849-853. https://doi.org/10.1007/1-4020-0611-X_1068
[16] Marsh, L. & Onof, C. (2008). Stigmergic epistemology, stigmergic cognition. *Cogn. Syst. Res., 9*(1-2), 136-149. https://doi.org/10.1016/j.cogsys.2007.06.009
[17] Dorigo, M. & Stützle, T. (2019). Handbook of Metaheuristics; Ant colony optimization: Overview and recent advances. *International Series in Operations Research and Management Science, 272*, 311-351. Springer, Cham. https://doi.org/10.1007/978-3-319-91086-4_10
[18] Stutzle, T. & Dorigo, M. (1999). ACO Algorithms for the Traveling Salesman Problem. Evol. Algorithms Eng. Comput. Sci. Recent Adv. Genet. Algorithms, Evol. Strateg. Evol. Program. Genet. Program. Ind. Appl.
[19] Mirjalili, S. (2022). Ant Colony Optimization MATLAB code. https://seyedalimirjalili.com/aco
[20] Sebesta, R. W. (2012). *Concepts of programming languages*, 10th ed. Colorado: University of Colorado.

Authors' contacts:

Luka Olivari, mag. ing. mech., lecturer
(Corresponding author)
Polytechnic of Sibenik,
Trg Andrije Hebranga 11, 22 000 Šibenik, Croatia
(022) 311-060, lolivari@vus.hr

Luca Olivari, mag. math, professor assistant
Polytechnic of Sibenik,
Trg Andrije Hebranga 11, 22 000 Šibenik, Croatia
(022) 311-060, lolivari1@vus.hr