ISSN 1846-6168 (Print), ISSN 1848-5588 (Online)
https://doi.org/10.31803/tg-20230825125127

Original scientific paper
Received: 2023-08-25, Accepted: 2023-09-12

Reducing ACO Population Size to Increase Computational Speed

Luka Olivari

Abstract: Ant Colony Optimization (ACO) is a powerful metaheuristic algorithm widely used to solve complex optimization problems in production and logistics. This paper presents a methodology for enhancing the ACO performance when applied to Traveling Salesman Problems (TSP). By reducing the number of ants in the colony, the algorithm's computational speed improves but solution quality is sacrificed. An optimal number of ants to produce the best results in the shortest time is specific to the problem at hand and can't be defined generally. This paper investigates the effect of ant population reduction relative to the problem size by measuring its impact on solution quality and execution time. Results show that for certain problem sizes ant population and execution time can be halved with practically no reduction in solution quality, or they can be reduced 5 times at the price of slightly worse solution quality. Reduction of ant population is much more impactful on reduction of execution time than it is on solution quality.

Keywords: ACO; Ant Colony Optimization; colony population; number of ants; speed up

1 INTRODUCTION

Ant Colony Optimization (ACO) is considered to be a reliable and efficient algorithm for solving many problems in production such as facility layout design to determine the optimal arrangement of machines to minimize material handling; job scheduling on the CNC machines to optimize the scheduling of jobs on machines in manufacturing processes, ensuring efficient resource utilization and minimizing completion times; tool path optimization on CNC machines to increase production and reduce costs; and many more problems that can be approximated with Traveling Salesman Problem (TSP). ACO is also often used in logistics for Vehicle Routing Problem (VRP) to optimize the routing and scheduling of vehicles, reducing travel distances and operational costs.

Both TSP and VRP belong to the NP-hard problem category [1, 2], implying that solving them using exact methods becomes significantly more challenging as the problem size increases. Due to long calculations, exact methods are not feasible for practical use. Heuristics and metaheuristics, such as ACO, are used for finding near-optimal solutions in time acceptable for practical use. Fast execution time is increasingly important as many of today's real-world problems are dynamic in nature, and solutions need to be found on the go. ACO is considered to be a more popular algorithm than any other metaheuristics with publications in highly reputed journals [3] whose convergence has been analytically proven. [4] These are some of the reasons why ACO still attracts the attention of many researchers to further its performance. It's important to clarify that an algorithm's performance encompasses both efficiency (finding solutions with minimal resource consumption, like time) and effectiveness (achieving solutions with the highest fitness or quality).

ACO population size corresponds to the number of generated solutions in each iteration. What is called "an ant" is actually one generated solution by the algorithm in one iteration. As a larger population means more generated solutions, it is expected that more ants will have a higher chance of producing a better-quality solution. Also, a bigger population means more computational time is needed to generate all solutions in each iteration.

This research investigates how reducing the ant colony population relative to the problem size affects the efficiency and effectiveness of the ACO algorithm when applied to TSP instances. In essence, the study seeks to optimize the trade-off between computational speed and solution quality. Reducing the number of ants in each iteration is expected to reduce the execution time of the ACO algorithm, with an expected trade-off of lower solution quality.

The literature survey shows that an optimal number of ants as an integer is specific to the case at hand and can't be defined generally. This paper is unique because it investigates a number of ants in relation to the size of the problem in the hope of finding a pattern that can be applied generally to problems of similar size. A similar research was conducted but it focused on the Ant Colony System (ACS) unlike the Ant System (AS) which is a focal point of this paper.

The aim of this paper is to quantify the impact of reducing the ant colony population (i.e., the number of ants) on the efficiency and effectiveness of the Ant System algorithm, focusing on problem sizes ranging from 100 to 200 nodes. Additionally, it aims to provide guidelines for enhancing the performance of ACO algorithms used to tackle similar problems. The novelty of this research lies in its focused investigation into the impact of population size reduction in ACO and the identification of population size "sweet spots" for balancing efficiency and effectiveness in solving TSP instances.

The authors hope that the research findings can serve as benchmark data for comparing the efficiency and effectiveness of ACO variations and other optimization algorithms in solving TSP and related problems. Also, these insights can help practitioners using ACO algorithms for optimization problems to make it more feasible to find solutions in dynamic and time-sensitive situations which can lead to cost savings in terms of increased operational efficiency and reduced computing resources.

Section 2 provides an overview of the Ant System and speed-up techniques unique to ACO algorithms, as well as a literature survey. Section 3 presents the time analysis of AS, details the methodology and experiment, and finally presents the results of the research. In Section 4, we discuss findings, present guidelines, and conclude with Section 5.

2 ANT COLONY OPTIMIZATION

Ant Colony Optimization (ACO) is a metaheuristic framework that incorporates algorithms inspired by the indirect coordination of ants in nature. Simple agents, ants, without memory and unaware of each other are able to solve complex tasks by modifying their environment. Ants communicate indirectly by depositing pheromone trails and marking paths from the colony to food sources. Pheromone quantities on different paths vary. Shorter paths will accumulate more pheromones than longer paths due to pheromone evaporation, as it takes an ant more time to travel the longer path, pheromones will have more time to evaporate.

If there are multiple possible paths to reach the destination an ant will select a path semi-randomly based on pheromone levels. Shorter paths with higher pheromone concentrations hold a greater probability of selection. Repeated use of a shorter path results in added pheromones, increasing its appeal until competing paths lose all pheromones, rendering them unviable options. The described mechanism is imitated by ACO to solve complex problems like TSP and VRP. [5]

ACO metaheuristic framework encompasses a variety of ant-inspired algorithms, including the Ant System (AS), the first algorithm in the ACO family proposed in 1992 by Marco Dorigo in his PhD thesis. [6] AS was introduced to solve the classical Traveling Salesman Problem (TSP). [7] The objective of TSP is to find the shortest path while visiting every city exactly once and then return to the origin. The problem is represented with a complete graph in which nodes represent cities. The solution to TSP is a Hamiltonian cycle of minimal length. Ants in the AS algorithm "move" through the graph, from node to node, until all nodes are visited, and ant returns to the starting node. Ants choose the next node probabilistically based on pheromone levels on edge between nodes and heuristic information i.e., the distance between nodes. After the tour is complete, ants mark their path with pheromones according to path length i.e., solution quality.

To avoid infeasible paths that might have acquired large pheromone amounts by chance, a pheromone evaporation mechanism is employed. Before the deposition of pheromones in the current iteration of the algorithm, evaporation occurs reducing the overall pheromone levels on all edges between nodes. The evaporation rate is set to values between 0 and 1. The 0 represents no evaporation at all, and 1 represents 100% evaporation of pheromones.

Several techniques are available to enhance algorithm performance, including coding style adjustments, and algorithm refinements. Usually, coding style speed-up techniques shouldn't impact the algorithm's effectiveness only its efficiency. Coding style techniques include reducing the complexity of calculations if possible, minimizing redundant operations, using vectorization instead of loops, minimizing function calls to avoid overhead, avoiding unnecessary display of the results and visualizations, etc.

Improvements made to the algorithm usually intend to enhance efficiency or effectiveness. Sometimes, improvements are realized in both, as is the case with the Rank-Based Ant System (RAS). [8] This variant of AS reinforces pheromone deposition of the top-ranked solutions, effectively increasing the convergence rate and quality of the solution, and simultaneously reducing the computational time as only a limited number of ants update their pheromone trails. It is a considerable reduction in computation, compared to the AS where all ants update their pheromone trails. Another improvement of AS is the MAX-MIN Ant System (MMAS), where typically the iteration-best or overall-best ant alone updates pheromone trails. [9] MMAX has a slower convergence rate compared to AS or RAS, but it produces better solutions in the long run.

Certain efficiency-focused methods can compromise the algorithm's effectiveness. In such scenarios, the trade-off between efficiency gains and potential drawbacks should be carefully evaluated. One such method is the use of *Candidate lists*. For big problems, when constructing a solution, an ant has a large number of possible choices for movement which causes an increase in execution time. Using *Candidate lists*, a list of nearest neighbors for each node in the problem, the computation can be accelerated. During the construction of the tour, an ant can move to nodes that are not on the *Candidate list* only if every node on the list is already visited. Although an optimal solution often can be found within a reasonably small number of nearest neighbors, this technique can make it impossible to find the optimal solution. [5, 10]

On the other hand, increasing the effectiveness of the algorithm often comes at the price of longer computational time. A good example is combining ACO with Local Search (LS) algorithms. ACO and LS complement each other rather nicely, as ACO is able to quickly find good quality near-optimal solutions, and Local Search is used to fine-tune those solutions as its effectiveness is largely influenced by the starting solution. There are many ways to combine ACO and LS, use it only to improve the final solution or use it to improve solutions in each iteration. Also, advanced LS algorithms produce better-quality solutions, but the general rule is always the same, better-quality solutions come at the price of longer execution time. [5, 11]

The construction of the tour is independent for each ant, which means that ACO naturally allows *parallel implementation* of the algorithm. Communication overhead can be a problem for smaller problems, resulting in longer computational time than it was initially. Because of this, and many other factors that influence *parallel implementation*, the exact problem size, when parallelization becomes beneficial, should be determined. [5] A case where multiple colonies run on multiple processors is common, although the easiest and most effective way of parallelizing ACO is to independently run many ACO algorithms. [7, 8]

Finally, the choice of programming language could significantly impact algorithm efficiency. This may not be the case for small problem sizes where a solution is generated in a fraction of a second. However, it becomes a considerable influence for large problems. Generally, lower-level programming languages, like C, exhibit greater speed compared to higher-level counterparts like C# or Python. The reason is that higher-level programming languages are user-oriented to be more accessible to a wider audience, while lower-level languages are machine-oriented, which makes them easier to process by the machine, but it makes coding more complex compared to higher-level languages.

Another aspect to consider is the distinction between compiled and scripted languages. Compiled programming languages with superior speed come at the price of overhead. Using compiled language, a user must edit, compile, run, and debug code to test it. Scripted languages allow interactive coding and easy testing of parts of the code but are generally considered to be slower. MATLAB, as a higher-level, scripted language proved to be slower than other programming languages, such as C#, R, or Python for small problem sizes. As problem size increases MATLAB turns out to be a very efficient programming language with faster execution speed than the mentioned programming languages, which could be a result of optimized internal functions and processes. [14, 15]

2.1 Literature Survey

In [16] authors try to overcome the low efficiency of the Ant Colony System (ACS), an extension of AS, in solving TSP within a limited time. The authors define the correlation coefficient between the initial population size and the number of nodes in the problem. Improvement was tested on benchmark instances ranging from 30 to 127 nodes.

In [10] authors present an analysis of the number of ants in the Ant Colony System (ACS) algorithm. The paper studies the effect of gradually changing the number of ants from 1 to 100 on the algorithm's behavior. Tests were performed on eil51 and kroA100 TSP benchmark instances. The authors conclude that a large number of ants did enhanced the algorithm performance, and for tested problems small number of ants in ACS is recommended.

On the other hand, in [17] authors discuss the application of hyperpopulated ant colonies to solve TSP. The authors propose an increased number of ants in the colony or assigning more colonies to the same problem to reduce the number of iterations in finding the solution. The parallel implementation makes it possible to reduce processing time.

In [18] authors investigate the number of ants (in the range of 3 to 12) used in a hybrid method based on ACO and Artificial Neural Networks in relation to the number of iterations, penalized objective function, and optimization time. The authors note that the optimum number of ants is specific to the dataset considered, and report that for their data set best results were achieved with five ants, and increasing the number of ants resulted in increased execution time.

In [19] authors introduce k-means clustering to group nodes in TSP and apply ACO to individual groups. Finally, using the connection technique join these groups into a single route. This way computational time of the ACO algorithm is reduced to 32% of the original run without clustering. Improvement was tested on benchmark instances ranging from 30 to 150 nodes.

In [20] authors made the basic alterations in the Ant Colony System (ACS) and Max-Min Ant System (MMAS) with respect to the Ant System (AS). Also, they compare the results based on different parameter adjustments for chosen algorithms. One parameter adjustment was the number of ants. The authors increase the population size from 16 ants to 52 ants in five steps. According to their findings, the MMAS execution time grows fastest with increased population compared to the other two algorithms.

In [21] authors introduce a novel *Partial ACO* (P-ACO) variant for solving larger TSP cases, achieved through the reduction of extensive memory demands, utilization of parallel CPU hardware, and the introduction of a substantial efficiency-enhancing strategy. By partially modifying the best tour only this approach resulted in increased efficiency and effectiveness of the algorithm. The algorithm was applied to TSP instances of up to 200 000 nodes and achieved solutions that are 5-7% longer than the best-known tours so far.

In [22] authors study the ACO algorithm's performance variations with the number of ants. They conclude that less ants favors exploration, while more ants favors exploitation, so the optimal number of ants depends on each specific problem.

In [23] and [24] authors have studied the influence of the number of ants on the performance of the MMAS algorithm, applied to the multi-objective problem of wireless sensor network design. In the first paper, they fixed the number of iterations and increased the number of ants from 1 to 10 incrementally. In the second paper, they compared the influence of the number of ants and the number of iterations on solution quality.

In [25] authors focus on the number of ants in the Simplified Ant Colony Algorithm (SACO) using the optimization of grillage structure. They conclude that there is a correlation between the number of ants and the number of design variables as more ants are required to achieve the optimum solution in the cases with more design variables. This correlation cannot be defined with a regular function.

In [26] authors propose a warm-up procedure for ACO, which initializes the pheromone matrix to provide a good starting point to obtain better results in fewer iterations. The warm-up procedure is based solely on the graph that formalizes the problem. The procedure was tested on benchmark instances as well as on a simulation of the real warehouse which was represented by the graph with 380 nodes.

In this study [27] the effect of optimization parameters in ACO, such as the number of ants, was investigated. The authors used 21 colonies with 5 to 5000 ants. Depending on the increase in the number of ants, the number of iterations decreased, and the optimization time increased nonlinearly. The authors conclude that the optimum number of ants is crucial for reaching the best solution in a short time.

In [28] authors propose a dynamic adaptive ACO algorithm (DAACO) that dynamically determines the population size of the colony to prevent falling into the local optimum, it also implements a hybrid local selection strategy to increase the quality of solution and reduce the execution time. Algorithm was tested on benchmark instances ranging from 291 to 1577 nodes with results showing that DAACO has advantages in convergence and solution quality compared to similar ACO algorithms.

3 METODOLOGY AND EXPERIMENT

3.1 Time Analysis of the Ant System

In this paper Ant System (AS), the first algorithm in the ACO framework, was applied to Traveling Salesman Problem (TSP).

Ant System algorithm consists of the following integral parts: *Initialize Algorithm*, *Main Loop*, and *Display Results*. The *Main Loop* is further divided into several steps: *Construct Tour* using the *Roulette Wheel* function, *Calculate Fitness*, *Update Pheromones*, and *Optional Actions*. The procedure of AS can be visualized with the pseudo-code presented in Fig. 1.

```
Ant System
  Initialize Algorithm
  Main Loop
    For i = 1 : Max. Iterations
      Construct Tour
          Roulette Wheel
      Calculate Fitness
      Update Pheromones
      Optional Actions
    end Main Loop
  Display Results
end Ant System
```

Figure 1 Ant System in pseudo-code [author]

The problem and parameters (such as pheromone evaporation rate and stopping criteria) are defined in the *Initialising Algorithm* section of the code. In this case, the stopping criteria are the maximum number of iterations. Initial pheromone levels are deposited according to the solution obtained with the Nearest Neighbour algorithm. *Construct Tour* manages ants in the colony that independently "walk the graph" i.e., construct solutions. Individual ant chooses the next node using the *Roulette Wheel* function based on pheromone levels and heuristic information. Each ant represents one possible solution to the TSP. When all tours are constructed, their lengths are calculated using the *Calculate Fitness* function. Before each ant deposits pheromones based on tour fitness using the *Update Pheromone* function, existing pheromones must partially evaporate. *Optional Activities* can include a Local Search algorithm to further improve tours, depositing additional pheromones on the best-so-far tour, etc. This paper did not incorporate any optional activities within the ACO algorithm. Finally, solution (order of visiting nodes) and solution fitness (tour length) are displayed.

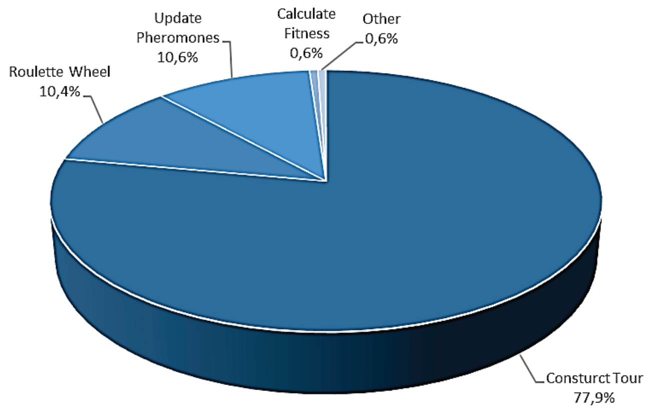

Figure 2 Time analysis of Ant System [author]

For time analysis, the AS algorithm used in this paper was applied to the *kroA100* benchmark problem and analyzed using MATLAB's *Run and Time* option. The colony population was 100 ants, and the stopping criteria were 500 iterations. Time analysis, illustrated in Fig. 2, shows that most of the computational time was consumed by the *Construct Tour* function. Considering that the *Roulette Wheel* function is called from within the *Construct Tour* function it is clear that constructing solutions takes almost 90% of the total time needed to run the algorithm.

3.2 Methodology

As mentioned before, the efficiency of the algorithm refers to computational speed which is the execution time of an algorithm for a limited number of iterations. Effectiveness refers to solution quality which is measured by tour length, a shorter tour indicates higher quality.

To quantify the impact of the reduced number of ants on the efficiency and effectiveness of the Ant System, the algorithm was applied to classic TSP benchmark instances kroA100, kroA150, and kroA200 found in TSPLIB [30]. These benchmark problems were chosen because of their round number of nodes in order to minimize the need for rounding adjustments in ant numbers during experimentation. Recommended parameters from [5] were used.

Parameters are as follows: $\alpha = 1$, $\beta = 5$, $\rho = 0.5$, and initial pheromone levels before the first iteration, are calculated according to the equation $\tau_0 = m/C^{nn}$. Where the number of ants is denoted with m and C^{nn} stands for the tour length obtained through the Nearest Neighbor algorithm. The recommended number of ants (m) is the same as the number of nodes (n) in the problem ($m = n$) for general good performance. [5]

A total of 12 tests were conducted for each benchmark problem (kroA100, kroA150, and kroA200).

First, results for each benchmark problem were obtained using 100% of the recommended number of ants. For kroA100, 100 ants were used. For kroA150, 150 ants were used. For kroA200, 200 ants were used. For future reference, these first tests will be called *Test 1*, as other tests will be compared to it.

Subsequent tests involved a gradual reduction of ants by 10%. Test 2 used 90% of the recommended ant population, Test 3 used 80%, Test 4 used 70%, and so on, down to Test 10, which employed 10% of the recommended number of ants. The final two tests were carried out with 6% and 2% of the recommended ant count.

The tour lengths of subsequent tests were compared to the optimal solution and the tour lengths obtained in *Test 1* for each benchmark problem. The execution time of subsequent tests was compared to the execution time of *Test 1*.

Experiments were performed using MATLAB R2021a on the Windows 10 64-bit operating system. The computer configuration was Intel(R) Core(TM) i5-10210U CPU 2.10 GHz, installed RAM 8 GB. The algorithm used is accessible through the GitHub repository [31]. Results are averaged for 10 independent runs due to the probabilistic nature of ACO and to reduce the influence of background processes.

3.3 Experiment results

Test results for each benchmark problem are presented in Tabs. 1, 2, and 3. The first column (*No. Ants*) shows the percentage of ants used in the test - 100% corresponds to the number of nodes in the problem ($m = n$). For the kroA100 benchmark, *Test 1* (first row) employed 100 ants, followed by 90 ants in the second test (second row), and so forth.

Table 1 Test results for benchmark instance kroA100

No. Ants	Solution quality	Deviation from the optimum	Deviation from the Test 1	Execution time (s)	Reduction of execution time
100%	22756	6.9%	-	135.2	-
90%	22800	7.1%	0.2%	120.5	89.1%
80%	22955	7.9%	0.9%	105.4	78.0%
70%	23007	8.1%	1.1%	92.65	68.5%
60%	23051	8.3%	1.3%	79.4	58.7%
50%	23088	8.5%	1.5%	66.7	49.3%
40%	23193	9.0%	1.9%	53.6	39.6%
30%	23180	8.9%	1.9%	40.6	30.0%
20%	23251	9.3%	2.2%	27.55	20.4%
10%	23343	9.7%	2.6%	14.5	10.7%
6%	23510	10.5%	3.3%	9.2	6.8%
2%	24170	13.6%	6.2%	3.4	2.5%

The second column (*Solution quality*) shows the lengths of an average solution obtained in 10 runs of the algorithm. In the third column (*Deviation from the optimum*) obtained solutions are compared to the optimal solution. The percentage of deviation from the optimal solution is given. It's calculated using the equation ((*average solution length* / *optimal solution length*) $*$ 100%) – 100%). This value represents the percentage by which the average tour length surpasses the optimal tour length. For instance, when using 100 ants in the kroA100 benchmark test, the resulting tour was 6.9% longer than the optimal solution.

In the fourth column (*Deviation from the Test 1*) results of subsequent tests (tests 2 to 12) are compared to the *Test 1*. It presents a comparison of tour lengths obtained with a reduced ant count against those obtained using the recommended number of ants (100% ants). The values show how much longer or shorter are the tour lengths compared to *Test 1*. For example, the average solution for benchmark problem kroA100 using 90 ants is 0.2% longer than the average solution using 100 ants. A negative percentage means that the tour was shorter than the average solution in *Test 1*.

The fifth column (*Execution time* [s]) shows the average execution time for one run of the algorithm. Execution time was measured using *tic* and *toc* functions in MATLAB, and the results are expressed in seconds.

The sixth column (*Reduction of execution time*) shows the difference in execution time between tests 2 to 12 with the reduced number of ants and *Test 1*. For example, for benchmark problem kroA100 using 90 ants produced a solution in 120.5 seconds, which is 89,1% of the time needed to produce a solution using 100 ants (135,2 seconds) in *Test 1*.

Table 2 Test results for benchmark instance kroA150

No. Ants	Solution quality	Deviation from the optimum	Deviation from the Test 1	Execution time (s)	Reduction of execution time
100%	28733	8.3%	-	375.1	-
90%	28620	7.9%	-0.4%	337.5	90.0%
80%	28552	7.6%	-0.6%	299.6	79.9%
70%	28616	7.9%	-0.4%	264.6	70.5%
60%	28742	8.4%	0.0%	225.65	60.2%
50%	28727	8.3%	0.0%	189.9	50.6%
40%	28831	8.7%	0.3%	152.45	40.6%
30%	28811	8.6%	0.3%	115	30.7%
20%	28887	8.9%	0.5%	76.7	20.4%
10%	29094	9.7%	1.3%	39.05	10.4%
6%	29379	10.8%	2.2%	23.9	6.4%
2%	29973	13.0%	4.3%	9.8	2.6%

Table 3 Test results for benchmark instance kroA200

No. Ants	Solution quality	Deviation from the optimum	Deviation from the Test 1	Execution time (s)	Reduction of execution time
100%	32129	9.4%	-	811.2	-
90%	32172	9.5%	0.1%	736.3	90.8%
80%	31878	8.5%	-0.8%	643.3	79.3%
70%	32071	9.2%	-0.2%	563.4	69.5%
60%	32189	9.6%	0.2%	482.5	59.5%
50%	32206	9.7%	0.2%	401.6	49.5%
40%	32326	10.1%	0.6%	322	39.7%
30%	32403	10.3%	0.9%	243.6	30.0%
20%	32347	10.1%	0.7%	163.3	20.1%
10%	32983	12.3%	2.7%	82.4	10.2%
6%	33452	13.9%	4.1%	49.7	6.1%
2%	34170	16.4%	6.4%	17.4	2.1%

Data from the tables is visually represented in Figs. 3, 4, and 5 for benchmark instances kroA100, kroA150, and kroA200, respectively.

It is clearly shown that computational time increases linearly with the number of ants. Computational time is directly related to the number of ants. For example, using 50% of ants will complete the calculation in half the time compared to *Test 1*. 50 ants in kroA100 instance will complete the calculation in 66,7 seconds, which is roughly 50% of the total time needed to produce a solution using 100 ants (235,2 seconds) in the same problem.

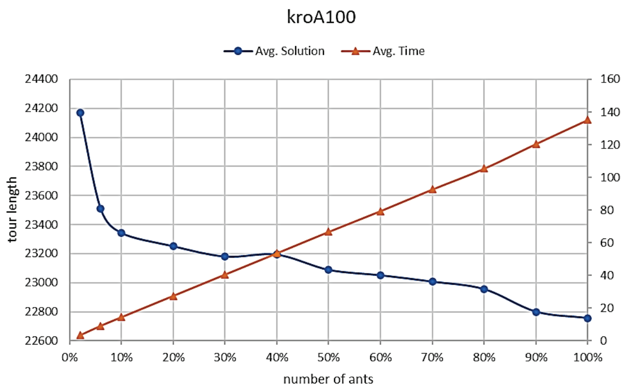

Figure 3 Tour length and execution time for kroA100

For benchmark problem kroA100, using the recommended number of ants (*Test 1*) produced a solution 6.9% longer than the optimum. Using the recommended number of ants, the kroA150 solution was 8.3% longer, and the kroA200 solution was 9.4% longer than the optimal solution. Using 100% of the ant population produces increasingly worse solutions as the problem size grows.

The smallest ant count in these tests was 2% of the recommended population size, which expectedly reduced computational time to approximately 2% of execution time compared to *Test 1*. The solution obtained with 2% of the recommended population size was 13.6% longer than the optimal solution for the kroA100, 13% longer for the kroA150, and 16.4% longer for the kroA200. Compared with results obtained with *Test 1*, using 2% of ants produced tours 6.2% longer for kroA100, 4.3% longer for kroA150, and 6,4% longer for kroA200. Using a very small population generates noticeably worse solutions compared to *Test 1*.

In all three cases, the solution quality drastically improves when the ant population is increased from 2% to 20% of the recommended population size. For example, the solution for kroA100 using 2% of the recommended population deviates 6,2% from the solution obtained in *Test 1*. While using 20% of the recommended population, the solution deviates 2,2% from the solution obtained in *Test 1*. Solutions for kroA200 using 2% ants deviate 6,4% from the solution obtained in *Test 1*, while using 20% of ants solution deviate 0,7% from the solution obtained in *Test 1*.

Increasing the population up to 50% still increases solution quality but at a slower rate. For example, using 50% of ants in kroA100 problem, the solution is 1,5% longer compared to the solution obtained in *Test 1*.

For kroA100 problem, increasing population size beyond 50% gradually increased solution quality, and using 100% ants produced the best results.

For kroA150 and kroA200 problems, increasing the population beyond 50% didn't show a significant increase in solution quality. It is interesting that using 70 or 80% of the population in both cases produced slightly better results compared to *Test 1* (using 100% of the ant population), but this can be credited to the probabilistic nature of ACO.

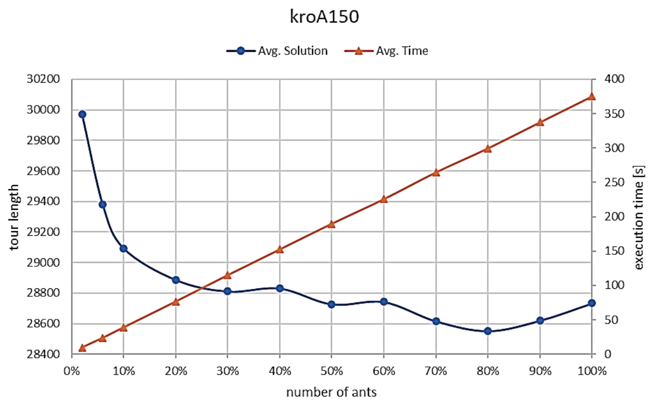

Figure 4 Tour length and execution time for kroA150

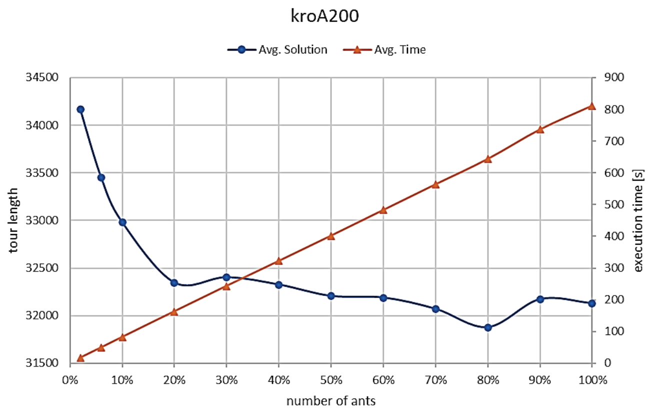

Figure 5 Tour length and execution time for kroA200

4 DISCUSSION

As indicated in the literature survey, some papers (like [18], [22], and [25]) try to determine an optimal number of ants as a constant number and conclude that the optimal number of ants is specific to the case at hand and can't be defined generally. In this paper, we investigate the population size of the Ant System (AS) in relation to the size of the problem to find a good compromise between computational speed and solution quality. Similar research was conducted for other ACO algorithms, specifically Ant Colony System (ACS) in [16] and [29] in which authors initialize population size relevant to the number of nodes in the problem.

Decreasing the ant population decreases solution quality and computational time. The decrease in computational time is much more prominent than the decrease in solution quality. The computational time scales linearly with the proportion of the recommended ant number, equating to the corresponding fraction of the ant population. This means that by reducing population size computational time can be reduced predictably.

Tests show that reducing the number of ants has a stronger effect on the solution quality of smaller problem sizes (100 nodes) than it has on larger problem sizes (200 nodes), this could be because of the smaller absolute number of ants.

Solution quality increases rapidly with the increase of population size from 2% to 20% of the recommended population size, after which it gradually increases up to 50% of the recommended population size. Using more than 50% of the recommended number of ants doesn't increase solution quality noticeably.

For problems with 150 and 200 nodes using 80% of ants produced better results than using 100% of ants. That effect could be because ACO is a probabilistic method. The influence of randomness, or "luck", could be decreased with more runs of the algorithm to average out the results. But this effect could also indicate that using more ants over a certain threshold doesn't necessarily improve solution quality.

For problem sizes similar to kroA150 and kroA200, two values could be considered "sweet spots" for a good efficiency/effectiveness ratio, using 50% or 20% of ants. With 50% of the population size computational time could be halved with solutions quality remaining approximately the same as in *Test 1*. Using 20% of ants produces 0.5% and 0.7% longer solutions for kroA150 and kroA200 benchmark instances compared to *Test 1* while computation is five times faster.

For benchmark instance kroA100, and problems of similar size, "sweet spot" is an even lower number of ants, concretely 10% of the population. In that case, computational time would be 10 times faster. An increase in computational speed comes at the price of a noticeable 2.6% longer solution. The second "sweet spot" is using 70 or 80% of the population which produces roughly 1% longer solution compared to *Test 1*.

It should be noted that using improved variants of the ACO algorithm would produce better overall results in a shorter time. For example, in a Rank-based Ant System (RAS) only a limited number of ants deposit pheromones which increases computational speed compared to AS. In MAX-MIN Ant System (MMAS) only one ant deposits pheromones in each iteration, which reduces computational time even more. It is proven that RAS and MMAS produce better solutions than basic AS. [5]

For future research, the influence of reduced population size on the improved ACO algorithms such as the MAX-MIN Ant System, the Rank-Based Ant System, and many new variants are often proposed in the literature. Also, bigger problems should be tackled as benchmark problems in TSPLIB can go up to tens of thousands of nodes.

5 CONCLUSIONS

It has been a while since Ant Colony Optimization (ACO) established its place as a reliable and efficient algorithm for solving many problems in production and logistics, but it still attracts the attention of many researchers who are determined to increase its efficiency and effectiveness. As an increased number of problems in the real world are dynamic in nature it is paramount to increase the computational speed of ACO to be useful in practice.

One method of increasing ACO efficiency is reducing the number of ants which considerably increases computational speed but slightly lowers solution quality. As the optimal number of ants depends on a specific case, we focused on the reduction of population size relevant to the number of nodes in the problem.

For problems of approximate size 150 - 200 nodes, the number of ants can be reduced by 50%, effectively halving computational time while solution quality remains roughly the same. For the same problem sizes, by reducing the number of ants to 20%, computational speed can be 5 times faster for a 0,5% to 0,7% reduction of solution quality. The reduction of ant population has a much stronger, positive, effect on computational speed than it has a negative effect on solution quality.

The limitation of this research lies in the probabilistic nature of ACO. In each test, the algorithm was run 10 times to reduce the influence of "luck" in generating solutions. More runs are always better, but also more time-consuming. Theoretically, the infinite number of runs would eliminate the effect of lucky guesses, but that is not possible to do because of obvious reasons.

In future research, the influence of reduced population size on bigger problems and improved ACO variants should be investigated.

6 REFERENCES

[1] Laporte, G. (1992). Traveling Salesman Problem: An overview of exact and approximate algorithms. *Eur. J. Oper. Res., 59*(2), 231-247.

[2] Kumar, S. N. & Panneerselvam, R. (2012). A Survey on the Vehicle Routing Problem and Its Variants. *Intell. Inf. Manag., 4*(3), 66-74. https://doi.org/10.4236/iim.2012.43010

[3] 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. *IEEE Comput. Intell. Mag., 15*(1), 52-63. https://doi.org/10.1109/MCI.2019.2954644

[4] Gutjahr, W. J. (2000). A Graph-based Ant System and its convergence. *Futur. Gener. Comput. Syst., 16*(8), 873-888. https://doi.org/10.1016/S0167-739X(00)00044-3

[5] Dorigo, M. & Stützle, T. (2004). *Ant colony optimization*. Cambridge, Massachusetts: The MIT Press.

[6] Dorigo, M. (1992). Optimization, Learning and Natural Algorithms. *PhD thesis, Politec. di Milano, Italy*.

[7] 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

[8] Bullnheimer, B., Hartl, R. F. & Straub, C. (1997). A New Rank Based Version of the Ant System. *A Comput. study*, 1, 1-16.

[9] Stuetzle, T. & Hoos, H. (1997). MAX-MIN Ant System and local search for the traveling salesman problem. *Proc. IEEE Conf. Evol. Comput. ICEC*, 309-314. https://doi.org/10.1109/icec.1997.592327

[10] Gambardella, L. M. & Dorigo, M. (1996). Solving symmetric and asymmetric TSPs by ant colonies. *Proc. IEEE Conf. Evol. Comput.*, 622-627. https://doi.org/10.1109/icec.1996.542672

[11] Olivari, L. (2023). Comparison of improved ACO algorithms

for tool path optimization in multi-hole drilling. *Proc. ICIL*, 75-78.
[12] Middendorf, M., Reischle, F. & Schmeck, H. (2002). Multi colony ant algorithms. *J. Heuristics, 8*(3), 305-320. https://doi.org/10.1023/A:1015057701750
[13] Stutzle, T. (1998). Parallelization strategies for ant colony optimization. *Int. Conf. Parallel Probl. Solving from Nature. Heidelb. Springer Berlin*, 722-731. https://doi.org/10.1007/bfb0056914
[14] Olivari, L. & Olivari, L. (2022). Influence of Programming Language on the Execution Time of Ant Colony Optimization Algorithm. *Tehnički Glasnik, 16*(2), 231-239. https://doi.org/10.31803/tg-20220407095736
[15] 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
[16] Liu, F., Zhong, J., Liu, C., Gao, C. & Li, X. (2018). A novel strategy of initializing the population size for ant colony optimization algorithms in TSP. *ICNC-FSKD 2017 - The 13th Int. Conf. Nat. Comput. Fuzzy Syst. Knowl. Discov.*, 249-253. https://doi.org/10.1109/FSKD.2017.8393166
[17] Siemiński, A. (2016). Using hyper populated ant colonies for solving the TSP. *Vietnam J. Comput. Sci., 3*(2), 103-117. https://doi.org/10.1007/s40595-016-0059-z
[18] Sivagaminathan, R. K. & Ramakrishnan, S. (2007). A hybrid approach for feature subset selection using neural networks and ant colony optimization. *Expert Syst. Appl., 33*(1), 49-60. https://doi.org/10.1016/j.eswa.2006.04.010
[19] Chang, Y. C. (2017). Using k-means clustering to improve the efficiency of ant colony optimization for the traveling salesman problem. *IEEE Int. Conf. Syst. Man, Cybern. SMC 2017*, 379-384. https://doi.org/10.1109/SMC.2017.8122633
[20] Jangra, R. & Kait, R. (2017). Analysis and comparison among Ant System; Ant Colony System and Max-Min Ant System with different parameters setting. *The 3rd IEEE Int. Conf.*, 1-4. https://doi.org/10.1109/CIACT.2017.7977376
[21] Chitty, D. M. (2018). Applying ACO to Large Scale TSP Instances. *Advances in Intelligent Systems and Computing, 650*, 104-118.
[22] Abdelbar, A. M. & Salama, K. M. (2018). The effect of the number of ants parameter in the ACOR algorithm: A run-time profiling study. *IEEE Symp. Ser. Comput. Intell. SSCI 2017 - Proc.*, 1-8. https://doi.org/10.1109/SSCI.2017.8280799
[23] Fidanova, S., Marinov, P. & Paparzycki, M. (2014). Multi-objective ACO algorithm for WSN layout: performance according to number of ants. *Int. J. Metaheuristics*, 3(2), p. 149. https://doi.org/10.1504/ijmheur.2014.063145
[24] Fidanova, S. & Marinov, P. (2013). Number of Ants Versus Number of Iterations on Ant Colony Optimization Algorithm for Wireless Sensor Layout. *Proc. Work. ICT New Mater. Nanotechnologies, Bankya*, 90-93.
[25] Aydin, Z. (2018). Determination the number of ants used in ACO algorithm via grillage optimization. *Uludağ Univ. J. Fac. Eng.*, 22(3), 251-262. https://doi.org/10.17482/uumfd.298586
[26] Neroni, M. (2021). Ant Colony Optimization with Warm-Up. *Algorithms, 14*(10), p. 295. https://doi.org/10.3390/a14100295
[27] Yilmaz, A. H. & Aydin, Z. (2022). Examination of parameters used in ant colony algorithm over truss optimization. *Balıkesir Üniversitesi Fen Bilim. Enstitüsü Derg., 24*(1), 263-280. https://doi.org/10.25092/baunfbed.955408
[28] Liu, H., Lee, A., Lee, W. & Guo, P. (2023). DAACO: adaptive dynamic quantity of ant ACO algorithm to solve the traveling salesman problem. *Complex Intell. Syst.*. https://doi.org/10.1007/s40747-022-00949-6
[29] Alobaedy, M. M., Khalaf, A. A. & Muraina, I. D. (2017). Analysis of the number of ants in ant colony system algorithm. *The 5th Int. Conf. Inf. Commun. Technol. ICoIC7 2017*, 3-7. https://doi.org/10.1109/ICoICT.2017.8074653
[30] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/ (accessed: 16.1.2023)
[31] https://github.com/l-olivari/AS2023 (accessed: 23.8.2023)

Author's contacts:

Luka Olivari, mag. ing. mech., senior lecturer
Šibenik University of Applied Sciences,
Trg Andrije Hebranga 11, 22 000 Šibenik, Croatia
(022) 311-060, lolivari@vus.hr