Potential of Using the Ant Colony Optimization Algorithm for Optimal Network Path Selection

Oleksandra Yaroshevska and Veronika Kirova
Anhalt University of Applied Sciences, Bernburger Str. 57, 06366 Köthen, Germany
yaroshevska.ob@gmail.com, veronika.kirova@hs-anhalt.de

Keywords: Network Optimal Path, Ant Colony Optimization (ACO) Algorithm, Dijkstra's Algorithm, Lowest Common Ancestor (LCA), Pheromone, RTT, Python.

Abstract: The article considers the possibilities of using the Ant Colony Optimization algorithm to find the shortest path in the network based on the selected criteria. Its performance is compared to Dijkstra's algorithm and LCA algorithm, which is widely used in different network routing protocols. An overview of the ACO algorithm, including its two primary components, the "ant" and "pheromone," is provided, highlighting its efficiency for the optimal network path selection. Detailed schemes, parameters and formulas of the ACO algorithm implementation in terms of networking are shown. A comparative analysis of the performance and execution time of the ACO and two compared algorithms for the optimal network path based on Round Trip Time criteria in networks of varying scale, ranging from small to highly branched networks with thousands of nodes, is discussed. Finally, the results are analysed, and the potential for ACO to serve as a complementary algorithm to Dijkstra's and LCA in future network applications is explored.

1 INTRODUCTION

1.1 Optimal Network Path Selection Problem

As modern technologies continue to evolve and the demand for network infrastructure grows, the challenge of building extensive local and global networks becomes increasingly urgent. A key factor in maintaining network efficiency is identifying optimal routes between nodes.

Using optimal routes between nodes in a network is critical for several reasons (Figure 1).

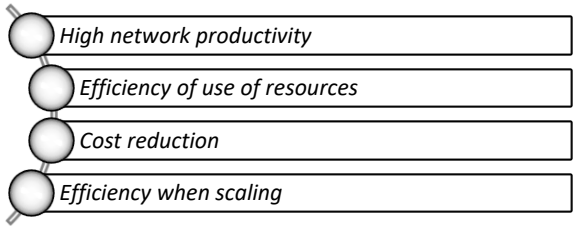

Figure 1: Reasons for the importance of route optimization in the network.

Each reason is explained detailed below [1]:

1) With optimal routing, data takes the shortest or most efficient path between nodes, reducing data transmitter delays. This facilitates faster communication and better user experience.
2) Optimal routing helps balancing the traffic by preventing certain paths from becoming congested while others are not used enough. Thus, the network works more efficiently without overloading the infrastructure.
3) There is a reduction in the need for additional infrastructure, energy consumption and maintenance that may arise from inefficient routing.
4) Smoother scaling is provided so that new nodes or connections do not decrease network efficiency.

Channel reservation is also an additional option for route optimization. This means that in case if one route fails, data can be quickly rerouted through alternate paths, increasing network reliability.

1.2 Methods to Solve the Problem and the Main Research Goal Description

Among the possible solutions to the problem of optimizing paths in a network, two groups of methods are distinguished: traditional and metaheuristic [2]. Currently, it is common to use traditional methods in networks. One of the most widespread methods for finding the best path is Dijkstra's algorithm [3]. Its advantage consists in the existence of a stable scheme using a path matrix, which helps to find the shortest or the most efficient path depending on the specific needs of the network. However, such a scheme is quite rigid, which is a drawback and may lead to difficulties in cases where more precise adjustment of the algorithm's parameters is required.

In addition to traditional approaches, modern algorithmic techniques such as the Lowest Common Ancestor (LCA) are increasingly being considered for network path optimization, especially in hierarchical or tree-structured networks. LCA algorithms are particularly effective when the network can be represented as a rooted tree and rapid queries between node pairs are required. By preprocessing the network structure, LCA allows for efficient determination of the closest shared ancestor of two nodes, which can significantly improve routing decisions in applications such as multicast routing, hierarchical clustering, and certain types of peer-to-peer networks. The use of LCA methods enhances the adaptability of routing protocols and supports real-time decision making with low computational overhead.

The other approach to solving the problem of finding the optimal path is metaheuristic methods. They are based on heuristic principles and use search in a large solution space to find approximate but high-quality answers, often operating with the processes that are inherent in nature. One of these methods is the Ant Colony Optimization (ACO) algorithm [4-6]. The basic concept of this algorithm was created based on observations of how ants find the shortest path to a food source.

The decision-making process about the optimal path includes elements of probability theory and is based on the concept of pheromones, which ants leave in nature depending on the quality of the path they have traversed. In a real network, the quality of a route could be considered, for example, as the level of delay between nodes, bandwidth or other important parameters. This article examines the potential use of ACO compared to Dijkstra's and LCA algorithms and provides a description of the problems for which ACO can be used.

2 EXPERIMENTAL TOPOLOGY

2.1 Dijkstra’s Algorithm Realization

To determine the efficiency of ACO, Dijkstra’s algorithm was implemented using various methods of data storage and processing to achieve the most valid results. Python was chosen as the programming language due to its convenience and the wide range of tools available for data analysis.

The first method involves using a standard list with element sorting to determine the smallest Round Trip Time (RTT) value, which is used in this experiment as the main parameter for path efficiency [7]. RTT is the time required to send a data packet from the sender to the receiver and return a response back. This parameter is important for evaluating network latency, as it reflects the speed of data transmission between nodes. A smaller RTT indicates faster transmission and better network performance.

The second method involves using a binary tree, where the smallest RTT value is the root of the tree. Further comparison based on processing speed determines the best method for data handling to be used as a standard when evaluating the efficiency of ACO (see section 2.3.1).

The implementation of both variations of Dijkstra’s algorithm was based on existing algorithmic frameworks [3]. This custom implementation is important to ensure that the input data in the comparison of Dijkstra’s and ACO algorithms are identical and can be adjusted during the experiment, as this is one of the factors ensuring the validity of the experiment.

The main stages of Dijkstra’s algorithm work are described below:

1) A priority queue and RTT matrices from the key node to each of the other nodes in the network are created.
   The priority queue is a list or binary tree according to one of the two methods described above. It includes "node - RTT" value pairs to identify the best paths. In this case, nodes are considered as vertices of an undirected graph, and the paths with RTT values are considered as the edges of the graph [8].
2) Each neighbor of the current node is visited.
   In the process of moving to neighboring nodes, each one is added to the priority queue along with its RTT. It is important to use nodes that have not been visited before to prevent cycles and incorrect algorithm behavior.
3) The previous matrix data is compared with the current paths to each of the neighboring nodes.

If the current path is better, the matrix is updated with the new data.

4) The path with the smallest RTT is selected. The node to which this path leads becomes the next current node.
   In the case of a list, further data sorting occurs. In the case of a binary tree, the best option is found at the root of the tree.

The algorithm repeats until every vertex of the graph is visited. As a result, the final matrix contains the best paths from the key node to every other node in the network.

In this implementation, the dictionaries are used as an analogy to matrices to improve the algorithm's productivity. Using dictionaries in Python is efficient due to their speed in handling elements (search, addition, deletion, etc.). Dictionaries are based on hash tables, which allow data to be accessed by key in O(1) time in most cases. They are also convenient for storing large amounts of data with "key-value" pairs, ensuring high performance when analyzing extensive networks with many nodes.

Two main dictionaries were created:

- Shortest path dictionary.
- Last neighbor dictionary.

The first dictionary contains information about the shortest path to each node. The second dictionary stores information about the second last node that must be visited before reaching the destination. For example, the second last node for the fifth node it could be the third, for the tenth node – the twelfth, and so on. This creates a chain effect when traversing from the starting (key) node to others.

In this way not only can the RTT of the shortest path to each node be determined, but the entire sequence of nodes along the path can be traced. This ensures accurate comparison with the ACO algorithm, where the sequence may vary depending on the specified parameters. Both dictionaries are output by the algorithm upon completion, allowing each to be used as needed.

2.2 LCA algorithm realization

To complement the comparison with the ACO algorithm, the LCA (Lowest Common Ancestor) algorithm was implemented to evaluate scenarios where hierarchical relationships between nodes are critical, such as in tree-based or partially hierarchical network topologies. Python was selected for the implementation due to its rich set of tools and the ability to prototype algorithmic logic efficiently.

In this realization, the network is modeled as a rooted tree, where each node may have multiple children, and all connections are unidirectional from parent to child. This reflects situations where pathfinding is restricted by hierarchical constraints or parent-child dependencies. The goal of the LCA algorithm is to find the common ancestor node that is lowest (i.e., deepest) in the tree for any two given nodes [9]. This approach is particularly useful in applications involving tree traversal, organizational hierarchies, or data clustering [10].

The LCA implementation is based on the binary lifting technique, which allows for efficient querying of the lowest common ancestor in logarithmic time. The preprocessing phase is designed to prepare necessary lookup tables that speed up each individual LCA query, optimizing the performance for networks with frequent ancestor-related queries.

The main stages of the algorithm are described below:

1) Tree Construction and Initialization. Each node is represented by an instance of a TreeNode class containing its value and a list of children. During initialization of the LCA class, the root node and the total number of nodes in the tree (n) are passed as parameters. Arrays are created to store the depth and parent of each node, as well as a binary lifting table (up) which enables ancestor lookup at various powers of two.
2) Depth-First Search (DFS) Traversal for Preprocessing. A depth-first traversal of the tree is performed starting from the root node. During traversal, each node's depth and parent are recorded. The up table is filled such that up[i][j] stores the 2^j-th ancestor of node i. This preprocessing allows any node to be moved upward by any power-of-two number of levels in constant time, which is essential for efficient LCA computation.
3) LCA Query Execution. To determine the lowest common ancestor of two nodes u and v, the algorithm first equalizes their depths by moving the deeper node upward. Then, starting from the highest level of the binary lifting table, both nodes are lifted together until they converge. The final result is the parent of the converging point, which represents the lowest common ancestor.
4) Performance and Application. This realization of the LCA algorithm ensures an O(n log n) preprocessing time and O(log n) query time, making it highly efficient for repeated ancestor queries in large hierarchical trees. While not directly focused on shortest paths or RTT as in

Dijkstra's or ACO, the LCA algorithm provides a foundational utility in hierarchical routing or clustering scenarios, offering a complementary perspective in network analysis.

The LCA implementation outputs the ancestor node shared by any two nodes in the shortest hierarchical path, enabling the tracing of common routes and structural relationships within tree-based network representations. This makes it a valuable comparative model alongside Dijkstra's and ACO algorithms in evaluating different types of network structures and their respective search efficiencies.

2.3 Ant Colony Optimization Algorithm Realization

ACO is based on two main terms: "ant" and "pheromone" [4]. The algorithm consists of several iterations, after each of which pheromone levels are updated. Pheromone is represented by a numerical value $0 < ph < 1$. The higher this value, the greater the probability of choosing a particular path. One iteration consists of several steps, each of which is a complete path from the start node to the final node. Such a step is analogous to an ant traveling the distance from its home to a food source in nature. For simplicity, further in the article, these steps will be referred to as ants.

The flowchart of the algorithm is shown in Figure 2.

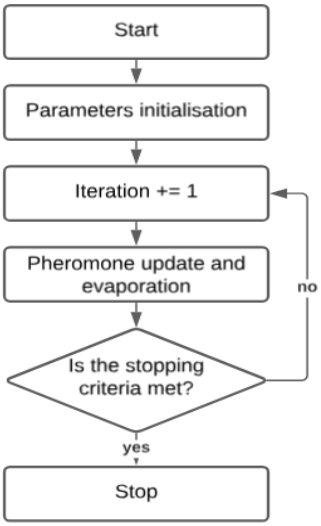

Figure 2: Base ACO block-scheme.

The main parameters of ACO are as follows:

- number of iterations;
- number of ants;
- initial pheromone level;
- pheromone decay rate;
- parameters of importance of the pheromone and RTT when calculating the probability of path selection;
- coefficients required for the mathematical calculations of probability.

After each iteration, the pheromone levels are updated, which influences subsequent iterations. Thus, the pheromone amount on paths that are traveled more frequently increases, raising the likelihood of those paths being used again while reducing the likelihood of using less efficient paths.

The stopping criterion for the algorithm in its standard implementation is the achievement of the set number of iterations. However, additional criteria may be added to increase efficiency.

The flowchart of the algorithm within each iteration is shown in Figure 3.

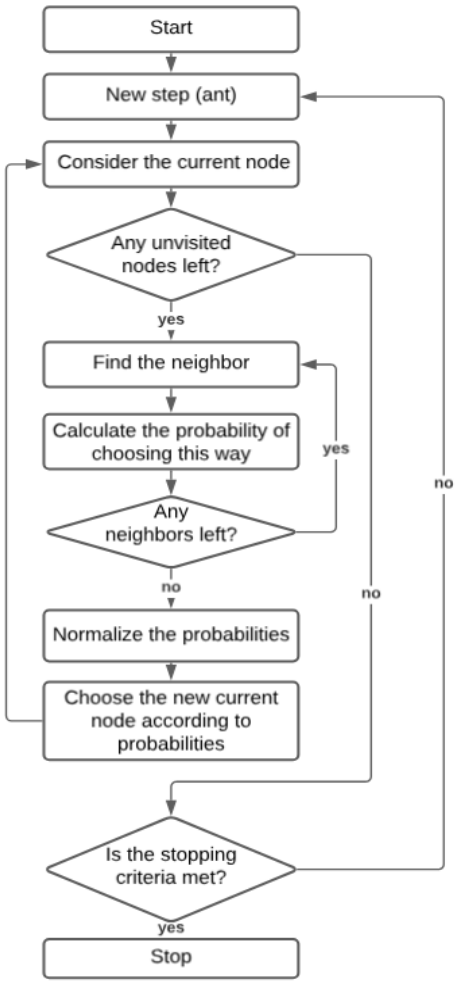

Figure 3: ACO block-scheme inside the iteration.

As shown in Figure 3, each iteration consists of a certain number of steps (ants). Each ant traverses the complete path from the starting node to the final node, after which it stores information about the path as a sequence of nodes.

After the iteration is complete, the pheromone levels on all the paths that were traversed are updated. The number of iterations and ants is determined empirically, depending on the network's scalability.

It is important to note that the probability of any path should not reach zero as long as it remains accessible, since the most optimal path may include sections with suboptimal RTT values at certain stages. The overall value across the entire path from the start to the final node will be the most optimal. Therefore, it's important to maintain a pheromone level that allows for a small probability of selecting alternative paths to those that were previously chosen.

The elements that make up the path of one ant are described in more detail below:

1) Finding the neighbors of the current node. At the first stage, the starting node is considered the current one. Subsequently, the current node will be the one the ant moves to on the path toward the final node.
2) Calculating the probability of transitioning to each neighboring node.
   This calculation is based on two main formulas.

$$P(i) = ph^{\alpha} * \frac{k}{RTT^{\beta}}. \quad (1)$$

   Formula (1) reflects the transition strength from the current node to another specific node. This strength, also referred to as the "desire" to transition, depends on the pheromone level and the inverse value of RTT. Thus, the lower the RTT value, the greater the transition strength.

   The coefficient k is selected based on the RTT values for the specific network and is determined empirically. Coefficients α and β are used to increase the influence of pheromone or RTT. In this experiment, both are set to one, which means that the influence of pheromone and RTT is equally weighted. The initial pheromone value is set to 0.3 and is either increased or decreased depending on which paths are traversed.

$$P(norm) = \frac{P(i)}{\sum_{i=1}^{n} P(i)} \quad (2)$$

   Formula (2) is required for normalizing the transition strength and calculating the actual probability within the range 0 < P < 1. Normalization occurs after evaluating the transition strengths for all neighboring nodes, as it requires the total sum of these strengths.
3) Selecting the next node.A scale from 0 to 1 is used for node selection. A randomly chosen number falls within a range that corresponds to a specific neighboring node based on the transition probability previously calculated. For example, with a 50% probability, half of the scale is covered. The transition to the next node then occurs.

   The condition for exiting the cycle is reaching the final node. The cycle is repeated according to the number of ants, which is one of the parameters of the algorithm.

At the end of the algorithm's execution, information is provided regarding the best path found and its total RTT value.

2.3 Results of Comparison by the Execution time Parameter

2.3.1 Dijkstra's Algorithm Realizations Comparison

To study the performance of the algorithms, an emulation of RTT data was carried out, obtained from networks of various scalability. The data was automatically generated according to the specified number of nodes and saved in a document for convenient access and the ability to rerun the experiment with different algorithms.

For this study, the number of connections between network nodes was set to 70% of all possible connections. An essential parameter is maintaining connectivity between all nodes, meaning that each node must have at least one connection to every other node.

The comparison results based on execution time performance are shown in Table 1.

Table 1: Execution time comparison of the Dijkstra's algorithm realizations.

Network branching (nodes / connections)		Dijkstra (binary tree)	Dijkstra (list)
10	32	0,00s	0,00s
500	87325	0,18s	0,32s
1000	349650	0,69s	1,41s
1500	786975	1,76s	3,76s
2000	1399300	3,22s	6,88s
2500	2186625	5,07s	12,47s
3000	3148950	7,12s	18,93s

Diagram of execution time comparison of Dijkstra's algorithm realizations is shown in Figure 4.

In Figure 4, the vertical axis represents the execution time of the algorithm, while the horizontal axis indicates the number of nodes in the network. It is important to note that the algorithm's execution time is stable and fluctuates within the hundredths and thousandths of a second due to a consistent pathfinding system, which allows for a reduction in the number of trials during the research to ten for networks with identical parameters.

From the data obtained in the first phase of the study, the advantage of Dijkstra's algorithm based on binary trees can be observed. In networks with a small number of nodes, this advantage is minimal and amounts to less than 1 second; however, in more complex networks, the difference becomes pronounced, highlighting the importance of using more efficient data storage and processing methods compared to standard lists.

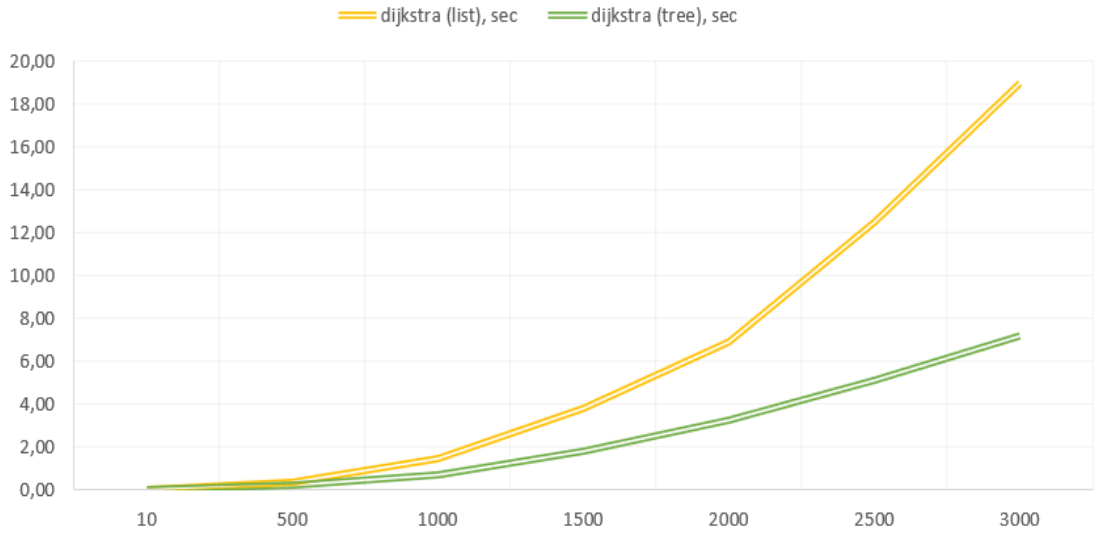

Figure 4: Diagram of execution time comparison of Dijkstra's algorithm realizations.

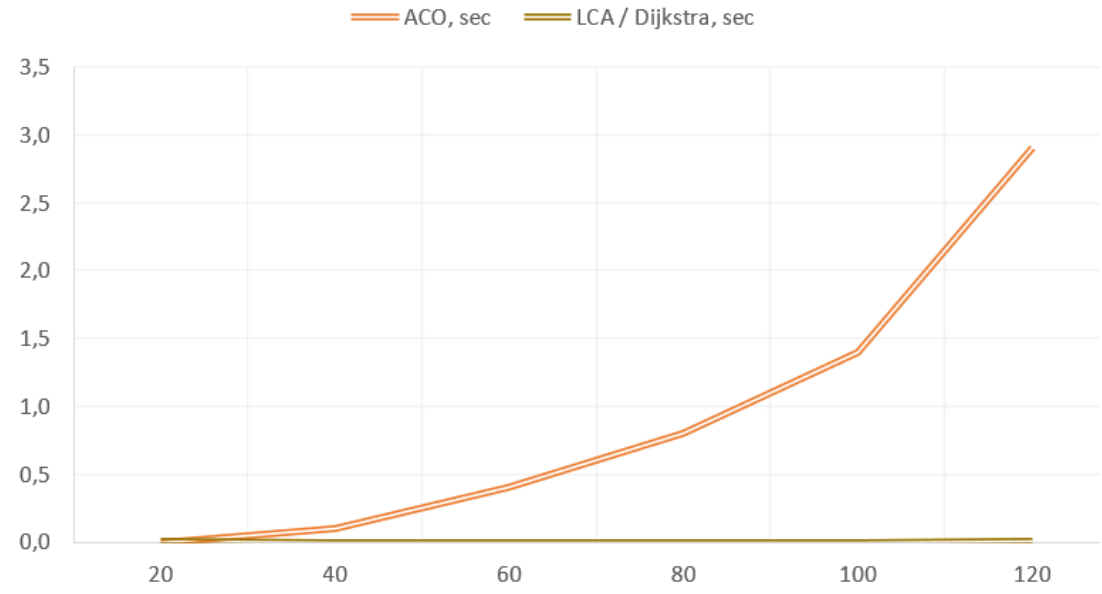

Figure 5: Diagram of execution time comparison of ACO, Dijkstra's and LCA algorithms.

2.3.2 ACO, Dijkstra's and LCA Algorithms Comparison

Based on the results of the first phase of the study, Dijkstra's algorithm was implemented using a binary tree. The next phase involves comparing ACO, Dijkstra's and LCA algorithms based on execution time to determine the potential for fully utilizing ACO in networks.

The comparison results are shown in Table 2 and the diagram of execution time comparison is shown in Figure 5.

The execution time analysis clearly demonstrates that ACO exhibits significantly lower performance in terms of speed when compared to the other two algorithms. Furthermore, it is important to note that the difference in execution times becomes more pronounced as the network size increases, as evidenced by the cases involving 20-node and 120-node topologies.

While Dijkstra's algorithm and LCA demonstrate comparable performance in small-scale networks, their efficiency diverges in larger topologies. For instance, in a simulated network consisting of 1,000 nodes, the average Round Trip Time (RTT) for queries using the LCA algorithm was measured at approximately 0.02 seconds, whereas Dijkstra's algorithm required an average of 0.71 seconds to complete equivalent path computations. This substantial difference highlights the superior scalability of the LCA approach in hierarchical network structures, where its logarithmic query complexity enables faster execution compared to the graph-based traversal required by Dijkstra's algorithm.

Table 2: Execution time comparison of ACO, Dijkstra's and LCA algorithms.

Network branching (nodes / connections)		Dijkstra (binary tree)	LCA	ACO
20	133	0,01s	0,01s	0,01s
40	546	0,01s	0,01s	0,1s
60	1239	0,01s	0,01s	0,4s
80	2212	0,01s	0,01s	0,8s
100	3465	0,01s	0,01s	1,4s
120	4998	0,01s	0,01s	2,9s

Additionally, it should be emphasized that using ACO in complex networks with more than 500 nodes is not effective at this stage of the algorithm's implementation and requires additional parameters to improve execution time.

3 PROPOSED SOLUTION

3.1 ACO Potential

Considering the execution time of Dijkstra's algorithm in networks with a large number of nodes (see section 2.3.1), as well as its efficient use in modern networks, it is proposed to utilize a different potential of ACO instead of merely competing on speed and subsequently replacing the basic path optimization algorithm.

ACO has certain application features that are absent in traditional algorithms due to their rigid working structure. These features involve dynamic parameter tuning and the ability to find alternative paths. This is achieved through the use of probabilistic elements and random selection. However, it is important to clarify that "random selection" in this context does not mean purely arbitrary choices. Rather, the decisions are guided by specific criteria such as Round Trip Time (RTT) values and the internal characteristics of the algorithm, which work together to improve overall efficiency and performance.

The search for alternative paths is a localized task within the network's operation, meaning that it can be performed without the need to gather new RTT measurements or to completely reconstruct all the paths within the network. This makes ACO particularly suitable for real-time optimizations where it is impractical to recalculate the entire network.

Such flexibility in path selection can prove useful in addressing various network management tasks, including the following:

1) Temporarily reducing the amount of traffic on a heavily congested path, allowing for better load balancing and preventing bottlenecks in data flow.
2) Identifying an alternative path for the transfer of large volumes of data, especially when the primary route is suboptimal for such specialized tasks.
3) Discovering an optimal path that bypasses a specific node, which may be temporarily unavailable or malfunctioning, thereby maintaining network connectivity and minimizing disruptions during such outages.

These capabilities make ACO a valuable tool for addressing network issues that require dynamic and responsive solutions.

3.2 Alternative Path Concept Realization

To efficiently store and manage data regarding the shortest paths discovered by the algorithm, several key data structures are utilized within the code:

- A temporary list of tuples "path – RTT value" that each ant passed in a given iteration;
- a temporary list for storing the best path during the current iteration;
- a dictionary for storing all the best paths found in each iteration, along with the corresponding RTT values, for easy output and future use.

The lists are called "temporary" because they are updated at the beginning of each new iteration, helping to collect new information about the paths traveled.

The flowchart for collecting and processing the information obtained during the algorithm's execution is shown in Figure 6.

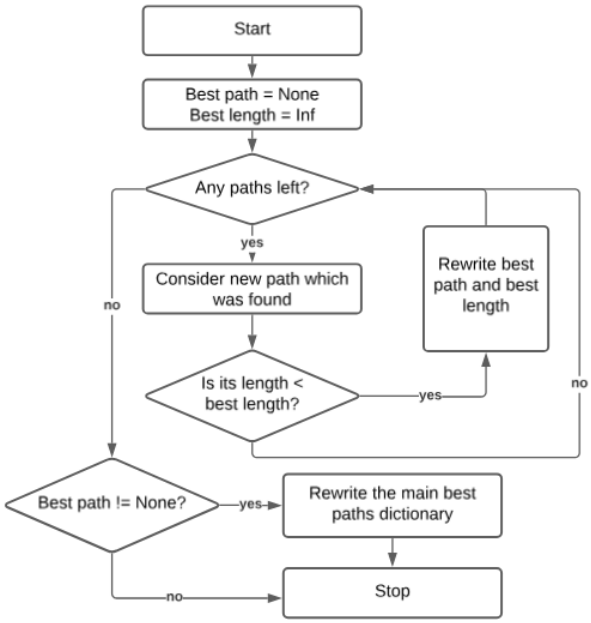

Figure 6: Block-scheme of alternative path concept realization.

This part of the algorithm begins operation after a list of all paths within the iteration has been formed. As the algorithm progresses, a dictionary is gradually populated with the best results, which is then passed on for further use at the end of the algorithm's execution.

After sorting process according to user preferences, any number of paths close to the optimal can be selected. This allows for local route adjustments where necessary within the network, without consuming excessive network resources.

4 CONCLUSIONS

This article explores the performance and potential of the Ant Colony Optimization (ACO) algorithm compared to Dijkstra's and LCA algorithms in network pathfinding tasks.

Dijkstra's algorithm was selected as a benchmark due to its status as a classical and widely accepted method for finding the shortest path in graphs, offering a deterministic and well-understood approach to pathfinding. In contrast, the LCA algorithm was chosen to represent more modern, structurally optimized methods, particularly suited for hierarchical or tree-based networks, where rapid ancestor queries can significantly reduce computation time.

The comparative analysis demonstrates that while ACO offers adaptability and robustness in dynamic or uncertain environments, it lags behind in terms of computational speed. Dijkstra's and LCA algorithms show comparable performance in small networks; however, LCA outperforms Dijkstra in large-scale networks due to its logarithmic complexity in query processing. These findings suggest that although ACO holds promise in flexible and heuristic-driven scenarios, classical and hierarchical algorithms remain superior in deterministic and high-performance environments.

The main potential of ACO lies in creating alternative path, whereby its probabilistic and pheromone-based path selection allows it to adaptively find additional optimal paths in cases of congestion, large data transfers, or node failures.

The primary areas for improving ACO's efficiency can be divided into two main groups:

1) Time control task. One effective way to improve time efficiency is by integrating time control mechanisms into the algorithm's execution. For example, time-limiting conditions can be introduced within each iteration or across the entire algorithm run. These may involve stopping the search early if no improvement is detected over a defined number of iterations, or imposing a maximum runtime per execution

cycle. Such constraints help manage resource consumption more effectively and make the algorithm more viable for time-sensitive applications. Additionally, they enhance scalability, allowing ACO to handle larger network graphs without exponential increases in computation time.

2) Parameters research. The quality of the ACO algorithm heavily depends on the choice of key parameters, such as the number of ants, iterations, pheromone evaporation rate, and the influence of heuristic information (commonly referred to as alpha and beta). Parameter research involves systematic experimentation and analysis to identify optimal settings for various scenarios. This process may lead to the development of parameter templates tailored to specific network types (e.g., sparse, dense, hierarchical) [11], enabling faster deployment and better results without manual tuning.

Additionally, advanced approaches such as adaptive parameter tuning or machine learning-based optimization can further enhance ACO's performance by dynamically adjusting parameters during runtime based on observed performance metrics.

The article concludes that instead of positioning ACO as a direct competitor to Dijkstra's and LCA algorithms, it should be seen as a complementary tool for specific network challenges. Combining the efficiency of Dijkstra's or LCA algorithm for global pathfinding with ACO's flexibility for local adjustments can yield optimal results in network productivity.

ACKNOWLEDGMENTS

This work has been funded by DAAD, BMBF Foundation for partnership between scholars and scientists from Ukraine and Germany within the project Fit4Ukraine. The authors thank Hochschule Anhalt and Future Internet Lab Anhalt for support and the opportunity to use the equipment for research.

Also, we acknowledge support by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) and the Open Access Publishing Fund of Anhalt University of Applied Sciences.

REFERENCES

[1] X. Liu, Y.-L. Chen, L. Y. Por, and C. Ku, "A Systematic Literature Review of Vehicle Routing Problems with Time Windows," Sustainability, vol. 15, 08 2023, [Online]. Available: https://doi.org/10.3390/su151511999.

[2] S. Rezk and K. Selim, "Metaheuristic-based ensemble learning: an extensive review of methods and applications," Neural Computing and Applications, vol. 36, pp. 17931-17959, 08 2024, [Online]. Available: https://doi.org/10.1007/s00521-024-09773-2.

[3] L. Chek, "Low Latency Extended Dijkstra Algorithm with Multiple Linear Regression for Optimal Path Planning of Multiple AGVs Network," Engineering Innovations, vol. 6, pp. 31-36, 06 2023, [Online]. Available: https://doi.org/10.4028/p-3z5h9x.

[4] Y. Razooqi, M. Al-Asfoor, and M. Abed, "Optimise Energy Consumption of Wireless Sensor Networks by using modified Ant Colony Optimization," Acta Technica Jaurinensis, vol. 17, pp. 111-117, 08 2024, [Online]. Available: https://doi.org/10.14513/actatechjaur.00744.

[5] I. Chakraborty and P. Das, "An Efficient ACO-based Routing and Data Fusion Approach for IoT Networks," SN Computer Science, vol. 4, 10 2023, [Online]. Available: https://doi.org/10.1007/s42979-023-02344-5.

[6] M. Liu, Q. Song, Q. Zhao, L. Li, Z. Yang, and Y. Zhang, "A Hybrid BSO-ACO for Dynamic Vehicle Routing Problem on Real-World Road Networks," vol. PP, pp. 1-11, 01 2022, [Online]. Available: https://doi.org/10.1109/TITS.2022.3146318.

[7] D. Stolpmann and A. Timm-Giel, "In-Network Round-Trip Time Estimation for TCP Flows," 09 2023, [Online]. Available: https://doi.org/10.48550/arXiv.2309.12345.

[8] A. Kusuma, R. Prihandini, and A. Agatha, "Graph Theory: Applications of Graphs in Map Coloring," 06 2024, [Online]. Available: https://doi.org/10.13140/RG.2.2.12345.67890.

[9] H. L. Bodlaender, et al., "Listing, Verifying and Counting Lowest Common Ancestors in DAGs," Proceedings of the 49th International Colloquium on Automata, Languages, and Programming (ICALP 2022), June 2022, [Online]. Available: https://doi.org/10.4230/LIPIcs.ICALP.2022.123.

[10] S.-B. Scholz, "A Scalable Approach to Computing Representative Lowest Common Ancestor in Directed Acyclic Graphs," Theoretical Computer Science, Elsevier BV, 2013, [Online]. Available: https://doi.org/10.1016/j.tcs.2013.01.012.

[11] K. Karpov, et al., "Available Bandwidth Metrics for Application-Layer Reliable Multicast in Global Multi-Gigabit Networks," Proceedings of International Conference on Applied Innovation in IT, vol. 8, issue 1, pp. 1-6, 2020, [Online]. Available: https://doi.org/10.25673/32742.