Efficiency Evaluation of Shortest Path Algorithms

Mariusz Głąbowski, Bartosz Musznicki, Przemysław Nowak and Piotr Zwierzykowski
Poznan University of Technology, Faculty of Electronics and Telecommunications
Chair of Communication and Computer Networks, Poznan, Poland
bartosz@musznicki.com, przemyslaw.nowak@inbox.com

***Abstract*—While the ever growing computational capabilities of devices that are used for man-machine interaction are taken for granted, the need to find their most optimum use is as important as ever. This issue is particularly relevant when considering solutions where the determination of the shortest path between given points (nodes) is one of the basic operations. In more complex executions of the shortest paths, sets of paths with the shortest distance between a single initial (source) point and all other destination points, as well as between all pairs of points, are to be found. For each of these approaches, individual algorithms with specific features have been worked out over the past decades. With that in mind, the present article seeks to explore this problem and is structured in such a way as to describe some of the selected algorithms solving the shortest path problem, and to analyse the efficiency of these algorithms during their operation in directed graphs of different type. The study shows that the efficiency varies among algorithms under investigation and allows to suggest which one ought to be used to solve a specific variant of the shortest path problem.**

Keywords-shortest path; algorithms; efficiency; evaluation

I. Introduction

The foundations for the present evaluation of the algorithms presented in this article are given by the research studies on shortest path problem solving using Ant Colony Optimization (ACO) metaheuristic approach [1]. It is just in the initial stage in the assessment of the potential in the applications of the ACO algorithm that the authors decided to start an in-depth analysis of those algorithms that represented a more traditional approach to the problem. As a result of the following studies, relevant tests have been carried out which are to be presented and compared in this article. It should be stressed that both well-known [2] and less commonly used algorithms are presented as long as they provide a possibility of finding the optimal solution having first satisfied some pre-defined initial requirements. Heuristic ACO algorithms have not been included in the presented evaluation for the simple reason that their operation does not, in fact, guarantee finding a solution that would always be optimal [3]. Moreover, the results obtained on the basis of ACO can be strongly dependent on the structure of the graph and there is no guarantee that any solution of any kind would be found at all [1].

In the process of careful investigation of publications related to the shortest path problem numerous books and papers have been studied. The most of comparison papers are either directed at specific aspects and applications of the algorithms [4]–[6] or are focused on comparing new concepts with more classical methods [7], [8]. Some papers are concerned with asymptotic computational complexity [9]–[12] while other works are aimed at empirical computational complexity analysis of a number of algorithms based on implementation and simulation [5], [13]–[16]. In this paper, we decided to follow the latter approach to build this article upon experimental findings with respect to practical performance of a range of 12 closed-form complexity algorithms for solving shortest path problems. The introduced homogeneous data structure representing graphs under scrutiny is carefully discussed. Owing to the well-defined data structure, the results can be directly compared what is critical to conclusively evaluate the efficiency.

The contents of the subsequent sections are arranged as follows. Section II shows the problem of the shortest path and lists some of its applications. The relevance to and relationship with the shortest path tree is discussed in Section III. In addition, a description of the two groups of algorithms that have been put to the analysis is presented. The data structure that represents the graphs under consideration is discussed in Section IV. Later on, in Section V, the graphs in which simulations were carried out are described. The description is followed by Section VI that will focus on the presentation and discussion of the results of the study. Finally, in Section VII, the article is summed up with conclusions.

II. Problem of the shortest path

For the directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{A})$, where N is the set of nodes (vertices) and A is the set of arcs (edges), we assign the cost a_{ij} to each of its edges $(i, j) \in \mathcal{A}$ (alternatively, this cost can be also called the length). We denote the biggest absolute value of an edge cost by C. For the resulting path $(n_1, n_2, \ldots, n_k)$, its length can be expressed by (1).

$$a_{ij} = \sum_{i=1}^{k-1} a_{n_i n_{i+1}} \qquad (1)$$

A path is called the shortest path if it has the shortest length from among all paths that begin and terminate in given vertices. The shortest path problem involves finding paths with shortest lengths between selected pairs of vertices. The initial vertex will be designated as s, while the end vertex as t.

A number of basic variants of the shortest path problem can be distinguished [17]:

- finding the shortest path between a pair of vertices.
- finding the shortest paths with single initial vertex.
- finding the shortest paths with single end vertex.
- finding the shortest paths between all pairs of vertices.

In solving the problem of the shortest path we shall apply the following assumptions (which, in the case of some specific algorithms, may not be required).

- *Costs of the edge a_{ij} are integers* (this requirement applies to only some of the algorithms). In the case of the real costs of the edge, we can convert summations to integers multiplying them by an appropriately high number. Imaginary values would introduce unnecessary complications with their representations in computer-mediated activities.
- *There is a directed path between the pairs of vertices under consideration.*
- *The graph does not include negative cycles.* The problem of the shortest path with negative cycles is $\mathcal{NP}$-hard (impossible to be presented using a polynomial algorithm).
- *The graph is a directed graph.* In the case of the undirected graph with non-negative weights, it is easy to to transform it into a directed graph.

The solution for the problem of the shortest path finds its application in a number of areas such as transportation or routing in communication networks [2], [18], [19] and is often related to searching for the shortest path tree in a graph.

III. Algorithms for solving Shortest Path problems

The following subsections of this Section focus on the algorithms for a determination of the shortest paths between a given single initial vertex and all the remaining vertices of the graph. It can be proved that the shortest paths from one node of a graph to all of the remaining nodes create a shortest paths tree [17], [20]. A characteristic feature of this tree is the fact that its root is formed from the initial (source) vertex, all of its edges are directed in the direction opposite to the vertex, and each path that can be created from the initial vertex to any other vertex is the shortest path to this vertex.

The algorithms solving shortest path problems that are briefly discussed in the following subsections have been evaluated through an efficiency analysis. Each of the algorithms has particular features that eventually lead to their differences in their properties and performance. On account of their possible applications, the algorithms have been, in turn, divided into two categories.

A. Single-Source Shortest Paths problem

The following subsections of this Section focus on the algorithms for a determination of the shortest paths between a given single initial vertex and all the remaining vertices of the graph.

1) Generic algorithm: The operation of the generic algorithm [21] is based on iterative checking of edges from the vertex under consideration i and on label setting for vertex j, in which a given edge terminates, to $d_j = d_i + a_{ij}$, in the case when $d_j > d_i + a_{ij}$. To store the vertices that are to be checked, the list V is used, called *candidates list*. The way vertices are stored in this list, as well as the method determining the addition and the retrieval of vertices to and from it, is frequently the major factor that distinguishes individual algorithms under consideration. In the case of the generic algorithm, the candidates list is a *FIFO* queue in which operations of additions and retrieval of a vertex to the end of it or from its head, respectively, are performed.

2) Dijkstra's algorithm: Dijkstra's algorithm is presumably the best known algorithm for finding the shortest path in the directed graph [22]. The basic difference between this algorithm and the generic algorithm is the way in which vertices are drawn from the candidates list — the selected vertex is the vertex that has the smallest label from all available vertices in the list:

$$d_i = \min_{j \in V} d_j \tag{2}$$

This causes the vertex with its label set, as well as all vertices that are in the path from the initial vertex to this particular vertex, to have the minimum value of the label and to not be added again to the candidates list. The total number of operations that the Dijkstra's algorithm needs to perform to solve the shortest path problem is $O(N^2)$.

3) Dijkstra's algorithm using a heap: It is not possible to decrease the number of operations that are performed in order to check labels, because this would not make it possible to guarantee the optimal solution finding — each edge has to be checked at least once. A selection of an optimal data structure that represents the list of candidates makes it possible, in turn, to reduce significantly the computational complexity of the operation of the selection of a vertex from the candidates list [23]. Here, heaps (also known as priority queues) can serve ideally the purpose. Using Fibonacci heap we can solve the shortest path problem using Dijkstra's algorithm and performing $O(A + N \log N)$ operations.

4) Dial's algorithm: Another way to reduce the number of operations accompanying the selection of a vertex from the candidates list is a division of the list into buckets [24]. Each bucket B_k stores only vertices with a given label k. This causes lengths of edges to have to be integers and non-negative. The computational complexity of the Dial's algorithm is $O(A + NC)$. What is crucial to understand, is that the bucket deletion and insertion operations require linear time and not more than NC buckets need to be examined by the procedure [14]. The higher the absolute value of an arc cost C, the more operations need to be performed by the algorithm, and thus, the performance gain related to the usage of buckets dramatically diminishes. Therefore, for small values $C \ll N$, Dial's algorithm performs very well in practice.

5) Bellman-Ford algorithm: The Bellman-Ford algorithm belongs to algorithms of the *label-correcting* type that treat all labels for vertex distances as temporary until the last iteration, after which all labels are set to optimal values [25]. This algorithm provides a possibility to solve the shortest path problem in graphs with negative lengths of edges. In the case when a negative cycle is found, the algorithm yields falsehood as the result of its operation. This algorithm makes $N-1$ iterations in which it checks A edges. Its computational complexity is then equal to $O(NA)$.

6) D'Esopo-Pape algorithm: The D'Esopo-Pape algorithm uses the candidates list in the form of a queue [26]. Vertices that are to be checked are always retrieved from the head of the list. However, the place a given vertex is added to in the candidate list depends on whether the vertex has already been located in this list. If so, it is added to its head, otherwise — to the end of the list.

7) SLF algorithm: The Small Label First algorithm (*SLF*) seeks to manage the candidates list in such a way as to make vertices with small labels located as close to the head of the list as possible [27]. The reason for this operation is the fact that the smaller the label of a vertex that is retrieved from the candidates list, the lower the probability that this vertex will be forwarded to the list once again. This algorithm, just as the two following algorithms, attempts to reach the characteristic operation of Dijkstra's algorithm with a lower computational outlay.

8) LLL algorithm: The Large Label Last algorithm (*LLL*) attempts to achieve the operation that is similar to that of the previous algorithm using a specific method for the retrieval of vertices from the candidates list [28]. The addition of vertices to the candidates list is not defined in any way. However, the method for their retrieval from the list is defined. Each time when a vertex is to be taken from the list, the average value of the labels of the vertices in the list is calculated. Then, the label of the vertex that is at the head of the list is compared with this average. If the label of the vertex is higher than the average, the vertex is moved to the end of the list. Otherwise, the vertex is returned as the one that has to be considered in this iteration.

9) SLF/LLL algorithm: The *SLF/LLL* combines the *SLF* algorithm method for the addition of vertices to the candidates list and the *LL* algorithm method for their retrieval from the list [21]. The *SLF/LLL* algorithm requires a lower number of iterations to solve the shortest path problem than the algorithms it combines. This is done, however, at the cost of the increased number of necessary calculations.

B. All-Pairs Shortest Path problem

The following subsections present algorithms that are dedicated to finding the shortest paths between all pairs of vertices.

1) The doubling algorithm: Algorithm's operation is based on iterative calculation of the shortest paths for all vertices composed of an increasing number of edges [29]. It starts with paths that are composed of just one edge, and then checks whether paths that are composed of two edges would not be shorter. This operation is then repeated until all paths that are composed of $N-1$ edges are checked. Bearing in mind the fact that a path that is composed of more than $N-1$ edges cannot be shorter than the shortest path, we know that $D^n = D^{(N-1)}$ for all $n \geq N-1$. This gives the ultimate computational complexity of the algorithm equal to $\Theta(n^3 \log_2 N)$.

2) Floyd-Warshall algorithm: The Floyd-Warshall algorithm obtains what the previous algorithm was capable of, using a different approach and achieving at the same time lower computational complexity equal to $\Theta(N^3)$ [30], [31].

3) Johnson algorithm: For sparse graphs (i.e., those in which the number of edges is far lower than N^2) it is possible to improve the process of calculation of the shortest paths between all pairs of vertices using Johnson algorithm [32]. For this purpose, the two algorithms discussed earlier, i.e., the Bellman-Ford algorithm and Dijkstra's algorithm (most favourably in its form with a heap), are used. If we choose to apply the implementation of Dijkstra's algorithm with Fibonacci heap, then we are obliged to perform $O(NA + N^2 \log N)$ operations to calculate the shortest paths between all the pairs of vertices in a sparse graph. Using a binary heap would result in an increase in the number of necessary operations to $O(NA \log N)$.

IV. Data structure representing graphs

To represent graphs during the simulation, a double associative adjacency array was used. This structure is composed of two associative arrays — one (external), representing vertices from which edges originate, and the other (internal) representing all vertices which edges for a given row of the first matrix (table) join. Such a representation provides an opportunity to minimize shortcomings of typical structures,

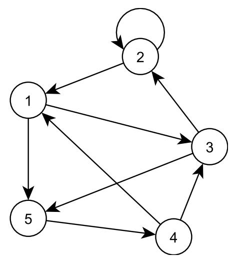

Figure 1. Exemplary directed graph

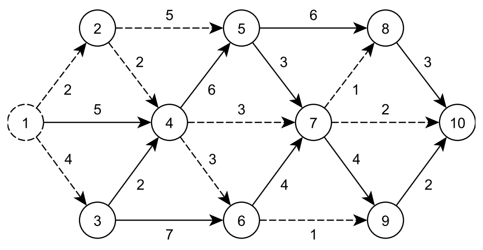

Figure 2. Manually created *custom* graph in which the edges marked with the dashed line create a shortest paths tree with the root in node 1

such as the list of edges or the adjacency matrix, providing at the same time appropriately low computational complexity for individual operations. The applied structure makes it possible to store additional information about edges, e.g., weights or costs. A homogeneous method for the projection (mapping) of graphs for all simulated algorithms ensures further comparability of the results of simulations.

The operation of the structure may differ depending on the implementation of the associative array and is dependable on the programming language used if embedded structures are used. The most crucial operation is the operation of checking whether a given key is in the array, hence structures that handle this best, e.g., hash tables or self-balancing binary search trees, are applied. Additionally, we can adjust the operation of the double associative adjacency array for our particular needs and thus make it possible, for example, to sort vertices in the internal array which a given edge joins using a heap.

For a graph with edge weights, the double, associative adjacency array T_{2asoc} can be written as follows:

$T_{2asoc} = T_{ext}$	external array
$T_{2asoc}[i] = T_{ext}[i] = T_{int_i}$	internal array for edge coming out from vertex i
$T_{2asoc}[i][j] = T_{int_i}[j] = a_{i,j}$	edge weight (i, j)

For example, the graph in Fig. 1 will be mapped in the following way:

$$\begin{array}{lll}
T_{2asoc}[1] & = T_{int_1} & \\
T_{2asoc}[1][3] & = T_{int_1}[3] & = a_{1,3} \\
T_{2asoc}[1][5] & = T_{int_1}[5] & = a_{1,5} \\
T_{2asoc}[2] & = T_{int_2} & \\
T_{2asoc}[2][1] & = T_{int_2}[1] & = a_{2,1} \\
T_{2asoc}[2][2] & = T_{int_2}[2] & = a_{2,2} \\
T_{2asoc}[3] & = T_{int_3} & \\
T_{2asoc}[3][2] & = T_{int_3}[2] & = a_{3,2} \\
T_{2asoc}[3][5] & = T_{int_3}[5] & = a_{3,5} \\
T_{2asoc}[4] & = T_{int_4} & \\
T_{2asoc}[4][1] & = T_{int_4}[1] & = a_{4,1} \\
T_{2asoc}[4][3] & = T_{int_4}[3] & = a_{4,3} \\
T_{2asoc}[5] & = T_{int_5} & \\
T_{2asoc}[5][4] & = T_{int_5}[4] & = a_{5,4}
\end{array}$$

Characteristic features of the structure:

- required memory: $O(N + A)$
- effective memory complexity for directed sparse graphs
- effective execution of graph algorithms that require to reach all vertices adjacent to a given vertex (logarithmic complexity)
- capacity of remembering parallel edges
- effective execution of checking whether the graph includes a given edge (logarithmic complexity)
- effective execution of addition and removal of edges of a graph (logarithmic complexity)
- possibility of a substitution of the internal associative table with some other structure, e.g., in order to sort vertices in which a given edge terminates by the weight of the edge (e.g., using binary, Fibonnaci, binomial or Relaxed heap)
- fairly complicated in its execution

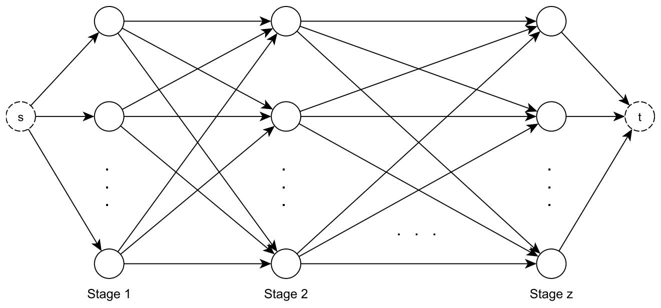

Figure 3. A graph that presents the problem of the shortest path in a multi-stage graph

TABLE I. STRUCTURE OF THE GRAPHS USED IN THE SIMULATION

graph	vertices	edges	
		number	lengths
custom	10	19	$\langle 1, 7\rangle$
multistage	52	420	$\langle 1, 9\rangle$
random	25	125	$\langle 1, 9\rangle$

V. Graphs used in the simulation

To examine the efficiency and performance of the algorithms during their operation in different graphs, directed graphs constructed manually and those that were generated pseudo-randomly were used. To discuss the results, the 3 representative graphs described in Table I were selected. Graph *custom* shown in Fig. 2 was created manually from 10 vertices that were joined together by 19 edges.

Another graph that was used in the tests is the graph that is characteristic for a multi-stage shortest path problem. An exemplary graph is presented in Fig. 3. The multi-stage graph used in the tests, *multistage*, has 5 stages, each having 10 vertices. The lengths of edges were generated randomly from within the interval $\langle 1, 9\rangle$. The *random* graph was generated randomly, without loops, and with 5 edges coming out of each of the vertices.

VI. Results of the simulations of the algorithms

All the tests were carried out in a simulation environment prepared in C# programming language. In order to achieve reliable results, each algorithm was performed 100 times for each of the graphs. To eliminate the influence of the simulation environment, extreme results were rejected and then the average of the remaining results was calculated.

Table II shows the running times of the algorithms tested for the graphs discussed in Section V. The results are divided into two groups — algorithms solving Single-Source Shortest Paths problem (*SSSP*) and algorithms solving All-Pairs Shortest Path problem (*APSP*).

The graph *custom* was solved by all *SSSP* algorithms in almost identical times. Of all the algorithms only two deserve a mention here — Dijkstra's algorithm with a heap (that operated within the longest time), and *SLF* (that solved the problem slightly quicker than the rest). The results that were very similar to that of the *SLF* algorithm were also shared by Dijkstra's algorithm, Dial's algorithm and the *LLL* algorithm. From the group of the *APSP* algorithms, it was the Floyd-Warshall algorithm that fared the best, being less than twice as long as the *SSSP* algorithms. The remaining algorithms needed about twice as much time to find all paths.

The graph characteristic for the multi-stage shortest path problem (*multistage*) brought a significant increase in differences between *SSSP* algorithms. Again, the *SLF* algorithm was the quickest, whereas Bellman-Ford and D'Esopo-Pape algorithms handled the problem the worst. Except Dijkstra's algorithm with a heap, which was performing slightly longer than the rest, the remaining algorithms had similar running times. This situation for the *APSP* algorithms was different than in the case of the previous graph — Johnson algorithm was the quickest and the doubling algorithm was the slowest.

The last graph under consideration (*random*) was solved the quickest in the *SSSP* mode by the *SLF* algorithm, with Dijkstra's algorithm as the runner up and the Bellman-Ford and the D'Esopo-Pape algorithms well behind the two. The latter two were the worst as compared to all involved *SSSP* algorithms. This time, the quickest *APSP* algorithm was the Floyd-Warshall algorithm. Johnson algorithm performed slightly worse, while the doubling algorithm was the worst (the longest) of the lot.

The procedures that solve the *SSSP* problem best include the *SLF* algorithm, that had the shortest times for each tested graph, and Dijkstra's algorithm, that always performed with a quite similar time. The *LLL* and the *SLF/LLL* algorithms performed very well and did not generate solutions over times that differ much from those provided by the quickest algorithm. The generic algorithm and Dial's algorithm performed slightly better or slightly worse depending on the chosen graph. Dijkstra's algorithm with a heap had some problems and, instead of performing quicker than Dijkstra's algorithm, was slower. In this particular case, this can be most probably explained by the missing optimization of the heap that formed the base for the algorithm. Undoubtedly, however, an improvement in the running time during which solutions are provided is still possible. At least, an improvement in the running time needed for the algorithm to generate solutions is possible. As it is clear from Fig. 4, for the Bellman-Ford and D'Esopo-Pape algorithms, the worst case occurs far too often, which may result from both non-optimal implementation and from the possibility of their operation on graphs that were unsuitable for them. The D'Esopo-Pape algorithm was much quicker to solve graphs, but irrespective of the fact it underperformed far too much as compared to the rest of the algorithms. Underperformance of the latter group of algorithms is particularly visible in graphs that have a higher number of edges, which results from the assumptions, as they were, that served as a basis

TABLE II. COMPARISON OF RUNNING TIMES FOR THE ALGORITHMS SOLVING THE SHORTEST PATH PROBLEM IN MICROSECONDS

group	algorithm	graph		
		custom	multistage	random
SSSP	generic	112	312	163
	Dijkstra	100	324	148
	DijkstraHeap	146	466	200
	Dial	104	322	172
	Bellman-Ford	119	3252	511
	D'Esopo-Pape	113	1260	239
	SLF	96	262	143
	LLL	102	336	155
	SLF/LLL	112	318	161
APSP	doubling algorithm	324	47678	4756
	Floyd-Warshall	184	16045	2057
	Johnson	418	9309	2959

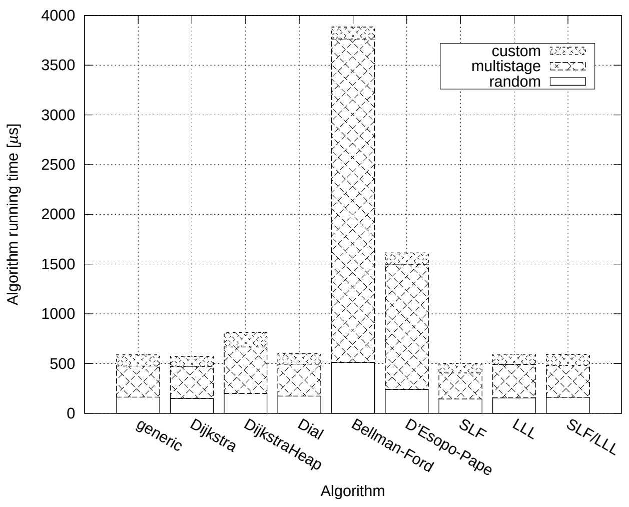

Figure 4. Chart of aggravated running times of the algorithms solving the shortest path problem with one initial vertex (*SSSP*)

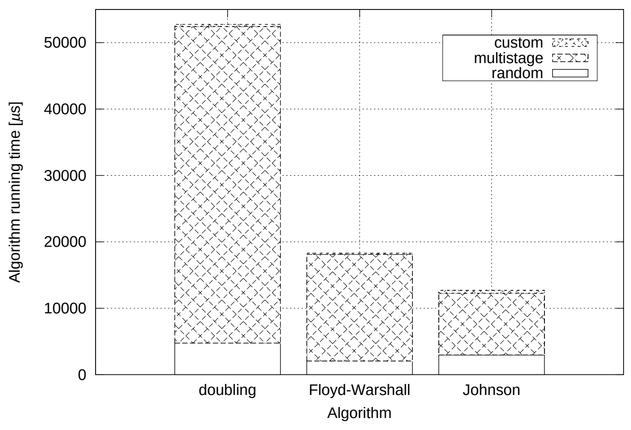

Figure 5. Chart of aggravated running times of the algorithms solving the shortest path problem between all pairs of vertices (*APSP*)

for their design.

The *APSP* algorithms were decidedly varied across different performance dimensions, in particular in relation to the time necessary to generate results, which is clearly shown in Fig. 5. The doubling algorithm was the slowest and performed several times slower than the competitors. The Floyd-Warshall algorithm was the fastest for 2 graphs, while for the third graph it was in second place. The differences in the time needed for graphs to be solved are in its case significant as compared to Johnson algorithm that overall turned out to be the fastest one.

VII. Conclusion

This article presented 12 algorithms solving the shortest path problem and provided an evaluation of their efficiency. The study showed that in a prepared simulation environment that ensured directed graphs of different type to be provided, the weakest aggregated time results from among all the available algorithms solving the Single-Source Shortest Paths problem were those of, in the descending order, the Bellman-Ford and the D'Esopo-Pape algorithms. The fastest algorithm was Small Label First algorithm, slightly faring better than Large Label Last algorithm. From the pool of the algorithms dedicated for All-Pairs Shortest Path problem, the doubling algorithm performed decidedly worst, while the best results were those of Floyd-Warshall algorithm (two graphs) and Johnson algorithm (one graph).

In addition to the presentation of run-time relationships between the algorithms, the study indicated the importance and significance of an appropriate choice of a method destined to solve the problem that would be the most efficient for a type of the graph structure to be used. Moreover, details concerning the implementation as well as the architecture of the structures for the representation of data can significantly influence the performance of an algorithm.

References

[1] M. Głąbowski, B. Musznicki, P. Nowak, and P. Zwierzykowski, "Shortest Path Problem Solving Based on Ant Colony Optimization Metaheuristic," International Journal of Image Processing & Communications, Special Issue: Algorithms and Protocols in Packet Networks, vol. 17, no. 1–2, 2012, pp. 7–17.

[2] B. Y. Wu and K.-M. Chao, Spanning Trees and Optimization Problems. USA: Chapman & Hall/CRC Press, 2004.

[3] C. Blum and A. Roli, "Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison," ACM Computing Surveys, vol. 35, no. 3, September 2003, pp. 268–308.

[4] R. Vasappanavara, E. V. Prasad, and M. N. Seetharamanath, "Comparative Studies of Shortest Path Algorithms and Computation of Optimum Diameter in Multi Connected Distributed Loop Networks," Multi-, Inter-, and Transdisciplinary Issues in Computer Science and Engineering, vol. 2, no. 1, January 2006, pp. 62–67.

[5] B. V. Cherkassky, L. Georgiadis, A. V. Goldberg, R. E. Tarjan, and R. F. Werneck, "Shortest Path Feasibility Algorithms: An Experimental Evaluation," ACM Journal of Experimental Algorithmics, vol. 14, 2009.

[6] K. Gutenschwager, A. Radtke, S. Völker, and G. Zeller, "The Shortest Path - Comparison of Different Approaches and Implementations for the Automatic Routing of Vehicles," in Proceedings of the 2012 Winter Simulation Conference, Berlin, Germany, 9–12 December 2012.

[7] U. Lauther, "An Experimental Evaluation of Point-To-Point Shortest Path Calculation on Road Networks with Precalculated Edge-Flags," in Proceedings of Ninth DIMACS Implementation Challenge, Piscataway, NJ, USA, 13–14 November 2006.

[8] Y. Sharma, S. C. Saini, and M. Bhandhari, "Comparison of Dijkstra's Shortest Path Algorithm with Genetic Algorithm for Static and Dynamic Routing Network," International Journal of Electronics and Computer Science Engineering, vol. 1, no. 2, 2012, pp. 416–425.

[9] S. Pettie, "On the Comparison-Addition Complexity of All-Pairs Shortest Paths," in Proceedings of ISAAC 2002, 13th International Symposium on Algorithms and Computation, Vancouver, BC, Canada, 21–23 November 2002.

[10] J. Hershberger, S. Suri, and A. Bhosle, "On the Difficulty of Some Shortest Path Problems," ACM Transactions on Algorithms, vol. 3, no. 1, 2007.

[11] R. Cohen and G. Nakibly, "On the Computational Complexity and Effectiveness of N-hub Shortest Path Routing," IEEE/ACM Transactions on Networking, vol. 16, no. 3, 2008, pp. 691–704.

[12] L. Roditty and U. Zwick, "On Dynamic Shortest Paths Problems," Algorithmica, vol. 61, no. 2, 2011, pp. 389–401.

[13] B. L. Golden, “Shortest Path Algorithms: A Comparison,” Massachusetts Institute of Technology, Operations Research Center, Tech. Rep., October 1975.

[14] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, “Shortest paths algorithms: Theory and experimental evaluation,” Mathematical Programming, vol. 73, no. 2, 1996, pp. 129–174.

[15] P. Biswas, P. K. Mishra, and N. C. Mahanti, “Computational Efficiency of Optimized Shortest Path Algorithms,” International Journal of Computer Science & Applications, vol. 2, no. 2, 2005, pp. 22–37.

[16] C. Demetrescu, S. Emiliozzi, and G. F. Italiano, “Experimental Analysis of Dynamic All Pairs Shortest Path Algorithms,” ACM Transactions on Algorithms, vol. 2, no. 4, 2006, pp. 578–601.

[17] R. K. Ahuja, T. L. Magnati, and J. B. Orlin, Network Flows: Theory, Algorithms and Applications. Englewood Cliffs, N.J.: Prentice-Hall, 1993.

[18] K. Stachowiak, J. Weissenberg, and P. Zwierzykowski, “Lagrangian relaxation in the multicriterial routing,” in IEEE AFRICON, Livingstone, Zambia, September 2011, pp. 1–6.

[19] B. Musznicki, M. Tomczak, and P. Zwierzykowski, “Dijkstra-based Localized Multicast Routing in Wireless Sensor Networks,” in Proceedings of CSNDSP 2012, 8th IEEE, IET International Symposium on Communication Systems, Networks and Digital Signal Processing, Poznań, Poland, 18–20 July 2012.

[20] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms. MIT Press, 1990.

[21] D. P. Bertsekas, Network Optimization: Continuos and Discrete Models. Belmont, Massechusetts: Athena Scientific, 1998.

[22] E. W. Dijkstra, “A note on two problems in connexion with graphs,” Numerische Mathematik, vol. 1, 1959, pp. 269–271.

[23] S. Saunders, “A Comparison of Data Structures for Dijkstra’s Single Source Shortest Path Algorithm,” 5 November 1999.

[24] R. B. Dial, “Algorithm 360: shortest-path forest with topological ordering,” Communications of the ACM, vol. 12, November 1969, pp. 632–633.

[25] J. Bang-Jensen and G. Gutin, Digraphs: Theory, Algorithms and Applications. London: Springer-Verlag, December 2008.

[26] U. Pape, “Implementation and efficiency of Moore-algorithms for the shortest route problem,” Mathematical Programming, vol. 7, no. 1, 1974, pp. 212–222.

[27] D. P. Bertsekas, “A Simple and Fast Label Correcting Algorithm for Shortest Paths,” Networks, vol. 23, 1993, pp. 703–709.

[28] D. P. Bertsekas, F. Guerriero, and R. Musmanno, “Parallel asynchronous labelcorrecting methods for shortest paths,” Journal of Optimization Theory and Applications, vol. 88, February 1996, pp. 297–320.

[29] E. Dekel, D. Nassimi, and S. Sahni, “Parallel Matrix and Graph Algorithms,” SIAM Journal on Computing, vol. 10, no. 4, 1981, pp. 657–675.

[30] R. W. Floyd, “Algorithm 97: Shortest path,” Communications of the ACM, vol. 5, June 1962, p. 345.

[31] S. Warshall, “A Theorem on Boolean Matrices,” Journal of the ACM, vol. 9, January 1962, pp. 11–12.

[32] D. B. Johnson, “Efficient Algorithms for Shortest Paths in Sparse Networks,” Journal of the ACM, vol. 24, January 1977, pp. 1–13.