Review and Performance Analysis of Shortest Path Problem Solving 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, Poznań, Poland,
e-mail: mariusz.glabowski@put.poznan.pl, przemyslaw.nowak@inbox.com, piotr.zwierzykowski@put.poznan.pl
** INEA S.A., Poznań, Poland, e-mail: bartosz.musznicki@inea.com.pl

***Abstract*—The development of concepts derived from the generic approach to solving the problem of the shortest path resulted in numerous and various algorithms that appeared over the past decades. The studies on the most basic operation aimed at the determination of the shortest path between two given points in a graph (in other words, often a network) have resulted in sophisticated solutions designed for more and more demanding applications. Those include finding the sets of paths with the shortest distance between all pairs of nodes or searching for a shortest path tree. The aim of the present article is to give the reader an introduction to the problem of the shortest path and a detailed review of two groups of selected algorithms designed to solve particular problems. In the study described herein, different algorithms have been examined for their efficacy in their operation in directed graphs of different type represented in a well-defined data structure. The empirical simulation-based analysis proves that the performance varies among algorithms under investigation and allows to suggest, which methods ought to be used to solve specific variants of the shortest path problem and which algorithms should be avoided or used with caution.**

Keywords-shortest path; algorithms; review; performance; analysis

I. Introduction

This article is an extended version of a conference paper presented at AICT 2013, The Ninth Advanced International Conference on Telecommunications [1]. It introduces numerous additions, such as, more information on the problem of the shortest path, a detailed description of approaches and algorithms being tested, and a discussion of new simulation results. The foundations for the present review and performance analysis of selected algorithms are given by the research studies on shortest path problem solving using Ant Colony Optimization (ACO) metaheuristic approach [2]. It is just in the initial stage in the assessment of the potential in the applications of the ACO algorithms 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 investigations, relevant tests have been carried out. They are presented and compared in this article. It should be stressed that both well-known [3] 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 [4]. 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 [5].

The contents of the subsequent sections are arranged as follows. Section III shows two distinct approaches to the definition of the problem of the shortest path, lists some of its applications, and introduces the key assumptions, which shall be applied. Section IV is aimed at presenting the general types of shortest path problem solving algorithms. A detailed description and discussion of the two groups of algorithms that have been put to the analysis are presented in Section V. In addition, the relevance to and relationship with the shortest path tree is discussed. The data structure that represents the graphs under consideration is discussed in Section VI. Then, in Section VII, the graphs in which the simulations were carried out are described. The description is followed by Section VIII that will focus on the presentation and discussion of the results of the study. Finally, in Section IX, the article is summed up with conclusions.

II. Related work

In the process of careful investigation of publications related to the shortest path problem, numerous books and papers have been studied. The bulk of comparison papers are either directed at specific aspects and applications of the algorithms [6]–[9] or are focused on comparing new concepts with more classical methods [10], [11]. Some papers are concerned with asymptotic computational complexity [12]–[15], while other works are aimed at empirical computational complexity analysis of a number of algorithms based on implementation and simulation [7], [16]–[19]. 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 that have not been compared before. 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, which is critical in conclusive evaluation of the efficiency.

III. 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 [see (1)]. For the resulting path $(n_1, n_2, \dots, n_k)$, its length a_P can be expressed by (2).

$$C = \max_{(i,j)\in\mathcal{A}} a_{ij} \tag{1}$$

$$a_P = \sum_{i=1}^{k-1} a_{n_i n_{i+1}} \tag{2}$$

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 (start) vertex will be designated as s, while the end (goal) vertex as t.

The problem of the shortest path can be also expressed differently [20]. If we go back to the original definition and define a_{ij} as the cost (and not the length), the problem can be reduced in its essence to a transmission of one flow unit between a pair of vertices as cheaply as it is possible. The problem is defined as follows: minimize (3a) limited by (3b) and (3c).

$$\sum_{(i,j)\in\mathcal{A}} a_{ij} x_{ij} \tag{3a}$$

$$0 \leq x_{ij}, \qquad \forall\, (i, j) \in \mathcal{A} \tag{3b}$$

$$\sum_{\{j:(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j:(j,i)\in\mathcal{A}\}} x_{ji} = \begin{cases} 1, & \text{for } i = s \\ -1, & \text{for } i = t \\ 0, & \text{otherwise} \end{cases} \tag{3c}$$

This makes it possible to formulate the shortest path problem by defining a linear function that is analogous to the function that defines the minimum cost flow problem. To illustrate the comparison drawn above, let us assign a flow vector x that is described by

$$x_{ij} = \begin{cases} 1, & \text{for } (i, j) \in P \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

to a randomly selected path P from s to t.

Then x can be a solution to the problem (3), while the cost x is equal to the path length P. Hence, if vector x in the representation of formula (4) is the optimum solution to the problem (3), then the relevant corresponding path P is the shortest path.

A number of basic variants of the shortest path problem can be distinguished [21]:

- *finding the shortest path between a pair of vertices*
  For a given pair of vertices s and t the shortest path between them should be found. It should be mentioned here that so far no algorithm is known that solves this problem asymptotically in its worst case better than the best algorithm for the problem with one initial vertex.
- *finding the shortest paths with single initial vertex*
  For a given vertex s the shortest path between the vertex and each of the vertices $i \in \mathcal{N}$ is to be found.
- *finding the shortest paths with single end vertex*
  This is a reverse of the previous variant — the shortest path from each of the vertices $i \in \mathcal{N}$ to a given vertex t is to be found. By reversing each of the edges of the graph, the previous problem is obtained.
- *finding the shortest paths between all pairs of vertices*
  The shortest path between each pair of the vertices i and j that belong to $\mathcal{N}$ is to be found. A solution to this problem can be obtained by a solution of the shortest path problem with one initial vertex for each of the vertices in the graph.

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).

- *The graph is a directed graph.* In the case of the undirected graph with non-negative weights, it is easy to transform it into a directed graph.
- *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).
- *There is a directed path between the pairs of vertices under consideration.*
- *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.

The solution to the problem of the shortest path finds its application in a number of areas such as transportation or routing in communication networks [3], [22], [23] and is often related to searching for the shortest path tree in a 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 [21], [25]. 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.

IV. Types of Shortest Path problem solving algorithms

The shortest path algorithms are characterized by certain common features — they are iterative, while their operation is based on assigning to particular vertices distance labels that are currently the best distances from the beginning of the path that is to be found. During the performance of these algorithms a set of valid vertices that can be taken into consideration is maintained. The method for a representation of this set may vary depending on a particular algorithm and can be representative for it. The difference in these algorithms is based on the method of updating the distance labels and a selection of a vertex expected to leave the mentioned set. Therefore, we can divide the shortest path algorithms into two groups [24]:

- *label-setting algorithms*
  This type of algorithms is characterized by a permanent setting of the distance label of one of the vertices in each iteration. This is equivalent to a single removal of a given vertex from the set of vertices under scrutiny. The most computationally complex part in these algorithms is mainly a selection, in each of the iterations, of a vertex with the lowest distance label from among the vertices that belong to the set of vertices under scrutiny. Algorithms from this group can be additionally applied only to acyclic graphs with defined (e.g., integer) edge lengths, or in the case when edges have non-negative lengths.
- *label-correcting algorithms*
  Unlike the algorithms of the previous group, algorithms of the type *label-correcting* treat all distance labels of vertices as temporary until the last iteration, whereupon all labels are set to the optimum value. This is translated then into a multiple addition of a vertex to the set of vertices under consideration and its multiple removal from the set. Due to the above, a choice of a vertex in each of the iterations is less computationally complex. Algorithms of this type can be used to solve all classes of the shortest path problem, including those with negative lengths of edges.

Label-setting algorithms can be viewed as a particular case of *label-correcting* algorithms. This means that *Label-correcting* algorithms can be used for solving more general cases of the problem. *Label-setting* algorithms for the case of non-negative lengths (costs) of edges have lower pessimistic complexity, which does not necessarily have to translate into better expected (average) complexity. All discussed *label-correcting* algorithms achieve identical pessimistic computational complexity. Differences in effectiveness of the algorithms can be seen in their practical applications or with particular graphs.

By taking into consideration practical applications of the algorithms under study, a numbers of factors are in favour of *label-correcting* algorithms. These algorithms are more elastic and, in consequence, can be better adjusted to making use of additional initial data for a given graph. They can be also better adjusted to a problem in providing a solution to which they have been used — it is even possible in some cases to make distance labels set only once (for one vertex to be added and removed from the set of considered vertices). This equalizes the most important advantage of *label-setting* algorithms.

In practice, graphs that have edges with negative costs are rare, while for these cases good *label-correcting* algorithms have better expected complexity than *label-setting* algorithms. It is so because, beside the required $O(A)$ number of operations for the algorithms of both types that is needed to check each of the edges at least once, the *label-setting* algorithms require approximately additional operations with their number proportional to N, while the number of operations of additional *label-correcting* algorithms approximately increases linearly with A. For rare graphs the ratio of additional operations of both groups of algorithms is much favourable for *label-correcting* algorithms, whereas for dense graphs, for *label-setting* algorithms.

V. 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.

The algorithms solving shortest path problems that are briefly discussed in the following subsections have been evaluated through 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 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 [20] 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.

The algorithm starts to check from the initial vertex s, with initial conditions defined by (5).

$$\begin{aligned} V &= \{s\}, \qquad d_s = 0 \\ d_i &= \infty, \qquad \forall\, i \neq s \end{aligned} \tag{5}$$

The algorithm checks individual edges of the initial vertex and, if an appropriate condition is satisfied, sets the labels of the vertex in which a given edge terminates, adding it to the candidates list if it is not already there. The procedure is then repeated until the list of candidates is empty.

During the performance of the algorithm labels are monotonically non-increasing and if $d_i < \infty$, then vertex i has appeared on the candidates list V at least once.

2) Dijkstra's algorithm: Dijkstra's algorithm is presumably the best known algorithm for finding the shortest path in the directed graph [26]. The method is an algorithm of the type *label-setting*, which means that once considered vertex does not appear again on the list of candidates, while its label, once it is set, is ultimate and denotes the shortest distance from the initial vertex to this vertex.

The initial conditions are illustrated in (5), while an additional constraint is non-negativity of the length of the edge (6).

$$a_{ij} \geq 0 \tag{6}$$

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:

$$d_i = \min_{j \in V} d_j \tag{7}$$

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 number of iterations of the algorithm is equal to the number of vertices N. During each iteration, two operations are performed — the choice of a vertex from among all vertices on the list of candidates, and scanning and, should the need arises, setting of distance labels. The choice of a vertex in its worst case requires $O(N)$ operations, which in conjunction with the number of iterations gives $O(N^2)$ operations. Checking of the labels is performed A times, since in each iteration the algorithms checks all edges that start in the vertex under scrutiny, whereas each vertex is considered only once. $O(A)$ is not taken into account because it is far smaller than $O(N^2)$. Therefore, 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 optimum solution finding — each edge has to be checked at least once. A selection of an optimum data structure that represents the candidates list makes it possible, in turn, to reduce significantly the computational complexity of the operation of the selection of a vertex from the candidates list [27]. 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 [28]. 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. When this is the case, labels can take on values from 0 to $(N-1)C$. This gives $(N-1)C+1$ of different values of the labels and, at the same time, buckets that have to be scanned in increasing order until the first non-empty bucket is found. After a given vertex is checked, it is removed from the bucket and scanning in the next iteration starts with this particular bucket. As a result, once emptied bucket is not checked again any more. It happens so because the currently checked vertex always has the lowest (smallest) label from among any other vertices in the buckets and, since lengths of edges are non-negative, while setting labels in a given iteration none of the label will be set to a value that is lower than the value of the label of the vertex that is being checked.

A good structure for the implementation of buckets is the two-way list. The list allows all operations (checking whether a bucket is empty, addition of a vertex and its removal from the bucket) to be performed in time $O(1)$. Taking it all into consideration, the choice of a vertex requires $O(NC)$ operations, which, after taking into account $O(A)$ operations for checking and setting labels, results in the computational complexity of Dial's algorithm being $O(A + NC)$. 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 [17]. 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, i.e., the ones treating all labels for vertex distances as temporary until the last iteration, after which all labels are set to optimum values [29]. 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 false return as the result of its operation. Because of this particular method of operation of the algorithm, the candidates list is not required. The initial conditions are in accordance with (5), though with the omission of the list V. The algorithms checks all edges of the graph $N-1$ times, which allows the minimum labels in the graph to be propagated. In its final stage, the algorithm checks whether any of the labels is non-optimum — this situation happens only in the case of the occurrence of a negative cycle in the graph, which is reported by the algorithm by yielding false returns. 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 [30]. 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 placed in this list. If this is the case, it is added to its head, otherwise — to the end of the list. This is caused by the fact that a modification of the label of vertex i can be followed by a modification of vertices j such that $(i,j) \in \mathcal{A}$. A quicker updating of the vertices in which the edge that starts in the considered vertex terminates effects in the optimum of the solution to be quicker achieved. Such an operation of the algorithm results in its good results in practice. There are instances, however, when the algorithm completely cannot cope with, and the number of additions of some vertices to the candidates list is non-polynomial.

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 [31]. 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. The algorithms are designed for graphs with non-negative edges, though they also operate otherwise (there is no guarantee then that they will perform better).

In each of its iteration the algorithm *SLF* checks the vertex that is placed at the head of the candidates list. The place of the addition of a vertex after its label has been changed depends on the value that is taken on by the label. If the vertex label of the vertex that is to be added to the candidates list is lower or equal to the label of the vertex that is currently at the head of the list, this vertex is added as the first in the list. Otherwise, the vertex is added at the very end of the candidates list.

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 [32]. 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 *LLL* algorithm method for their retrieval from the list [20]. 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. To speed up the process, parallel computing methods can be applied.

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: The algorithm's operation is based on iterative calculation of the shortest paths for all vertices composed of an increasing number of edges [33]. 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. This procedure has some similarity with matrix multiplication [25].

Matrices that are used in the algorithms $D^m = \{d_{ij}^m\}$ and $Pred^m = \{pred_{ij}^m\}$ have the initial values (8a) and (8b), respectively.

$$d_{ij}^1 = \begin{cases} 0, & \text{for } i=j \\ a_{ij}, & \text{for } (i,j) \in \mathcal{A} \\ +\infty, & \text{otherwise} \end{cases} \tag{8a}$$

$$pred_{ij}^1 = \begin{cases} 0, & \text{for } i=j \\ i, & \text{for } (i,j) \in \mathcal{A} \\ 0, & \text{otherwise} \end{cases} \tag{8b}$$

In its simplest case, the matrix D^1 would be multiplied by itself $N-2$ times to take into account paths that have $1, 2, \ldots, N-1$ edges. The matrices that correspond to particular iterations would be as in (9), though such a case would effect in the complexity at the level $\Theta(N^4)$.

$$\begin{aligned} &D^1 \\ &D^2 &&= D^1 \cdot D^1 \\ &D^3 &&= D^2 \cdot D^1 \\ & &&\vdots \\ &D^{(N-1)} &&= D^{(N-2)} \cdot D^1 \end{aligned} \tag{9}$$

The knowledge of the values of all matrices D is not, however, necessary — it is the matrix $D^{(N-1)}$ that needs special attention. Hence, the doubling algorithm, instead calculating successive matrices D, calculates only its powers of 2 [see (10)].

$$\begin{aligned} &d_{ij}^{2^m} = \min_k\{d_{ik}^{2^{(m-1)}} + d_{kj}^{2^{(m-1)}}\}, \\ &i, j, k \in \mathcal{N},\ m = 1, 2, \ldots, \lceil \log_2(N-1) \rceil \end{aligned} \tag{10}$$

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 $n \geq N-1$. In such a case we have a sequence of matrices [see (11)].

$$\begin{aligned} &D^1 \\ &D^2 &&= D^1 \cdot D^1 \\ &D^4 &&= D^2 \cdot D^2 \\ & &&\vdots \\ &D^{2^{\lceil \log_2(N-1) \rceil}} &&= D^{2^{\lceil \log_2(N-1) \rceil - 1}} \cdot D^{2^{\lceil \log_2(N-1) \rceil - 1}} \end{aligned} \tag{11}$$

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 $\Theta(N^3)$ [34], [35]. The algorithm analyses the internal vertices of the path $P = (i, n_1, n_2, \ldots, n_k, j)$, i.e., those that are neither the initial (original) vertex nor the goal vertex. For the given path P, these are the vertices from the set $\{n_1, n_2, \ldots, n_k\}$.

Assuming that $\mathcal{N} = \{1, 2, \ldots, N\}$, for a certain k, let us consider the sub-set $\{1, 2, \ldots, k\}$ and all paths from i to j whose internal vertices belong to this sub-set, for each pair of the vertices $i, j \in \mathcal{A}$. From among all the paths we denote the shortest path as P. The assumption is that the graph does not include non-negative cycles and, thus, this path is a simple path (does not have repeated vertices or edges). We analyse this path against the shortest paths from i to j that have the set of internal vertices limited to the sub-set $\{1, 2, \ldots, k-1\}$.

Depending on whether k is an internal vertex of path P, we can draw the following conclusions. If k is not an internal vertex of the path P, then it means that its internal vertices are limited to the sub-set $\{1, 2, \ldots, k-1\}$. Then, P is the shortest path also when the set of its internal vertices is equal to $\{1, 2, \ldots, k\}$. Intuitively, this means that an expansion of the set of internal vertices does not change the shortest path. If, however, k is the internal vertex of the path P, we divide it into two paths P_1 from i to k and P_2 from k to j. Both paths are the shortest paths, while the set of their vertices is limited to the sub-set $\{1, 2, \ldots, k-1\}$, since vertex k is the goal vertex of path P_1 and the initial vertex of path P_2. This means, in turn, that by dividing the path into two shortest paths we can limit the set of their internal vertices. In both cases we obtain the shortest path for a given k using the shortest path for $k-1$.

The matrices used in this algorithm are exactly the same as the matrices used in the doubling algorithm. The initial values for these matrices are identical with equation (5) with the only difference that, instead d_{ij}^1 and $pred_{ij}^1$, we define respectively d_{ij}^0 and $pred_{ij}^0$.

On the basis of the earlier considerations and initial conditions we obtain the recurrent formula (12).

$$d_{ij}^k = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}, \qquad i, j, k \in \mathcal{N} \tag{12}$$

The formula illustrates precisely in how the length of the shortest path is dependent on whether k is its internal vertex and whether uses the values obtained for $k-1$.

3) Johnson's 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's algorithm [36]. 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. Therefore, the Johnson's algorithm has a number of particular properties and limitations of both algorithms. It can determine whether a graph includes a negative cycle (just like the Bellman-Ford algorithm) and requires non-negative lengths of the edge (like Dijkstra's algorithm). Getting round this limitation is possible thanks to an appropriate transformation of the graph presented in (13).

In the initial stage of the Johnson's algorithm, the graph is being modified in order to get rid of edges with negative lengths. Such a transformation has to guarantee additionally that the shortest paths in the graph do not change. To achieve that, the graph is being added the additional vertex s that is to be the initial vertex for the Bellman-Ford algorithm. To each of the earlier vertices the edge that starts in s with the length equal to 0 is also added.

$$\begin{gathered} \mathcal{G}' = (\mathcal{N}', \mathcal{A}') \\ \mathcal{N}' = \mathcal{N} \cup \{s\}, \qquad s \notin \mathcal{N} \\ \mathcal{A}' = \mathcal{A} \cup \{(s, j) : j \in \mathcal{N}\} \\ a' = a, \qquad a'_{sj} = 0, \qquad \forall\, j \in \mathcal{N} \end{gathered} \tag{13}$$

Thus created graph $\mathcal{G}'$ has no paths that would include the vertex s except those that start in it, and includes negative cycles only when the graph $\mathcal{G}$ has included such cycles. If the graph $\mathcal{G}$ does not include negative cycles, then, after the execution the Bellman-Ford algorithm on the graph $\mathcal{G}'$ with the vertex s as the initial vertex, we obtain the vector h that defines the lengths of the shortest paths in this graph. The vector is then used to modify the lengths of edges in such a way as not to make them non-negative, in line with (14).

$$a'_{ij} = a'_{ij} + h_i - h_j, \qquad i, j \in \mathcal{A}' \tag{14}$$

Then, for each vertex i that belongs to $\mathcal{A}$ Dijkstra's algorithm is applied to calculate all the shortest paths that start in it. After the calculation of the lengths of the paths that start in a given vertex, they are modified in such a way as to reflect and correspond to the lengths of paths in the original graph [see (15)].

$$d_{ij} = d'_j + h_j - h_i, \qquad i, j \in \mathcal{A} \tag{15}$$

In this way, the matrix $D = \{d_{ij}\}$ is obtained. The matrix includes the lengths of the paths between all pairs of the vertices.

In the Johnson's algorithm, Dijkstra's algorithm is performed N times and it is the latter algorithm that significantly influences the computational complexity of the whole algorithm. 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 $O(NA \log N)$.

VI. 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, 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)

Therefore, 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 (all edges between the same pair of verices)
- 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

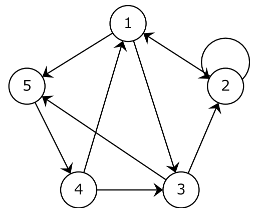

Figure 1. Exemplary directed graph

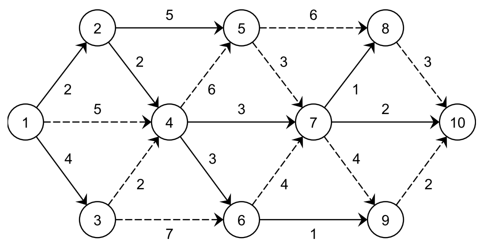

Figure 2. Manually created *custom 1* graph in which the edges marked with the solid line create a shortest paths tree with the root in node 1

TABLE I. STRUCTURE OF THE GRAPHS USED IN THE SIMULATION

graph	vertices	edges	
		number	lengths
custom 1 (c1)	10	19	$\langle 1, 7\rangle$
custom 2 (c2)	20	43	$\langle 0, 9\rangle$
multistage 1 (ms1)	52	420	$\langle 1, 9\rangle$
multistage 2 (ms2)	86	249	$\langle 1, 10\rangle$
random 1 (r1)	25	125	$\langle 1, 9\rangle$
random 2 (r2)	100	628	$\langle 1, 20\rangle$

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

VII. 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 6 representative graphs described in Table I were selected. Graph *custom 1* shown in Fig. 2 was created manually from 10 vertices that were joined together by 19 edges. The *custom 2* graph was created manually as well and consists of 20 vertices and 43 edges. Another two graphs that were used in the tests are the graphs that are characteristic for a multi-stage shortest path problem. An exemplary graph is presented in Fig. 3. The first multi-stage graph used in the tests, *multistage 1*, has 5 stages, each having 10 vertices. The lengths of edges were generated randomly from within the interval $\langle 1, 9\rangle$. The second multi-stege graph, *multistage 2*, has 30 stages, each having 3 vertices, and therefore, the structure comprises 249 edges. The *random 1* graph was generated randomly, without loops, and with 5 edges coming out of each of the vertices. The last graph under scrutiny, *random 2*, was also generated randomly, without loops, but with 3 to 10 edges coming out of each of the vertices.

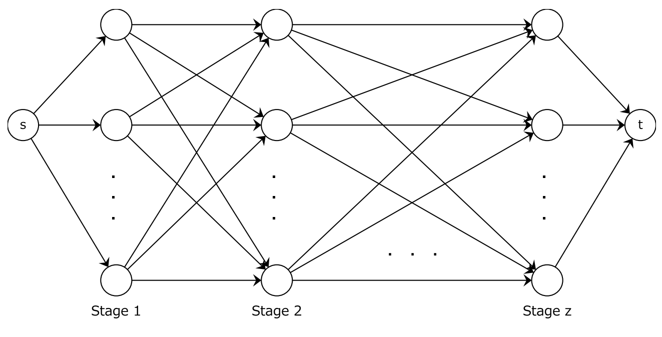

Figure 3. A general structure of a multi-stage graph

VIII. 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 (execution) times of the algorithms tested for the graphs discussed in Section VII. 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 best results for each graph are highlighted in bold text, and the worst are in italics.

The graph *custom 1* 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 next graph the simulations were performed on, *custom 2*, was salved in the group of *SSSP* algorithms in the shortest time by the *SLF* algorithm. Bellman-Ford algorithm and Dijkstra's algorithm with a heap were the slowest ones. The remaining algorithms finished in quite similar times. In the case of *APSP* algorithms Johnson's was the fastest, being slightly better than Floyd-Warshall algorithm and over 3 times faster than the doubling algorithm.

The first graph characteristic for the multi-stage shortest path problem *multistage 1* brought a significant increase in

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

algorithm	graph					
	c1	c2	ms1	ms2	r1	r2
generic	112	127	312	247	163	603
Dijkstra	100	122	324	258	148	405
DijkstraHeap	*146*	176	466	323	200	518
Dial	104	128	322	282	172	395
Bellman-Ford	119	*217*	*3252*	3097	*511*	*8526*
D'Esopo-Pape	113	147	1260	*4171*	239	1222
SLF	**96**	**111**	**262**	**236**	**143**	**376**
LLL	102	121	336	274	155	422
SLF/LLL	112	132	318	288	161	431
doubling alg.	324	*2594*	*47678*	*233517*	*4756*	*364714*
Floyd-Warshall	**184**	1031	16045	70420	**2057**	112880
Johnson	*418*	**879**	**9309**	**12970**	2959	**41904**

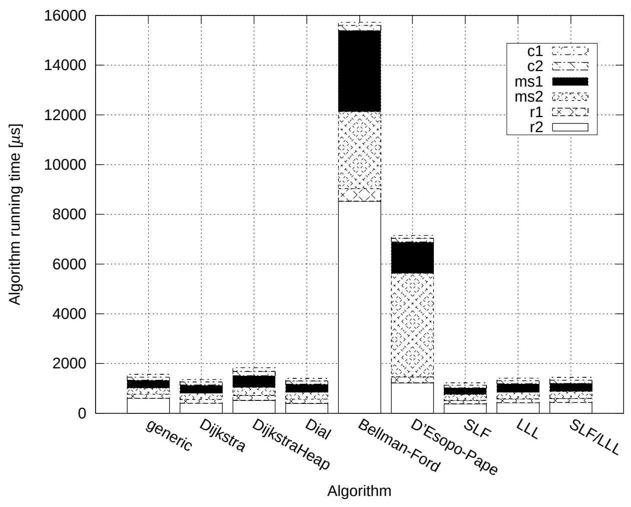

Figure 4. Chart of running (execution) times of the algorithms solving the shortest path problem with one initial vertex (*SSSP*)

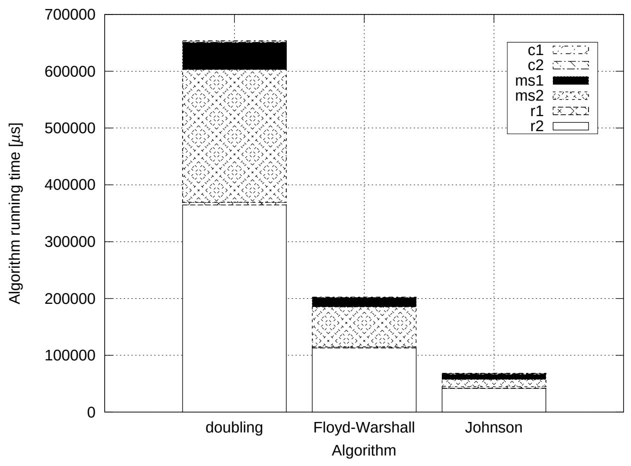

Figure 5. Chart of running (execution) times of the algorithms solving the shortest path problem between all pairs of vertices (*APSP*)

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 exactly as in the case of the previous graph — Johnson's algorithm was the quickest and the doubling algorithm was the slowest, while the distance between Johnson's and Floyd-Warshall algorithms increased.

The *SLF* proved to be the quickest for the *multistage 2* graph, and hence, it was faster than the generic algorithm and the *SLF/LLL* algorithm that came second and third, accordingly. The D'Esopo-Pape and the Bellman-Ford algorithms performed the worst and were 10 to 14 times slower than other algorithms. In the group of *APSP* algorithms the Johnson's algorithm took the shortest time to solve the problem, about 5 times faster than Floyd-Warshall algorithm and about 18 times faster than the doubling algorithm.

Another graph under consideration, *random 1*, 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's algorithm performed slightly worse, while the doubling algorithm was the worst (the longest) of the lot.

The last graph, *random 2*, consists of the highest number of edges. However, it poses no problem for the *SLF* algorithm to solve it in the shortest time in the group of *SSSP* algorithms. Dial and Dijkstra's algorithms gave good results as well. Bellman-Ford algorithm operated clearly longer as compared to the rest of the algorithms. The D'Esopo-Pape algorithm was also slow once again. If we take a look at the *APSP* algorithm the results are analogical to the most of the prior results. Johnson's algorithm is the most time-efficient with the Floyd-Warshall algorithm being almost 3 times slower and the doubling algorithm to be far behind, and thus, the last one in the race.

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 execution 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 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 remaining graphs 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's algorithm that overall turned out to be the fastest one.

IX. Conclusion

This article provides a detailed presentation of 12 algorithms solving the shortest path problem and presents an analysis of their performance. 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 Dijkstra's 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 Johnson's algorithm.

In addition to the presentation of run-time relationships between the algorithms, the study indicates 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 that is to be used. Moreover, it is worthwhile to remember that 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.

Future work could focus on conducting experiments in larger graphs, including those obtained by using Internet-like topology generators, and in the structures that reflect the relationships in the society or complex databases. The operation in graphs with a power-law distribution of node degrees may prove to be interesting and useful as well. Furthermore, the performance measured and evaluated in FLOPS (FLoating point Operations Per Second), instead of average running time, may give a new and broader insight into the problem.

Acknowledgement

This work has been partially supported by a grant from Switzerland through the Swiss Contribution to the enlarged European Union (PSPB-146/2010, CARMNET).

References

[1] M. Głąbowski, B. Musznicki, P. Nowak, and P. Zwierzykowski, "Efficiency evaluation of shortest path algorithms," in Proceedings of AICT 2013, The Ninth Advanced International Conference on Telecommunications, Rome, Italy, 23–28 June 2013, pp. 154–160.

[2] 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.

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

[4] 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.

[5] M. Głąbowski, B. Musznicki, P. Nowak, and P. Zwierzykowski, "An in-depth discussion of challenges related to solving shortest path problems using ShortestPathACO based algorithms," in Information Systems Architecture and Technology; Knowledge Based Approach to the Design, Control and Decision Support, J. Świątek, L. Borzemski, A. Grzech, and Z. Wilimowska, Eds. Wrocław, Poland: Oficyna Wydawnicza Politechniki Wrocławskiej, 2013, pp. 77–88.

[6] 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 Trans-disciplinary Issues in Computer Science and Engineering, vol. 2, no. 1, January 2006, pp. 62–67.

[7] 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.

[8] 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.

[9] M. Piechowiak, P. Zwierzykowski, and M. Stasiak, "Multicast routing algorithm for packet networks with the application of the lagrange relaxation," in *Proceedings of NETWORKS 2010, 14th International Telecommunications Network Strategy and Planning Symposium*, Warsaw, Poland, September 2010, pp. 197–202.

[10] 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.

[11] 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.

[12] 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.

[13] J. Hershberger, S. Suri, and A. Bhosle, "On the difficulty of some shortest path problems," ACM Transactions on Algorithms, vol. 3, no. 1, 2007.

[14] 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.

[15] L. Roditty and U. Zwick, "On dynamic shortest paths problems," Algorithmica, vol. 61, no. 2, 2011, pp. 389–401.

[16] B. L. Golden, "Shortest path algorithms: a comparison," Massachusetts Institute of Technology, Operations Research Center, Tech. Rep., October 1975.

[17] 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.

[18] 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.

[19] 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.

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

[21] R. K. Ahuja, T. L. Magnati, and J. B. Orlin, Network flows: Theory, algorithms and applications. Englewood Cliffs, N.J.: Prentice-Hall, 1993.

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

[23] 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.

[24] F. B. Zhan and C. E. Noon, "A comparison between label-setting and label-correcting algorithms for computing one-to-one shortest paths," Journal of Geographic Information and Decision Analysis 4, 2000.

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

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

[27] S. Saunders, "A comparison of data structures for Dijkstra's single source shortest path algorithm," Honours project, University of Canterbury, Department of Computer Science and Software Engineering, 5 November 1999.

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

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

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

[31] D. P. Bertsekas, "A simple and fast label correcting algorithm for shortest paths," Networks, vol. 23, 1993, pp. 703–709.

[32] 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.

[33] E. Dekel, D. Nassimi, and S. Sahni, "Parallel matrix and graph algorithms," SIAM Journal on Computing, vol. 10, no. 4, 1981, pp. 657–675.

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

[35] S. Warshall, "A theorem on boolean matrices," Journal of the ACM, vol. 9, January 1962, pp. 11–12.

[36] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," Journal of the ACM, vol. 24, January 1977, pp. 1–13.