Asia-Pacific Journal of Operational Research
Vol. 30, No. 2 (2013) 1250054 (25 pages)

DOI: 10.1142/S0217595912500546

A NOVEL LINEAR ALGORITHM FOR SHORTEST PATHS IN NETWORKS

DRAGAN VASILJEVIC

Department of Operations Management
Faculty of Organizational Sciences, University of Belgrade, Serbia
vasiljevic@fon.bg.ac.rs

MILOS DANILOVIC*

Department of Operations Management
Faculty of Organizational Sciences, University of Belgrade, Serbia
danilovicm@fon.bg.ac.rs

Published 23 January 2013

We present two new linear algorithms for the single source shortest paths problem. The worst case running time of the first algorithm is $O(m + C \log C)$, where m is the number of edges of the input network and C is the ratio of the largest and the smallest edge weight. The pseudo-polynomial character of the time dependence can be overcome by the fact that Dijkstra's kind of shortest paths algorithms can be implemented "from the middle", when the shortest paths to the source are known in advance for a subset of the network vertices. This allows the processing of a subset of the edges with the proposed algorithm and processing of the rest of the edges with any Dijkstra's kind algorithm afterwards. Partial implementation of the algorithm enabled the construction of a second, highly efficient and simple linear algorithm. The proposed algorithm is efficient for all classes of networks and extremely efficient for networks with small C. The decision which classes of networks are most suitable for the proposed approach can be made based on simple parameters. Experimental efficiency analysis shows that this approach significantly reduces total computing time.

Keywords: Single source problem; priority queues; weight ratio; heaps; buckets.

1. Introduction

The shortest path problem is network N together with the set ψ of pairs of network vertices between which shortest paths are to be evaluated. Depending on the ψ, different groups of problems can be distinguished. Single Source Shortest Path problem (*SSSPp*) is determining the shortest paths from a given source vertex s to all other vertices, $\psi_s = \{(s, v) \mid v \in V\}$. *SSSPp* is a core problem among the shortest paths problems and is one of the oldest fundamental problems in the algorithm theory. Since 1959 almost all developments concerning this problem have evolved around the famous Dijkstra's algorithm (Dijkstra, 1959) (hereinafter Dij-Alg stands for Dijkstra's algorithm or some of its modifications). *SSSPp* appears in countless practical applications.

*Corresponding author

Known results: The original version of Dij-Alg ran in $O(m + n^2)$ time, where m is number of edges and n is number of vertices in a network. So far main efforts in solving the shortest paths problems have aimed to linearize the quadratic term, usually by using a priority queue (Johnson, 1975, 1977; Raman, 1996, 1997), a data structure that maintains a set of elements and supports insert, decrease-key and extract-min operations. Different implementations of priority queues use heaps and buckets. Among different heaps, the algorithm of Fredman and Tarjan (1987) using Fibonacci heap has the best average time dependence $O(m + n \log n)$ while operation times for bucket-based implementations (Ahuja *et al.*, 1990; Cherkasky *et al.*, 1996; Denardo and Fox, 1979; Dial, 1969) depend on the ratio between the largest edge weight and the smallest non-zero edge weight, C and in general case are $O(m+n \log C)$. Mikkel Thorup algorithm (1999, 2000) has $O(m)$ time in a word RAM model and uses the component hierarchy, which is generated with linear-time pre-processing. The shortest paths problems can also be solved by quantum algorithms (Aghaei *et al.*, 2009), which can do several operations simultaneously due to their wave-like properties, but must be implemented on special quantum computers.

Physical complexity of algorithms and selection of hardware on which they can be implemented resulted in the need to find new, simple procedures for solving the problem in general, or to develop specialized algorithms, which would be efficient for certain classes of networks. Various speed-up techniques have been proposed based on some underlying network properties (Lauther, 2004; Sedgewick and Vitter, 1986; Wagner and Willhalm, 2003). Recently, published works are more concerned with multiple single-source single-target shortest paths computations using natural hierarchical decompositions or pre-processing (Bast *et al.*, 2007; Goldberg *et al.*, 2006; Kohler *et al.*, 2005; Mohring *et al.*, 2005; Sanders and Schultes, 2005; Schulz *et al.*, 2002). A special effort has been invested to evaluate combinations of various techniques (Bauer and Delling, 2008; Holzer *et al.*, 2005).

Our results: In Dij-Alg, one spends $n * O(n)$ time on choosing a vertex, minimizing distance to the source, so a decrease in the running time bound requires a speed-up in finding this minimal distance. Almost all papers on *SSSPp* in last 50 years have moved the field of investigation from the graph theory to the theory of data structures, trying to find this minimal distance in constant time. Dij-Alg searches solution space by increasing the distance from the source. This is obtained by choosing the vertex with the minimum temporary distance from the source in each step, and this procedure is critical for the efficiency of the algorithm. In this work, we present a new algorithm for *SSSPp*, which can be applied on any network with positive edge weights. In the presented algorithm, partitioning of edges is proposed according to their weights, so search of the solution space is performed in "wave-fronts" in which temporary distances from the source belong to the same weight group. The worst case running time of the algorithm is $O(m + C \log C)$. Pseudo-polynomial term $C \log C$, can be replaced with a constant, due to the fact that Dij-Alg can be implemented "from the middle" in the situations when the

shortest paths to the source are known in advance for a subset of the network vertices. Linear procedure for transition to Dij-Alg is presented and statement is proved that, when data obtained with this procedure are applied in Dij-Alg, its time of execution is equal to the time it spends on an instance whose size is equal to the number of unexplored vertices. Thus the algorithms based on this approach are, in the worst case, as efficient as the best-known algorithms and in some cases they are much better. Partial implementation of the algorithm enabled the construction of a second, highly efficient and simple linear algorithm, which outperformed known algorithms in all tested instances.

The paper is organized as follows. Section 2 gives some definitions and abbreviations, which are used in this paper. Section 3 presents the structure of Dij-Alg and presents partial Dij-Alg and proves the statement that it gives correct shortest paths from the source to all the network vertices. In Sec. 4, computational complexity of partial Dij-Alg is analyzed. The hypothesis, that its running time is equal to the running time of the Dij-Alg is used in the instance of order $n - K$, where K is the number of vertices for which the shortest paths to the source are known, is proved. In Sec. 5 statements, which determine the composition and properties of the proposed partitioning of edges, are proved. Section 6 describes the proposed algorithms and Sec. 7 outlines algorithm's time complexities. Experimental results are presented in Sec. 8. Finally, Sec. 9 discusses important advantages of this approach and performance gains due to special instances of the problem.

In this paper, we have limited our discussion to networks with no negative arc weights, so presented solutions refer to non-negative single-source shortest path problem, *NSSSPp*.

2. Preliminaries

Graph $G = (V, E)$ consists of a finite and non-empty set V of the vertices and a set of edges $E \subseteq \{(v_i, v_j) \,|\, v_i, v_j \in V,\ v_i \neq v_j\}$. Network $N = (G, w)$ is a graph G together with a function w, which joins vectors or other functions to its vertices and/or edges. In the shortest paths problems, w is a real-valued function $w : E \to \mathbb{R}$. Real number $w(v_i, v_j)$ or $w(e_{ij})$ is a weight of edge, $e_{ij} = (v_i, v_j) \in V$. The ratio of the largest edge weight $w^N_{\max}$ to the smallest non-zero edge weight $w^N_{\min}$ in N is network weight ratio C. Definitions of other standard terms in the graph theory can be found in a well-known reference given under (Ahuja *et al.*, 1993). Several abbreviations and definitions of terms less often encountered will be given here.

A path PG of length k in G is a finite sequence of vertices and edges: $v_0, e_1, v_1, e_2, \ldots, e_k, v_k$, provided that $e_i (i = 2, 3, \ldots, k)$, starts in v_{i-1} (which is an endpoint for e_{i-1}) and ends in v_i (which is a starting point for e_{i+1}). Some nodes may be repeated in a path, but if they are not, the path is elementary. A path is usually given by the sequence of nodes: $v_0, v_1, \ldots, v_k$, where adjacent vertices in sequence are adjacent in the path, too. The vertices v_0 and v_k are linked by PG_{0k} and are called its end vertices or ends. The distance $d_G(v_i, v_j)$ in G of two vertices v_i and v_j is the length of a shortest path PG^*_{ij} in G; if no such path exists, we set

$d_G(v_i, v_j) = \infty$. The first vertex after v in the shortest path is first descendant of v in G, $\mathrm{dsc}_1(v)$, the next one is second descendant, $\mathrm{dsc}_2(v)$ and so on. Likewise, vertices before v are, first ancestor of v in G, $\mathrm{anc}_1(v)$, second ancestor $\mathrm{anc}_2(v)$ and so on. Denote by G_d^i the sub-graph of G consisting of $\mathrm{dsc}_i(v)$. The eccentricity of v_i is $\varepsilon(v_i) = \max d_G(v_i, v_j)$ over all v_j in G. Radius G is $r(G) = \min \varepsilon_G(v)$ over all v in G, and the diameter $D(G) = \max(\varepsilon_G(v))$ over all v in G.

Path from s to t in a network N is a path $P_{st} = PG_{st}$, whose weight function w extends in the following manner:

$$w_{st} = \sum_{e \in P_{st}} w(e),$$

where the sum is over all edges in path P_{st}. A shortest path from s to t in a network N is path P_{st}^* for which the path weight $w(P_{st})$ is minimal in comparison to weights of all paths from s to t. If all cycles in the network have positive weight, P_{st}^* is elementary. In addition it could be inferred that any sub-path of a shortest path is the shortest path for its end vertices. The distance $d_N(s, t)$ in N of two vertices s and t is $w(P_{st}^*)$. The objective of the *NSSSPp* is to find a set $NSSSP = \{P_{st}^*,\ \forall t \in V\}$.

3. Partial use of Dij-Alg

Known *SSSPp* algorithms for networks with no negative edge weights belong to the group of very efficient exact algorithms whose main task is to linearize their time complexity. In such cases, it is very important to define each step in detail as well as the maximum time they needed to solve a problem instance of size n. The general structure of Dij-Alg, given by Johnson (1977), was basic for detailed representation of the Algorithm 1.

Algorithm 1. Dij-Alg for solving (N, ψ_s) problem

```
1  INPUT: w(a,b), ∀(a,b) ∈ E
2  d(v) = ∞ ∀v ≠ s
3  d(s) = 0, anc(s) = null
4  TS = {s}, SS = ∅
5  while TS ≠ ∅ do
6      begin
7          Choose u ∈ TS
8          TS = TS\{u}, SS = SS ∪ {u}
9          for each (u,v) ∈ E and v ∉ SS do
10           if d(v) > d(u) + w(u,v) then
11             begin
12               d(v) = d(u) + w(u,v), anc(v) = u
13               TS = TS ∪ {v}
14             end
15     end
16 OUTPUT: (d(v), anc(v)), ∀v ∈ V
```

In Step 7, u with the minimal $d(u)$ is chosen, when all edge weights are non-negative. At the end of this procedure:

$$d(v) = d_N(s, v), \quad \forall\, v \in V, \tag{1}$$

$$\mathrm{anc}(v) = \mathrm{anc}_{P^*_{sv}}(v). \tag{2}$$

Almost all known Dij-Algs differ only in Steps 7 and 13, i.e., in defining the queue together with certain procedures that insert a new element into the queue, delete a specified element, and find and delete an element that minimizes priority in the queue. Critical Steps 7 and 8 are executed n times, i.e., choice of a vertex is performed n times.

For many instances of the *SSSPp* that arise in practice, the networks have some underlying properties, such as special topological or geometric structure or special edge weight distribution, which could be exploited to speed-up the computations. On the other hand, Dij-Alg is almost perfect, so the question arises whether Dij-Alg could be applied "from the middle" in the situations when some of the shortest paths are known in advance or whether they could be computed in a more efficient way.

Dij-Alg maintains three sets for keeping track of vertices: *the solution set* (SS) of vertices for which the shortest distance has been already computed, together with these distances to source and the ancestors of this vertices, *a temporary set* (TS), which holds vertices that have associated currently best distance but have no determined shortest distance and can be arranged in some heap, and *the set of unexplored vertices* (US), i.e., vertices which are not in SS or TS. If a subset $\{u_i\}$ of the network vertices for which their shortest paths to the source s are known to exists, there are enough data to construct SS, TS and US in a unique way. SS consists of $\{u_i\} \cup \{s\}$ together with uniquely defined $d(u_i)$ and $\mathrm{anc}(u_i)$. TS holds neighbors $\{v_j\}$ of vertices in SS, which are not in SS. Temporary distances $d(v_j)$ and ancestors $\mathrm{anc}(v_j)$ for vertices in TS are computed in the procedure TRANSIT (Algorithm 2).

Procedure TRANSIT can be incorporated now in the shortest path algorithm (Algorithm 3).

Theorem 1. *The function* $d(v)$ *obtained by Algorithm* 3 *gives the distances from* s *and the function* $\mathrm{anc}(v)$ *gives the ancestor of* v *in the shortest path from* s *to* v.

Proof. Through the iterations in Algorithm 3, it holds $d(v) \geq d_N(s, v),\ \forall\, v \in V$. We prove that through the iterations:

$$d(v) = d_N(s, v), \quad \forall\, v \in \mathrm{SS}. \tag{3}$$

– At the start of the algorithm SS holds vertices from the input shortest paths, so (3) is satisfied for any $v \in \mathrm{SS}$;
– The execution of TRANSIT, does not change SS;

Algorithm 2. Procedure TRANSIT

```
1  INPUT: SS, w(a,b), ∀(a,b) ∈ E
2  d(v) = ∞ ∀v ∉ SS
3  TS = ∅
4  for each u ∈ SS
5    for each {(u,v) | ((u,v) ∈ E, v ∉ SS)}
6      begin
7          if d(v) = ∞ then (TS = TS ∪ {v}, go to Step 10)
8          if d(v) > d(u) + w(u,v)
9              begin
10               d(v) = d(u) + w(u,v)
11               anc(v) = u
12             end
13     end
14 end
15 OUTPUT: TS, (d(v), anc(v)), ∀v ∈ TS
```

Algorithm 3. Partial-Dij-Alg for solving (N, ψ_s) problem

```
1  INPUT: SS, w(a,b), ∀(a,b) ∈ E
2  apply TRANSIT (SS), obtain TS, (d(v), anc(v)), ∀v ∈ TS
3  while TS ≠ ∅ do
4      begin
5          Choose u ∈ TS | d(u) = min(d(t), ∀t ∈ TS)
6          TS = TS\{u}, SS = SS ∪ {u}
7          for each (u,v) ∈ E and v ∉ SS do
8            if d(v) > d(u) + w(u,v) then
9              begin
10               d(v) = d(u) + w(u,v), anc(v) = u
11               TS = TS ∪ {v}
12             end
13     end
14 OUTPUT: (d(v), anc(v)), ∀v ∈ V
```

– The only change of SS in the rest of the algorithm is performed in Step 6:

$$\mathrm{TS} = \mathrm{TS}\backslash\{u\}, \quad \mathrm{SS} = \mathrm{SS} \cup \{u\},$$

for the $u \in \mathrm{TS}$, chosen by:

$$d(u) = \min(d(t),\ \forall t \in \mathrm{TS}), \tag{4}$$

so it suffices to show that $d(u) = d_N(s, u)$. Suppose u is the first vertex through the iterations with:

$$d(u) > d_N(s, u). \tag{5}$$

According to:

$$d(v) = d_N(s, \text{anc}(v)) + w(\text{anc}(v), v), \quad \text{anc}(v) \in \text{SS}, \tag{6}$$

and the rule that SS has no loops, this implies that P^*_{su} exists with some vertices, other than u which are not in SS. For the same reason, these vertices must be the last ones in the sequence of the path vertices. Let $P^*_{su} = (s, v_1, v_2, \ldots, v_k, u)$ and let v_i be the first vertex in the sequence not in SS. According to the rule that any sub-path of a shortest path is the shortest path for its end vertices, we have $P^*_{sv_i} = (s, v_1, v_2, \ldots, v_i)$ and $d(v_i) = d_N(s, v_i)$. Referring to (5), this implies $d(v_i) = d_N(s, v_i) \leq d_N(s, u) < d(u)$. If $v_i \in$ TS this contradicts (4). On the other hand, $v_i \notin$ US, because $d(v_i) \geq d(\text{anc}(v_i)) \geq d(\text{anc2}(v_i)) \geq \ldots$ and some of the ancestors of v_i belong to TS. This completes the proof. □

Theorem 1 has two important corollaries: (i) Composition of SS can be arbitrary and (ii) TS obtained by TRANSIT holds all necessary information for further steps of partial Dij-Alg.

Dij-Alg expands outwards from source s, expanding steadily the network region for which distances and the shortest paths are known. This expansion should be orderly, firstly incorporating the closest vertices and then moving on to those farther away. Figure 1 presents the structures of SS, TS and US before executing Step 7 in one of the Algorithm 1 loops. Bold lines represent final shortest paths and normal lines represent edges, which connect vertices from SS and TS. There are no edges between two vertices in TS and there are no loops in the sub-graph composed of the vertices from SS and TS. Other edges are presented as dashed lines.

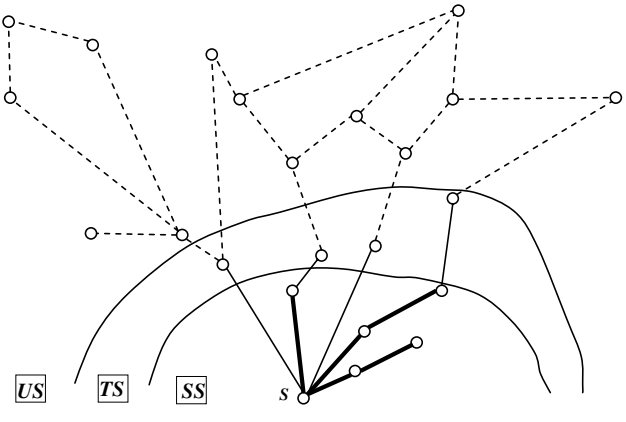

Fig. 1. Composition of SS, TS and US in Dij-Alg.

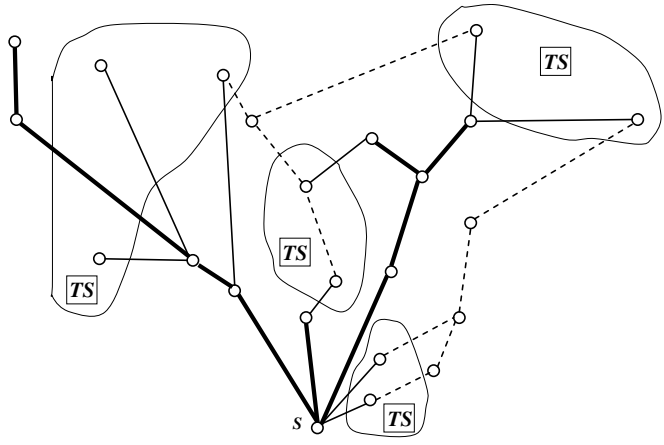

Fig. 2. Composition of SS, TS and US in Partial-Dij-Alg.

On the other hand, when some of shortest paths are known in advance, SS holds vertices with arbitrary distances to the source. After execution of TRANSIT, every temporary distance $d(v)$ of the vertices in TS is the smallest one, compared to all $d(u) + w(u, v)$ distances, where u is anc(v) from SS and where $d(u)$ is the final smallest distance from u to the source.

Figure 2 presents the composition of TS after execution of TRANSIT when SS is composed of the paths, which are drawn in bold lines. Some edges are missing from Fig. 1 because a vertex from TS can be connected with a vertex from SS with one edge only. On the other hand, for the same reason, some edges from Fig. 2 are missing in Fig. 1.

Although different, in both cases compositions of SS affect further execution of algorithm through TS only indirectly. After execution of TRANSIT, for each vertex v from TS holds:

$$d(v) = \min(d_N(s, \text{anc}(v)) + w(\text{anc}(v), v)), \quad \forall\, (\text{anc}(v), v) \in \text{E} \,|\, \text{anc}(v) \in \text{SS},$$

so anc(v) is the only link, which connects SS and TS. The conclusion is that SS may contain arbitrary shortest paths from s, if TS is obtained by TRANSIT.

The shortest paths to the source are known for all the vertices in SS. Therefore, according to the rule that every sub-path of a shortest path is the shortest path for its end vertices, SS holds all the vertices contained in any shortest path of the vertices in SS. Thus, use of any shortest path algorithm on these vertices would not change their distance to the source. Additional data for the vertices in SS, $d(u_i)$ and anc(u_i), are the same as those obtained by Dij-Alg. The TS holds all neighbors of SS vertices, which are not in SS. Additional data for TS — namely, current temporary distances $d(v)$, $v \in$ TS, after execution of TRANSIT, may differ from the temporary distances $d(v)$ obtained by Dij-Alg, according to (6), only if anc(v) is different in those two procedures. After completing TRANSIT procedure, partial Dij-Alg starts

from a TS vertex v that has the minimum $d(v)$, proceeding afterwards in the same way as any other Dij-Alg, with an exception that it "skips" the vertices which are already in SS, so, at the end of the partial Dij-Alg, all the $d(v)$ values which are different in two procedures become equal $d_N(s, v)$ values. This completes the proof that TS obtained by TRANSIT holds all necessary information for further steps of the partial Dij-Alg.

4. Computational Complexity of Partial Dij-Alg

The time complexity of Algorithm A, as a function which maps natural number n to the maximal time $\mathrm{T}^{\mathrm{A}}(n)$ needed by Algorithm A for solving a problem instance of size n, can be measured several ways. Here we have used the unit time model in which $\mathrm{T}^{\mathrm{A}}(n)$ is measured by the number of elementary arithmetic operations that the algorithm performs. Actually, in a network algorithm, T^{A} is a function of n and m, where m varies from $m \sim n$ to $m \sim n^2$ for connected networks without parallel edges. Depending on the model of computation, the set of elementary operations varies, so in the word RAM model which needs array addressing, unit-time word operations are: Addition, subtraction, comparison and arbitrary shifts, i.e., multiplication and division by powers of two. In strong RAM model certain other word operations take unit time, like ACO operations and multiplication.

In the analysis of the complexity of Dij-Alg and partial Dij-Alg, we shall denote by T^{Ak} the complexity of Algorithm k.

Dij-Alg: Algorithm 1 has loop L_1 between Steps 6 and 15 with n iterations and loop L_2, nested in L_1, between Steps 10 and 14 which is executed $\deg(u)$ times, where $\deg(u)$ is a degree of a vertex u. As a result, the loops cost $O(\sum \deg(v_i))$, where the sum is over all network vertices, so loop complexity is $O(m)$. Step 1 is executed n times, while all other Steps except 7, 8, 12 and 13 are $O(1)$, which gives a total running time of:

$$O(n * O(\text{Steps 7 and 8}) + m * O(\text{Steps 12 and 13})). \tag{7}$$

If SS and TS are implemented to allow constant time additions and deletions, for example as doubly linked lists, Steps 8, 12 and 13 are $O(1)$, so:

$$\mathrm{T}^{\mathrm{A1}}(m, n) = O(m + n * O(\text{Step 7})).$$

Step 7 is crucial and almost all papers on this problem in last 50 years invested effort to make $O(\text{Step 7})$ constant. Selected vertex u, in some of them (Goldberg, 2001) has not necessarily a current minimal d, but always $d(u)$ is equal to the correct distance $d_N(s, u)$. It was mentioned in the introduction of this paper that of different heaps, the algorithm of Fredman and Tarjan using Fibonacci heap has the best average time dependence $O(m + n \log n)$ and that operation times for bucket-based implementations depend on C and in general case are $O(m + n \log C)$. In general, time dependence can be given as:

$$\mathrm{T}^{\mathrm{A1}}(m, n) = O(m + f(n)). \tag{8}$$

Partial Dij-Alg: Let $G_{SS}(V_{SS}, E_{SS})$, $|V_{SS}| = n_{SS}, |E_{SS}| = m_{SS}$, be a sub-graph of G induced by $V \backslash \text{SS}$ and let E_{ST}, $|E_{ST}| = m_{ST}$ be a set of edges having one vertex in SS and another in TS. Step 5 in Algorithm 3 implies that u with $\min(d(u))$ was chosen from TS, like in original Dij-Alg, but, depending on the circumstances, heaps or buckets can be used in the same way as in any other shortest paths algorithm. If the procedure considers heaps or buckets, they are set in the same way but for a smaller number of vertices — specifically, for n_{SS} vertices. In other words, after executing Algorithm 2, the most suitable algorithm for the shortest paths can be applied. Thus, the time complexity of Algorithm 3 consists of:

$$\mathrm{T}^{\mathrm{A3}} = \mathrm{T}^{\mathrm{A2}} + \mathrm{T}^{\mathrm{PQ}} + \mathrm{T}^{\mathrm{A1}},$$

where T^{A2} is the time complexity of TRANSIT, T^{PQ} is the time complexity of setting priority queue for the rest of the procedure and T^{A1} represents the time complexity of the applied Dij-Alg on the unexplored vertices. As there is no restriction on the type of Dij-Alg applied, one can obviously conclude that T^{A1} stands for the time complexity of the best Dij-Alg on the specified circumstances.

Theorem 2. *Given a network $N = (G, w) = ((V, E), w)$ and a subset of G, $SS = \{P^*_{sv}\}$, which contains some of the shortest paths from the source s, shortest paths $P^*_{sv}, \forall\, v \in V$ can be found in time $O(m_{ST} + m_{SS} + f(n_{SS}))$.*

Proof. One can obviously conclude that the number of steps in TRANSIT is proportional to m_{ST}. Loops in Steps 4 and 5 are executed for every edge with one vertex in SS. As the edges with both end vertices in SS are of no further interest for the algorithm, they can be removed in constant time by a pre-processing procedure, so:

$$O(\mathrm{T}^{\mathrm{A2}}(m_{ST}, |\mathrm{TS}|)) = O(m_{ST}).$$

After completing TRANSIT, the rest of the procedure is performed independently, apart from the vertices currently in SS. As proven above, TS obtained by TRANSIT holds all necessary information for further steps of the partial Dij-Alg, thus, from the complexity point of view, the situation is the same as when original Dij-Alg should be applied to G_{SS}. Additional $\mathrm{T}^{\mathrm{PQ}}(|\mathrm{TS}|)$ needed for creating heaps or buckets has been, in fact, incorporated in $\mathrm{T}^{\mathrm{A1}}(m_{SS}, n_{SS})$ of the applied Dij-Alg. The only difference is that in partial Dij-Alg, insertion of $|\mathrm{TS}|$ vertices into a heap or a bucket is done in advance. Thus,

$$\mathrm{T}^{\mathrm{A3}}(m_{SS}, n_{SS}) = \mathrm{T}^{\mathrm{A2}}(m_{ST}, |\mathrm{TS}|) + \mathrm{T}^{\mathrm{PQ}}(|\mathrm{TS}|) + \mathrm{T}^{\mathrm{A1}}(m_{SS} - m_{ST}, n_{SS})$$

$$O(\mathrm{T}^{\mathrm{PQ}}(|\mathrm{TS}|)) = O(|\mathrm{TS}|) < O(m_{ST})$$

$$O(\mathrm{T}^{\mathrm{A3}}(m_{SS}, n_{SS})) = O(m_{ST}) + O(m_{ST}) + O(\mathrm{T}^{\mathrm{A1}}(m_{SS} - m_{ST}, n_{SS})),$$

and, according to (8):

$$O(\mathrm{T}^{\mathrm{A3}}(m_{SS}, n_{SS})) = O(m_{ST} + m_{SS} + f(n_{SS})), \tag{9}$$

which completes the proof. □

5. Partitioning of Network Vertices and Edges

In this section, we describe the partitioning of network vertices into buckets according to their distances to the source and the partitioning of network edges according to their weights. Our algorithms are based on this bucket implementation and, depending of the instance of the problem, single or multilevel buckets can be applied. It is not the aim of this paper to introduce a new data bucketing structure, so further explanation should use single-level bucketing, noting that the multilevel buckets can be applied whenever it is necessary.

Denote by B_i a bucket $i, i = 1, \ldots, k$, where k is a positive integer. Each B_i corresponds to an interval $[a_i, b_i)$ with $a_{i+1} = b_i$. Denote by BV a bucket which contains a set of vertices and by BE a bucket containing a set of edges. Write $V = (BV_0 \cup BV_1 \cup BV_2 \cup \cdots \cup BV_k)$, where $BV_0 = \{s\}$ and where $a_i \leq d_N(s, v) < b_i, \forall\, v \in BV_i$. Let v^i denote the vertex from BV_i, v^{i+} denotes vertex from $BV_{i+} = (BV_i \cup BV_{i+1} \cup \cdots \cup BV_k)$ and v^{i-} denotes vertex from $BV_{i-} = (s \cup BV_1 \cup BV_2 \cup \cdots \cup BV_{i-1})$. Similarly, $N^i(G^i(V^i, E^i), w^i)$ denotes the sub-network of N consisting of vertices v^i, and likewise for other definitions. Let $\boldsymbol{w^{i+}_{\min}}$ denote $\min(w^{i+})$. We consider a vertex bucket structure having $a_i = b_{i-1}$ and $b_i = a_i + \boldsymbol{w^{i+}_{\min}}$, $b_0 = \boldsymbol{w^{N}_{\min}}$. Similarly, write $E = BE_1 \cup BE_2 \cup \cdots \cup BE_C$, where $a_i \leq w(e^i) < b_i, \forall\, e^i \in BE_i, a_i = \boldsymbol{i} * \boldsymbol{w^{N}_{\min}}$ and $b_i = (i+1) * \boldsymbol{w^{N}_{\min}}$. Finally, denote by $B^i(v) = \{v_a \,|\, e(v, v_a) \in BE_i\}$ and $B^i(BV_k) = \{v_a | e(v_x, v_a) \in BE_i \bigwedge v_x \in BV_k\}$.

The following Theorem 3 characterizes proposed bucketing.

Theorem 3. *For the partitioning of the network vertices into buckets* s, $BV_1, BV_2, \ldots, BV_k$, *the following holds*:

(i) $k \leq \varepsilon(s) * C$;
(ii) $SSSP \cap E^i = \varnothing, i = 1, 2, \ldots, k$;
(iii) $SSSP \cap (v_1^{i-}, v_2^{i+}, v_3^{i-}) = \varnothing,\ \forall\, v^{i-}, \forall\, v^{i+}, i = 1, 2, \ldots, k$.

Proof. (i) For any network holds $d_N(s, v) \leq \varepsilon(s) * w^N_{\max} \,|\, \forall\, v \in V$. Consider relaxation $b_i = a_i + w^N_{\min}$, so $k = \max(d_N(s, v))/w^N_{\min} \leq \varepsilon(s) * w^N_{\max}/w^N_{\min} = \varepsilon(s) * C$. Obviously, narrower bounds for $d_N(s, v)$ and k can be obtained from:

$$d_N(s, v) \leq \sum_{i=1}^{\varepsilon(s)} w_{\max}^{N(G^i_d, w)};$$

(ii) Let $SSSP \cap E^i = (v_1^i, v_2^i)$. That implies $d_N(s, v_1^i) + w(v_1^i, v_2^i) = d_N(s, v_2^i)$ or $d_N(s, v_2^i) + w(v_1^i, v_2^i) = d_N(s, v_1^i)$, i.e., $w(v_1^i, v_2^i) = |d_N(s, v_1^i) - d_N(s, v_2^i)| < b_i - a_i = w^{i+}_{\min}$, which contradicts the bucketing structure;

(iii) Stands for every shortest path, i.e., value of the distance from the source to the vertex which is in between two other vertices in the shortest path sequence must be in between the values of the distances from the source to these vertices, i.e., $d_N(s, v_2^{i+}) < d_N(s, v_1^{i-})$ or $d_N(s, v_2^{i+}) < d_N(s, v_3^{i-})$ which, again, contradicts the proposed bucketing structure. □

If, according to the available hardware, division and multiplication by powers of two are unit time operations and $w_{\min}^N > 0$, any interval $[w_{\min}, w_{\max}]$ can be reduced to $[1, C+1)$ and normalized $\{\bar{w}_i\} = \{\frac{w_i}{w_{\min}^N}\}$ and $\overline{d_N}(s,v) = \{\frac{d_N(s,v)}{w_{\min}^N}\}$ can be considered. In this case, bucket bounds are $a_i = i$ and $b_i = i+1$. Further explanations will consider these normalized bounds and weights, nothing that the original bounds and weights can be used with the same procedures, whenever the normalization is not a linear process. We proceed with Observation 4.

Observation 4. *$i + j \le \overline{d_N}(s, v^i) + \bar{w}(e^j) < i + j + 2$, i.e., the first descendant v_a of the vertex v^i in the shortest path from the source, where $e(v^i, v_a) \in BE_j$ belongs to BV_{i+j} or BV_{i+j+1}.*

As a corollary of Observation 4, we also derive some properties of the first ancestor of v^i in $P^*_{sv^i}$.

Corollary 5. *Let $v_a = \text{anc}_1(v^i)$ in $P^*_{sv^i}$. Then:*

(i) *if $e(v_a, v^i) \in BE_j$ then:*

$$\begin{aligned} &v_a \in (BV_{i-j} \text{ or } BV_{i-j-1}), \quad \text{for } i-j > 1, \\ &v_a \in BV_1, \quad \text{for } i-j = 1, \\ &v_a \in BV_0, \quad \text{for } i = j. \end{aligned} \tag{10}$$

(ii) $d^*(v^i) = \overline{d_N}(s, v_a^j) + \bar{w}(e(v_a^j, v^i)) = \min(\overline{d_N}(s, v^{i-}) + \bar{w}(e(v^{i-}, v^i)))$, $\forall\, e(v^{i-}, v^i) \in E$.

In the proposed algorithm the vertex label $d^*(v^i)$ will denote the shortest normalized distance from s to v^i and $d(v^i)$ the temporary shortest normalized distance through the iterations of the algorithm. According to Observation 4 and Corollary 5, the following Theorem 6 characterizes the main procedure in the proposed algorithm.

Theorem 6. *If all $d^*(v^{i-})$ are known, and $B_i, B_{i+1}, \ldots, B_k$ contains first neighbors v_x^{i+} of v^{i-}, satisfying: $d(v_x^{i+}) = \min(\overline{d_N}(s, \text{anc}(v_x^{i+})) + \bar{w}(e(\text{anc}(v_x^{i+}), v_x^{i+})))$, $\forall\, v^{i-} = \text{anc}(v_x^{i+})$, $\forall\, e(\text{anc}(v_x^{i+}), v_x^{i+})) \in E$, within the same bounds as $BV_i, BV_{i+1}, \ldots, BV_k$, then $B_j = BV_j$ and $d(v^j) = d^*(v^j)$, where B_j is the first non-empty bucket in the sequence $B_i, B_{i+1}, \ldots, B_k$.*

Proof. Knowledge of all $d^*(v^{i-})$ implies that, likewise in Dij-Alg, $v^j = \text{dsc}_1(v_a^{i-})$ having lowest $\overline{d_N}(s, v^{i-}) + \bar{w}(e(v^{i-}v^{i+}))$ determines that $e(v_a^{i-}, v^j)$ is in the shortest

path from the source and that $\overline{d_N}(s, v^j)$ is the lowest label value among all v^{i+}. That further means buckets $B_i, B_{i+1}, \ldots, B_{j-1}$ remain empty all the time. Theorem 6 expands the set of the vertices, which can be determined as the shortest paths vertices, to the whole bucket B_j. According to Theorem 3 (statements (ii) and (iii)), no shortest path contains more than one vertex from B_j, which completes the proof. □

6. A New Algorithm

A new algorithm, for arbitrary network N with positive edge weights is a labeling algorithm with the strategy to expand from the source in wave-fronts, instead of vertex by vertex expansion such as labeling of vertices with Dij-Alg. We assume, without loss of generality, primarily for the simplicity of explanation that available hardware performs division and multiplication by powers of two in unit time, so, during the initialization, algorithm computes normalized $\{\bar{w}_i\}$ for every edge in N. When $w^N_{\min} > 0$, any interval $[w^N_{\min}, w^N_{\max}]$ can be reduced to $[1, C + 1)$.

In this section, two versions of the algorithm are presented, Algorithm 4 with pre-processing and Algorithm 5 without pre-processing.

Algorithm 4: A linear pre-processing procedures can be constructed for grouping and rearranging of the adjacency list of N, so $B^i(v)$ contains the first neighbors of v having normalized edge weights within $[i, i + 1)$, so $e(v, v_a) \in BE_i$, for each $v_a \in B^i(v)$. In general case, multilevel buckets (Cherkasky *et al.*, 1999) can be used, but in cases when the edge ratio is satisfactorily small, simple and adaptive procedures can be constructed. Use of redundant data, such as group of pointers where ith pointer points to the ith bucket in the adjacency list for every vertex and auxiliary vertex state array which stores the state of vertex v_i on the ith position, enable obtaining the following in a constant time:

— Creating buckets of arbitrary widths;
— Reaching all elements in a bucket;
— Tracking the position of the specified bucket, like low-level bucket for each neighbor in the adjacency list.

Generally, first j buckets $B^j(s)$, of the source neighbors can be empty, so integer $fneb = j + 1$, can be used as the input parameter to speed-up computation.

Algorithm, in each Step i, maintains four sets for keeping track of vertices:

— The solution set SS $= \{v^{i-}\}$ of vertices for which the shortest distance has been already computed;
— The final set BV_i of vertices whose normalized distances to the source belongs to $[i, i + 1)$, which holds vertices that have determined shortest distance;
— The temporary set BV_{i+1} of vertices, which holds vertices that have associated currently best distance but have no determined shortest distance;
— Set of unexplored vertices US $= \{v^{(i+2)+}\}$.

Algorithm 4. A new shortest paths algorithm with pre-processing

```
1  INPUT: w̄(v_a, v_b), ∀(v_a, v_b) ∈ E; fneb
2  i = fneb
3  BV_0 = {s}; BV_i = B^i(s)
4  SS = BV_0 ∪ BV_i; US = V\SS
5  d(v) = ∞ ∀v ∉ SS
6  while US ≠ ∅ do
7      BEGIN
8          i+ = 1
9          for k = 1 to i − fneb do
10             begin
11                 for each v ∈ BV_{i−k} | BV_{i−k} ≠ ∅ do
12                     for each u = B^k(v) | u ∈ US do
13                       begin
14                         if d(u) > d(v) + w̄(v, u) then
15                         begin
16                           d(u) = d(v) + w̄(v, u), anc(u) = v
17                           if d(u) < i + 2 then BV_i = BV_i ∪ v
                                 else BV_{i+1} = BV_{i+1} ∪ v
18                         end 15
19                       end 13
20             end 10
21         BV_i = BV_i ∪ B^i(s); change BE width
22         SS = SS ∪ BV_i; US = US\(BV_i ∪ BV_{i+1})
23     END 7
24 OUTPUT: (d_N(s, v), anc(v)), ∀v ∈ V
```

For the clarity of explanation, algorithm is presented in a simplified form. Main intent was to present a structure of the algorithm as simple as possible, so some detailed steps were committed. All steps of the algorithm, which affect the overall efficiency, will be presented in detail in the next part of the paper, within the scope of the complexity analysis.

The strategy of the algorithm is that at each iteration the whole BV_i, instead of a single vertex, is transferred to SS. Based on Theorem 6, BV_i is populated in Steps 17 and 21. Step 17 also populates the bucket BV_{i+1}, so, according to the presented algorithm, situations can occur in which same vertex can be in both buckets. Thanks to the fact that at any iteration the content of only two buckets can be changed, there is no need to remove the vertex from BV_{i+1} when its affiliation to BV_i is determined. Namely, exact content of BV_i is important no sooner than the execution of the Step 22 (when SS is populated, i.e., when all the iterations of the layer i are terminated). At this step, using auxiliary vector, the exact BV_i can

be obtained in a constant time. It should be noted that in this Step exact content of BV_{i+1} is not required.

First six steps of the algorithm populate SS with s and $B^i(s)$, where i is the index of the first non-empty bucket of the source neighbors. For all other vertices, $d(v) = \infty$. The main loop in the algorithm includes Steps 7–23. At the end of each iteration, a new layer is populated with the vertices with determined shortest distances from the source. The loop between Steps 10 and 20 passes through the previously determined buckets and adds edges to the determined shortest paths. The buckets meet conditions (i) and (ii) of the Corollary 5. Step 21 corresponds to (10) and Step 22 is shown only for clarity of the algorithm presentation. In reality, there is no need for storing SS and US, because all the information is already stored in BV. In the Step 21, the width of edge buckets is increased each time the lowest level buckets are emptied. As mentioned above, changing the width of buckets is a unit time operation: k-times wider buckets are obtained when one in k of the pointers is considered.

Algorithm 5: Unlike the Algorithm 4 that maintains only two buckets, BV_i and BV_{i+1}, Algorithm 5 always maintains C buckets. Specifically, in the ith iteration, buckets $B(BV_i) = \{B_1(BV_i), \ldots, vB_C(BV_i)\}$ can be populated, where $B_a(BV_i), a = 1, \ldots, C$, denotes the bucket of vertices u satisfying conditions:

(i) $\bar{w}(v,u) \in [a, a+1)$, where $v \in BV_i$;
(ii) $\bar{w}(v,u)$ has the minimal value among all v from BV_i.

A simple procedure ALGPB(BV_b) for populating buckets $B(BV_b)$, presented below, is the core of the Algorithm 5. This procedure allows trivial formulation of Algorithm 5. It was assumed that all vertices have labels, which are either $d(v) = \infty$ or their normalized temporary distances.

ALGPB(BV_b). Procedure for populating buckets $B(BV_b)$

```
1  INPUT: b; BV_b; w̄(v_i, v_j), ∀(v_i, v_j) ∈ BE_{b+}; d(v), ∀v ∈ V
2  for each v ∈ BV_b do
3      for each u | (v,u) ∈ BE_{b+}, u ∈ US do
4          begin
5              if d(u) > d(v) + w̄(v,u) then
6              d(u) = d(v) + w̄(v,u), anc(u) = v
7              a = ⌊d̄(u)⌋ − b
8              B_a(BV_b) = B_a(BV_b) ∪ u
9              V_{b+a} = V_{b+a} ∪ u
10         end
11 OUTPUT: V_{b+1}, ..., V_{b+C}; (d(v), anc(v)), ∀v ∈ B(BV_b)
```

As in Algorithm 4, some steps are entered here only because of the clarity of explanation. Actually, storing $B(BV_b)$ is not necessary, because all the information exist in $V_{b+1}, \ldots, V_{b+C}$. It should be noted that after execution of ALGPB(BV_b), some buckets $V_{b+1}, \ldots, V_{b+C}$ can be empty.

Now Algorithm 5, without bucketing of adjacency lists in pre-processing can be formulated:

Algorithm 5. A new shortest paths algorithm without pre-processing

```
1  INPUT: w̄(v_a, v_b), ∀(v_a, v_b) ∈ E
2  i = 0
3  BV_0 = {s}
4  SS = BV_0; US = V\SS
5  d(v) = ∞ ∀v ∉ SS
6  while US ≠ ∅ do
7     perform ALGPB(BV_i), change V_{i+1}, ..., V_{i+C}
8     change BE width
9     SS = SS ∪ BV_i; US = US\BV_i
10    i++
11 continue 6
12 OUTPUT: (d_N(s, v), anc(v)), ∀v ∈ V
```

7. Complexity of Proposed Algorithms

As mentioned above, during the initialization, algorithms compute normalized $\{\bar{w}_i\}$ for every edge in N. If arbitrary shifts are unit-time word operations, the position k of the most significant bit of the smallest edge weight determines the normalization value. In that case $\{\bar{w}_i\} \neq \{\frac{w_i}{w^N_{\min}}\}$, but $\{\bar{w}_i\} = \{\frac{w_i}{2^{k-1}}\}$. Any interval $[w^N_{\min}, w^N_{\max}]$ can be reduced to $[1, C+1)$ and although $\bar{w}^N_{\min}$ is not necessary equal to one, it is still in the first bucket $[1, 2)$.

For Algorithm 4, simple $O(m)$ procedure can be constructed for grouping and rearrangement of adjacency list of a given network. Different pointers can be added to speed up computation, such as group of pointers where ith pointer points to the ith bucket in adjacency list for every vertex. The main difference from (Cherkasky *et al.*, 1996, 1999; Denardo and Fox, 1979; Goldberg, 2001) is that, when once created, no change in buckets composition (deletion, moving from bucket to bucket) is performed in presented application.

It is obvious from the listing of Algorithm 4 that each edge is treated at most once by the procedure. "At most" is based on the statement (ii) of Theorem 3, i.e., that edges with both vertices from the same bucket are not considered and on the fact that the first descendants of the source from the lowest level bucket are transferred to SS in unit time. In the certain instances of the problem just this should save the significant computational time. As a trivial example, proposed algorithm

performs shortest paths on complete Euclidian networks, complete networks with $C < 2$, networks having $\deg(s) \to n$, $C < 2$ in a constant time.

We now analyze a worst case bound on the running time of the Algorithm 4. Denote by TA_k, the time complexity of the Step k of the algorithm. We have $\mathrm{TA}_{2-4} = O(1)$, $\mathrm{TA}_5 = O(m)$. Loop 7–23 iterates through each ith layer and Loop 10–20 iterates through each of the previously populated layers. Condition $BV_{i-k} \neq \varnothing$ in Step 11 need not be checked due to the auxiliary linked list consisting of only populated previous layers. If any of previous layers is empty, the procedure simply skips this iteration. Loop 13–19 iterates trough all $u = B^k(v) \,|\, u \in \mathrm{US}$. This is the lowest nested loop so its number of iterations determines the complexity of the whole algorithm. If $B^k(v)$ is populated, one edge is treated in each iteration, which fits in $\mathrm{TA}_{7-23} = O(m)$. For efficiency reasons, strategy of the algorithm is not to move v from $B^k(v)$ when $d^*(v)$ is determined, so the condition $u \in \mathrm{US}$ must be checked. However, this does not deny $\mathrm{TA}_{7-23} = O(m)$: If $u \in \mathrm{US}$, a edge, not explored yet is treated, which further means that in each iteration different edges are treated. All other operations are elementary, so the overall $\mathrm{T}^{\mathrm{A4}} = O(m)$.

If $B^k(v)$ is empty, algorithm spends $O(1)$ for each such situation. The algorithm runs in $\mathrm{T}^{\mathrm{A4}} = O(m + \varphi)$ time, where φ is the total number of times, empty $B^k(v)$ is chosen, which is in the worst case $C \log C$. To overcome that additional time, Algorithm 4 can be modified in the way that $B^k(BV_{i-k})$ instead of $B^k(v)$, $v \in BV_{i-k}$ is considered in Loop 13–19. Labels of the vertices v_a^i from $B^k(BV_{i-k})$ satisfy the conditions of Theorem 6: $d(v_a^i) = \min(\overline{d_N}(s, \mathrm{anc}(v_a^i)) + \bar{w}(e(\mathrm{anc}(v_a^i), v_a^i))), \mathrm{anc}(v_a^i) \in BV_{i-k},\ \forall\, e(\mathrm{anc}(v_a^i), v_a^i)) \in E$.

The main difference between Algorithms 4 and 5 is that in the ith iteration all neighbors of vertices from BV_{i-1} are arranged in buckets. Each edge is treated at most twice, once for storing its head vertex into bucket and once when that edge is included or excluded from SS. It implies that a worst case bound on the running time of the algorithm is still $\mathrm{T}^{\mathrm{A5}} = O(m + \varphi)$ where $\varphi = O(nC)$. Using multilevel buckets a worst case bound on the running time of the algorithm is then $\mathrm{T}^{\mathrm{A5}} = O(m + n \log C)$.

Ability to partially apply Dij-Alg, without loss of overall efficiency, provides further reduction of T^{A4} and T^{A5}, namely, one can chose in advance the upper bound of C which should be treated by the new algorithms. Thus $C \log C$ is constant and the time dependence becomes $\mathrm{T}^{\mathrm{A4}} = O(m')$ and $\mathrm{T}^{\mathrm{A5}} = O(m' + n') = O(m')$, where m' and n' refers to vertices and edges which are treated by these algorithms. According to (9) overall complexity becomes:

$$\mathrm{T}(m, n) = O(m') + O(m - m', f(n - n')) = O(m, f(n - n')), \tag{11}$$

where $f(n - n')$ refers to the best known algorithms for the considered instances.

8. Experimental Results

It can be easily deduced, from the complexity analysis of the proposed algorithms that for both algorithms the most appropriate network classes are networks with

small C and, for Algorithm 4, networks with uniform edge weight distributions. The experimental results on these networks verified an extreme efficiency of the proposed algorithms. The aim of this experimental analysis was, however, to evaluate the effectiveness of the algorithms, on the most difficult instances for the proposed approach. For this purpose, we evaluate the efficiency of our algorithms through experiments on networks from 9th DMACS Implementation Challenge, Demetrescu *et al.* (2006). These well known and mostly used instances include random generated networks and road networks of Western Europe and United States. Results obtained on the road networks were analyzed in detail since these instances represent the huge real-life networks with roughly 20 million vertices. All results relate to Algorithm 5, noting that all the most important conclusions are also valid for Algorithm 4.

Algorithm 5 was implemented in C++ and compiled with Microsoft Visual C++ 2010. All tests were performed on Intel Core 2 Duo 2.2 GHz PC with 4 GB installed memory. It should be noted that this hardware is a modest option, both in terms of speed and in terms of available memory.

Road Networks: The networks representing the USA road networks belong to the 9th Implementation DIMACS Challenge dataset. Complete USA has 23,947,347 vertices (road intersections) and 58,333,344 directed edges (road segments). Table 1 presents data for USA and eleven sub-networks. Data include maximal out-degree (*odeg*), maximal in-degree (*ideg*), number of bread-first layers when vertex 1 is chosen as a source vertex (*layers*), number of vertices (*vertices*), number of edges (*edges*), minimal edge weight (*minl*) and maximal edge weight (*maxl*). Column *bfsdist* represents maximal distance from the vertex 1, obtained by the bread-first search. Columns *layers* and *bfsdist* show an estimate, which can be easily calculated for any vertex and can be used as preliminary data.

Presented data refers to travel distances. Travel times are also included in data, but here presented results are restricted to travel distances, as the conclusions are common in both cases. Experiment consists of running Algorithm 5 on each network

Table 1. USA road networks from 9th Implementation DIMACS Challenge dataset.

Name	Description	odeg	ideg	layers	bfsdist	vertices	edges	minl	maxl
NY	New York City	8	8	619	1,809,021	264,346	733,846	1	36,946
BAY	Bay Area	7	7	522	2,870,859	321,270	800,172	1	94,305
COL	Colorado	8	8	1,081	9,480,952	435,666	1,057,066	1	137,384
FLA	Florida	8	8	1,920	12,716,366	1,070,376	2,712,798	1	214,013
NW	Northwest USA	9	9	1,959	15,475,876	1,207,945	2,840,208	1	128,569
NE	Northeast USA	9	9	1,108	4,644,388	1,524,453	3,897,636	1	63,247
CAL	California and Nevada	8	8	1,895	19,587,992	1,890,815	4,657,742	1	215,354
LKS	Great Lakes	8	8	3,240	22,956,713	2,758,119	6,885,658	1	138,911
E	Eastern USA	9	9	2,878	14,272,708	3,598,623	8,778,114	1	200,760
W	Western USA	9	9	3,137	29,429,749	6,262,104	15,248,146	1	368,855
CTR	Central USA	9	9	3,826	30,916,841	14,081,816	34,292,496	1	214,013
USA	USA	9	9	6,261	55,395,482	23,947,347	58,333,344	1	368,855

Table 2. Number of source vertices for different districts.

NY	BAY	COL	FLA	NW	NE	CAL	LKS	E	W	CTR
532	438	323	131	116	92	74	51	39	22	10

Table 3. Obtained results for Algorithm 5.

Name	vertices	edges	maxl	dimacs	alg5	mb
NY	264,346	733,846	36,946	585	162	6
BAY	321,270	800,172	94,305	600	203	7
COL	435,666	1,057,066	137,384	813	366	5
FLA	1,070,376	2,712,798	214,013	2,142	755	6
NW	1,207,945	2,840,208	128,569	2,455	855	6
NE	1,524,453	3,897,636	63,247	3,566	855	8
CAL	1,890,815	4,657,742	215,354	4,343	1,297	8
LKS	2,758,119	6,885,658	138,911	6,605	1,812	7
E	3,598,623	8,778,114	200,760	9,459	2,341	8
W	6,262,104	15,248,146	368,855	17,838	4,074	9
CTR	14,081,816	34,292,496	214,013	49,355	11,820	11
USA	23,947,347	58,333,344	368,855	83,365	18,547	12

from Table 1. In each case sources are taken from the list, given with the data set. Number of different source vertices for each network is presented in Table 2.

Table 3 presents results obtained by Algorithm 5 (*alg*5) together with results obtained by the reference DIMACS NSSP solver in column *dimacs* (solver use an efficient implementation of Goldberg's algorithm (2001)). For each network, we report the average time per source processing in milliseconds. Column *mb* presents maximal bucket occupancy through the iterations of Algorithm 5.

A significant advantage of Algorithm 4 was expected, since Algorithm 5 works almost like bread-first search with higher consumption of memory than Dij-Alg. What is an important practical result is that on all tested instances, both on road networks and random networks, no memory problems were reported. It must be noted that coding of Algorithm 5 for this experiments did not include any kind of multilevel bucketing, so the influence of φ in $O(m + \varphi)$ was actually $\varphi = O(nC)$, noting that the impact of $O(nC)$ on the total elapsed time was negligible. Table 4 shows the comparison of total elapsed times and times spent on the execution of φ part of the Algorithm 5. Figure 3 graphically presents comparison of results from Table 3.

USA road networks are very convenient to demonstrate two other practical advantages of the proposed approach. First is partial use of Dij-Alg and second is the approximate application of algorithms. Analyzing the edge weight distribution of the road networks, one can notice that for any network it can be defined an subset of edges, with more than 80% of the total number of edges, such that ratio of the longest and the shortest edge from this subset is significantly less than total weight ratio. For example, for NY, there are 689,524 edges (94% of the total number

Table 4. Impact of φ on elapsed time.

Name	alg5	φ	%
NY	162	5	2.95
BAY	203	7	3.54
COL	366	24	6.58
FLA	755	39	5.14
NW	855	40	4.67
NE	855	12	1.39
CAL	1,297	51	3.94
LKS	1,812	58	3.22
E	2,341	38	1.61
W	4,074	78	1.92
CTR	11,820	82	0.69
USA	18,547	143	0.77

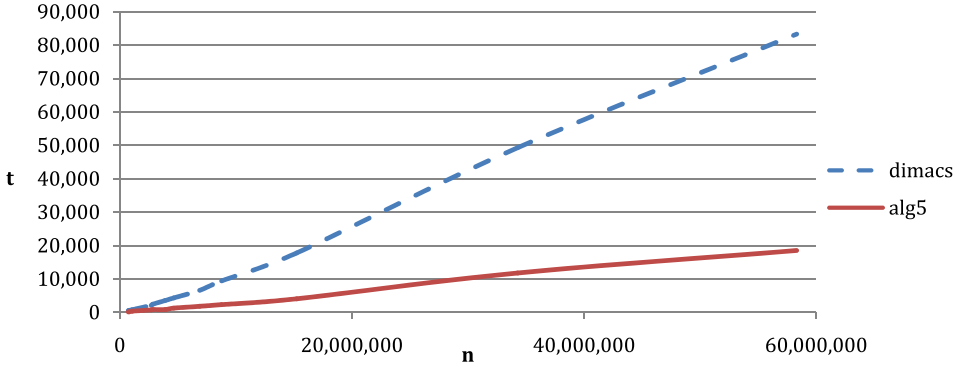

Fig. 3. Comparison of DIMACS and Algorithm 5 on USA road networks.

of NY edges), having weights between 300 and 6,000 (ratio 20). Also, there are 695,388 edges having weights between 1 and 6,000, i.e., almost all edges are shorter than 6,000 and only a few are between 6,000 and 36,946. This directly implies the application of Partial-Dij-Alg, if available hardware resources are critical.

The approximate application of algorithm can be shown also on NY example. Algorithm user who wants information about the certain distance, needs that information within limited accuracy, for example, if the distance is 1,000 km, an error of 100 m is usually negligible in practice. Maximal edge length (road segment) for NY is 36,946. The following procedure was carried out: All edge weights were divided by 100, and all normalized weights, less than one were rounded to one. Algorithm 5 was then applied on such modified network, having $C = 370$, instead $C = 36{,}946$. At the end, approximate distances have been replaced with the real ones in obtained distance tree. Complete procedure was more than two times faster than the original one. What is more interesting, all the obtained distances were exactly the same as the distances obtained by the original version. This procedure was carried out on all others road instances and only a few distances had slightly modified values.

Random Networks: DIMACS Challenge dataset includes procedures for creating random networks. We conducted experiments on both the suggested families, *Random graphs* and *Grid graphs*. All networks are sparse, as for dense networks arc scans dominate the running time. There are two main differences between these random classes; one is the range of the edge weight values and another number of labeled vertices at a point during the execution. For the *Random graphs*, the average number of labeled vertices at a point of a shortest path computation is large. Vertices of the *Grid graphs* form a two-dimensional $x \times y$ grid. *Square grids* have $x = y$ and moderate number of labeled vertices at any time during an execution of Dij-Alg, while *Long grids* have y fixed to a small constant and x growing with the number of vertices. On these grids, the number of labeled vertices is small. For each of these three families, two problem sub-families were considered. For the first, n-family, n grows and $C = n$. Values of n are $n = 2^i,\ i = 10, \ldots, 21$, while for the second, C family, n is fixed, $n = 2^{20}$ and C grows, $C = 4^i,\ i = 0, \ldots, 15$. Table 5 presents data for Random4-n family having $m = 4n$.

This family is very suitable for the proposed algorithms. All graphs from this family has less than 20 bread-first layers, so obtained results are among the best known reported results on similar hardware quality. Corresponding C family is presented in Table 6.

Thanks to a negligible impact of $O(nC)$ on a total elapsed time, slope of elapsed time as a function of C is very small.

Although the results for grid families were quite satisfactory, the proposed algorithm is definitely not the best choice for this class of networks. This conclusion is based on the fact that these families are very specialized, and for them, specific algorithms can be constructed. Algorithms 4 and 5 are general-purpose algorithms, so they do not exploit specific properties of specialized networks.

Conclusion derived from the random experimental results was the same as for the USA road networks: Even for the largest networks, no lack of memory occurred. It should be noted that a large number of networks from tested families had zero

Table 5. Execution times of Algorithm 5 on Random4-n families.

Name	vertices	arcs	minl	maxl	t[ms]	mb
Random4-n.10.0.gr	1,024	4,096	0	1,024	1	6
Random4-n.11.0.gr	2,048	8,192	0	2,048	1	7
Random4-n.12.0.gr	4,096	16,384	0	4,096	1	7
Random4-n.13.0.gr	8,192	32,768	1	8,192	2	6
Random4-n.14.0.gr	16,384	65,536	0	16,384	7	7
Random4-n.15.0.gr	32,768	131,072	0	32,768	17	8
Random4-n.16.0.gr	65,536	262,144	0	65,536	37	9
Random4-n.17.0.gr	131,072	524,288	0	131,072	89	7
Random4-n.18.0.gr	262,144	1,048,576	0	262,143	204	8
Random4-n.19.0.gr	524,288	2,097,152	0	524,288	366	7
Random4-n.20.0.gr	1,048,576	4,194,304	0	1,048,576	932	8
Random4-n.21.0.gr	2,097,152	8,388,608	0	2,097,152	1,983	9

Table 6. Execution times of Algorithm 5 on Random4-C families.

Name	vertices	arcs	minl	maxl	t[ms]	mb
Random4-C.1.0.gr	1,048,576	4,194,304	0	4	562	190,420
Random4-C.2.0.gr	1,048,576	4,194,304	0	16	573	51,735
Random4-C.3.0.gr	1,048,576	4,194,304	0	64	602	13,320
Random4-C.4.0.gr	1,048,576	4,194,304	0	256	613	3,420
Random4-C.5.0.gr	1,048,576	4,194,304	0	1,024	632	912
Random4-C.6.0.gr	1,048,576	4,194,304	0	4,096	646	253
Random4-C.7.0.gr	1,048,576	4,194,304	0	16,384	689	76
Random4-C.8.0.gr	1,048,576	4,194,304	0	65,536	769	34
Random4-C.9.0.gr	1,048,576	4,194,304	0	262,144	833	15
Random4-C.10.0.gr	1,048,576	4,194,304	0	1,048,576	932	8
Random4-C.11.0.gr	1,048,576	4,194,304	0	4,194,304	1,004	5
Random4-C.12.0.gr	1,048,576	4,194,304	5	16,777,213	991	6
Random4-C.13.0.gr	1,048,576	4,194,304	22	67,108,809	1,338	3
Random4-C.14.0.gr	1,048,576	4,194,304	44	268,435,449	1,064	5
Random4-C.15.0.gr	1,048,576	4,194,304	143	1,073,741,424	1,038	5

weight edges, which could be treated by separate procedures in order to improve efficiency.

9. Concluding Remarks

Rapid development of computer hardware has made it possible to efficiently process huge databases on the cheapest personal computers. Every day, CPU memory is getting cheaper and the processing time more expensive. In such circumstances, some standard ways of evaluating algorithms become unsuitable for practical applications. Also, worst case analysis is an excellent tool for the theoretical comparison of algorithms, but only objectively derived experimental analysis can respond to the comparison of algorithms having equal worst case complexity. This imposes an importance hierarchy of fulfilling the parameters of the experimental test. In practical applications of algorithms this order would be: (1) time efficiency, (2) simplicity of algorithm formulation and (3) maximum size of instances that can be processed on the available hardware, noting that the increase in CPU time caused by the larger memory addressing is included in (1). Experimental results presented in the previous section have shown that the proposed algorithms can counteract the known algorithms in all these parameters.

The most important contribution of the paper is contained in the Eq. (11), which is the result of two ideas: Expansion in wave-fronts from the source and the partial use of Dij-Alg. This equation shows that the proposed algorithm can be, depending on the processed instance, more efficient than the best existing algorithms. Although the idea of the partial use of Dij-Alg is very simple and on the level of an easy exercise, there is no article in the literature in which this idea was exploited.

Presented algorithms apply a different approach from the majority of the known algorithms. Beside the algorithm expansion in wave-fronts from the source, there are three main differences between this approach and approach which is applied in most

of the known shortest paths algorithms. First, for Algorithm 4, the main bucketing is performed on edges instead on temporary distances from the source, using the benefits that the range of edge weights is much narrower than the range of the temporary vertex distances to the source, so the number of buckets is significantly decreased. Second, for both algorithms, there is no need to move nodes from bucket to bucket, because a vertex for any reason does not have to be deleted from its bucket. Finally, for the certain instances of the problem, only subset of the edges can be considered, so one can decide in advance which range of values should be treated by the procedure and, possibly, if the time requirements are a dominant factor, use redundant data to speed-up computation. The possibility that only a subset of edges can be selected by the proposed algorithm and the ability to partially apply Dij-Alg without loss of overall efficiency, allows us to make a hybrid algorithm which contains proposed algorithm and any, most convenient Dij-Alg as its parent algorithms.

It is easy to conclude which classes of networks and what source properties are most suitable for the proposed algorithms. In the first, groups are networks with low weight ratio of its edges. Network parameters, like weight ratio for the whole network, weight distribution, weight ratio of k-neighbors or composition of the intervals obtained by edge groupings, can be easily obtained in pre-processing and used for making decisions about the composition of the subset of the network edges which should be treated by the algorithm. Another network characteristic, suitable for this application, refers to a graph parameters such as diameter, density, array of vertex degrees and source parameters, such as eccentricity, degree and cardinality of k-neighbors. All these parameters indicate that the most favorable condition for the Algorithm 4 is when a significant number of examined vertices exist in a relatively small number of graph layers. Finally, for the uniform edge weight distributions, Algorithm 4 is extremely efficient. For the arbitrary distributions, the value φ from $\mathrm{TA} = O(m + \varphi)$ can be obtained in pre-processing and the decisions for the further steps of the algorithm can be taken, i.e., use of the proposed algorithm on the entire network or processing only the subset of the given network with proposed algorithm and processing the rest of the edges with any Dij-Alg afterwards.

References

Aghaei, MRS, ZA Zukarnain, YA Zainuddin and MH Ali (2009). A hybrid algorithm for finding shortest path in network routing. *Journal of Theoretical and Applied Information Technology*, 5(3), 360–365.

Ahuja, RK, K Mehlhorn, JB Orlin and RE Tarjan (1990). Faster algorithms for the shortest path problem. *Journal of the Association for Computing Machinery*, 37, 213–223.

Ahuja, RK, TL Magnanti and JB Orlin (1993). *Network Flows: Theory, Algorithms, and Applications.* Englewood Cliffs, NJ: Prentice Hall.

Bast, H, S Funke, D Matijevic, P Sanders and D Schultes (2007). In transit to constant time shortest-path queries in road networks. In *Proc. 9th Int. Workshop on Algorithm Engineering and Experiments-ALENEX 2007*, New Orleans, USA, pp. 46–60.

Bauer, R and D Delling (2008). SHARC: Fast and robust unidirectional routing. In *Proc. 10th Int. Workshop on Algorithm Engineering and Experiments-ALENEX 2008*, San Francisco, USA, pp. 13–26.

Cherkasky, BV, AV Goldberg and T Radzik (1996). Shortest-paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73, 129–174.

Cherkasky, BV, AV Goldberg and C Silverstein (1999). Buckets, heaps, lists and monotone priority queues. *SIAM Journal on Computing*, 28(4), 1326–1346.

Demetrescu, C, AV Goldberg and DS Johnson (2006). Shortest Paths. *9th DIMACS Implementation Challenge*. Available at http://www.dis.uniroma1.it/_challenge9/, Accessed on September 1, 2012.

Denardo, EV and BL Fox (1979). Shortest-route methods: 1. Reaching, pruning and buckets. *Operations Research*, 27, 161–186.

Dial, RB (1969). Algorithm 360: Shortest path forest with topological ordering. *Communications of the ACM*, 12, 632–633.

Dijkstra, EW (1959). A note on two problems in connexion with graphs. *Numerical Mathematics*, 1(1), 269–271.

Fredman, ML and RE Tarjan (1987). Fibonacci heaps and their use in improved network optimization algorithms. *Journal of the Association for Computing Machinery*, 34(3), 596–615.

Goldberg, AV (2001). A simple shortest path algorithm with linear average time. In *Proc. 9th European Symp. on Algorithms-ESA 2001 in Lecture Notes in Computer Science*, 2161, pp. 230–241.

Goldberg, AV, H Kaplan and RF Werneck (2006). Reach for A*: Efficient point-to-point shortest path algorithms. In *SIAM Workshop on Algorithms Engineering and Experimentation (ALENEX 06), Society for Industrial and Applied Mathematics*, Miami, FL.

Holzer, M, F Schulz, D Wagner and T Willhalm (2005). Combining speed-up techniques for shortest-path computations. *Journal of Experimental Algorithmics*, 10, 1–18.

Johnson, DB (1975). Priority queues with update and finding minimum spanning trees. *Information Processing Letters*, 4(3), 53–57.

Johnson, DB (1977). Efficient algorithms for shortest paths in sparse networks. *Journal of the Association for Computing Machinery*, 24(1), 1–13.

Kohler, E, RH Mohring and H Schilling (2005). Acceleration of shortest path and constrained shortest path computation. In *Proc. 4th Int. Workshop on Efficient and Experimental Algorithms-WEA 2005 in Lecture Notes in Computer Science*, 3503, 126–138.

Lauther, U (2004). An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In *Geoinformation and Mobilitat*, Vol. 22, pp. 219–230. Munster: IfGI Prints, Institut fur Geoinformatik Universitat.

Mohring, RH, H Schilling, B Schutz, D Wagner and T Willhalm (2005). Partitioning graphs to speed up Dijkstra's algorithm. In *Proc. 4th Int. Workshop on Efficient and Experimental Algorithms-WEA 2005 in Lecture Notes in Computer Science*, 3503, 89–202.

Raman, R (1996). Priority queues: Small, monotone and trans-dichotomous. In *Proc. 4th Annual European Symp. Algorithms in Lecture Notes in Computer Science*, 1136, 121–137.

Raman, R (1997). Recent results on single-source shortest paths problem. *SIGACT News*, 28, 81–87.

Sanders, P and D Schultes (2005). Highway hierarchies hasten exact shortest path queries. In *Proc. of the 13th European Symp. Algorithms-ESA 2005 in Lecture Notes in Computer Science*, 3669, 568–579.

Schulz, F, D Wagner and K Weihe (2002). Using multi-level graphs for timetable information. In *Proc. 4th Int. Workshop on Algorithm Engineering and Experiments-ALENEX in Lecture Notes in Computer Science*, 2409, 43–59.

Sedgewick, R and J Vitter (1986). Shortest paths in Euclidean graphs. *Algorithmica*, 1(1–4), 31–48.

Thorup, M (1999). Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the Association for Computing Machinery*, 46(3), 362–394.

Thorup, M (2000). On RAM priority queues. *SIAM Journal on Computing*, 30(1), 86–109.

Wagner, D and T Willhalm (2003). Geometric speed-up techniques for finding shortest paths in large sparse graphs. In *Proc. 11th European Symp. on Algorithms-ESA 2003 in Lecture Notes in Computer Science*, 2832, 776–787.

Dragan Vasiljevic is Associate Professor and Head of Chair of Computer Integrated Manufacturing and Logistics at FOS (Faculty of Organizational Sciences) of University of Belgrade. He received his PhD degree in the field of logistics and operations management at the same university. He published more than 100 papers in peer-reviewed journals, conference proceedings and industry reports. He is the author or co-author of the following three books: Computer Integrated Logistics, Models and Trends, Logistics and Supply Chain Management and Kaizen, and Japanese Path to Business Excellence, which has been used as a course books at the FOS' Department of Engineering and Operations Management. His current research interests and areas of expertise include supply networks, high-performance manufacturing and lean logistics. Dr Vasiljevic has extensive experience in numerous consulting projects done for domestic and foreign companies and also serves on several editorial boards.

Milos Danilovic was born in Belgrade, Serbia, on April 1985. He received his Bachelor Degree (2009) and Master Degree (2011) from Faculty of Organizational Sciences (FOS), University of Belgrade. Now, he is a PhD student in the field of Operations Management at FOS. He currently works as a Teaching Assistant at Department of Operations Management at FOS. He published several papers as an author or co-author in conference proceedings. His main research interests and areas of expertise include combinatorial optimization problems, supply networks, scheduling theory, operations management and computer integrated manufacturing systems.