Complete Pareto Front of the Minimal Length Maximal Capacity Shortest Path Problem

Lasko M. Laskov[1*] and Marin L. Marinov[1]

[1*] Department of Informatics, New Bulgarian University, 21 Montevideo Str., Sofia, 1618, Bulgaria.

Contributing authors: llaskov@nbu.bg; mlmarinov@nbu.bg;

Abstract

Shortest path problem in a network is a fundamental task in combinatorial optimization. Many applications in practice involve optimization two conflicting criteria and from a biobjective shortest path problems. Since there does not exist a single solution that is optimal with respect to both objective functions, we are interested in a special set of solutions for which any of the two criteria cannot be improved without declining the other: *Pareto optimal set* or *Pareto front*.
In this paper we analyze in details the biobjective shortest path problem in the case in which the first objective function is a linear one (minimal length paths) and the second one is a nonlinear bottleneck function (maximal capacity paths). We present an exact solution based on two modifications of the Dijkstra's algorithm that finds the complete description of all Pareto optimal solutions. We provide detailed proofs of the correctness and the computational complexity of the presented algorithms and numerical examples that illustrate their execution.

Keywords: combinatorial optimization, biobjective shortest path problem, Pareto front

1 Introduction

Shortest path problem in networks is a fundamental task in both graph theory [1] and combinatorial optimization [2]. In its standard single-criterion formulation, it is the problem of finding a path between a source and a destination vertex in a network, where the length of the path, defined as the sum of the weights of the edges that constitute it, is minimized.

While the above single criterion formulation has numerous applications, many problems in practice require more than one, and in particular, two criteria to model more accurately the task being of interest. Optimization problems that consists of two objective functions are usually referred to as bicriteria or *biobjective* optimization problems and are a subject of extensive research in computational optimization (see for example [3], [4], [5], [6]).

We exclude the trivial situation in which the two objective functions of a biobjective shortest path problem allow us to find a single solution that is optimal for both of them simultaneously, and we focus on the case in which they are conflicting. If the two objective functions are in contradiction, or even they are incommensurable [7], we are interested in finding a special set of *nondominated* solutions that in literature is referred to as *Pareto optimal set* [8] or *Pareto front* [9].

Depending on the concrete problem that is solved, the cardinality of the Pareto optimal set can be a relatively big number, and at the same time, all nondominated solutions can be considered equally preferable from mathematical point of view. Even though in practice usually a single Pareto optimal solution is selected, the detailed information about the Pareto front can be used for a valuable analysis that can clarify the interdependencies between decision variables, objective function and problem constraints [8]. Also, a precise description of the Pareto front will give the decision maker the complete picture needed to make an informed decision.

The choice of the particular objective functions determines different types of biobjective shortest path problems. In his notable work [10] Hansen describes ten different types of biobjective path problems by combining pairwise six objective functions. He introduces two criteria in the network (length and cost) and defines the bicriteria path problems by defining the objective functions that are applied on each of them. In our research we focus on one of these problems: with the first criterion being a linear function representing the path length and the second criterion being a nonlinear *bottleneck* function representing the path capacity.

Hansen clearly defines that a path that is not efficient is dominated by at least one efficient path. Then the goal that is formulated in [10] is to find the *minimal complete set* of the efficient paths. His approach is generalized by Martins [11] for the multicriteria case and the latter work became the basis of multiple approaches in the literature. Since then, many of the published works that aim to find the minimal complete set of the nondominated solutions or an approximation of the Pareto front [5] are based on heuristic methods [12], [9], and some of them are based on exact algorithms [13], [3].

In general, the number of elements of the complete Pareto front can be a large number [7] and the problem of their calculation is NP-hard [14]. In recent literature is a common practice to adopt methods of artificial intelligence in the case of NP-hard optimization problems [15]. On the other hand, it has been shown that in practice many bicriteria shortest path problems lead to a number of Pareto optimal solutions that is small enough to make the calculation of the complete Pareto front feasible [14]. Indeed, in literature there are a number of works that calculate all Pareto optimal solutions of a predefined bibojective shortest path problem, for example [16], [4], [6].

The goal of our research is to find all Pareto optimal solutions of the minimal length maximal capacity shortest path problem. We propose an exact label-setting technique that is based on two helper modifications of the Dijkstra's algorithm [17] that are used to represent both all shortest paths and maximal capacity paths in the input network. The general algorithm takes advantage from both helper methods and constructs the complete description of the Pareto front. We provide detailed mathematical proofs of the correctness and computational complexity of the proposed algorithms.

The paper is structured as follows. In Section 2 we provide basic notations and definitions, we formulate the two helper problems and the general problem that is solved in our research. In Section 3 we describe the proposed algorithms together with their proofs, numerical examples that illustrate them, and we discuss the effectiveness of the computer program implementation. Finally, in Section 4 we give our conclusions.

2 Problem formulation

2.1 Basic notations and definitions

With $G = (V, E)$ we denote the digraph (directed graph) that has $n = |V|$ number of vertices and $m = |E|$ number of directed edges. We label the vertices of G with the natural numbers from 1 to n, and without loss of generality $V = \{1, 2, \ldots, n\}$. We denote each edge $e \in E$ with the ordered pair (u, v), where $u, v \in V$ are vertices and e is directed from u to v, and then $E \subseteq V^2$.

We will define two weight functions on the set of edges of the digraph. The first weight function $f : E \to \mathbb{R}_+$ assigns to each edge (u, v) the positive real number $f(u, v)$. We will call $f(u, v)$ the *length* of the edge (u, v).

The second weight function is $g : E \to \overline{\mathbb{R}}_+$, where $\overline{\mathbb{R}}_+ = \mathbb{R} \cup \{\infty\}$. It assigns to each edge $(u, v) \in E$ the positive real number $g(u, v)$ if the edge (u, v) is restricted, or ∞ if the edge is not restricted. We will call the value $g(u, v)$ the *capacity* of the edge (u, v) and it defines the restriction that is imposed on the edge.

The digraph $G = (V, E)$ with the two weight functions that are defined on its edges form the *network* $N = (V, E, f, g)$ [18]. We adapt the standard adjacency list representation of a graph [19] to form the adjacency list Adj of a network.

The *adjacency list* Adj of the network $N = (V, E, f, g)$ is an array of $n = |V|$ lists, where each list $Adj[u]$, $u \in V$, contains a triple $(v, f(u, v), g(u, v))$ for each edge $(u, v) \in E$, where $f(u, v)$ is the length and $g(u, v)$ is the capacity of the edge. It has the following general form:

$$Adj = [\langle (v, f(u, v), g(u, v)), \ldots \rangle, \ldots]. \tag{1}$$

We will denote that the adjacency list is an attribute of the network with $N.Adj$.

A *path* in the network N is a finite sequence of vertices and edges of the type:

$$v_0, e_1, v_1, e_2, \ldots, v_{t-1}, e_t, v_t, \tag{2}$$

where $v_j \in V$, for all $j = 0, 1, \ldots, t$ are distinct vertices and $e_i = (v_{i-1}, v_i)$ is the edge from v_{i-1} to v_i with $e_i \in E$ for all $i = 1, 2, \ldots, t$. The path consists of $t + 1$ vertices and t edges. We will call v_0 the *source vertex* of the path and v_t – the *terminal vertex*.

A path that connects the source vertex v_0 and the terminal vertex v_t is called a (v_0, v_t)-*path* and is represented by the ordered sequence α of distinct vertices:

$$\alpha = (v_0, v_1, \ldots, v_t). \tag{3}$$

Note that in a given network there may be many (v_0, v_t)-paths.

For each path $\alpha = (v_0, v_1, \ldots, v_t)$ we define its *path length* as the function

$$x(\alpha) = \sum_{i=1}^{t} f(v_{i-1}, v_i) \tag{4}$$

and we define its *path capacity* as the function

$$y(\alpha) = \min_{1 \le i \le t} \{g(v_{i-1}, v_i)\}. \tag{5}$$

The two functions $x(\alpha)$ and $y(\alpha)$ define the *objective functions* of the shortest path optimization problem that is given in Definition 2. Note that $x(\alpha)$ is a linear function, while $y(\alpha)$ is a nonlinear bottleneck function.

In our considerations we will focus on paths with source vertex $v_0 = 1$, which will not limit the possible cases, because we can relabel the vertices of the network N if we want to consider paths with another staring vertex. For convenience, we will denote the set of all $(1, u)$-paths with W_u.

For each vertex $u \in V$ with $\delta(u)$ we denote the *distance* form the source vertex $v_0 = 1$ to the terminal vertex $v_t = u$, where:

$$\delta(u) = \min_{\alpha \in W_u} \{x(\alpha)\}. \tag{6}$$

If there is no $(1, u)$-path in the network, we set $\delta(u) = \infty$.

For each $u \in V$ with $\kappa(u)$ we denote its *vertex capacity*, where:

$$\kappa(u) = \max_{\alpha \in W_u} \{y(\alpha)\}. \tag{7}$$

In the case in which a $(1, u)$-path in the network does not exist, then $\kappa(u) = -\infty$.

For each set of $(1, u)$-paths W_u, *Pareto optimal path* is given by Definition 1.

Definition 1 We call the path $\alpha \in W_u$ *Pareto optimal* when there does not exist another path $\beta \in W_u$, for which any of the following two conditions is fulfilled:

- $x(\beta) < x(\alpha)$ and $y(\beta) \ge y(\alpha)$;
- $x(\beta) \le x(\alpha)$ and $y(\beta) > y(\alpha)$.

Based on the definition of Pareto optimal path, we can now define the biobjective optimization problem that we examine in this paper.

Definition 2 Given the network $N = (V, E, f, g)$, starting vertex 1 and terminal vertex n, the *minimal length maximal capacity shortest path problem* is the problem to find a Pareto optimal path $\alpha \in W_n$, where W_n forms the *feasible set*.

A shortest path problem Definition 2 may have many solutions $\alpha \in W_n$. The set of all such Pareto optimal paths is called *Pareto optimal set* or *Pareto front*.

Definition 3 We denote the set of all Pareto optimal paths in the network N with P, where P forms the *Pareto front* of the biobjectove shortest path problem.

We say that two paths α and β are *equivalent*, denoted $\alpha \sim \beta$, when $x(\alpha) = x(\beta)$ and $y(\alpha) = y(\beta)$. Also, we will say that the path β is *dominated* by the path α, denoted $\beta \prec \alpha$, when $x(\alpha) < x(\beta)$ and $y(\alpha) \geq y(\beta)$ or $x(\alpha) \leq x(\beta)$ and $y(\alpha) > y(\beta)$.

Definition 4 For all *classes of equivalent Pareto optimal paths* P_i,

$$P = \bigcup_{i=1}^{K} P_i, \tag{8}$$

where K is the number of such classes. For short, we will call P_i *equivalence classes*.

With the respect of the distance (6) and capacity (7), we define a subnetwork and digraph respectively that are used formulate the two helper algorithms given in Section 3.1 and Section 3.2.

Definition 5 We will say that $\widehat{N} = (V, \widehat{E}, f, g)$ is a *shortest paths subnetwork* in the network $N = (V, E, f, g)$, if the following two properties hold:

1. Every $(1, n)$-shortest path in N is also a $(1, n)$-path in $\widehat{N}$.
2. Every $(1, n)$-path in $\widehat{N}$ is a $(1, n)$- shortest path in N.

Definition 6 We say that $\widetilde{G} = (V, \widetilde{E})$ is a *maximal capacity digraph* of the network $N = (V, E, f, g)$, if the following two properties hold:

1. Every $(1, n)$-maximal capacity path in N is also a $(1, n)$-path in $\widetilde{G}$.
2. Every $(1, n)$-path in $\widetilde{G}$ is also a $(1, n)$-maximal capacity path in N.

2.2 The complete Pareto front problem

Based on the definitions of the Pareto optimal path in Definition 1 and the biobjective shortest path problem in Definition 2, we will formulate the complete Pareto front problem that is the subject of the presented research.

Problem 1 (Complete Pareto front) Calculate the set P of all Pareto optimal paths $\alpha \in W_n$ that form the Pareto front of the minimal length maximal capacity shortest path problem (Definition 2).

To solve the complete Pareto front problem stated above, we will examine separately the two single-objective problems that are defined by the objective functions (4) and (5) respectively. Based on the fact that the solution of a shortest path problem, in the general case, will consist of a whole set of optimal paths rather than a unique single solution, we define the following two helper problems.

Problem 2 (Shortest paths subnetwork) Given the network $N = (V, E, f, g)$, compute the shortest paths subnetwork $\widehat{N} = (V, \widehat{E}, f, g)$.

Problem 3 (Maximal capacity paths digraph) Given the network $N = (V, E, f, g)$, compute the maximal capacity paths digraph $\widetilde{G} = (V, \widetilde{E})$.

An important property of both shortest paths subnetwork $\widehat{N}$ and maximal capacity paths digraph $\widetilde{G}$ is that they allow us to construct respectively the complete list of all paths with minimal length, and the complete list of all paths with maximal capacity.

3 The proposed shortest path algorithms

In this section we propose two modified versions of the Dijkstra's algorithm [17] that solve the two helper problems given in Problem 2 and Problem 3, and the final algorithm that generates the complete description of the Pareto front as given in Problem 1. The implementation of the algorithms assumes that the network $N = (V, E, f, g)$ is represented using the adjacency list $N.Adj$. Without loss of generality, the source and terminal vertices are selected $v_0 = 1$ and $v_t = n$ respectively.

Table 1 Fibonacci heap functions for min/max-priority queue Q

Function	Description	Complexity
$insert(Q, v)$	inserts an element v into Q	$\Theta(1)$
$extract(Q)$	extracts and returns the min/max element from Q	$\Theta(\log n)$
$update(Q, v, k)$	decrease/increase the element v with the new key k	$\Theta(1)$

Both helper algorithms use a priority queue Q abstract data type (ADT) that is implemented using the *Fibonacci heap* data structure [20]. Using the technique of amortized analysis it is proved that Fibonacci heap introduces a significant speedup of the Dijkstra's algorithm to $O(n \log n + m)$ (see [19]), and this fact plays an important role in the computational complexity analysis of the proposed method.

In the algorithm that constructs the shortest paths subnetwork (Section 3.1) Q is a min-priority queue, while in the algorithm that constructs the maximal capacity

digraph (Section 3.2) Q is a max-priority queue. In both cases we will use the Fibonacci heap functions with their corresponding implementations, as they are given in Table 1 along with their corresponding computational complexities [19].

3.1 Shortest paths subnetwork

We will solve the first helper Problem 2 to construct the subnetwork $\widehat{N} = (V, \widehat{E}, f, g)$ that is composed by the all shortest paths in N, as given in Definition 5. The main function of the modified Dijkstra's algorithm is $minsum(G.Adj)$ (given in Algorithm 2) which returns the adjacency list of the network $\widehat{N}$.

The main concept of the algorithm is that it splits the set of all vertices of the input network V into two subsets: the subset V_0 of vertices that are not yet traversed, and the subset $U = V \setminus V_0$ of the vertices that are traversed. As the algorithm visits the vertices, it guarantees that

$$\delta(v) \geq \delta(u), \tag{9}$$

for each vertex $v \in V_0$ and each vertex $u \in U$. Before the execution of the first iteration, $V_0 = V$ and $U = \varnothing$. Then, on each of the n consecutive iterations one vertex is selected from V_0 and is transferred into U.

The algorithm manages three data structures: an array d, an adjacency list of ingoing neighbors p, and a min-priority queue Q that is implemented with a Fibonacci heap.

For each vertex $u \in V$ the array d stores the current estimate of the distance of $(1, u)$-path. In the initial stage of the algorithm all elements of d are set to ∞, except $d[0]$ which is set to 0 (see line 2 of Algorithm 2).

The adjacency list p stores the ingoing neighbors of the shortest path digraph $\widehat{G}$ that forms the network $\widehat{N}$. It is an array of lists in which the element $p[v]$ stores a list of all vertices u, such that $(u, v) \in \widehat{E}$.

The min-priority queue Q contains the set of vertices V_0 that are not yet traversed by the algorithm. It is keyed by the estimates of the shortest path lengths stored in d, which means that when a vertex u is inserted into Q its key will be $d[u]$.

Algorithm 1 Relaxation function applied on the distance attribute

```
1:  function relax(u, v)
2:      new ← d[u] + f(u, v)
3:      if d[v] > new then
4:          d[v] ← new
5:          update(Q, v, new)          ▷ decrease the key of v to store new
6:          p[v] ← {u}
7:      else if d[v] = new then
8:          p[v] ← p[v] ∪ {u}          ▷ store alternative paths
9:      end if
10: end function
```

One each step of the algorithm it performs the relaxation function in the Algorithm 1 on a selected edge (u, v). It calculates a new estimate of the distance of the corresponding $(1, v)$-path for the vertex v (line 2). If the new estimate surpasses the current, it replaces the values $d[v]$, the key in Q is decreased, and the vertex u is stored as a parent of v in the shortest paths ingoing adjacency list p. Otherwise, if the new estimate is equal of the current one, the edge (u, v) is stored as an alternative branch in p.

Algorithm 2 Constructs shortest path subnetwork $\widehat{N}$

```
1:  function minsum(N.Adj)
2:      d[1] = 0, d[i] = ∞, for i = 2, 3, ..., n
3:      p[i] = ⟨∅⟩, for i = 1, 2, ..., n
4:      for each u ∈ V do
5:          insert(Q, u)                          ▷ min-priority queue keyed by d
6:      end for
7:      while Q ≠ ∅ do
8:          u ← extract(Q)
9:          for each vertex v ∈ N.Adj[u] do
10:             relax(u, v)
11:         end for
12:     end while
13:     N̂.Adj ← outadj(p)
14:     return d[n], N̂.Adj
15: end function
```

The vertex u that is selected as the starting vertex of the edges that are relaxed, is determined by the minimum element in the min-priority queue Q, returned by $extract(Q)$ on line 8 of Algorithm 2. The main loop of the algorithm will stop when all vertices are transferred from V_0 to U and Q becomes empty. Finally, the outgoing adjacency list $\widehat{N}.Adj$ of the shortest paths network is constructed from the ingoing adjacency list p by the function $outadj(p)$. This function (Algorithm 3) composes each edge (u, v) from the digraph $\widehat{G}$ given by p and it takes the corresponding edge length $f(u, v)$ and capacity $g(u, v)$ from the original adjacency list $N.Adj$.

Proposition 1 *The function $minsum(N.Adj)$ correctly constructs the shortest path subnetwork $\widehat{N}$.*

Proof We will analyze the **while** loop of Algorithm 2 using the method of mathematical induction. We denote the set of all vertices that are traversed after the k-th iteration with U_k and the set of their outgoing neighbors with $X_k = \{v : v \in N.Adj[u], u \in U_k\}$. With $V_k = V \setminus U_k$ we denote all the vertices that are not yet traversed after the k-th iteration. We will prove that for the current states d_k and p_k of the array d and the adjacency list p, the set of properties, given in Property 1 and Property 2, hold.

Algorithm 3 Construct the outgoing $\widehat{N}.Adj$ from the ingoing adjacency list p

```
1: function outadj(p)
2:     for each end vertex v ∈ p do
3:         for each start vertex u ∈ p[v] do
4:             N̂.Adj[u] ← N̂.Adj[u] ∪ {(v, f(u,v), g(u,v))}
5:         end for
6:     end for
7:     return N̂.Adj
8: end function
```

Property 1 For the k-th iteration of the Algorithm 2 we define the following properties:

1. $d_k[v] = \begin{cases} \delta(v), & \text{if } v \in U_k \\ \min\limits_{u \in U_k, (u,v) \in E} \{\delta(u) + f(u,v)\}, & \text{if } v \in V_k \cap X_k \\ \infty, & \text{if } v \in V_k \setminus X_k \end{cases}$
2. $U_k = \{u_1, u_2, \ldots, u_k\}$ and $\delta(u_1) \leq \delta(u_2) \leq \ldots \leq \delta(u_k)$, where $u_1 = 1$.
3. For each $v \in V_k \cap X_k$ exists a path $\alpha = (1, v_1, \ldots v_l, v)$ such that $x(\alpha) = d_k[v]$, the vertices in the path $\alpha_1 = (1, v_1, v_2, \ldots, v_l)$ belong to U_k, and $x(\alpha_1) = \delta(v_l)$.
4. $\delta(u) \leq \delta(v)$ for each $u \in U_k$ and each $v \in V_k$.

Property 2 For each vertex v the adjacency list $p[v]$ contains all vertices $u \in U_k$ for which there exists a $(1, v)$-path α with the following properties:

1. The vertex u is the one before the last vertex in α.
2. All vertices in α without v are elements of U_k.
3. $x(\alpha) = d_k[v]$.

Base case. It is defined by the first two iterations of the **while** loop. We verify directly that Property 1 and Property 2 are fulfilled for $U_2 = \{u_1, u_2\}$, $u_1 = 1$, X_2, $V_2 = V \setminus U_2$, d_2 and p_2.

Inductive step. We assume that after the k-th iteration $U_k = \{u_1, u_2, \ldots u_k\}$, X_k, V_k, d_k and p_k satisfy Property 1 and Property 2. Let $Q \neq \varnothing$ (there are vertices in the set V_k) and the loop enters its $(k+1)$-st iteration. We will prove that for U_{k+1}, X_{k+1}, V_{k+1}, d_{k+1} and p_{k+1}, the two properties hold.

The call to the $extract(Q)$ function on line 8 ensures that the vertex u_{k+1} that is selected from the priority queue is such that $d_k[u_{k+1}] = \min\limits_{v \in V_k} \{d_k[v]\}$.

We will note that if $\min\limits_{v \in V_k} \{d_k[v]\} = \infty$, then for for each vertex $v \in V_k$ a $(1, v)$-path does not exist and both d_k and p_k will not change during the execution of the algorithm. In this case $d_k[u_{k+1}] = \delta(u_{k+1})$ and p is the ingoing adjacency list of the digraph $\widehat{G}$ that defines the network $\widehat{N}$.

We examine the case in which $\min\limits_{v \in V_k} \{d_k[v]\} < \infty$. The selection of the vertex u_{k+1} defines the sets $V_{k+1} = V_k \setminus \{u_{k+1}\}$ and $U_{k+1} = U_k \cup \{u_{k+1}\}$.

We will prove that for the selected vertex u_{k+1}, the equality $d_k[u_{k+1}] = \delta(u_{k+1})$ holds. Let $\alpha = (1, v_1, \ldots, v_{s-1}, v_s, v_{s+1}, \ldots, v_l, u_{k+1})$, be an arbitrary $(1, u_{k+1})$-path. With s we denote the smallest possible index for which $v_s \notin U_k$. Then $v_i \in U_k$ for $i = 1, 2, \ldots, s-1$

and $\alpha_1 = (1, v_1, \ldots, v_{s-1}, v_s)$ is a $(1, v_s)$-path for which all inner vertices are from U_k. From the inductive assumption Property 1.1 we get that $x(\alpha_1) \geq d_k[v_s]$. From the selection of the vertex u_{k+1} it follows that $d_k[v_s] \geq d_k[u_{k+1}]$ and then $x(\alpha_1) \geq d_k[v_s] \geq d_k[u_{k+1}]$. Since $x(\alpha) = x(\alpha_1) + f(v_s, v_{s+1}) + \cdots + f(v_l, u_{k+1}) \geq x(\alpha_1) \geq d_k[u_{k+1}]$, it follows that $d_k[u_{k+1}] = \delta(u_{k+1})$.

From the inductive assumption $\delta(u) \leq \delta(v), \forall u \in U_k, \forall v \in V_k$ it follows that

$$\delta(u_k) \leq \delta(u_{k+1}), \tag{10}$$

which proves that Property 1.2 holds for U_{k+1}.

Now we will prove that $\delta(u) \leq \delta(v)$, $\forall u \in U_{k+1}$ and $\forall v \in V_{k+1}$. From the inequality (10) it follows that it is enough to prove that $\delta(u_{k+1}) \leq \delta(v)$, $\forall v \in V_{k+1}$.

Let $v \in V_{k+1}$ and α be an $(1, v)$-path for which $x(\alpha) = \delta(v)$. Also, let $\alpha = (1, v_1, \ldots, v_{s-1}, v_s, v_{s+1}, \ldots, v_l, v)$, where s is the smallest index for which $v_s \notin U_k$. Then, in analogy with above proof, we get that $\delta(v) = x(\alpha) \geq f(1, v_2) + f(v_2, v_3) + \cdots + f(v_{s-1}, v_s) \geq d_k[v_s] \geq d_k[u_{k+1}] = \delta(u_{k+1})$, which proves Property 1.4.

The inner **for** loop (lines 9 to 11 of Algorithm 2) modifies the vector d_k and the adjacency list p_k and we get d_{k+1} and p_{k+1}. During this modification, only the components $d_k[v]$ and $p_k[v]$ for which $v \in N.Adj[u_{k+1}]$ can be changed. From Algorithm 1 and the fact that $d_k[u_{k+1}] = \delta(u_{k+1})$ it follows

$$d_{k+1}[v] = \min\{d_k[v], \delta(u_{k+1}) + f(u_{k+1}, v)\}, \tag{11}$$

for all $v \in N.Adj[u_{k+1}]$. Besides that, for $v \notin N.Adj[u_{k+1}]$ it is fulfilled that $d_{k+1}[v] = d_k[v]$. Particularly, $d_{k+1}[v] = d_k[v] = \infty$ for each $v \in V_{k+1} \setminus X_{k+1}$.

Let $v \in V_{k+1} \cap X_{k+1}$. The following two cases are possible.

- $v \notin X_k$ and then $d_k[v] = \infty$. From (11) it follows that $d_{k+1}[v] = \delta(u_{k+1}) + f(u_{k+1}, v)$.
- $v \in X_k$ and from the inductive assumption $d_k[v] = \min\limits_{u \in U_k, (u,v) \in E} \{\delta(u) + f(u, v)\}$.

Therefore

$$d_{k+1}[v] = \min_{u \in U_{k+1}, (u,v) \in E} \{\delta(u) + f(u, v)\}, \tag{12}$$

for all $v \in V_{k+1} \cap X_{k+1}$.

The observation that for each v it holds $d_k[v] \geq d_{k+1}[v] \geq \delta(v)$ and therefore $d_{k+1}[v] = \delta(v), v \in U_{k+1}$, finalizes the verification of Property 1.

We will prove that Property 2 also holds after the $(k+1)$-st iteration of the **while** loop. Let $\alpha = (1, v_1, \ldots, v_l, v)$, is a path such that $x(\alpha) = d_{k+1}[v]$ and $v_j \in U_{k+1}$, $j = 1, 2, \ldots l$. We will prove that $v_l \in p_{k+1}[v]$.

If $v_l = u_{k+1}$, then by definition $v_l \in p_{k+1}[v]$.

Let $v_l \neq u_{k+1}$. Then there exists $s \leq k$, such that $v_l \in U_s$ and $v_l \notin U_{s-1}$. In this case $d_{s+1}[v] \leq \delta(v_l) + f(v_l, v) \leq x(\alpha) = d_{k+1}[v]$. Since $s + 1 \leq k + 1$, then $d_{s+1}[v] \geq d_{k+1}[v]$ and therefore $d_{s+1}[v] = \delta(v_l) + f(v_l, v) = d_{k+1}[v]$. Then $v_l \in p_{s+1}[v]$ and $p_{s+1}[v] \subseteq p_{k+1}[v]$.

The **while** loop completes after n iterations. From the proof above, it follows that for each vertex v from V the following two statements hold:

- $d_n[v] = \delta(v)$;
- $p_n[v]$ contains every vertex $u \in V$ that is the one before the last vertex in $(1, v)$-path with length $\delta(v)$.

Let $\alpha = (1, v_1, \ldots, v_l, v)$ is a path, such that $x(\alpha) = \delta(v)$. Then for the path $\alpha_j = (1, v_1, \ldots, v_j)$, $j = 1, 2, \ldots, l$ it holds that $x(\alpha_j) = \delta(v_j)$. Besides that, from Property 2 it follows that $v_l \in p_n[v]$ and for each $j = 2, 3, \ldots, l$ it holds $v_{j-1} \in p_n[v_j]$. Hence, α is $(1, v)$-path in $\widehat{N}$.

The opposite statement is also true. Let $\alpha = (1, v_1, \ldots, v_l, v)$ is a $(1, v)$-path in $\widehat{N}$. We will prove that $x(\alpha) = \delta(v)$. Since $v_0 = 1 \in p_n[v_1]$, then $\delta(v_1) = f(1, v_1)$. By construction $v_1 \in p_n[v_2]$ and therefore $\delta(v_2) = \delta(v_1) + f(v_1, v_2) = x(\alpha_2)$, where $\alpha_2 = (1, v_1, v_2)$. By analogy we prove that $x(\alpha) = \delta(v)$.

The correctness of the function *outadj*(p) (Algorithm 3) completes the proof of the proposition. □

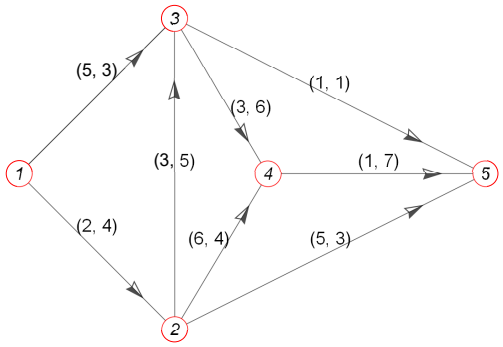

Fig. 1 Example network N_1 composed by five vertices and eight edges with length and capacity given next to them

Example 1 Let N_1 is a network (see Figure 1), represented by the following adjacency list:

$$N_1.Adj = [\langle(2, 2, 4), (3, 5, 3)\rangle, \langle(3, 3, 5), (4, 6, 4), (5, 5, 3)\rangle, \langle(4, 3, 6), (5, 1, 1)\rangle, \langle(5, 1, 7)\rangle, \langle\varnothing\rangle]. \tag{13}$$

We will illustrate the calculations of the function $minsum(N_1.Adj)$.

Firstly, the algorithm initializes the data structures $d_0 = [0, \infty, \infty, \infty, \infty]$, $p = [\langle\varnothing\rangle, \langle\varnothing\rangle, \langle\varnothing\rangle, \langle\varnothing\rangle, \langle\varnothing\rangle]$ and the min-priority queue Q that stores $V_0 = \{1, 2, 3, 4, 5\}$ keyed by the corresponding values in d_0.

The first iteration of the **while** loop results in the set of traversed vertices $U_1 = \{1\}$, $V_1 = V_0 \backslash \{1\} = \{2, 3, 4, 5\}$, $d_1 = [0, 2, 5, \infty, \infty]$ and $p_1 = [\langle\varnothing\rangle, \langle 1\rangle, \langle 1\rangle, \langle\varnothing\rangle, \langle\varnothing\rangle]$. The outgoing neighbors of the traversed vertices are represented in the set $X_1 = \{2, 3\}$.

Because $V_1 \neq \varnothing$, the **while** loop executes its second iteration. The function $extract(Q)$ defines that $u_2 = 2$, since $\min\{2, 5, \infty, \infty\} = 2 = d_1[2]$. Then the set of the traversed vertices becomes $U_2 = \{1, 2\}$, and the vertices that are not traversed are $V_2 = V_1 \backslash \{2\} = \{3, 4, 5\}$. The outgoing neighbors of U_2 are the elements of the set $X_2 = \{2, 3, 4, 5\}$. Since the neighbors of the vertex 2 in $N_1.Adj[2]$ are the vertices 3, 4 and 5, the **for** loop will modify only $d_1[j]$ and $p_1[j]$ for $j \in \{3, 4, 5\}$ using the $relax(u, v)$ function. The result is $d_2 = [0, 2, 5, 8, 7]$ and $p_2 = [\langle\varnothing\rangle, \langle 1\rangle, \langle 1, 2\rangle, \langle 2\rangle, \langle 2\rangle]$.

Since $V_2 \neq \varnothing$, the **while** loop enters its third iteration. We will note that in each iteration of the loop, the algorithm transfers a vertex, that is selected by the $extract(Q)$ function, from the set V_k into the set U_k. Therefore, the **while** loop will have that much iterations, as the number of vertices in the network. In the examined case, the loop will have five iterations.

The values of U_k, d_k and p_k for each of the next three iterations $k = 3, 4$ and 5 are:

- $U_3 = \{1, 2, 3\}$, $d_3 = [0, 2, 5, 8, 6]$, $p_3 = [\langle\varnothing\rangle, \langle 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle, \langle 3\rangle]$;

- $U_4 = \{1, 2, 3, 5\}$, $d_4 = [0, 2, 5, 8, 6]$, $p_4 = [\langle\varnothing\rangle, \langle 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle, \langle 3\rangle]$;
- $U_5 = \{1, 2, 3, 5, 4\}$, $d_5 = [0, 2, 5, 8, 6]$, $p_5 = [\langle\varnothing\rangle, \langle 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle, \langle 3\rangle]$.

We will note that after each iteration, the set of the outgoing neighbors of the traversed nodes either remains unchanged, or is increased. In this example, $X_2 = X_3 = X_4 = X_5$.

From Proposition 1 it follows that the distance array is $d_5 = [0, 2, 5, 8, 6]$ and the digraph $\widehat{G}_1$ of the shortest paths subnetwork $\widehat{N}_1$ has ingoing adjacency list stored in $p_5 = [\langle\varnothing\rangle, \langle 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle, \langle 3\rangle]$.

In the particular example, the shortest paths subnetwork $\widehat{N}_1$ results by removing the two edges $(2, 5)$ and $(4, 5)$ from N_1.

Using the ingoing adjacency list, the function $outadj(p)$ constructs the outgoing adjacency list $\widehat{N}_1.Adj = [\langle(2, 2, 4), (3, 5, 3)\rangle, \langle(3, 3, 5), (4, 6, 4)\rangle, \langle(4, 3, 6), (5, 1, 1)\rangle, \langle\varnothing\rangle, \langle\varnothing\rangle]$. Algorithm 2 returns the shortest distance 6 and the adjacency list $\widehat{N}_1.Adj$.

Using exhaustive search we verify that the subnetwork $\widehat{N}_1$ contains exactly two $(1, 5)$-paths $\alpha = (1, 3, 5)$ and $\beta = (1, 2, 3, 5)$ and their length is $x(\alpha) = x(\beta) = 6$.

Proposition 2 *The computational complexity of the function $minsum(N.Adj)$ is $O(n \log n + m)$.*

The proof of Proposition 2 follows directly from the proof of the computational complexity of the Dijkstra's algorithm in the case in which Fibonacci heap is used to implement the priority queue ADT [19]. The advantage of the Fibonacci heap over other implementations such as the binary heap in this case is in the computational complexity of the $insert(Q, v)$ and $update(Q, v, k)$ functions (see Table 1). Note that $update(Q, v, k)$ which is triggered in the relaxation function in Algorithm 1, decreases the key of the element v by replacing it with the new key k. Updating a key value also may require heap restructure, which in this case is implemented efficiently.

3.2 Maximal capacity paths digraph

In this section we will provide an algorithm that solves the second helper Problem 3 and constructs maximal capacity digraph $\widetilde{G} = (V, \widetilde{E})$ that is given in Definition 6.

The function $maxmin(G.Adj)$ in Algorithm 6 constructs the adjacency list of the outgoing neighbors $\widetilde{G}.Adj$ of the maximal capacity digraph by first calculating the capacity of the destination vertex n by calling the function $capacity(N.Adj)$. Algorithm 5 is a modification of the Dijkstra's algorithm that is based on the relaxation function in Algorithm 4 which calculates the maximal capacity of the vertex n of the network.

This modification of the Dijkstra's algorithm manages an array d, which in this case stores the estimates of corresponding vertices' capacities, and a max-priority queue Q that is keyed by the values in d. As in the case of the first helper algorithm described in Section 3.1, Q is implemented using a Fibonacci heap.

At the initialization stage of the $capacity(N.Adj)$ function, all elements of the array d are initialized with $-\infty$, except $d[1]$ which is set to ∞. All vertices are inserted into the max-priority queue Q that represents the set V_0 of vertices that are not yet traversed by the algorithm. When a vertex u is inserted into Q, its key is $d[u]$.

Algorithm 4 Relaxation function applied on the capacity attribute

```
1: function relax(u, v)
2:     new ← min {d[u], g(u, v)}
3:     if d[v] < new then
4:         d[v] ← new
5:         update(Q, v, new)                ▷ increase the key of v to store new
6:     end if
7: end function
```

On each iteration of the **while** loop of Algorithm 5, the vertex with the maximal capacity estimate u is extracted from the priority queue, and its neighbors are processed by the relaxation function in Algorithm 4. Thus, on each iteration a vertex u is removed from the set V_0, and it is transferred into the set $U = V \setminus V_0$ of traversed vertices.

When an edge is relaxed (Algorithm 4), a new estimate for the capacity of the end vertex is calculated as the minimum of the current estimate and the corresponding weight $g(u, v)$. If the new estimate surpasses the existing one, it replaces it in the array d and also the corresponding key is increased in the Fibonacci heap by the function call on line 5.

Algorithm 5 Calculate the capacity of the vertex n

```
1: function capacity(N.Adj)
2:     d[1] = ∞, d[i] = −∞, for i = 2, 3, . . . , n
3:     for each u ∈ V do
4:         insert(Q, u)                     ▷ max-priority queue keyed by d
5:     end for
6:     while Q ≠ ∅ do
7:         u ← extract(Q)
8:         for each vertex v ∈ N.Adj[u] do
9:             relax(u, v)
10:        end for
11:    end while
12:    return d[n]
13: end function
```

The **while** loop of the *capacity*$(N.Adj)$ function will terminate when the priority queue Q does not contain any vertices, or with other words, when V_0 is empty and all vertices (except the destination vertex n) have been traversed by the algorithm. Then the element with index n of the vector d will contain the capacity of the destination vertex.

Proposition 3 *The function capacity*$(N.Adj)$ *correctly calculates the capacity of the destination vertex* n *in the network* N*.*

Proof In analogy to the proof of Proposition 1, we denote the set of vertices that are not yet traversed on the k-th iteration with V_k, the set of all traversed with U_k and the set of outgoing neighbors of the vertices in U_k with X_k. We will use the method of mathematical induction to analyze the calculations of the **while** loop of the algorithm.

Base case. The base case of the induction is defined by the first two iterations of the **while** loop.

The *first iteration* results in the extracted node $u_1 = 1$, and the corresponding sets $U_1 = \{1\}$ and $X_1 = \{v : v \in N.Adj[1]\}$. The components of the capacity estimates array d_0 are modified and $d_1[v] = g(1, v)$ for all v that are adjacent to the vertex 1.

The *second iteration* extracts u_2 from Q, such that its corresponding key $d_1[u_2]$ is maximal. We will assume that $d_1[u_2] \neq -\infty$, otherwise u_1 is an isolated vertex in N. The corresponding sets are defined $U_2 = U_1 \cup \{u_2\}$ and $X_2 = \{v : v \in N.Adj[1] \cup N.Adj[u_2]\}$.

Now let α is an arbitrary $(1, u_2)$-path. We will examine the two possibilities:

1. α contains only two vertices. Then $y(\alpha) = g(1, u_2) = d_1[u_2]$.
2. $\alpha = (1, v, \ldots, u_2)$, where $v \neq u_2$. Then $y(\alpha) \leq g(1, v) = d_1(v) \leq d_1(u_2)$.

This proves that $d_1[u_2] = \kappa(u_2)$.

The relaxation step is performed for each $v \in G.Adj[u_2]$ and as a result $d_2[v] = \max\{d_1[v], \min\{\kappa(u_2), g(u_2, v)\}\}$. Therefore, $d_2[v] = -\infty$, for each $v \in V_2 \setminus X_2$. Besides that, for every $v \in V_2 \cap X_2$ exists a path α, such that $y(\alpha) = d_2[v]$ and $\alpha = (1, v)$ or $\alpha = (1, u_2, v)$.

From the definition of $d_2[v]$ it follows that $d_1[v] \leq d_2[v] \leq \kappa(v)$, for all $v \in V_0$. For each $v \in V_2 \setminus X_2$ it holds that $d_2[v] = -\infty$. Now we will prove that for each $u \in U_2$ and each $v \in V_2$ the following inequality holds:

$$\kappa(u) \geq \kappa(v). \tag{14}$$

Let $\alpha = (1, v_2, \ldots, v_l, v)$ is a path, such that $y(\alpha) = \kappa(v)$. Two cases are possible:

1. $v_2 = u_2$ and then $y(\alpha) \leq g(1, u_2) = \kappa(u_2)$;
2. $v_2 \neq u_2$ and then $v_2 \notin U_2$ and $y(\alpha) \leq g(1, v_2) = d_1[v_2] \leq d_1[u_2] = \kappa(u_2)$.

Therefore, for the path α it holds that $\kappa(v) = y(\alpha) \leq \kappa(u_2)$. Since $\kappa(u_2) \leq \kappa(1)$, the inequality (14) is proved.

Inductive step. We assume that the **while** loop has performed k iterations and the sets V_k, U_k and X_k are defined. We also assume that Property 3 holds.

Property 3 For the k-th iteration of the Algorithm 5 we define the following properties:

1. $d_k[v] = \begin{cases} \kappa(v), & \text{if } v \in U_k \\ \max\limits_{u \in U_k, (u,v) \in E} \{\min\{\kappa(u), g(u, v)\}\}, & \text{if } v \in V_k \cap X_k \\ -\infty, & \text{if } v \in V_k \setminus X_k \end{cases}$
2. $U_k = \{u_1, u_2, \ldots, u_k\}$ and $\kappa(u_1) \geq \kappa(u_2) \geq \ldots \geq \kappa(u_k)$, where $u_1 = 1$.
3. For each $v \in V_k \cap X_k$ exists a path $\alpha = (1, v_2, \ldots v_l, v)$ such that $y(\alpha) = d_k[v]$, the vertices in the path $\alpha_1 = (v_1, v_2, \ldots, v_l)$ belong to U_k and $y(\alpha_1) = \kappa(v_l)$.
4. $\kappa(u) \geq \kappa(v)$ for each $u \in U_k$ and each $v \in V_k$.

We will prove that Property 3 holds for the sets V_{k+1}, U_{k+1}, X_{k+1} and the array d_{k+1} that are defined by the $(k+1)$-st iteration of the algorithm.

It is clear that if $\max\limits_{v \in V_k}\{d_k[v]\} = -\infty$, then all vertices from V_k are unreachable from the source vertex $u_1 = 1$. In this case the array d_k remains unmodified by the iterations of the

while loop and $\kappa(v) = -\infty$ for each $v \in V_k$. That is why we will examine the case in which $\max_{v \in V_k}\{d_k[v]\} \neq -\infty$.

On line 7 the current u_{k+1} is extracted from the priority queue Q and by definition, it is the vertex with the maximal corresponding key. Also, this sets $U_{k+1} = U_k \cup \{u_{k+1}\}$ and $X_{k+1} = \{v : v \in X_k \cup N.Adj[u_{k+1}]\}$ are defined. We will prove that:

$$d_k[u_{k+1}] = \kappa(u_{k+1}). \tag{15}$$

Let $\beta = (1, v_1, \ldots, v_{s-1}, v_s, \ldots, v_l, u_{k+1})$ is an arbitrary $(1, u_{k+1})$-path. We denote with s the smallest index, such that $v_s \notin U_k$. Then $v_j \in U_k$ for each $j = 1, 2, \ldots, s-1$. For the path $\beta_1 = (1, v_1, \ldots, v_{s-1}, v_s)$, from Property 3.1 it follows that $y(\beta_1) \leq \min\{\kappa(v_{s-1}), g(v_{s-1}, v_s)\} \leq d_k[v_s]$. Therefore the inequality holds:

$$y(\beta) \leq y(\beta_1) \leq d_k[v_s] \leq d_k[u_{k+1}], \tag{16}$$

which proves equality (15).

Since $u_{k+1} \in V_k$ and $u_k \in U_k$, then from Property 3.4 it follows that $\kappa(u_k) \geq \kappa(u_{k+1})$. This proves that Property 3.2 is fulfilled for the set U_{k+1}.

Now we will prove that for each $v \in V_{k+1}$, the following inequality is fulfilled:

$$\kappa(u_{k+1}) \geq \kappa(v). \tag{17}$$

Let $\gamma = (1, v_1, \ldots, v_{s-1}, v_s, v_{s+1}, \ldots, v_l, v)$ is a path, such that $y(\gamma) = \kappa(v)$. With s we denote the smallest index for which $v_s \notin U_k$. In analogy to inequality (16) we prove that for the path $\gamma_1 = (1, v_1, \ldots, v_{s-1}, v_s)$ it holds $y(\gamma) \leq y(\gamma_1) \leq d_k[v_s] \leq d_k[u_{k+1}]$. This inequality proves (17), because $d_k[u_{k+1}] = \kappa(u_{k+1})$ and because of the choice of γ, it holds that $y(\gamma) = \kappa(v)$.

The inequality (17) and the already proved Property 3.2 for the set U_{k+1}, prove Property 3.4 for U_{k+1} and V_{k+1}.

The relaxation function modifies the array d_k for all vertices $v \in N.Adj[u_{k+1}]$, so that $d_{k+1}[v] = \max\{d_k[v], \min\{\kappa(u_{k+1}), g(u_{k+1})\}$. For the vertices that are not adjacent to u_{k+1}, the components of d_{k+1} remain unchanged.

For the array d_{k+1}, it hods that $d_k[v] \leq d_{k+1}[v]$ for all $v \in V_0$. Particularly, for each $u \in U_{k+1}$ it holds that $\kappa(u) = d_k[u] \leq d_{k+1}[u] \leq \kappa(u)$, or in other words $d_{k+1}[u] = \kappa(u)$.

We will complete the proof of Property 3.1 for d_{k+1} by examining the following two cases: $v \in V_{k+1} \setminus X_{k+1}$ and $v \in V_{k+1} \cap X_{k+1}$.

If $v \in V_{k+1} \setminus X_{k+1}$, then $v \notin \bigcup_{u \in U_{k+1}} N.Adj[u]$ and from the inductive assumption it follows that $d_{k+1}[v] = d_k[v] = -\infty$.

Now we will examine the case in which $v \in V_{k+1} \cap X_{k+1}$.

If $v \notin N.Adj[u_{k+1}]$, then from the relaxation function follows that $d_{k+1}[v] = d_k[v]$ and from the inductive assumption the following holds:

$$\begin{aligned} d_{k+1}[v] = d_k[v] &= \max_{u \in U_k, v \in N.Adj[u]}\{\min\{\kappa(u), g(u,v)\}\} \\ &= \max_{u \in U_{k+1}, v \in N.Adj[u]}\{\min\{\kappa(u), g(u,v)\}\}. \end{aligned} \tag{18}$$

If $v \in V_{k+1} \cap N.Adj[u_{k+1}]$, then from relaxation function and the inductive assumption:

$$\begin{aligned} d_{k+1}[v] &= \max\left\{\max_{u \in U_k, v \in N.Adj[u]}\{\min\{\kappa(u), g(u,v)\}, \min\{\kappa(u_{k+1}), g(u_{k+1}, v)\}\right\} \\ &= \max_{u \in U_{k+1}, v \in N.Adj[u]}\{\min\{\kappa(u), g(u,v)\}. \end{aligned} \tag{19}$$

Let $v \in V_{k+1} \cap X_{k+1}$. We will prove that there exists a path $\alpha = (1, v_1, \ldots, v_l, v)$, such that $y(\alpha) = d_{k+1}[v]$, the vertices of the path $\alpha_1 = (1, v_1, \ldots, v_l)$ belong to U_{k+1} and $y(\alpha_1) = \kappa(v_l)$.

Indeed, if $d_{k+1}[v] = d_k[v]$, then the above statement follows from the inductive assumption. If $d_{k+1}[v] \neq d_k[v]$, then $d_{k+1}[v] = \min\{\kappa(u_{k+1}), g(u_{k+1}, v)\}$. In this case, according to the inductive assumption, there exists a path $\alpha_1 = (1, v_1, \ldots, v_l, u_{k+1})$, such that $y(\alpha_1) = d_k[u_{k+1}] = \kappa(u_{k+1})$ and $v_j \in U_k$ for each $j = 1, 2, \ldots, l$. Then, the path $\alpha = (1, v_1, \ldots, v_l, u_{k+1}, v)$ has capacity $y(\alpha) = \min\{\kappa(u_{k+1}), g(u_{k+1}, v)\} = d_{k+1}[v]$ and the vertices of α_1 belong to U_{k+1}.

With this we have proved that Property 3 holds for the sets U_{k+1}, V_{k+1}, X_{k+1} and the array d_{k+1} that are defined by the $(k+1)$-st iteration of the algorithm. Besides that, in the base step of the induction we have proved that Property 3 hods for the second iteration of the algorithm. From here we can conclude that the $(n-1)$-st iteration defines the corresponding U_{n-1}, V_{n-1}, X_{n-1} and d_{n-1} for which Property 3 holds. In particular, $d_{n-1}[u] = \kappa(u)$ for each $u \in U_{n-1}$, and the set $V_{n-1} = \{v\}$ contains a single vertex. Then, for an arbitrary path $\alpha = (1, v_1, \ldots, v_l, v)$, for which $v_j \neq v$, and with the help of Property 3.1 we get:

$$y(\alpha) \leq \min\{\kappa(v_l), g(v_l, v)\} \leq d_{n-1}[v]. \tag{20}$$

The inequality (20) shows that $d_{n-1}[v] = \kappa(v)$, completing the proof of the proposition. □

We will illustrate the Algorithm 5 with the following Example 2.

Example 2 Using Algorithm 5 we will calculate the capacity array for the input network N_1 with adjacency list (13) that is shown in Figure 1.

The initialization steps of the algorithm determine $d_0 = [\infty, -\infty, -\infty, -\infty, -\infty]$ and $V_0 = \{1, 2, 3, 4, 5\}$, where V_0 is stored in the max-priority queue Q keyed by the corresponding values in d_0. No vertices are traversed yet and both sets U_0 and X_0 are empty.

On the *first iteration* of the algorithm, the vertex $u_1 = 1$ is extracted from Q which now stores $V_1 = \{2, 3, 4, 5\}$. The set of traversed vertices is $U_1 = \{1\}$, and the corresponding set of their neighbors is $X_0 = \{2, 3\}$. The relaxation function edits the array d_0 and we get $d_1 = [\infty, 4, 3, -\infty, -\infty]$.

The *second iteration* extracts the vertex $u_2 = 2$ from the max-priority queue Q and now it stores $V_2 = \{3, 4, 5\}$. The set of the traversed vertices is $U_2 = \{1, 2\}$ with its corresponding set of neighbors $X_2 = \{2, 3, 4, 5\}$. In Figure 1 it is obvious that that the capacity of the vertex 2 is $\kappa(2) = d_1[2] = 4$. This is not true for the vertex 3 because $\kappa(3) \neq d_1[3] = 3$. The relaxation function modifies the elements with indexes 3, 4 and 5 of the array d_1. The result is $d_2 = [\infty, 4, 4, 4, 3]$.

The **while** loop of the algorithm executes two more iterations in which it calculates:

- $U_3 = \{1, 2, 3\}$, $V_3 = \{4, 5\}$ and $d_3 = [\infty, 4, 4, 4, 3]$;
- $U_4 = \{1, 2, 3, 4\}$, $V_3 = \{5\}$ and $d_4 = [\infty, 4, 4, 4, 4]$.

From the proof of Proposition 3 it follows that $d_4[5] = \kappa(5)$ and therefore the capacity array is $d_4 = [\infty, 4, 4, 4, 4]$.

Based on the function *capacity*($N.Adj$), we will formulate Algorithm 6 that constructs the maximal capacity digraph $\widetilde{G}$, given in Definition 6. The algorithm stores in the digraph $\widetilde{G}$ only those edges from the network N, that have capacity that is bigger or equal to the capacity of the vertex n.

Algorithm 6 Compose the adjacency list of the maximal capacity digraph $\widetilde{G}$

```
1:  function maxmin(N.Adj)
2:      c ← capacity(N.Adj)
3:      for each u ∈ V do
4:          for each vertex v ∈ N.Adj[u] do
5:              if g(u, v) ≥ c then
6:                  G̃.Adj[u] ← G̃.Adj[u] ∪ {v}
7:              end if
8:          end for
9:      end for
10:     return c, G̃.Adj
11: end function
```

On the initial step, the function $maxmin(N.Adj)$ calculates the capacity of the vertex n using the $capacity(N.Adj)$ function and stores it in c. Then the algorithm consecutively visits all the edges of N, as they are stored in the adjacency list of the network. When the capacity of a given edge is bigger or equal to c, it is stored in the adjacency list of $\widetilde{G}$.

Proposition 4 *The function $maxmin(N.Adj)$ correctly constructs the maximal capacity digraph $\widetilde{G}$.*

Proof From the proof of Proposition 3, it follows that c is the capacity of the vertex n in the network N.

Let α is a $(1, n)$-path in the network N with capacity $y(\alpha) = c$. Then, each edge (u, v) in α has a capacity $g(u, v) \geq c$. Therefore, α is a $(1, n)$-path in $\widetilde{G}$.

Now let β is a $(1, n)$-path in $\widetilde{G}$. By the definition of the adjacency list of $\widetilde{G}$, each edge (u, v) in β has capacity $g(u, v) \geq c$, therefore $y(\beta) \geq c$. Since c is the capacity of the vertex n, then $y(\beta) \leq c$ and hence, $y(\beta) = c$. □

Example 3 We will calculate the maximal capacity digraph $\widetilde{G}_1$ of the input network N_1 with adjacency list (13) that is shown in Figure 1.

In Example 2 we have examined in details how the function $cpacity(N.Adj)$ calculates the array of capacities $d = [\infty, 4, 4, 4, 4]$ of the network N_1. Particularly, in line 2 of Algorithm 6 we get that the capacity of the vertex $n = 5$ is $c = d[5] = 4$. The outer **for** loop will execute four iterations – one for each vertex that has a list of outgoing neighbors in $N_1.Adj$.

The inner **for** loop will include each edge with capacity bigger or equal to $c = 4$ into the corresponding list in $\widetilde{G}_1.Adj$. Thus, consecutively we get the adjacency lists that construct:

$$\widetilde{G}_1.Adj = [\langle 2\rangle, \langle 3, 4\rangle, \langle 4\rangle, \langle 5\rangle, \langle \varnothing\rangle]. \tag{21}$$

Figure 2 shows the maximal capacity digraph $\widetilde{G}_1$. From the representation of the graph it is obvious that there are exactly two $(1, 5)$-paths: $\alpha = (1, 2, 4, 5)$ and $\beta = (1, 2, 3, 4, 5)$. We can verify directly that $y(\alpha) = y(\beta) = 4 = c$.

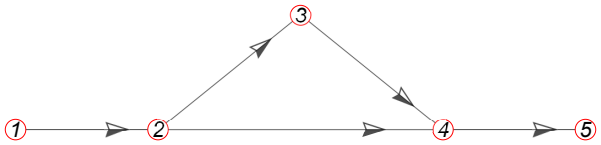

Fig. 2 The maximal capacity digraph $\widetilde{G}_1$ of the example network N_1

Proposition 5 *The computational complexity of the function maxmin*$(N.Adj)$ *is* $O(n \log n + m)$.

As in the case of Proposition 2, the proof follows directly from the computational complexity of the Dijkstra's algorithm when the priority queue is implemented using a Fibonacci heap [19], because the function *capacity*$(N.Adj)$ follows exactly its steps. Also, the construction of the digraph $\widetilde{G}$ itself has computational complexity $O(m)$.

3.3 Complete Pareto front algorithm

We will describe the general concept of Algorithm 7 that constructs the list L of all equivalence classes P_i, given in Definition 4, using the following procedure:

1. Set $Y_0 = W_n$, where W_n is the set of all $(1, n)$-paths in N.
2. Calculate the distance $d_i = \min\limits_{\beta \in Y_0}\{x(\beta)\}$ and define the set of paths $X_i = \{\alpha \in Y_0 : x(\alpha) = d_i\}$.
3. Calculate the capacity $c_i = \max\limits_{\beta \in X_i}\{y(\beta)\}$ and define the equivalence classes $P_i = \{\alpha \in X : y(\alpha) = c_i\}$. Store P_i into the list L.
4. Define the restricted set of paths $Y_i = \{\beta \in Y_0 : y(\beta) > c_i\}$.
5. Define the set of paths $Z_i = Y_0 \setminus (Y_i \cup P_i)$.
6. If $Y_i = \varnothing$, then the procedure stops. Otherwise, set $Y_0 = Y_i$, and go to step 2.

Lemma 1 *Each iteration of the above procedure defines one equivalence class* P_i. *After a finite number of iterations the procedure stops and in* L *are stored all sets* P_i *that are given in Definition 4.*

Proof We will use the method of mathematical induction. We initialize the distance $d_0 = 0$ and capacity $c_0 = 0$, equivalence class $P_0 = \varnothing$, and the set of paths $Z_0 = \varnothing$, $Y_0 = W_n$, and the list $L = \langle \varnothing \rangle$.

Base case. The base case of the induction is defined by the first iteration of the procedure which results in:

$$d_1 = \min_{\beta \in Y_0}\{x(\beta)\}, X_1 = \{\alpha \in Y_0 : x(\alpha) = d_1\},$$
$$c_1 = \max_{\beta \in X_1}\{y(\beta)\}, P_1 = \{\alpha \in X_1 : y(\alpha) = c_1\},$$
$$L = \langle P_1 \rangle, Y_1 = \{\beta \in Y_0 : y(\beta) > c_1\}, Z_1 = Y_0 \setminus (Y_1 \cup P_1). \tag{22}$$

We will prove that the properties given in Property 4 hold for (22) in the case in which $k = 1$.

Property 4 The following properties hold, where k is the current number of iterations performed by the procedure.

1. $W_n = \bigcup_{i=1}^{k} (Z_i \cup P_i) \cup Y_k$, and the sets Z_i, P_i, where $i = 1, 2, \ldots, k$, and Y_k do not have common elements.
2. $P_1, P_2, \ldots, P_k$ are equivalence classes, for which:
   (a) $d_i = x(\alpha)$ and $c_i = y(\alpha)$ for all $\alpha \in P_i$, $i = 1, 2, \ldots k$;
   (b) $d_{i-1} < d_i$ and $c_{i-1} < c_i$ for all $i = 1, 2, \ldots k$.
3. Each path β is dominated by each path α, $\beta \prec \alpha$, where $\beta \in Z_i$ and $\alpha \in P_i$, for all $i = 1, 2, \ldots k$.
4. For each $\beta \in Y_k$ the inequalities $c_k < y(\beta)$ and $d_k < x(\beta)$ hold.
5. $L = \langle P_1, P_2, \ldots, P_k \rangle$.

The first two properties and the last one are obvious.

We will prove Property 4.3. From the condition $\alpha \in P_1$ it follows that $x(\alpha) = d_1$ and $y(\alpha) = c_1$, and from the condition $\beta \in Z_1$ it follows that $\beta \notin Y_1$ and $\beta \notin P_1$. Then, $y(\beta) \le c_1 = y(\alpha)$ and there are possible two cases:

1. $y(\beta) < c_1 = y(\alpha)$ and $x(\beta) \ge d_1 = x(\alpha)$;
2. $y(\beta) = c_1 = y(\alpha)$ and $x(\beta) > d_1 = x(\alpha)$, because $\beta \notin P_1$.

In both cases we have that $\beta \prec \alpha$.

Property 4.4 follows directly from the definitions of c_1 and d_1. Indeed, if $\beta \in Y_1$, then from the definition of Y_1 it follows that $y(\beta) > c_1$. Then, from the definition of c_1 it follows that $x(\beta) \ne d_1$, and therefore $x(\beta) > d_1$.

If $Y_1 = \varnothing$, then $W_n = P_1 \cup Z_1$ and P_1 is the equivalence class. This means that the number of equivalence classes $K = 1$ in (8), and the procedure stops.

If $Y_1 \ne \varnothing$, then the constant $K > 1$. In this case, for each $\beta \in Y_1$ the following inequalities hold:

$$c_1 < y(\beta) \quad \text{and} \quad d_1 < x(\beta). \tag{23}$$

Therefore, in this case P_1 is the set of Pareto optimal paths α and it is correctly included in the list L. In this case we set $Y_0 = Y_1$ and the procedure goes back to step 2 for its second iteration.

Inductive step. We assume that after $k \ge 1$ iterations of the procedure the sets P_i, Z_i, Y_k and numbers d_i and c_i for $i = 1, 2, \ldots, k$ are defined, for which Property 4 hods.

It is clear that if $Y_k = \varnothing$, then $W_n = \bigcup_{i=1}^{k} (Z_i \cup P_i)$ and the list L is composed correctly.

We assume that $Y_k \ne \varnothing$. Then the procedure executes its $(k+1)$-st iteration, which results in:

$$d_{k+1} = \min_{\beta \in Y_k} \{x(\beta)\}, X_{k+1} = \{\alpha \in Y_k : x(\alpha) = d_{k+1}\},$$

$$c_{k+1} = \max_{\beta \in X_{k+1}} \{y(\beta)\}, P_{k+1} = \{\alpha \in X_{k+1} : y(\alpha) = c_{k+1}\},$$

$$L = \langle P_1, P_2, \ldots, P_{k+1} \rangle, Y_{k+1} = \{\beta \in Y_k : y(\beta) > c_{k+1}\}, Z_{k+1} = Y_k \setminus (Y_{k+1} \cup P_{k+1}). \tag{24}$$

We will prove that for the sets and numbers in (24) hold properties given in Property 4. The proof will be composed by four steps.

Step 1. From the definitions of the sets P_{k+1}, Y_{k+1} and Z_{k+1} it follows that they do not have common elements, and it is fulfilled that $Y_k = Z_{k+1} \cup P_{k+1} \cup Y_{k+1}$. Taking into account the inductive assumption Property 4.1, we get that $W_n = \bigcup_{i=1}^{k+1} (Z_i \cup P_i) \bigcup Y_{k+1}$.

Step 2. Let $\alpha \in P_{k+1}$. Then from the definition of P_{k+1} it follows that $x(\alpha) = d_{k+1}$ and $y(\alpha) = c_{k+1}$. Since $\alpha \in X_{k+1} \subseteq Y_k$, from the inductive assumption Property 4.1 it follows $c_k < y(\alpha) = c_{k+1}$ and $d_k < x(\alpha) = d_{k+1}$.

Step 3. We will prove that for each $\beta \in Z_{k+1}$ and $\alpha \in P_{k+1}$, we have that $\beta \prec \alpha$. From $\beta \in Z_{k+1}$ it follows that $\beta \in Y_k$, $\beta \notin Y_{k+1}$ and $\beta \notin P_{k+1}$. Then, from the definition of d_{k+1} it follows that $x(\beta) \geq d_{k+1}$, and from the definition of Y_{k+1} it follows that $y(\beta) \leq c_{k+1}$. As a result, there are exactly two possibilities for $y(\beta)$:

1. $y(\beta) < c_{k+1}$ and $x(\beta) \geq d_{k+1}$, or
2. $y(\beta) = c_{k+1}$ and then $x(\beta) > d_{k+1}$, because $\beta \notin P_{k+1}$.

It is enough to note that if $\alpha \in P_{k+1}$, then $x(\alpha) = d_{k+1}$ and $y(\alpha) = c_{k+1}$, hence $\beta \prec \alpha$.

Step 4. Let $\beta \in Y_{k+1}$. We will prove that $y(\beta) > c_{k+1}$ and $x(\beta) > d_{k+1}$. The inequality

$$y(\beta) > c_{k+1} \tag{25}$$

follows directly from the definition of Y_{k+1}. Besides that, from $Y_{k+1} \subset Y_k$ it follows that $x(\beta) \geq d_{k+1}$. Something more, if we assume that $x(\beta) = d_{k+1}$, we will get a contradiction with the inequality (25). Indeed, if $x(\beta) = d_{k+1}$, then $\beta \in X_{k+1}$ and therefore $y(\beta) \leq c_{k+1}$. The resulting contradiction proofs that:

$$x(\beta) > d_{k+1}. \tag{26}$$

The two inequalities (25) and (26) prove that if $\alpha \in P_{k+1}$ and $\beta \in Y_{k+1}$, then α and β cannot be compared.

To finalize the proof, we note that in the base step of the induction we have proved that Property 4 holds for $k = 1$. The inductive step proves that after each consecutive k-th iteration, Property 4 holds. Since the elements of W_n are finite number and $P_i \neq \varnothing$ for each index i, then after a finite number of K iterations, the procedure stops. □

Corollary 1 *For the equivalence classes P_i, it hods that*

$$P_i = \{\alpha \in W_n : \ x(\alpha) = d_i \ \textit{and}\ y(\alpha) = c_i\}, \tag{27}$$

for each $i = 1, 2, \ldots, K$.

Corollary 2 *For P_K, the following property holds:*

$$P_K = \{\alpha \in X_1' : \ x(\alpha) = d_1'\}, \tag{28}$$

where $X_1' = \{\alpha \in W_n : y(\alpha) = \max\limits_{\beta \in W_n} \{y(\beta)\}$ and $d_1' = \min\limits_{\beta \in X_1'} \{x(\beta)\}$.

Corollary 3 *Let $\kappa(n) = \max\limits_{\beta \in W_n} \{y(\beta)\}$ is the capacity of the vertex n. If $c_1 = \kappa(n)$, then all Pareto optimal solutions are equivalent and $K = 1$.*

We will point out that the list L can be composed by firstly applying the Corollary 2 and separate the set P_K. Then consecutively separate the sets P_{K-1}, P_{K-2}, etc., until P_1 is separated.

We will concretize the described procedure for construction of the list of all equivalence classes with the function $front(N.Adj)$ that is given in Algorithm 7. In this algorithm, for each $i = 1, 2, \ldots, K$ we define a special digraph G_i, such that P_i is the list of its $(1, n)$-paths. The result is stored in the list L which in this representation will contain the adjacency lists $G_i.Adj$ of the digraphs G_i.

Apart from the functions $minsum(N.Adj)$ and $outadj(p)$, defined in Section 3.1, and functions $maxmin(N.Adj)$ and $capacity(N.Adj)$, defined in Section 3.2, in Algorithm 7 we will use the function $restrict(N.Adj, c)$. It takes as parameters the adjacency list of a network and a number c. The result is the adjacency list composed by the edges (u, v) of the input network for which $g(u, v) > c$.

Algorithm 7 Calculate the complete Pareto front given in Problem 1

```
1:  function front(N.Adj)
2:      R.Adj ← N.Adj                          ▷ copy of the adjacency list to be restricted
3:      c_0 ← capacity(N.Adj)
4:      more ← true
5:      while more = true do
6:          d, N.Adj ← minsum(N.Adj)
7:          if d = ∞ then
8:              more ← false
9:          else
10:             c_1, G̃.Adj ← maxmin(N.Adj)
11:             L ← L ∪ G̃.Adj
12:             if c_1 = c_0 then
13:                 more ← false
14:             else
15:                 R.Adj ← restrict(R.Adj, c_1)
16:             end if
17:             N.Adj ← R.Adj
18:         end if
19:     end while
20:     return L
21: end function
```

The Algorithm 7 returns as result the list L, where $|L| = K$, and each of the elements of the list give complete description of the sets P_i given in (8). If after the termination of the algorithm $K = 0$, then $(1, n)$-path does not exist and the vertex n is isolated.

We will note that, the list L allows us easily to construct a list P' of Pareto optimal solutions by taking a predefined number of elements from each class P_i. Particularly, we can compose the list P of all Pareto optimal solutions.

Theorem 1 *The function $front(N.Adj)$ terminates after K iterations of its* ***while*** *loop, where K is given in (8). Each set P_i, $i = 1, 2, \dots K$ defined in Definition 4, is the set of all $(1, n)$-paths of the sub-graph with an adjacency list given as the i-th element of the list L.*

Proof From the proof of Lemma 1 it follows that to prove Theorem 1, it is enough to show that Algorithm 7 implements the procedure that constructs the list of all equivalence classes.

Step 2 of the procedure is implemented with the function call on line 6 of the algorithm. From the correctness of the function $minsum(N.Adj)$, it follows that $d = \min\limits_{\beta \in W_n} \{x(\beta)\}$ stores the current distance of the vertex n, and that the adjacency list $N.Adj$ is replaced by the adjacency list of the shortest path subnetwork $\widehat{N}.Adj$, which uniquely defines the set X_i for the i-th iteration of the procedure.

The **if** statement on line 7 verifies whether the network N has at least one $(1, n)$-path. If $d = \infty$, the algorithm will terminate. Otherwise, line 10 of the algorithm calculates the capacity $c_1 = \max\limits_{\beta \in X_i} \{y(\beta)\}$ of the vertex n of the shortest path subnetwork. Also, the set of all $(1, n)$-paths in the digraph $\widetilde{G}.Adj$ is the set P_i of all paths from X_i that have capacity c_1. That is why, on line 11 the adjacency list $\widetilde{G}.Adj$ is included in the list L. This is how step 3 of the procedure is implemented in the algorithm. Besides that, from Lemma 1 it follows that P_i is the corresponding equivalence class from (8).

If the capacity of the shortest path subnetwork c_1 is equal to the capacity c_0 of the input network N, then the algorithm stops. In this case N does not contain $(1, n)$-path with capacity that is bigger than c_1, the set Y_i in step 4 of the procedure is the empty set, and the procedure also terminates.

Otherwise, it means that $c_1 < c_0$. In this case the algorithm calculates the restricted adjacency list and stores it into $R.Adj$. It contains the edges (u, v), for which $g(i, j) > c_1$. Therefore, the set Y_i in step 4 of the i-th iteration of the procedure is the set of all $(1, n)$-paths in R. Obviously, $Y_i \neq \varnothing$ exactly when c_1 is less than the capacity of the vertex n.

On line 17 the algorithm replaces the adjacency list $N.Adj$ with the adjacency list of the restricted subnetwork $R.Adj$. Then the algorithm proceeds the next iteration. Analogously, the procedure sets $Y_0 = Y_1$ and goes to step 2, since $Y_i \neq \varnothing$. Therefore, the **while** loop of the algorithm executes the same number of iterations as the procedure. □

The proof of the Theorem 1 is illustrated with the following Example 4.

Example 4 We will trace the execution of the function $front(N.Adj)$ for the input network N_1 that is given in (13) and Figure 1.

On the initial stage of the algorithm (lines 2 – 4) the adjacency list of the input network is copied in a temporary adjacency list $R.Adj$. The network R will be recomposed on line 15 using the function $restrict(R.Adj, c_1)$ for the capacity c_1 calculated on line 10. The capacity of the destination vertex $n = 5$ of input network N_1 is calculated $c_0 = 4$. Also, the Boolean variable *more*, that is used to control the iterations of the loop, is initialized with *true*.

Iteration $k = 1$ of the **while** loop in line 6 calculates the distance to vertex $n = 5$, $d = 6$, and updates $N.Adj$ to store the adjacency list of the shortest path subnetwork $\widehat{N}_1.Adj = [\langle(2, 2, 4), (3, 5, 3)\rangle, \langle(3, 3, 5), (4, 6, 4)\rangle, \langle(4, 3, 6), (5, 1, 1)\rangle, \langle\varnothing\rangle, \langle\varnothing\rangle]$. In Example 1 we have shown that $\widehat{N}_1$ has exactly two $(1, 5)$-paths $\alpha_1 = (1, 3, 5)$ and $\beta_1 = (1, 2, 3, 5)$, and both of them have the same length $x(\alpha_1) = x(\beta_1) = 6$. Also, traversing the network N_1, we find that the set of all its $(1, 5)$-paths is:

$$Y_0 = \{(1, 2, 5), (1, 3, 5), (1, 2, 3, 5), (1, 2, 4, 5), (1, 3, 4, 5), (1, 2, 3, 4, 5)\}. \tag{29}$$

We can directly verify that besides α_1 and β_1, Y_0 does not contain another $(1,5)$-path with length equal to $d = 6$, which follows from the correctness of the function $minsum(N.Adj)$. In the procedure, the set $\{\alpha_1, \beta_1\}$ is denoted with X_i for $i = 1$.

The **if** statement will terminate the algorithm if $d = \infty$, or in other words, the vertex n is isolated. In this particular case the condition is not fulfilled, and the algorithm proceeds to the body of the **else** statement.

On line 10 the algorithm calculates that the capacity of the terminal vertex in the current shortest paths subnetwork subnetwork is $c_1 = 1$ and composes the maximal capacity digraph $\widetilde{G}.Adj = [\langle 2,3\rangle, \langle 3,4\rangle, \langle 4,5\rangle, \langle\varnothing\rangle, \langle\varnothing\rangle]$. It can be directly verified that $\widetilde{G}$ contains exactly two $(1,5)$-paths: $\alpha_1 = (1,3,5)$ and $\beta_1 = (1,2,3,5)$. In this way, $\widetilde{G}$ uniquely defines the set $P_1 = \{\alpha_1, \beta_1\}$, and its adjacency list is stored as the first element of the list L.

If the two capacities c_1 and c_0 are equal, the algorithm will terminate. In this particular case, $c_1 = 1 < c_0 = 4$ and the algorithm will compose the adjacency list of of the restricted subnetwork with edge capacities less or equal to $c_1 = 1$: $R.Adj = [\langle(2,2,4),(3,5,3)\rangle, \langle(3,3,5),(4,6,4),(5,5,3)\rangle, \langle(4,3,6)\rangle, \langle(5,1,7)\rangle, \langle\varnothing\rangle]$.

Finally, the algorithm stores $R.Adj$ into $N.Adj$ and proceeds to iteration $k = 2$.

The calculations of the iterations $k = 2$ and $k = 3$ are given in Table 2. Note that on the third iteration the capacity of the terminal vertex in the current shortest paths subnetwork subnetwork is $c_1 = 4$ and the **if** statement on line 12 will terminate the algorithm.

Table 2 Calculations of iterations 2 and 3 of Algorithm 7

Iteration	Distance	Shortest paths subnetwork	Capacity	Maximal capacity digraph
$k = 2$	$d = 7$	$\widehat{N}.Adj = [\langle(2,2,4),(3,5,3)\rangle,$ $\langle(3,3,5),(4,6,4),(5,5,3)\rangle,$ $\langle(4,3,6)\rangle, \langle\varnothing\rangle, \langle\varnothing\rangle]$	$c_1 = 3$	$\widetilde{G}.Adj = [\langle 2,3\rangle, \langle 3,4,5\rangle,$ $\langle 4\rangle, \langle\varnothing\rangle, \langle\varnothing\rangle]$
$k = 3$	$d = 9$	$\widehat{N}.Adj = [\langle(2,2,4)\rangle,$ $\langle(3,3,5),(4,6,4)\rangle,$ $\langle(4,3,6)\rangle, \langle(5,1,7)\rangle, \langle\varnothing\rangle]$	$c_1 = 4$	$\widetilde{G}.Adj = [\langle 2\rangle, \langle 3,4\rangle,$ $\langle 4\rangle, \langle 5\rangle, \langle\varnothing\rangle]$

The above solution shows that the network N_1 has exactly five Pareto optimal paths, which are distributed in three equivalence classes: $L(1) = P_1 = \{(1,3,5),(1,2,3,5)\}$, $L(2) = P_2 = \{(1,2,5)\}$ and $L(3) = P_3 = \{(1,2,4,5),(1,2,3,4,5)\}$.

Theorem 2 *The computational complexity of the function $front(N.Adj)$ is $K \cdot O(n \log n + m)$, where K is the number of classes of Pareto equivalent paths.*

The proof follows directly from the computational complexity of the functions $minsum(N.Adj)$, $capacity(N.Adj)$ and $maxmin(N.Adj)$. Note that on line 3 the algorithm performs single call to the function $capacity(N.Adj)$ which has complexity $O(n \log n + m)$. From Theorem 1 we know that the **while** loop performs exactly K iterations, where K is the number of classes of Pareto equivalent paths. Each iteration involves a single call to functions $minsum(N.Adj)$ and $maxmin(N.Adj)$, while both of them have complexity $O(n \log n + m)$. Besides that, the function $restrict(R.Adj, c_1)$ has linear computational complexity $O(m)$, because its visits each edge of the network exactly once.

3.4 Implementation and numerical experiments

Along with the analytical analysis of the proposed algorithms, we provide computer implementation and we verified numerically their correctness and efficiency. The prototype of the described method is implemented in Wolfram Language [21] that allows us to take advantage of the system of symbolic computing Mathematica during the design of algorithms. The final version[1] of our software is implemented in programming language Julia [22] because of program compactness and computational efficiency.

Table 3 Execution time of our implementation for test networks with n vertices, m edges that result in K equivalence classes and total number of Pareto optimal solutions

Network	n	m	K	Total solutions	Execution time (sec.)
N_1	5	8	3	5	0.000030
N_2	11	30	6	10	0.000064
N_3	30	171	5	8	0.000123
N_4	40	248	6	10	0.000186
N_5	50	346	8	12	0.000293
N_6	50	1193	18	3063	0.000813

In Table 3 we provide the execution time of our implementation of the function $front(N.Adj)$ that constructs the complete description of the Pareto front of the biobjective shortest path problem. The tests are conducted with input networks with increasing number of vertices n and edges m. The number of equivalence classes K varies in the test networks and it is an important parameter for the execution time of the program (see Theorem 2). From the results in Table 3 it is obvious that the algorithm implementation completes in less than 0.001 second on a standard PC configuration, even for relatively big input networks.

4 Conclusion

In this paper we provide a detailed analysis of the biobjective shortest path problem in a network in which the first objective function is a linear function (shortest length), while the second objective is a bottleneck function (maximal capacity). We present an efficient exact algorithm that discovers a complete description of the Pareto front of the problem that has computational complexity $K \cdot O(n \log n + m)$. We provide detailed mathematical proofs of the correctness of the algorithms, along with detailed numerical examples that illustrate their execution.

The two helper algorithms to compose the shortest paths subnetwork and maximal capacity digraph, represent an efficient solutions of the corresponding single-criterion problems for calculation of all shortest paths and all maximal capacity paths in the network.

The result of the presented method does not only provide the complete list of all Pareto optimal paths of the examined problem, but also fully describes the classes P_i of equivalent Pareto optimal paths. This allows us to generate any minimal complete

[1] https://github.com/llaskov/Biobjective-ShortestPath-ParetoFront

set of efficient paths for the given problem by selecting a single representative of each of the classes P_i.

References

[1] Pemmaraju, S., Skiena, S.: Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica ®, pp. 323–334. Cambridge University Press, Cambridge (2003). https://doi.org/10.1017/CBO9781139164849

[2] Schrijver, A.: Combinatorial Optimization: Polyhedra and Efficiency. Algorithms and Combinatorics, vol. A–C. Springer, Berlin, Heidelberg, New York (2003). https://doi.org/10.1002/9781118600245

[3] Sedeño-noda, A., Colebrook, M.: A biobjective Dijkstra algorithm. European Journal of Operational Research **276**(1), 106–118 (2019) https://doi.org/10.1016/j.ejor.2019.01.007

[4] Brunelli, F., Crescenzi, P., Viennot, L.: On computing Pareto optimal paths in weighted time-dependent networks. Information Processing Letters **168**, 106086 (2021) https://doi.org/10.1016/j.ipl.2020.106086

[5] Beier, R., Röglin, H., Rösner, C., Vöcking, B.: The smoothed number of Pareto-optimal solutions in bicriteria integer optimization. Mathematical Programming **200**, 319–355 (2022) https://doi.org/10.1007/s10107-022-01885-6

[6] Craveirinha, J., Pascoal, M., Clímaco, J.: An exact approach for finding bicriteria maximally srlg-disjoint/shortest path pairs in telecommunication networks. INFOR: Information Systems and Operational Research **61**(3), 399–418 (2023) https://doi.org/10.1080/03155986.2023.2228021

[7] Kaisa, M.: Nonlinear Multiobjective Optimization, 1st edn. International Series in Operations Research & Management Science, vol. 12. Springer, Boston, USA (1999). https://doi.org/10.1007/978-1-4615-5563-6

[8] Branke, J., Deb, K., Miettinen, K., Słowiński, R. (eds.): Multiobjective Optimization: Interactive and Evolutionary Approaches. Lecture Notes in Computer Science. Springer, Berlin (2008). https://doi.org/10.1007/978-1-4615-5563-6

[9] Tušar, T., Filipič, B.: Visualization of Pareto front approximations in evolutionary multiobjective optimization: A critical review and the prosection method. IEEE Transactions on Evolutionary Computation **19**(2), 225–245 (2015) https://doi.org/10.1109/TEVC.2014.2313407

[10] Hansen, P.: Bicriterion path problems. Multiple Criteria Decision Making Theory and Application, 109–127 (1980) https://doi.org/10.1016/S1097-2765(03)00225-9

[11] Queirós Vieira Martins, E.: On a multicriteria shortest path problem. European Journal of Operational Research **16**(2), 236–245 (1984) https://doi.org/10.1016/0377-2217(84)90077-8

[12] Mohamed, C., Bassem, J., Taicir, L.: A genetic algorithms to solve the bicriteria shortest path problem. Electronic Notes in Discrete Mathematics **4**(1), 851–858 (2010) https://doi.org/10.1016/j.endm.2010.05.108

[13] Diakonikolas, I., Yannakakis, M.: Small approximate Pareto sets for biobjective shortest paths and other problems. SIAM Journal on Computing **39**(4), 1340–1371 (2010) https://doi.org/10.1137/080724514

[14] Müller-Hannemann, M.: On the cardinality of the Pareto set in bicriteria shortest path problems. Annals of Operations Research **147**, 269–286 (2006) https://doi.org/10.1007/s10479-006-0072-1

[15] Fidanova, S., Ganzha, M.: Ant colony optimization for workforce planning with hybridization. In: Ganzha, M., Maciaszek, L., Paprzycki, M., Slezak, D. (eds.) Proceedings of the 18th Conference on Computer Science and Intelligence Systems. Annals of Computer Science and Information Systems, vol. 35, pp. 955–959 (2023). https://doi.org/10.15439/2023F9586

[16] Namorado Climaco, J.C., Queirós Vieira Martins, E.: A bicriterion shortest path algorithm. European Journal of Operational Research **11**(4), 399–404 (1982) https://doi.org/10.1016/0377-2217(82)90205-3

[17] Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1**(1), 269–271 (1959) https://doi.org/10.1007/bf01386390

[18] Diestel, R.: Graph Theory, 5th edn. Springer, Berlin (2017). https://doi.org/10.1007/978-3-662-53622-3

[19] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press, Cambridge, Massachusetts (2009). https://doi.org/10.5555/1614191

[20] Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM **34**(3), 596–615 (1987) https://doi.org/10.1145/28869.28874

[21] Inc., W.R.: Mathematica, Version 14.0. Champaign, IL, 2024. https://www.wolfram.com/mathematica

[22] Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: A fresh approach to numerical computing. SIAM review **59**(1), 65–98 (2017) https://doi.org/10.1137/141000671