Noname manuscript No.
(will be inserted by the editor)

Exact Methods for Solving the Elementary Shortest and Longest Path Problems

Quoc Trung BUI, Yves DEVILLE · Quang Dung PHAM

Received: date / Accepted: date

Abstract We consider in this paper the problems of finding the elementary shortest and longest paths on a graph containing negative and positive cycles. These problems are NP-hard. We propose exact algorithms based on Mixed Integer Programming for their solution, employing different valid inequalities. Moreover, we propose decomposition techniques which are very efficient for cases with special structure. Experimental results show the efficiency of our algorithms compared with state of the art exact algorithms.

Keywords Elementary shortest path, Elementary longest path, Negative cycles, Mixed integer programming, Decomposition

1 Introduction

We consider in this paper the elementary shortest path problem (ESPP) and the elementary longest path problem (ELPP) for graphs with negative and positive cycles where the source and the destination can be fixed or non-fixed. The most general version of the problem is stated as follows. We are given a directed graph (also called a digraph) $G = (V_G, A_G, c)$ with the set of nodes V_G and the set of arcs A_G. Each arc $(u, v) \in A_G$ is associated with a cost c_{uv} which can be negative, positive or zero. An elementary path on G is a path that visits each node at most once. Given a set of source nodes $S \subseteq V_G$ and a set of destination nodes

Quoc Trung BUI, Yves DEVILLE
Université catholique de Louvain
ICTEAM Institute, B-1348 Louvain-la-Neuve, Belgium
Tel.: +3210472067
Fax: +3210450345
E-mail: quoc.bui@uclouvain.be, yves.deville@uclouvain.be

Quang Dung PHAM
Hanoi University of Science and Technology
School of Information and Communication Technology, Hanoi, Vietnam
Tel.: +84438692463
Fax: +84438692906
E-mail: dungpq@soict.hust.edu.vn

$T \subseteq V_G$, one problem is to find an elementary shortest path starting from a node in S and terminating at a node in T, denoted by ESPP(S, T, G), and the other problem is to find an elementary longest such path, denoted by ELPP(S, T, G). Different cases of these problems depending on the cardinality of S and T have been considered in the literature. For instance, when $S = \{s\}$ and $T = \{t\}$ with $s \neq t$, the resulting ESPP, denoted by ESPP(s, t, G), is the elementary shortest path problem on a digraph from a specified source node to a specified destination node [16,26]. When $S = T = \{d\}$ (d is a node in G), ESPP(S, T, G) becomes the profitable tour problem, which is denoted by PTP(d, G) [11,15,23,37]. When $S = \{s\}$ and $T = \{t\}$ with $s \neq t$, the resulting ELPP, denoted by ELPP(s, t, G), is the elementary longest path problem on a directed graph from a specified source node to a specified destination node [42]. When $S = T = V_G$, ELPP(V_G, V_G, G), denoted by ELPP(G), becomes the longest elementary path problem [29,42,43,55]. All these problems are NP-hard. Note that when the graph G does not contain negative cycles, then ESPP(S, T, G) is polynomially solvable, for example, by the well-known Bellman–Ford method. Our approach can also handle such problems on undirected graphs by replacing each edge by two opposite arcs with the same cost. For ease of presentation, we consider in this paper only directed graphs.

1.1 Equivalence between problems

We show in this section some equivalences between these problems in the sense that the problems can be polynomially transformed into each other.

1.1.1 Equivalence between ESPP and ELPP

Suppose given a graph $G = (V_G, A_G, c)$, $S, T \subseteq V_G$. Construct the graph $G' = (V_{G'}, A_{G'}, c')$ where $V_{G'} = V_G$, $A_{G'} = A_G$, $c'(e) = -c(e), \forall e \in A$. Clearly, if p is an optimal solution to ESPP(S, T, G), then p is also an optimal solution to ELPP(S, T, G'), and vice versa.

1.1.2 Equivalence between ESPP(S, T, G) and ESPP(s, t, G)

Given a graph $G = (V_G, A_G, c)$, and $S, T \subseteq V_G$, construct the graph $G' = (V_{G'}, A_{G'}, c')$ where $V_{G'} = V_G \cup \{s, t\}$ ($s, t \notin V_G$ are artificial nodes), $A_{G'} = A_G \cup \{(s, v) \mid v \in S\} \cup \{(v, t) \mid v \in T\}$, $c'_{uv} = c_{uv}, \forall v \neq u \in V_G$ and $c'_{sv} = 0, \forall v \in S, c'_{vt} = 0, \forall v \in T$. It is evident that if $p = \langle s, v_1, \ldots, v_k, t\rangle$ is an optimal solution to ESPP(s, t, G'), then $p' = \langle v_1, \ldots, v_k\rangle$ is an optimal solution to ESPP(S, T, G), and vice versa.

1.1.3 Equivalence between ESPP(S, T, G) and PTP(d, G)

When $S = T = \{d\}$ for d a node in G, ESPP(S, T, G) becomes PTP(d, G).

Given a graph $G = (V_G, A_G, c)$, construct the graph $G' = (V_{G'}, A_{G'}, c')$ where $V_{G'} = V_G \cup \{d\}$ ($d \notin V_G$ is an artificial node), $A_{G'} = A_G \cup \{(d, v) \mid v \in S\} \cup \{(v, d) \mid v \in T\}$, $c'_{uv} = c_{uv}, \forall v \neq u \in V_G$ and $c'_{dv} = 0, \forall v \in S, c'_{vd} = 0, \forall v \in T$. It is evident that if the tour $\mathcal{T} = \langle d, v_1, \ldots, v_k, d\rangle$ is an optimal solution to PTP(d, G), then $p' = \langle v_1, \ldots, v_k\rangle$ is an optimal solution to ESPP(S, T, G), and vice versa.

1.2 Related work

ESPP(s, t, G) is a special case of a well-studied problem: the elementary shortest path problem with resource constraints (ESPPRC), in which all resource constraints are removed. ESPPRC appears as a sub-problem in column-generation solution approaches for solving vehicle-routing problems (VRP) [51]. Dror in [17] proved that the ESPPRC is NP-hard. There are well-known labelling algorithms for solving the ESPPRC, based on dynamic programming, for example, [27]. Several techniques have been proposed for improving these labelling algorithms, such as dominance rules to prevent the search from considering paths that are shorter than the others already found [20], state space relaxation [12,48], and bounding [8]. In order to use labelling algorithms to solve ESPP(s, t, G), Drexl and Irnich [16] added a virtual resource as a limitation on the number of visited nodes, but this addition could not prevent the labelling algorithm from considering all subsets of nodes. In their paper, the authors stated that dynamic programming exact algorithms for ESPP(s, t, G) are not appropriate for complete graphs containing more than 20 nodes. This might be the case for problems without additional constraints. Using the bi-directional method of Gighini and Salani in [47], it can solve slightly larger cases. In short, labelling algorithms are not very good at solving ESPP(s, t, G).

Ibrahim et al. in [26] presented two mixed integer programming formulations for ESPP(s, t, G) and compared them in terms of the strength of their respective linear relaxations. These formulations have been tested on small graphs (containing no more than 25 nodes). The results obtained show that the commodity-flow formulation is stronger than the arc-flow formulation. Most recently, in [16], Drexl et al. have compared the integer versions of the formulations in terms of their computation times, memory requirements, and have assessed the quality of the lower bounds provided by an integer relaxation of the commodity-flow formulation. Some constraint relaxation techniques have been proposed for solving ESPP(s, t, G) to optimality. The approach of dynamic separation of sub-tour elimination constraints (SECs) applied to the arc-flow formulation outperforms the others[1]. A formulation applying traditional dynamic SEC separation schemata in [16] will be presented in detail later in this paper.

PTP(d, G) was first presented by Dell'Amico et al. [15], as a variant of the traveling salesman problem (TSP) with profits. In the TSP with profits, each node is associated with a certain quantity of profit, and the overall goal is to find a sub-tour that maximizes the profits collected from the visited nodes but that simultaneously minimizes the travel costs. The TSP with profits becomes PTP(d, G) when the two objectives are combined into one (minimizing the travel costs less the profits). Volgenan and Jonker in [54] showed that PTP(d, G) can be polynomially reduced to the much more studied NP-complete Asymmetric TSP (ATSP). When the cost matrix is symmetric and satisfies the triangle inequality, three approximation algorithms were developed in [11,23,37] for PTP(d, G) with approximation factors of $\frac{5}{2}, 2 - \frac{1}{n-1}$ and $1 + log(n)$ (n is the number of nodes). In the [19], Feillet et al. reviewed other heuristic and exact approaches for solving the TSP with profits.

[1] This arc-flow formulation is presented in Section 2.1.

Wong et al. [55] mentioned ELPP(G) on graphs in the context of peer-to-peer information retrieval networks where weights are associated with nodes. They also proposed a genetic algorithm for solving this problem. ELPP(G) has also been addressed in [49] for evaluating the worst-case packet delay of Switched Ethernet. The given Switched Ethernet is transformed into a delay computation model represented by a directed graph. On this particular directed acyclic graph, the longest path can be computed using dynamic programming. ELPP(G) also appears in the domain of high-performance printed circuit board design in which one needs to find the longest path between two specified nodes on a rectangle grid routing [52]. In [44], ELPP(G) was described in the context of multi-robot patrolling and [43] proposed a genetic algorithm for solving ELPP(G).

In [29], approximating algorithms for ELPP(G) (unweighted graph) were considered, and it was shown that no polynomial-time algorithm can find a constant factor approximation for the longest path problem unless P=NP. In [24], a heuristic algorithm was proposed for ELPP(G), based on the strongly connected components of the graph.

In our previous work in [42], we considered ELPP(G) on arbitrary undirected graphs and proposed two constraint-based techniques for its solution: an exact algorithm based on constraint programming (CP) and a constraint-based local search algorithm. To solve ELPP(G) on sparse undirected graphs with many bridges, we also proposed an efficient algorithm combining constraint-based techniques and dynamic programming.

1.3 Objective, contributions, and outline of the present paper

The main goal of the present paper is to propose exact algorithms for solving ESPP(s, t, G) in which G contains arbitrary arc costs (negative and positive) and may contain negative cycles. As discussed above, solving this problem also yields the solution of ESPP(S, T, G) and ELPP(S, T, G). In particular, we propose exact algorithms based on Mixed Integer Programming (MIP) for solving ESPP(s, t, G) employing different valid inequalities. We also propose decomposition techniques that are efficient in the presence of some special structures. Computational resultss show that our algorithms are more efficient than state of the art exact algorithms [16]. Moreover, using branch-and-cut algorithms, we improve our previous exact algorithm [42], based on Constraint Programming (CP), for solving ELPP(G). Computational results show a significant improvement with the new proposed approach.

1.3.1 Contributions

In the present paper, we focus on solving two problems: ESPP(s, t, G) and ELPP(G).

- **For ESPP(s, t, G):** We propose an exact algorithm based on MIP in which we
  - Introduce new valid inequalities for the ESPP, some of which are derived from the ATSP;
  - Propose an additional algorithm for generating SECs that can detect, in some cases, more SECs than the traditional algorithm in [16];

  - Propose a branch-and-cut algorithm with a constraint filter that solves this problem more quickly than the state of the art algorithm in [16];
  - Propose decomposition techniques that are very efficient for cases with special structures, e.g., directed graphs having many strongly connected components, and directed graphs in which the corresponding undirected graphs have many bridges.
- **For ELPP(G)**:
  - We adapt our previous preprocessing schema [42] that decomposes the undirected graph with many bridges into bridge-blocks so that it can be used for directed graphs with both positive and negative arc costs.
  - By applying our branch-and-cut-based algorithm (instead of applying a constraint-programming based algorithm) on each bridge-block, the overall performance is better than that of our previous paper [42].

1.3.2 Outline

The rest of this paper is structured as follows. Section 2 presents three integer programming formulations for ESPP(s, t, G). Some classes of inequalities that are valid for ESPP(s, t, G) are presented in Section 3. Section 4 describes in details our proposed algorithm for solving ESPP(s, t, G). Section 5 proposes decomposition techniques allowing the use of a dynamic programming schema for solving ESPP(s, t, G) and ELPP(G) on directed graphs with bridges. The experimental results are reported in Section 6. Finally, Section 7 concludes the paper and indicates some directions for future research.

2 Integer programming formulations for ESPP(s, t, G)

Given a directed graph $G = (V_G, A_G, c)$ with set of nodes V_G, set of arcs A_G, each arc $(i, j) \in A_G$ being associated with a cost $c_{ij} \in \mathbb{R}$, suppose that $s, t \in V_G$ are the source node and the destination node. A path from s to t in G is a sequence of nodes $p = \langle v_1, \dots v_n \rangle$, in which $v_1 = s$, $v_n = t$ and $(v_k, v_{k+1}) \in A_G, \forall k \in \{1, \dots, n-1\}$, and the cost of the path is $c(p) = \sum_{i=1}^{n-1} c_{v_i v_{i+1}}$.

For $S \subseteq V_G$, we define $\delta_G^+(S) = \{(i, j) \in A_G | i \in S, j \notin S\}$, $\delta_G^-(S) = \{(i, j) \in A_G | i \notin S, j \in S\}$, $\delta_G(S) = \delta_G^+(S) \cup \delta_G^-(S)$, and $A_G(S) = \{(i, j) \in A_G | i, j \in S\}$. For a node $v \in V_G$, we denote by $\delta_G^+(v)$ (resp. $\delta_G^-(v)$) the set $\delta_G^+(\{v\})$ (resp. $\delta_G^-(\{v\})$).

2.1 Arc-flow formulation: Model 1

This formulation defines only one type of binary variable x_{ij}, which represents whether or not the arc (i, j) is included in the solution. [9, 16, 26, 28].

$$minimize \sum_{(i,j) \in A_G} c_{ij} x_{ij} \tag{1a}$$

subject to

$$x(\delta_G^+(i)) - x(\delta_G^-(i)) = \begin{cases} 1, & i = s \\ -1, & i = t \\ 0, & i \in V_G \setminus \{s, t\} \end{cases} \tag{1b}$$

$$x_{is} = x_{ti} = 0, \forall (i,s), (t,i) \in A_G \tag{1c}$$

$$x(\delta_G^+(S)) - x(\delta_G^+(i)) \geq 0, \forall S \subseteq V_G, |S| \geq 2, t \notin S, \forall i \in S \tag{1d}$$

$$x_{ij} \in \{0,1\}, \forall (i,j) \in A_G \tag{1e}$$

Here, (1a), (1b), (1d) and (1e) represent, respectively, the objective function, the constraints of flow conservation, the SECs, and the integer constraints. In this formulation, there are $|V_G|^2$ variables. However, (1d) has an exponential number of constraints.

2.2 Commodity-flow formulation: Model 2

The following formulation has been studied in [16,26] for ESPP(s,t,G). This formulation has three types of binary variables. The first-type variables x_{ij} equal to 1 iff arc (i,j) is used in the shortest path. The second-type variables y_i equal to 1 iff node i is traversed by the shortest path. And the third-type variables z_k equal to 1 iff a flow from the source node s travels to node $k \in V_G \setminus \{s\}$ using the arc (i,j). The following formulation is presented in [16].

$$minimize \sum_{(i,j) \in A_G} c_{ij} x_{ij} \tag{2a}$$

subject to

$$z_{ij}^k \leq x_{ij}, \forall k \in V_G \setminus \{s,t\},\ (i,j) \in A_G, i \neq k, s \neq j, j \neq t \tag{2b}$$

$$\sum_{(s,j) \in \delta_G^+(s)} z_{sj}^k = y_k, \forall k \in V_G \setminus \{s,t\} \tag{2c}$$

$$\sum_{(i,j) \in \delta_G^+(i)} z_{ij}^k - \sum_{(j',i) \in \delta_G^-(i)} z_{j'i}^k = 0, \forall k \in V \setminus \{s,t\}, i \in V_G \setminus \{s,k,t\} \tag{2d}$$

$$\sum_{(j,k) \in \delta_G^-(k)} z_{jk}^k = y_k, \forall k \in V_G \setminus \{s,t\} \tag{2e}$$

$$\sum_{(i,j) \in \delta_G^+(i)} x_{ij} = y_i, \forall i \in V_G \setminus \{t\} \tag{2f}$$

$$\sum_{(j,i) \in \delta_G^-(i)} x_{ji} = y_i, \forall i \in V_G \setminus \{s\} \tag{2g}$$

$$\sum_{(s,j) \in \delta_G^+(s)} x_{sj} = 1 \tag{2h}$$

$$\sum_{(i,t) \in \delta_G^-(t)} x_{it} = 1 \tag{2i}$$

$$x_{is} = x_{ti} = 0, \forall (i,s), (t,i) \in A_G \tag{2j}$$

$$x_{ij} \in \{0,1\}, \forall (i,j) \in A_G \tag{2k}$$

$$y_i \in \{0,1\}, \forall i \in V \tag{2l}$$

$$z_{ij}^k \geq 0, \forall k \in V_G \setminus \{s,t\}, (i,j) \in A_G, i \neq k, s \neq j, j \neq t \tag{2m}$$

The justification of the constraints in this formulation is given in [16,26]. This formulation contains $O(|V_G|^3)$ variables and constraints. A weakness of this formulation is the large number of variables.

2.3 Miller–Tucker–Zemlin formulation: Model 3

This formulation uses another type of SEC, introduced in [31,36] introduced for the TSP. The SECs are in (3d), in which the constant number M is large enough (e.g., $M = |V_G|$), the binary variables x_{ij} represent whether the arc (i,j) is visited, and each integer variable p_i denotes the relative position of the visited node. The constraints (3d), the SECs, state that $p_i < p_j$ if $x_{ij} = 1$.

$$minimize \sum_{(i,j) \in A_G} c_{ij} x_{ij} \tag{3a}$$

subject to

$$x(\delta_G^+(i)) - x(\delta_G^-(i)) = \begin{cases} 1, & i = s \\ -1, & i = t \\ 0, & i \in V_G \setminus \{s,t\} \end{cases} \tag{3b}$$

$$x_{is} = x_{ti} = 0, \forall i \in V_G \tag{3c}$$

$$p_i + M x_{ij} + 1 \leq p_j + M, \forall (i,j) \in A_G \tag{3d}$$

$$x_{ij} \in \{0,1\}, \forall (i,j) \in A_G \tag{3e}$$

$$p_i \in \{1, \ldots, |V_G|\}, \forall i \in V_G \tag{3f}$$

In [16], Model 2 was shown to be less efficient than Model 1 when solving the ESPP(s,t,G). In addition to the variables of Model 1, Model 3 has $|V_G|$ more variables $(p_i, i \in A_G)$. In this paper, we investigate Model 1, exploiting various valid inequalities that have not been considered so far and a new separation strategy. We will show that our proposed algorithm is better than the state of the art algorithm of [16] in most cases. Altough Model 3 has a polynomial number of constraints, we show, in the experimental section, that that approach performs very poorly.

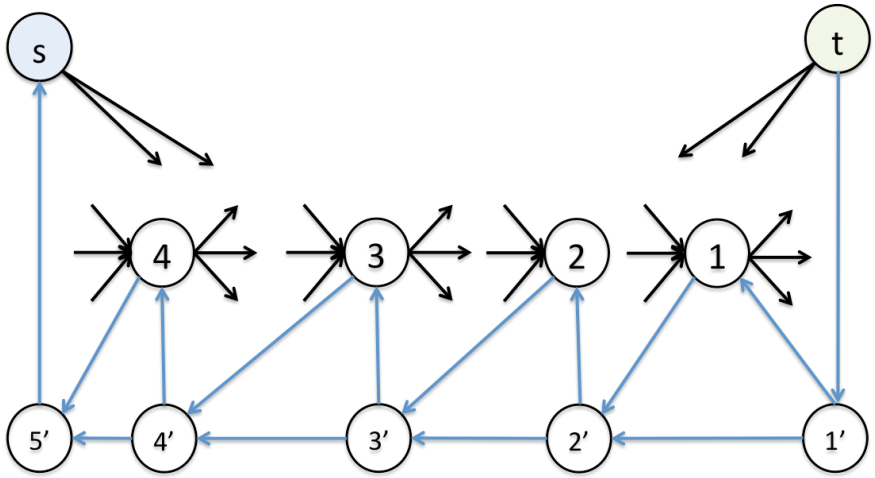

Fig. 1 Transformation from ESPP(s, t, G) to ATSP

3 Classes of valid inequalities for ESPP(s, t, G)

In [54], Volgenant and Jonker showed that PTP(d, G) can be polynomially reduced to ATSP. Based on this, we will show that ESPP(s, t, G) can be polynomially reduced to ATSP. Hence, some valid inequalities of ATSP can be applied to ESPP(s, t, G).

First of all, we construct a new directed graph $G' = (V_{G'}, A_{G'}, c')$ (called the transformed graph) from the directed graph $G = (V_G, A_G, c)$ (called the original graph) with $|V_G| - 1$ new nodes. That is, assuming that $V_G = \{s, v_1, \dots v_n, t\}$, we put $G' = (V_{G'}, A_{G'}, c')$, with $V_{G'} = V_G \cup NV$ and $NV = \{v'_1, \dots, v'_n, v'_{n+1}\}$. We also put $A_{G'} = A_G \cup NA$, where NA consists of $3n-1$ arcs: $(v'_1, v'_2), (v'_1, v_1), (v_1, v'_2), \dots,$ (v'_{n-1}, v'_n), (v'_{n-1}, v_{n-1}), (v_{n-1}, v'_n) and $(t, v'_1), (v'_{n+1}, s)$. In addition, $c'_{ij} = c_{ij}$ $\forall (i, j) \in G$ and $c'_{ij} = 0$ $\forall (i, j) \in NA$.

Figure 1 gives a simple example of the transformation: the original graph of 6 nodes is transformed to a graph of 11 nodes, which transformed graph has 5 new nodes $1', 2', 3', 4', 5'$ and 14 new arcs $(t, 1'), (1', 1), (1, 2'), (1', 2')$, $\dots (5', s)$.

The ATSP(G') problem can be modeled by a set of binary variables $X' = \{x_{ij} | (i, j) \in A_{G'}\}$ indicating whether the arc (i, j) is included in the solution [22]:

$$minimize \sum_{(i,j) \in A_{G'}} c_{ij} x_{ij} \tag{4a}$$

subject to

$$x(\delta^+_{G'}(i)) = x(\delta^-_{G'}(i)) = 1, \quad \forall i \in X' \tag{4b}$$

$$\sum_{i \in S} \sum_{j \in V_{G'} \setminus S} x_{ij} \geq 1, \quad S \subset V_{G'}, S \neq \emptyset \tag{4c}$$

$$x_{ij} \in \{0, 1\}, \ \forall (i, j) \in A_{G'}. \tag{4d}$$

We put $X = \{x_{ij} \in X' | (i, j) \in A_G\}$.

Theorem 1 *If $X'^* = \{x^*_{ij} \mid (i,j) \in A_{G'}\}$ is an optimal solution to ATSP(G'), then $X^* = \{x^*_{ij} \mid (i,j) \in A_G\}$ is an optimal solution to ESPP(s,t,G)*

Proof Suppose that X'^* corresponds to the shortest Hamiltonian cycle $C = \langle (s,u_1), (u_1,u_2), \ldots, (u_p,t), (t,u_{p+1}), \ldots, (u_{2n+1},s)\rangle$ (starting and terminating at s) on the transformed graph G'. We will show that the elementary path $P = \langle (s,u_1), (u_1,u_2), \ldots, (u_p,t)\rangle$ corresponds to X^* and is an optimal solution to ESPP(s,t,G). We have:

- Due to v'_j is in C, so either (v'_j, v'_{j+1}) or (v'_j, v_j) is in C.
- Due to v'_{j+1} is in C, so either (v'_j, v'_{j+1}) or (v_j, v'_{j+1}) is in C.

It follows that we have that node v'_{j+1} always stands after node v'_j in C (the cycle starting and terminating at s). In addition, node v'_1 is always after node t in the cycle (there is only one arc (t,v'_1) coming to node v'_1). We have that all nodes in NA stand after the node t in C. Hence, all nodes of P belong to G. Thus, X^* models P and the cost of P is equal to the cost of C. We now show that P has minimal cost.

Suppose that the cost of P is not minimal, and $P' = \langle (s,x_1), (x_1,x_2), \ldots, (x_p,t)\rangle$ is an elementary path in G whose cost is smaller than that of P. We extend P' to establish a Hamiltonian cycle C' in Algorithm 1.

Algorithm 1: Extend(P', G')

```
C' ← P';
Add arc (t, v'_1) to C';
foreach j ∈ 1..n + 1 do
    if v_j ∈ P' then
        Add the arc (v'_j, v'_{j+1}) to C'
    else
        Add (v'_j, v_j) and (v_j, v'_{j+1}) to C';
return C';
```

Clearly, the cost of C' is equal to that of P' and is thus smaller than the cost of C (a contradiction, as C is an optimal solution to ASTP(G')).

So, P is the shortest elementary path in G. ∎

As a result of Theorem 1, we can tackle ESPP(s,t,G) on the graph G by solving ATSP in G'. However, in this approach, the input size of the problem is increased by a factor of two (the number of nodes of G' is about twice as large as that of G). We therefore do not follow this approach. Instead, we exploit different valid inequalities of the model of ATSP(G') and integrate them into the model of ESPP(s,t,G). It is clear that if $\mathbb{I}(Y')$ is a valid inequality of Model 2.1, with $Y' \subseteq X'$, then $\mathbb{I}(Y)$ is a valid inequality of (1a)–(1e) with $Y \subseteq X$.

D_k, 2-Matching, and T_k are classes of valid inequalities for ASTP in variables in X' (see [5,7,18,21,22,25,39] for more details about these valid inequalities). From D_k and T_k, we can extract inequalities in variables in X and apply these inequalities to the solution of ESPP(s,t,G). In addition, we adapt the 2-Matching valid inequalities to establish valid inequalities for ESPP(s,t,G). As far as we know, these valid inequalities have not yet been exploited for solving ESPP(s,t,G). Moreover, we propose a simple class of valid inequalities which will be experimentally shown to be effective (see Section 3.4).

3.1 D_k-inequalities

For each cycle $\langle (i_1, i_2), \ldots, (i_k, i_1) \rangle$ of G ($k \in \{3, |V_G| - 1\}$), the two following inequalities are valid for ESPP(s, t, G):

D_k^--inequalities:

$$\sum_{j=1}^{k-1} x_{i_j i_{j+1}} + x_{i_k i_1} + 2\sum_{j=3}^{k} x_{i_1 i_j} + \sum_{j=4}^{k}\sum_{h=3}^{j-1} x_{i_j i_h} \leq k - 1 \tag{5}$$

and the D_k^+-inequalities:

$$\sum_{j=1}^{k-1} x_{i_j i_{j+1}} + x_{i_k i_1} + 2\sum_{j=3}^{k} x_{i_j i_1} + \sum_{j=3}^{k-1}\sum_{h=2}^{j-1} x_{i_j i_h} \leq k - 1. \tag{6}$$

3.2 T_k-inequalities

The T_k-inequalities that are valid for ESPP(s, t, G) on the original graph G are as follow.

For any $W \subset V_G \setminus \{s, t\}, 2 \leq |W| \leq |V_G| - 4, w \in W, p, q \in V_G \setminus W$, we have

$$x_{pw} + x_{pq} + x_{wq} + x(A_G(W)) \leq |W|. \tag{7}$$

3.3 2-Matching inequalities

The 2-Matching inequalities discovered by Edmonds [18] are widely used in branch-and-cut algorithms for TSP and its variations [5,25,39,41]. We adapt the 2-Matching to be valid inequalities for ESPP(s, t, G) as follows:

$$x(\delta_G(S)) \geq 2x(F) + 1 - |F|, \forall S \subset V_G, s \notin S, t \notin S, F \subset \delta_G(S), |F| \textit{ odd}. \tag{8}$$

Theorem 2 *Inequalities (8) are valid for ESPP(s, t, G).*

Proof Let P be the set of nodes of an elementary shortest path on the original graph G. We have $x(\delta_G(S)) \geq x(F)$ as $F \subseteq \delta_G(S)$ and $x(F) \leq |F|$. Hence $x(\delta_G(S)) - x(F) + |F| - x(F) \geq 0$.

In addition, because neither s and t are in S, we have $x(\delta_G(S)) = 2|P \cap S|$, which is even. So we have $x(\delta_G(S)) - x(F) + |F| - x(F) \geq 1$, as $|F|$ is odd. Then inequalities (8) are true.

Finally, we conclude that inequalities (8) are valid for ESPP(s, t, G). ∎

3.4 Maximum outflow inequalities

We propose a class of so-called maximum outflow inequalities:

$$\delta_G^+(v) \leq 1, \forall v \in V_G \setminus \{t\}. \tag{9}$$

This class imposes a constraint on the maximum outflow at each node of the graph as in each solution to ESPP(s, t, G), there is at most one arc leaving each node of G.

4 Solving ESPP(s,t,G) to optimality

Models 2 and 3 described in Section 2 have a polynomial number of constraints. It is thus possible to solve the problem by a MIP solver with these models. We will assess the performance of these branch-and-cut algorithms (two algorithms correspond to two models) in the experiments section. We denote these two branch-and-cut algorithms by *ESP_ComFlow* and *ESP_MTZ*.

In this section, we concentrate on proposing a branch-and-cut algorithm for solving ESPP(s,t,G) with Model 1. For the sake of readability, we denote that algorithm by *ESP_FltC*.

4.1 Branch and Cut schema of *ESP_FltC*

We present here a generic branch-and-cut procedure of *ESP_FltC*, (see [6,38] for more information on branch-and-cut procedures).

At a generic step of the branch-and-cut procedure, the original linear programming relaxation, $0 \leq x_{ij} \leq 1$, is enriched by additional valid inequalities for ESPP(s,t,G), and some of the binary variables are fixed either at their upper or lower bound. We denote by $\mathcal{C}$ the current family of valid inequalities for ESPP(s,t,G) and we assume that the linear system $Ax \geq b$ defining $\mathcal{C}$ contains at least all the inequalities in (1b) and (1c) of Model 1.

We denote by $\mathcal{F}_0$ and $\mathcal{F}_1$ the sets of variables that have been fixed at 0 and 1, respectively.

Let $\mathcal{K}(\mathcal{C},\mathcal{F}_0,\mathcal{F}_1) = \{x : Ax \geq b$
$x_{ij} = 0$ for $x_{ij} \in \mathcal{F}_0$
$x_{ij} = 1$ for $x_{ij} \in \mathcal{F}_1\}$
and let $LP(\mathcal{C},\mathcal{F}_0,\mathcal{F}_1)$ denote the linear program

$$Min\ cx : x \in \mathcal{K}(\mathcal{C},\mathcal{F}_0,\mathcal{F}_1)$$

which is assumed to be feasible, with a finite minimum.

The active nodes of the enumeration tree are represented by a list $\mathcal{NL}$ of ordered pairs $(\mathcal{F}_0,\mathcal{F}_1)$. Let x^* be the best known solution to ESPP(s,t,G) and UB stand for the current upper bound (the value of the best known solution x^*).

During the solution process, the branch-and-cut procedure maintains an inequality pool $\mathcal{IP}$ containing the inequalities generated. This procedure is depicted in Figures 2. Its different steps are developed in the following sections.

4.2 Preprocessing

Preprocessing is a very important algorithmic tool for solving large scale combinatorial problems. The main idea is to detect unnecessary information in the problem and to reduce the size of the problem by logical implications. In *ESP_FltC*, we apply some of following simple reduction methods [14,30,33–35,53].

1. **Preprocessing:** Removing useless nodes and arcs from the graph G.
2. **Initialization:** Let $\mathcal{C}$ be the set of constraints in the linear programming relaxation of ESPP(s,t,G) (containing only the inequalities in (1b) and (1c) of Model 1). Set $\mathcal{IP} = \emptyset$, $\mathcal{F}_0 = \mathcal{F}_1 = \emptyset$, $x^* = NULL$, $UB = +\infty$, and two temporary inequality sets $\mathcal{SI} = \mathcal{GI} = \emptyset$.
3. **Lower bounding:** Solve the linear problem $LP(\mathcal{C} \cup \mathcal{SI}, \mathcal{F}_0, \mathcal{F}_1)$. Clear the set of separated inequalities $\mathcal{SI}$. If there exists an optimal solution to the linear problem, denote it by $\overline{x}$. If the solution $\overline{x}$ is an elementary path, update the best known solution $x^* = \overline{x}$ and the upper bound value $UB = c\overline{x}$. If the solution $\overline{x}$ is not an elementary path, go to Step 5 (Inequality generation).
4. **Node selection:** If $\mathcal{NL} = \emptyset$, stop; otherwise, choose an ordered pair $(\mathcal{F}_0, \mathcal{F}_1)$ and remove it from $\mathcal{NL}$. Go to Step 3 (Lower bounding).
5. **Inequality generation:** Clear the set of generated inequalities $\mathcal{GI}$. Generate inequalities valid for ESPP(s,t,G) but violated by $\overline{x}$ and add them to $\mathcal{GI}$. If $\overline{x}$ is an integer-feasible solution, do $\mathcal{C} = \mathcal{C} \cup \mathcal{GI}$ and go to Step 3 (Lower bounding).
6. **Inequality selection:** Select some most violated inequalities from $\mathcal{GI}$ and then add them to $\mathcal{IP}$.
7. **Inequality detection:** All inequalities in $\mathcal{IP}$ that are violated by the solution $\overline{x}$ are inserted into $\mathcal{SI}$.
8. **Branching/Cutting decision:** If $\mathcal{SI}$ is not empty, go to Step 3 (Lower bounding).
9. **Branching:** Select an appropriate variable x_{ij} such that $\overline{x_{ij}}$ is fractional. Generate two subproblems corresponding to $(\mathcal{F}_0 \cup \{x_{ij}\}, \mathcal{F}_1)$ and $(\mathcal{F}_0, \mathcal{F}_1 \cup \{x_{ij}\})$, insert them into $\mathcal{NL}$. Go to Step 4 (Node selection).

Fig. 2 Branch-and-cut schema of *ESP_FltC*

4.2.1 Removing useless nodes

We can erase nodes that cannot be in any path from the source node to destination node. Such nodes can be detected by a simple search, for example, depth-first search.

- A forward search is used to find nodes connected from the source node, these nodes are collected into a set denoted by S_1.
- A backward search is used to find nodes connected to the destination node, these nodes are collected into a set denoted by S_2.

We can remove nodes in $V_G \setminus \{S_1 \cap S_2\}$ because they can not be in a path from the source node to the destination node.

4.2.2 Removing nodes of degree 1

A node with only one flow that arrives or leaves it is useless and can be removed.

1. Let v be a node such that $\delta_G^+(v) = \{u\}$ (there is only one flow going out from v). Then each arc $(i,v) \in \delta_G^-(v)$ can be replaced by an arc (i,u) of cost $c_{iv} + c_{vu}$.
2. Let v be a node such that $\delta_G^-(v) = \{u\}$ (there is only one flow arriving to v). Then each arc $(v,i) \in \delta_G^+(v)$ can be replaced by an arc (u,i) of cost $c_{uv} + c_{vi}$.

4.3 Node selection

The objective of the node selection step (Step 4) is to choose the next node to expand. A naive node selection strategy might lead to a huge search tree. By contrast, an intelligent node selection strategy leads quickly to a good feasible

solution that sharply reduces the gap between the upper bound and lower bound and proves the optimality of the current incumbent. There exist some well-known node-selection strategies, such as depth-first-search, breadth-first-search, and best-bound search.

In order to minimize the size of the search tree, in *ESP_FltC*, the *best-bound search* strategy, which selects the node with the best lower bound, is carried out. Its advantage is that, for a fixed sequence of branching decisions, it minimizes the number of nodes that are explored, because all nodes that are explored would have been explored independently of the upper bound [10,32].

4.4 Branching variable selection

In the branching step (Step 9), a fractional variable is chosen and two new subproblems are generated by setting in turn the value of the variable to 0, and to 1. Selecting the right branching variable is an important component of a branch-and-cut algorithm. Ideally, we would like to choose the branching variables that minimize the size of the search tree. A simple branching strategy applied is to select the variable with the largest integer violation: this is known as *maximum fraction branching* [5,30]. In practice, this rule is not efficient: it performs about as well as randomly selecting a branching variable [3].

The most successful branching strategies estimate the change in the lower bound after branching. Because we prune a node of the branch-and-bound tree whenever the lower bound of the node is greater than the current upper bound, we want to increase the lower bound as much as possible. In [3], Achterberg et al. also experimented with strong branching rules, pseudocost branching, strong branching, a hybrid of strong/pseudocost branching, pseudo cost branching with strong branching initialization, and reliability branching.

ESP_FltC is implemented in C++, using IBM Ilog Cplex Concert Technology, version 12.2. The default variable selection strategy of CPLEX is a variant of hybrid branching [2,3]. Based on the experiments in [3], we decided that *ESP_FltC* should carry out that default strategy.

4.5 Inequality generation

Given a solution $\overline{x}$ to $LP(\mathbb{C}, \mathcal{F}_0, \mathcal{F}_1)$ at a node of the branch-and-bound tree, we try to find valid inequalities that are not satisfied by $\overline{x}$, and add them to the model. Algorithms for finding these inequalities often rely on a support graph $G_s = (V_s, A_s)$ with $V_s = V$, $A_s = \{(i,j)|\overline{x}_{ij} > 0\}$ and each arc $(i,j) \in A_s$ is associated with a cost $c_s(i,j) = \overline{x}_{ij}$. For ease of presentation, we write $A_s^+(i) = \{j|(i,j) \in A_s\}$ and $A_s^-(i)\{j|(j,i) \in A_s\}$.

Algorithms for separating valid inequalities D_k, T_k are presented in [5,22]. For the generation of 2-Matching valid inequalities, we only separate inequalities with $|S| = 1$ in order to reduce the computational complexity [4,39,40].

4.5.1 Subtour elimination constraint generation

In model 1, the goal of generating an SEC that is violated by $\overline{x}$ (see Section 2.1) means finding a pair $\langle S, i\rangle$ in which $S \subseteq V_G \setminus \{t\}$ and $i \in S$ such that $\overline{x}(\delta_G^+(S)) <$

$\overline{x}(\delta_G^+(i))$. The algorithm for generating SECs in [16] works on the undirected auxiliary support graph $G_a = \{V_a, E_a\}$ in which $V_a = V$, $E_a = \{(i,j)|\overline{x}_{ij} + \overline{x}_{ji} > 0\}$ with a cost $c_e = \overline{x}_{ij} + \overline{x}_{ji}$ associated to each edge $e_{ij} \in E_a$[2]. We call this algorithm the traditional separation algorithm for finding SECs.

The traditional separation algorithm for finding SECs works as follows. In the first step, it checks whether there exist any isolated components in G_a that are not connected to s and t. If such an isolated component is found, an SEC (1d) is generated in which S contains all nodes of the isolated component, and i is randomly selected from S. Otherwise (i.e., no isolated component is found), the algorithm performs the second step which solves, for each node $v \in V_G \setminus \{t\}$, the maximum $v-t$-flow/minimum $v-t$-cut problem on G_a. If the maximum flow $v-t$ is less than the outflow from this node with respect to $\overline{x}$ (i.e., $\sum_{(v,u)\in\delta_G^+(v)} \overline{x}_{vu}$), then one SEC (1d) is generated; in this SEC $\langle S, i\rangle$, S is the set of nodes that are on the same side of the $v-t$-cut than v and i is the node v.

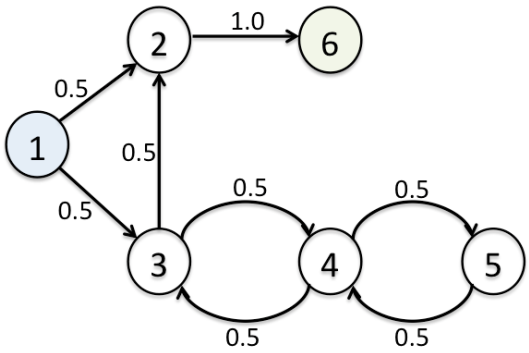

Fig. 3 In this example, the traditional separation algorithm for SECs cannot find any SECs.

In some cases, the traditional algorithm cannot detect any SECs: this is illustrated in Figure 3: there are violated SECs (1d) in this support graph that cannot be detected by the traditional algorithm, for example, with $S = \{3, 4, 5\}$ and i being the node 3 or 4.

As the traditional algorithm cannot always find SECs even if they exist, we propose in this section another heuristic algorithm for finding SECs which is simple but efficient. Our proposed algorithm extends from a promising node (node i in (1d)) to obtain the set of nodes (set S in (1d)) until an SEC is found or until the current set of nodes cannot be extended anymore. In our proposed algorithm, a promising node is an endpoint of an arc (u, v) such that $\delta_{G_s}^+(u) \geq 2$. For example, in Figure 3, nodes 1, 2, 3, 4, and 5 are promising nodes.

The first step of our heuristic separation algorithm for SECs is similar to the traditional separation algorithm, which checks for isolated components that are not connected to the source and the destination nodes. The second step of our heuristic algorithm tries to find an SEC by extending a promising node (as

[2] We would like to thank Michael Drexl, the author of [16], for this information.

described in Algorithm 2) instead of finding the max flow from every node $v \in V_a$ to t in G_a.

In Algorithm 2, the procedure extends from a promising node i. At each iteration, a new node is added to the set S (lines 7–8), the outflow from S is firstly recomputed (lines 9–10), then it checks for a violation of the SECs (lines 11–12). If a violated SEC is found, the procedure immediately stops. The procedure also stops immediately when S can not be extended (lines 13–15).

Algorithm 2: FindSEC($G_s = (V_s, A_s), i, t$)

Input: Support graph $G_s(V_s, A_s)$, the destination node t and a promising node i
Output: A pair $\langle S, i\rangle$ (if $S \neq \emptyset$ we have a violated SEC, otherwise, no violated SEC found corresponding to i)

1 Insert the promising node i into node set S and a queue Q;
2 $\delta^+_{G_s}(i) \leftarrow \sum_{v \in A^+_s(i)} c_s(i, v)$;
3 $OutFlowS \leftarrow \delta^+_{G_s}(i)$;
4 $Extend \leftarrow true$;
5 **while** $Extend$ **do**
6 Get and Remove top node u from the queue Q ;
7 **foreach** $v \in A^+_s(u) : v \notin S, v \neq t, Extend = true$ **do**
8 Insert v into S and Q;
9 $OutFlowS \leftarrow OutFlowS + \sum_{u \in A^+_s(v)} c_s(v, u)$;
10 $OutFlowS \leftarrow OutFlowS - \sum_{u \in S}(c_s(u, v) + c_s(v, u))$;
11 **if** $OutFlowS < \delta^+_{G_s}(i)$ **then**
12 $Extend \leftarrow false$;
13 break ;
14 **if** Q *is empty* **then**
15 $Extend = false$;
16 $S \leftarrow \emptyset$;
17 **return** $\langle S, i\rangle$;

Clearly, our proposed heuristic separation algorithm for SECs can find two SECs from the support graph in the Figure 3: $\langle\{3, 4, 5\}, 3\rangle$ and $\langle\{4, 5\}, 4\rangle$.

4.5.2 Generation of maximum outflow inequality

It is clear that maximum outflow inequalities can easily be computed by an algorithm with complexity of $O(|V_s|)$, checking for a violation of a maximum outflow inequality at each node of the support graph G_s.

4.5.3 Generating other inequalities

We also use other families of inequalities that are valid for a general mixed integer program. They are MIP mixed integer rounding inequalities, MIP zero-half inequalities, MIP Gomory fractional inequalities, etc.. Our algorithm *ESP_FltC* is implemented in CPLEX solver that provides an option for either generating these inequalities or not.

4.6 Inequality selection

The execution of *ESP_FltC* maintains an *inequality pool* $\mathcal{IP}$ that collects the inequalities generated. At each node, each time the algorithm produces a solution, the inequalities in the pool are checked as to whether they are violated by the solution. If some inequalities are violated by the solution, the algorithm adds them to the model of the current node and then re-optimizes. This procedure is iteratively executed until no inequality in the pool is violated.

The pool $\mathcal{IP}$ has an important role in *ESP_FltC* that becomes much stronger if its pool contains just the most violated inequalities. To reject useless violated inequalities, *ESP_FltC* must answer the two following questions:

1. When there are various classes of valid inequalities, how to decide whether one class is better than another? Although many branch-and-cut solvers report computational experience on this issue, it mostly remains an open question. In general, all valid inequalities are separated.
2. Within a class of inequalities, is one inequality better than another? This is also an open question. However, two widely used measures of the quality of an inequality in the same class are presented in [6]. Given an inequality $\alpha x \geq \beta$, its quality is computed by one of the two following measures:
   (a) The value $\beta - \alpha\overline{x}$ measures the traditional violation of the inequality.
   (b) The geometric measure uses the Euclidean distance between $\overline{x}$ and the hyperplane $\alpha x = \beta$, namely the distance between $\overline{x}$ and its orthogonal projection onto this hyperplane.

   In this paper, we use the traditional measure to order the inequalities belonging to the same class. However, we must determine a good threshold for each class of inequalities to arrange that the pool $\mathcal{IP}$ contains only the most violated inequalities.

5 Decompositions

In this section, we propose some decomposition techniques that extend our previous work [42] and support directed graphs with both positive and negative arc costs. In these decompositions, the given graph will be decomposed into bridge-blocks and strongly connected components. Our proposed branch-and-cut algorithm, described above, will be applied to each bridge-block or strongly connected component. The results obtained will then be aggregated using dynamic programming schemata to solve the ESPP and the ELPP.

Before describing the decompositions, we give some definitions and notation. Given a directed graph G, we write V_G for the set of nodes of G and A_G for the set of arcs of G. Given a set of nodes $S \subseteq V_G$, we denote by $G(S)$ the sub-graph induced by S in which $V_{G(S)} = S$ and $A_{G(S)} = \{(u,v) \in A_G | u, v \in S\}$. An auxiliary graph of G is a graph $G' = \{V_G, E_G\}$ where $E_G = \{(u,v) \mid (u,v) \in A_G \vee (v,u) \in A_G\}$. In a rooted tree, T, $T(v)$ denotes the set of descendant nodes of v in T, including v.

5.1 Computing bridge-blocks

Without loss of generality, we assume that the graph G is connected. The computation of bridges and bridge-blocks can be realized with a variant of the $O(|V|+|A|)$ depth-first search algorithm $DFS(r)$ of [50] on an auxiliary graph of G, denoted by G'. The variant algorithm we developed requires fixing a root node r of V_G and produces the following results:

- The set $B(G')$ of bridges of G';
- A spanning tree T of G', rooted at r, in which $f(v)$ is the father of a node v (by convention $f(r) = r$); the spanning tree will be composed of k spanning subtrees (one per each bridge-block), connected by the bridges;
- The set $\{S_1, \ldots, S_k\}$, where S_i is the set of nodes of bridge-blocks of G';
- The set R of roots of the different subtrees $T(S_i)$, i.e., $R = \{r_i | r_i$ is the root of $T(S_i), 1 \leq i \leq k\}$;
- $Exit(r_i)$, the set of exit nodes of $T(S_i)$, which are nodes in S_i connected to (the root of) some other bridge-block in T plus the node r_i, i.e., $Exit(r_i) = \{v \in S_i | \exists r_j \in R : f(r_j) = v\} \cup \{r_i\}$;
- $ChRoot(v)$, the child nodes of v in T that are outside the bridge-blocks of v; by the definition of a bridge block, such children are necessarily members of R, i.e., $ChRoot(v) = \{r_i \in R | f(r_i) = v\}$;
- $NextRoot(r_i)$ is the set of root nodes of bridge-blocks connected to S_i, i.e., $NextRoot(r_i) = \cup_{v \in S_i} ChRoot(v)$;

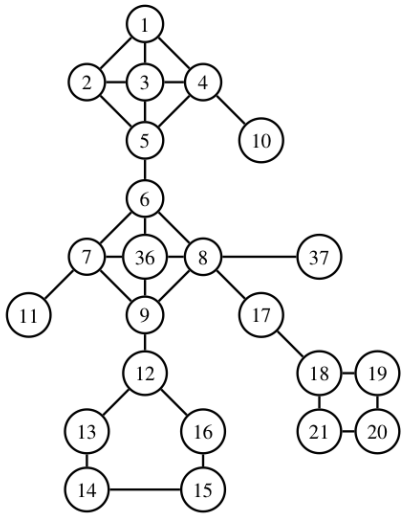

Fig. 4 The auxiliary graph G'

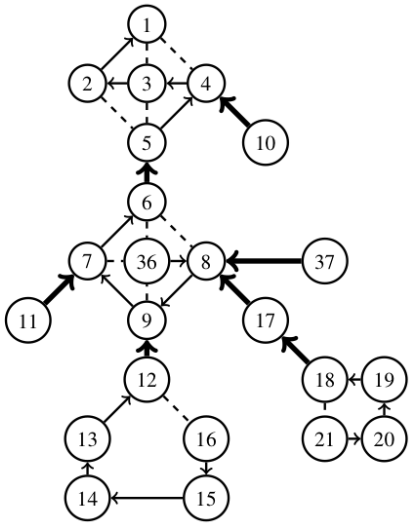

Fig. 5 The rooted spanning tree T (solid arcs are in T, bold arcs are bridges)

Figures 4 and 5 illustrate the result of DFS; we give some of the results: $S_1 = \{1, 2, 3, 4, 5\}, S_2 = \{6, 7, 8, 9, 36\}, S_3 = \{17\}, S_4 = \{18, 19, 20, 21\}, S_5 = \{37\}, S_6 = \{12, 13, 14, 15, 16\}, S_7 = \{11\}, S_8 = \{10\}.R = \{1, 6, 17, 18, 37, 12, 11, 10\}$, $NextRoot(1) =$

$\{6, 10\}, NextRoot(6) = \{11, 12, 17, 37\}, NextRoot(17) = \{18\}$. The other $NextRoot$ values are $\{\emptyset\}$. $B(G') = \{(6, 5), (10, 4), (11, 7), (12, 9), (17, 8), (18, 17), (37, 8)\}$.$ChRoot(1) = \{\emptyset\}, ChRoot(5) = \{6\}, ChRoot(8) = \{17, 37\}$ etc. $Exit(1) = \{1, 4, 5\}, Exit(6) = \{6, 7, 8, 9\}$, etc.

5.2 Computing strongly connected components

A directed graph is *strongly connected* if there exists a path from each node in the graph to every other node. The *strongly connected components* (SCCs) of a directed graph are the maximal strongly connected subgraphs. It is evident that if each SCC is contracted to a single node, the contracted directed graph is acyclic. When solving ESPP(s, t, G), we could focus on the SCCs that are on paths in the contracted graph from the SCC containing s to the SCC containing t.

Given a directed graph $G = (V_G, A_G, c)$ together with a source node s and a destination node t, we introduce the following notation:

- $\mathcal{S}$ are the set of nodes connected from s by an elementary path and $\mathcal{T}$ is the set of nodes that connect to t by an elementary path.
- $SCC = \{c_1, \ldots, c_n\}$ is the set of SCCs of G with $c_i = (V_{c_i}, A_{c_i}, c)$.
- We write $I(c_i) = \{v \in V_{c_i} \mid (u, v) \in A_G \wedge u \notin V_{c_i}\}$ and $O(c_i) = \{v \in V_{c_i} \mid (v, u) \in A_G \wedge u \notin V_{c_i}\}$. If c_i contains s, then s is included in $I(c_i)$. If c_i contains t, then t is included in $O(c_i)$.
- $Parent(c_i)$ is the set of SCCs connected to c_i by an arc, i.e.: $Parent(c_i) = \{c_j \in SCC | i \neq j \wedge \exists (u, v) \in V_G : u \in V_{c_j}, v \in V_{c_i}\}$.
- $Children(c_i)$ is set of SCCs connected from c_i by an arc, i.e.: $Children(c_i) = \{c_j \in SCC | i \neq j \wedge \exists (u, v) \in V_G : u \in V_{c_i}, v \in V_{c_j}\}$.

Before computing ESPP(s, t, G), the input graph is preprocessed in the two following steps:

- **Step 1** consists of removing nodes that cannot be a part of an elementary path from s to t from the graph. $\mathcal{S}$ and $\mathcal{T}$ are computed with a depth-first search algorithm [13]. Nodes in $V_G \setminus \{\mathcal{S} \cap \mathcal{T}\}$ are removed from G.
- **Step 2** consists of finding all SCCs in the graph. This computation can be realized with a variant of the $O(|V| + |A|)$ depth-first search algorithm in [13]. The variant of the depth-first search algorithm that we propose uses s as the root node. After this step, we have a set of SCCs: $SCC = \{c_1, \ldots, c_n\}$ and $I(c_i)$, $O(c_i)$, $Parent(c_i)$, $Children(c_i)$ for each c_i.

Figures 6 and 7 illustrate the result of the preprocessing. The overall goal is to determine the elementary shortest path from node 1 to node 19. After Step 1, nodes 3, 11, 12, 13, 14, 15, 20 are removed. After Step 2, $SCC = \{c_1, c_2, c_3\}, V_{c_1} = \{1, 2, 4, 5\}, V_{c_2} = \{6, 7, 8, 9, 10\}, V_{c_3} = \{16, 17, 18, 19\}, I(c_1) = \{1\}, I(c_2) = \{6\}, I(c_3) = \{16, 17\}, O(c_1) = \{5\}, O(c_2) = \{8, 10\}, O(c_3) = \{19\}, Parent(c_1) = \emptyset, Parent(c_2) = \{c_1\}, Parent(c_3) = \{c_2\}, Children(c_1) = \{c_2\}, Children(c_2) = \{c_3\}$, and $Children(c_3) = \emptyset$.

5.3 ELPP(G) on sparse directed graphs with many bridges

In our previous paper [42], we presented a dynamic programming approach for solving ELPP(G) on sparse undirected graphs with many bridges, but limited

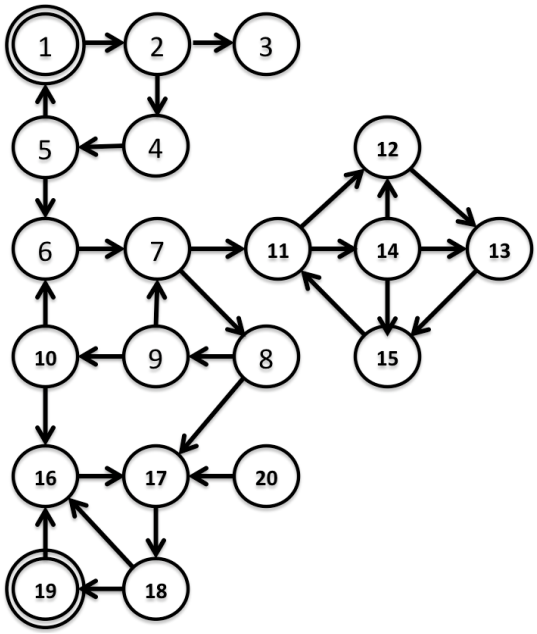

Fig. 6 The graph G

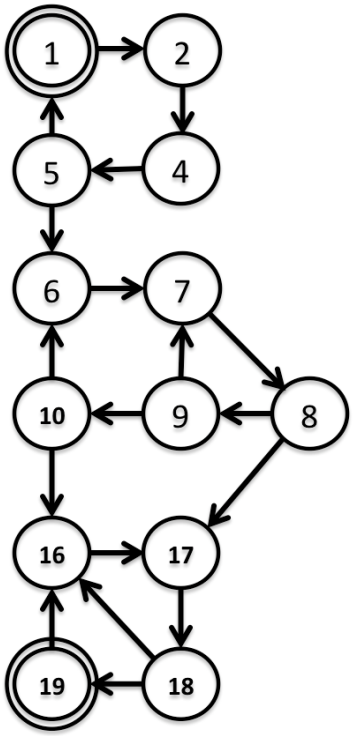

Fig. 7 The graph G after the preprocessing step)

to positive arc costs. In the present paper, we extend the approach to directed graphs with positive and negative arc costs. This new algorithm based on MIP is much faster than our previous algorithm in [42] that was based on Constraint Programming. We denote this new algorithm by *ELP_FltC_BB*.

The given directed graph is decomposed into bridge-blocks as described in Section 5.1. From the resulting data structures, we now introduce the following notations.

- $L(r_i)$: the cost of the elementary longest path in $G(T(r_i))$.
- $d(u, v)$: the cost of the elementary longest path from u to v in $G(S_i)$, with $u, v \in S_i$. By convention, $d(u, u) = 0$.
- $H_1(r_i)$: the cost of the elementary longest path starting from r_i in $G(T(r_i))$.
- $H_2(r_i)$: the cost of the elementary longest path ending at r_i in $G(T(r_i))$.
- $h_1(v)$: the cost of the elementary longest path in $G(T(v))$ starting from v, but without any arc in $G(S_i)$, with $v \in S_i$.
- $h_2(v)$: the cost of the elementary longest path in $G(T(v))$ ending at v, but without any arc in $G(S_i)$, with $v \in S_i$.

These last two quantities can be computed recursively as follows:

$$h_1(v) = max(max_{r_i \in ChRoot(v)} H_1(r_i) + c_{vr_i}, 0)$$

$$h_2(v) = max(max_{r_i \in ChRoot(v)} H_2(r_i) + c_{r_i v}, 0)$$

$$H_1(r_i) = max(max_{v \in S_i} d(r_i, v) + h_1(v), 0)$$

$$H_2(r_i) = max(h_2(v) + max_{v \in S_i} d(v, r_i), 0)$$

$h_1(v)$ and $h_2(v)$ are defined to be 0 when $ChRoot(v) = \emptyset$.

Using the above values, we can now compute $L(r_i)$, the cost of the elementary longest path in $G(T(r_i))$, with $r_i \in R$. We have $L(r_i) = max\{L_0(r_i), L_1(r_i), L_2(r_i)\}$, where $L_0(r_i)$ (resp. $L_1(r_i), L_2(r_i)$) is the cost of the elementary longest path in $G(T(r_i))$ containing no (resp. exactly one, at least 2) node of S_i. These values can be computed as follows.

- $L_0(r_i) = max_{r_j \in NextRoot(r_i)} L(r_j)$
- $L_1(r_i) = max_{v \in S_i} l(v)$ where

$$l(v) = \begin{cases} 0 & \text{if } ChRoot = \emptyset \\ max\{H_2(r_j) + c_{r_j v}, H_1(r_j) + c_{v r_j}\} & \text{if } ChRoot(v) = \{r_j\} \\ max_{r_k \neq r_j \in ChRoot(v)} H_2(r_k) + c_{r_k v} + c_{v r_j} + H_1(r_j) & \text{otherwise} \end{cases}$$

- $L_2(r_i) = max_{u \neq v \in S_i} h_2(u) + d(u, v) + h_1(v)$

The algorithm is composed of the sequential calls to the recursive methods ComputeFirstStep(r) and ComputeSecondStep(r) depicted in Algorithms 3 and 4, where r is the root node of the spanning tree T. Algorithm 3 computes the longest path in G containing some exit nodes and arc bridges. The variable f^* stores the cost of this path.

Algorithm 3: ComputeFirstStep(r_i)

foreach $r_j \in NextRoot(r_i)$ **do**
  ComputeFirstStep(r_j);
foreach $u, v \in Exit(r_i) : u \neq v$ **do**
  $d(u, v, S_i) \leftarrow$ solveELPP($\{u\}, \{v\}, G(S_i)$); ;
$d_1(r_i) \leftarrow$ solveELPP($\{r_i\}, S_i \setminus Exit(r_i), G(S_i)$) ;
$d_2(r_i) \leftarrow$ solveELPP($S_i \setminus Exit(r_i), \{r_i\}, G(S_i)$) ;
foreach $v \in S_i$ **do**
  $h_1(v) \leftarrow max_{r_j \in ChRoot(v)} H_1(r_j) + c_{v r_j}$;
  $h_2(v) \leftarrow max_{r_j \in ChRoot(v)} H_2(r_j) + c_{r_j v}$;
  $h_1(v) \leftarrow max\{h_1(v), 0\}; h_2(v) \leftarrow max\{h_2(v), 0\}$;
$H_1(r_i) \leftarrow max_{v \in S_i \setminus Exit(r_i)} d(r_i, v) + h_1(v)$;
$H_2(r_i) \leftarrow max_{v \in S_i \setminus Exit(r_i)} d(v, r_i) + h_2(v)$;
$H_1(r_i) \leftarrow max\{H_1, d_1(r_i), 0\}$; $H_2(r_i) \leftarrow max\{H_2, d_2(r_i), 0\}$;
$L_0 = max_{r_j \in NextRoot(r_i)} L(r_j)$;
$L_1 \leftarrow 0$;
foreach $v \in S_i : ChRoot(v) \neq \emptyset$ **do**
  if $ChRoot(v) = \{r_j\}$ **then**
    $l_1 \leftarrow max\{H_2(r_j) + c_{r_j v}, H_1(r_j) + c_{v r_j}\}$;
  else
    $l_1 \leftarrow max_{r_k \neq r_j \in ChRoot(v)} H_2(r_k) + c_{r_k v} + c_{v r_j} + H_1(r_j)$;
  $L_1 \leftarrow max\{L_1, l_1\}$;
$L_2 \leftarrow max_{u \neq v \in Exit(r_i)} h_2(u) + d(u, v) + h_1(v)$;
$L(r_i) \leftarrow max\{L_0, L_1, L_2, 0\}$;
$f^* \leftarrow max\{f^*, L(r_i)\}$;

The algorithm 4 completes the search by considering pairs of nodes without arc bridges, using f^* as a lower bound. The search is performed only if the positive cost W (the sum of the positive arc costs) of the bridge-block $G(S_i)$ is greater than f^* (see lines 4–5) because the cost of the any longest path on S_i is always less than or equal to W. The method solveELPP($G(S_i), f^*$) computes the cost of the elementary longest path in $G(S_i)$ so that the cost of the solution path is greater than f^*. That method is derived from any algorithm of ELPP with a lower bound constraint for the objective function. The data structure $L(r_i)$, $d(u, v)$, $H_1(r_i)$, $H_2(r_i)$, $h_1(v)$, $h_2(r_i)$ and f^* is global. At the end of the computation, f^* will be the cost of the optimal solution. The algorithm can be easily extended to also return the optimal path.

To compute ELPP(G) on an undirected graph, we replace each edge by two opposite arcs with the same cost as the edge. Our algorithm is slightly adapted to reduce the computation time. For a pair of nodes, it is not required to compute separately two elementary longest paths from one to the other, as the two paths have the same the length. This means that $H_1(r_i) = H_2(r_i)\ \forall$ root r_i and $h_1(v) = h_2(v)\ \forall v \in S$.

In [42], we used a Constraint Programming search method. Here, we use an MIP search method, adapted from *ESP_FltC* to solve ELPP(G). Furthermore, *ELP_FltC_BB* calls only once solveELPP($\{r_i\}, S_i \setminus Exit(r_i), G(S_i)$), and solveELPP($S_i \setminus Exit(r_i), \{r_i\}, G(S_i)$), replacing the various calls of the method solveELPP($\{u\}, \{v\}, G(S_i)$).

Algorithm 4: ComputeSecondStep(r_i)

```
1 foreach r_j ∈ NextRoot(r_i) do
2     ComputeSecondStep(r_j);
3 W ← positive cost of G(S_i);
4 if W > f* then
5     d ← solveELPP(G(S_i), f*);
6     f* ← max{f*, d};
```

5.4 ESPP(s, t, G) on sparse directed graphs with many bridge-blocks

In this section, we apply the decomposition technique described in Section 5.1 to the resolution of ESPP(s, t, G). The proposed algorithm, denoted by *ESP_FltC_BB*, has the following successive steps:

1. Decompose the given graph into bridge-blocks as described in Section 5.1
2. Construct a contracted graph $G_c = (V_{G_c}, E_{G_c})$:
   - V_{G_c} is the set of contracted nodes: each contracted node corresponds to a set of nodes of a bridge-block. We denote by $C(S)$ the contracted node corresponding to the bridge-block S.
   - For each pair of bridge-blocks S_i and S_j, if there exist $u \in S_i$ and $v \in S_j$ such that (u, v) is an edge of the auxiliary graph G', then $(C(S_i), C(S_j))$ is an edge of G_c.
3. Denote by S^s and S^t respectively the bridge-blocks containing s and t.

4. Find the unique path from $C(S^s)$ to $C(S^t)$ in G_c (this path is unique because G_c is a tree). Suppose this path is $\langle C(S^s) = C(S_0), C(S_1), \ldots, C(S_k) = C(S^t)\rangle$.
5. Two consecutive bridge-blocks S_i and S_{i+1} are connected by one edge (u, v) in G'. We denote by $Out(S_i)$ the node u and by $In(S_{i+1})$ the node v ($i = 0, 1, 2, \ldots, k-1$). We denote by $In(S_0)$ the node s and by $Out(S_k)$ the node t.
6. Apply the branch-and-cut algorithm proposed in Section 4 for finding the shortest elementary path from $In(S_i)$ to $Out(S_i)$ in S_i ($\forall i = 0, \ldots, k$).
7. Concatenate these paths, to establish the shortest elementary path from s to t in G. Note that if there is no arc connecting $Out(S_i)$ to $In(S_{i+1})$ in G with $0 \leq i \leq k-1$, then there is no solution to the original problem.

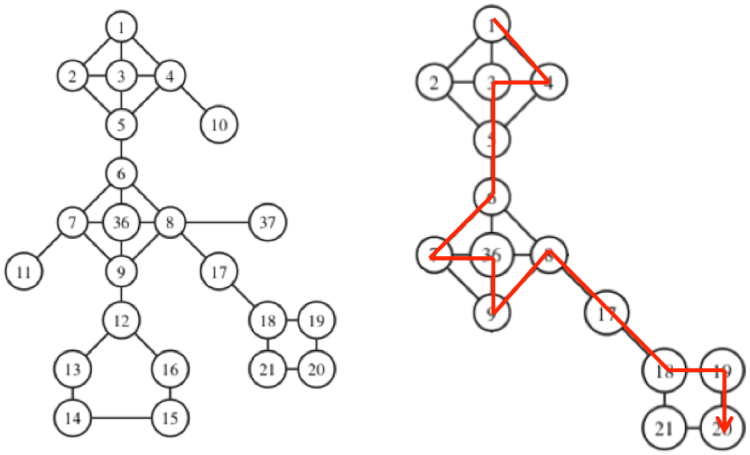

Fig. 8 ESPP(s, t, G) on an undirected graph with many bridge-blocks, $s = 1$, $t = 20$

To compute ESPP(s, t, G) on a sparse undirected graph, we replace each edge by two opposite arcs with the same cost as the edge.

Figure 8 illustrates the algorithm to compute the elementary shortest path from 1 to 20. After finding the unique path in the contracted graph, we have the sequence of bridge-blocks corresponding to this path: $S_0 = \{1, 2, 3, 4, 5\}, S_1 = \{6, 7, 8, 9, 36\}, S_2 = \{17\}, S_3 = \{18, 19, 20, 21\}$. The solution is the concatenation of the shortest elementary paths from 1 to 5 in S_0; from 6 to 8 in S_1; from 17 to 17 in S_2; and from 18 to 20 in S_3.

5.5 ESPP(s, t, G) on sparse directed graphs with strongly connected components

The algorithm *ESP_FltC_SCC* is depicted in detail in Algorithm 5. In lines 1–4, the variables $d(v)$ and $d(u, v)$ are initially assigned a very large value L (e.g., L is the sum of the positive arc costs in the graph). In lines 5–6, for each SCC c_i, the costs of the elementary shortest paths between a node in $I(c_i)$ and a node in $O(c_i)$ are computed by the algorithm *ESP_FltC*. Note that each SCC

is contracted to a single node: the contracted directed graph is acyclic. Method SortTopologocialOrdering(SCC) (line 9) sorts SCC into the sequence $S[1..|SCC|]$ in a topological ordering so that if there exist $u \in S[i]$ and $\in S[j]$ and $(u, v) \in A_G$, then $i < j$. After the ordering, $S[1]$ contains the node s and $S[|SCC|]$ contains the node t. The main idea is based on the well-known algorithm for finding the shortest path on a directed acyclic graph. Each SCC $S[i]$ in the sorted list is considered at each iteration (line 10). Lines 11–20 compute the shortest elementary path from s to nodes of $I(S[i])$ and $O(S[i])$. We use the intermediate variables $d'(v)$ to store temporarily the value of $d(v)$ in order to avoid corruption in case the intersection of $I(S[i])$ and $O(S[i])$ is not empty.

Algorithm 5: *ESP_FltC_SCC(s,t,G)*

```
1  foreach u, v ∈ V_G do
2      d(u, v) ← L ;
3  foreach v ∈ V_G do
4      d(v) ← L; d'(v) ← L;
5  SCC ← compute the set of strongly connected components of G;
6  sz ← |SCC|;
7  foreach c_i ∈ SCC, u ∈ I(c_i), v ∈ O(c_i) do
8      d(u, v) ← solveESPP(u, v, c_i);
9  S[1..sz] ← SortTopologocialOrdering(SCC) ;
10 for i ← 1 to sz do
11     foreach v ∈ I(S[i]) do
12         foreach (u, v) ∈ A_G such that u and v are not in the same strongly connected
           component do
13             if d(v) > d(u) + c_uv then
14                 d(v) ← d(u) + c_uv;
15     foreach v ∈ O(S[i]) do
16         foreach u ∈ I(S[i]) do
17             if d'(v) > d(u) + d(u, v) then
18                 d'(v) ← d(u) + d(u, v);
19     foreach v ∈ O(S[i]) do
20         d(v) ← d'(v);
21 return d(t);
```

6 Experiments

In this section, we compare, in terms of computation time, our algorithms introduced in this paper to the state of the art algorithms in solving the two problems ESPP(s, t, G) and ELPP(G). The characteristics of all the compared algorithms are summarized in Figure 9. The Reference column gives references providing experimental results.

Name	Problem	Underlying Model	On Graph	Reference
ESP_Drexl	ESPP	Model 1 + B&C	Both	[16]
ESP_ComFlow	ESPP	Model 2 + B&C	Both	[this paper]
ESP_MTZ	ESPP	Model 3 + B&C	Both	[this paper]
ESP_FltC	ESPP	Model 1 + B&C	Both	[this paper]
ESP_FltC_BB	ESPP with bridges	Model 1 + B&C	Both	[this paper]
ESP_FltC_SCC	ESPP with SCC	Model 1 + B&C + Dyn	Undirected	[this paper]
ELP_Drexl	ELPP	Model 1	Both	[16]
ELP_FltC	ELPP	Model 1 + B&C	Both	[this paper]
ELP_FltC_BB	ELPP with bridges	Model 1 + B&C + Dyn	Both	[this paper]
ELP_CP	ELPP	CP	Undirected	[42]
ELP_CP_BB	ELPP with bridges	CP + Dyn	Undirected	[42]
ELP_CP2_BB	ELPP	CP + Dyn	Both	[this paper]

Fig. 9 List of algorithms compared

6.1 Instances

We created a total of six classes of instances for the two problems to test the algorithms.

6.1.1 Instances for ESPP(s, t, G)

To test the algorithms in solving ESPP(s, t, G), we created the following three classes of instances:

- **SI1** consists of 15,000 instances, each instance was created from one of 420 directed graphs with one random pair of source and destination nodes. All the directed graphs were created in [16], in which 150 directed graphs were randomly generated and 270 directed graphs were extracted from the pricing sub-problems by a heuristic column generation algorithm for the asymmetric m-salesman travelling salesman problem [45]. All graphs contain at least one negative cycle. More details about the graphs are given in Table 1.

Graph group	Type	No. graphs	$\|V\|$	$\|A\|$	Arc cost range	Arc cost type
R_sparse_25	Random	20	26	300	$[-10; 10]$	Integer
R_sparse_50	Random	20	51	1225	$[-10; 10]$	Integer
R_sparse_100	Random	20	101	4950	$[-10; 10]$	Integer
R_dense_25	Random	30	26	553	$[-1000; 1000]$	Double
R_dense_50	Random	30	51	2353	$[-1000; 1000]$	Double
R_dense_100	Random	30	101	9703	$[-1000; 1000]$	Double
P_first_25	Pricing	30	28	651	$[-10^8; -9.48.10^7]$	Double
P_first_50	Pricing	30	53	2551	$[-10^8; -9.48.10^7]$	Double
P_first_100	Pricing	30	103	10101	$[-10^8; -9.48.10^7]$	Double
P_penultimate_25	Pricing	30	28	651	$[-10^7; 30000]$	Double
P_penultimate_50	Pricing	30	53	2551	$[-10^7; 30000]$	Double
P_penultimate_100	Pricing	30	103	10101	$[-10^7; 30000]$	Double
P_last_25	Pricing	30	28	651	$[-30000; 30000]$	Double
P_last_50	Pricing	30	53	2551	$[-30000; 30000]$	Double
P_last_100	Pricing	30	103	10101	$[-30000; 30000]$	Double

Table 1 420 directed graphs used to create 15,000 instances of the class **SI1**

- **SI2** consists of 5,000 instances that were created from 10 random connected directed graphs with random pairs of source and destination nodes. These directed graphs have many bridge-blocks. In each digraph, the cost of its arcs was generated by an uniform distribution with the range [-100; 100] and one-third of its arcs have a negative cost. In 5 out of 10 directed graphs, each bridge-block includes exactly 20 nodes. In the 5 other directed graphs, each of them has exactly 10 bridge-blocks.
- **SI3** consists of 3,000 instances that were created from 15 random connected directed graphs with many strongly connected components (50 nodes per one strongly connected component). In each digraph, the cost of the arcs was generated by an uniform distribution with the range $[-10.0^6, 10.0^6]$ and one-third of the arcs have a negative cost. More details about the directed graphs are given in Table 2.

Class	No. graphs	$\lvert V\rvert$	$\lvert A\rvert$	$\sharp SCC$	Arc cost range
g_scc_100	5	100	1903-2005	2	$[-10.0^6, 10.0^6]$
g_scc_500	5	500	10051-10068	10	$[-10.0^6, 10.0^6]$
g_scc_1000	5	1000	20130-20148	20	$[-10.0^6, 10.0^6]$

Table 2 15 directed graphs with many strongly connected component were used to generate 3,000 instances of the class **SI3**

6.1.2 Instances for ELPP(G)

- **LI1** consists of 9 instances from 9 random sparse planar undirected graphs without negative-cost arcs. These instances are taken from [42].
- **LI2** consists of 10 instances from 10 random planar connected unidirected graphs without negative-cost arcs. These graphs are built to include many bridge-blocks (10 nodes per bridge-block) and are also taken from [42].
- **LI3** consists of 10 instances created from 10 connected directed graphs with bridges. In each digraph, the costs of the arcs were generated by a uniform distribution with range [-100; 100] and one-third of the arcs have a negative cost.

6.2 Settings

We implemented all the algorithms in C++. For the algorithms based on mixed integer programming, we used IBM Ilog Cplex Concert Technology version 12.4, while for the algorithms based on constraint programming we use the Comet language [1]. The experiments were performed on XEN virtual machines with 1 core of a CPU Intel Core2 Quad Q6600 @2.40GHz and 1 GB of RAM running Linux. A time limit was set, of 20 minutes of CPU time, for each instance.

After extensive experiments using **irace** [46], we decide to set the parameters for *ESP_FltC* as follows:

- Separate SECs using our heuristic separation algorithm with a threshold of 0.25,

– Separate 2-Matching inequalities with a threshold of 0.1,
– Separate maximum outflow inequalities with a threshold 0.01,
– Separate other valid inequalities.

In the two algorithms *ESP_FltC_BB* and *ESP_FltC_SCC*, from Sections 5.4 and 5.5, the elementary shortest path between two specified nodes in each block and strongly connected component is computed by *ESP_FltC*.

In the algorithm *ELP_FltC_BB*, based on the equivalence between the problems in Section 1.1, we slightly adapted *ESP_FltC* to solve ELPP(S, T, G) by the following methods: solveELPP$(\{u\}, \{v\}, G(S_i))$, solveELPP$(\{r_i\}, S_i \setminus Exit(r_i), G(S_i))$, solveELPP$(S_i \setminus Exit(r_i), \{r_i\}, G(S_i)))$, and solveELPP$(G(S_i), f^*))$.

6.3 Solving ESPP(s, t, G)

The state of the art algorithm for ESPP(s, t, G), proposed in [16] and using the Arc-flow formulation (Model 1), is denoted by *ESP_Drexl*. We re-implemented this algorithm to compare it with the algorithms proposed in this paper, *ESP_MTZ*, *ESP_ComFlow*, *ESP_FltC*, *ESP_FltC_BB*, and *ESP_FltC_SCC*.

*6.3.1 Solving 15,000 instances of the class **SI1***

All instances of the class **SI1** were tested to compare 4 algorithms *ESP_Drexl, ESP_ComFlow, ESP_MTZ* and *ESP_FltC* in terms of their computation time.

Instance group	Algorithm	% optimal	B&B nodes (min./avg./max.)	Computation time (min./avg./max.)
Random_sparse_25	*ESP_ComFlow*	100	0/4.71/69	0.15/2.96/13.98
	ESP_MTZ	0	-/-/-	-/-/-
	ESP_Drexl	100	0/2.14/55	0.03/0.08/0.72
	ESP_FltC	100	0/1.82/47	0.02/0.16/15.92
Random_sparse_50	*ESP_ComFlow*	93.6	0/9.13/136	54.18/375.77/1194.59
	ESP_MTZ	0	-/-/-	-/-/-
	ESP_Drexl	100	0/5.52/159	0.06/0.45/18.64
	ESP_FltC	100	0/4.78/123	0.04/0.66/33.78
Random_sparse_100	*ESP_ComFlow*	0	-/-/-	-/-/-
	ESP_MTZ	0	-/-/-	-/-/-
	ESP_Drexl	100	0/4.25/146	0.23/2.09/31.01
	ESP_FltC	100	0/3.71/283	0.07/2.12/31.31

Table 3 Computational results for random sparse instances

The computational results are shown in Tables 3, 4, 5, 6, and 7, and summarized in Table 8. The tables contains the class of test instances and the instances as described in Section 6.1 (*Instance class* and *Instances*); the algorithm used to solve the problem (*Algorithm*); The number of nodes ($|V|$), the number of arcs ($|A|$), the number of edges ($|E|$), the number of bridge-blocks ($|B|$), the number of strongly connected components ($|SCC|$); the percentage of instances solved to optimality in a given limited computation time (*% Optimal*); the number of nodes in the branch-and-bound tree (*B & B nodes*); and the overall CPU time in seconds

Instance group	Algorithm	% optimal	B&B nodes (min./avg./max.)	Computation time (min./avg./max.)
Random_dense_25	*ESP_ComFlow*	100	0/5.19/75	1.35/7.66/36.76
	ESP_MTZ	0	-/-/-	-/-/-
	ESP_Drexl	100	0/3/64	0.03/0.12/2.66
	ESP_FltC	100	0/2.55/48	0.02/0.1/2.99
Random_dense_50	*ESP_ComFlow*	51.7	0/3.19/35	151.58/525.98/1199
	ESP_MTZ	0	-/-/-	-/-/-
	ESP_Drexl	100	0/10.39/154	0.08/1.25/19.56
	ESP_FltC	100	0/9.6/100	0.06/0.3/1.21
Random_dense_100	*ESP_ComFlow*	0	-/-/-	-/-/-
	ESP_MTZ	0	-/-/-	-/-/-
	ESP_Drexl	100	0/22.07/236	0.33/4.07/52.44
	ESP_FltC	100	0/19.81/223	0.32/2.94/34.97

Table 4 Computational results for random dense instances

Instance group	Algorithm	% optimal	B&B nodes (min./avg./max.)	Computation time (min./avg./max.)
P_first_25	*ESP_ComFlow*	100	0/9.16/404	1.92/9.5/63.97
	ESP_MTZ	0	-/-/-	-/-/-
	ESP_Drexl	100	0/7.65/222	0.03/0.19/14.08
	ESP_FltC	100	0/3.61/147	0.02/0.09/2.35
P_first_50	*ESP_ComFlow*	51.7	0/6.89/61	142.13/689.63/1798.69
	ESP_MTZ	0	-/-/-	-/-/-
	ESP_Drexl	100	0/35.03/1003	0.08/1.11/55.66
	ESP_FltC	100	0/24.51/1659	0.06/1.19/35.32
P_first_100	*ESP_ComFlow*	0	-/-/-	-/-/-
	ESP_MTZ	0	-/-/-	-/-/-
	ESP_Drexl	97.4	0/917.84/10845	0/52.54/1142.01
	ESP_FltC	99.2	0/871.63/15453	0.91/96.74/1181.18

Table 5 Computational results for pricing instances

Instance group	Algorithm	% optimal	B&B nodes (min./avg./max.)	Computation time (min./avg./max.)
P_penultimate_25	*ESP_ComFlow*	100	0/0.93/149	0.93/3.73/18.38
	ESP_MTZ	82.6	0/361033/2494670	0.06/176.94/1196.37
	ESP_Drexl	100	0/7.6/109	0.03/0.11/13
	ESP_FltC	100	0/1.04/62	0.02/0.09/3.57
P_penultimate_50	*ESP_ComFlow*	98.8	0/0.75/58	1.0/216.95/1182.66
	ESP_MTZ	19.4	0/339502/1262260	0.19/385.01/1144.67
	ESP_Drexl	100	0/15.89/252	0/1.16/53/04
	ESP_FltC	100	0/3.86/86	0/0.53/8.09
P_penultimate_100	*ESP_ComFlow*	0	-/-/-	-/-/-
	ESP_MTZ	0	-/-/-	-/-/-
	ESP_Drexl	97.5	9/1279.99/11565	1.42/108.71/1185.39
	ESP_FltC	99.9	0/398.44/9935	0/48.74/1114.69

Table 6 Computational results for pricing instances (continued)

(*Computation time*). For the rightmost columns, we give the minimum, average, and maximum values (*min./avg./max.*).

First of all, from the computational results we conclude that the algorithms *ESP_ComFlow* and *ESP_MTZ* are not efficient in solving ESPP(s, t, G) on medium and large-sized graphs compared with the two other algorithms *ESP_Drexl* and *ESP_FltC*. Within the time limit of 20 minutes, *ESP_MTZ* solved instances of 25

Instance group	Algorithm	% optimal	B&B nodes (min./avg./max.)	Computation time (min./avg./max.)
P_last_25	*ESP_ComFlow*	100	0/1.04/74	1.45/4.74/26.52
	ESP_MTZ	0	-/-/-	-/-/-
	ESP_Drexl	100	0/13.95/285	0.04/0.13/5.02
	ESP_FltC	100	0/1.76/64	0.03/0.11/4.67
P_last_50	*ESP_ComFlow*	95.2	0/2.99/203	0.68/256.19/1198.29
	ESP_MTZ	3	0/0/0	0.36/0.44/0.57
	ESP_Drexl	100	0/70.13/1859	0/4.62/215.17
	ESP_FltC	100	0/15.72/600	0/1.13/33.24
P_last_100	*ESP_ComFlow*	0	-/-/-	-/-/-
	ESP_MTZ	0	-/-/-	-/-/-
	ESP_Drexl	95.6	9/1505.53/10256	0.88/114.62/1186.68
	ESP_FltC	99.8	0/512.88/9649	0.85/58.62/1040.2

Table 7 Computational results for pricing instances (continued)

Algorithm	% optimal	B&B nodes (min./avg./max.)	Computation time (min./avg./max.)
ESP_ComFlow	62.62	0/4.25/404	0.93/183.23/1798.69
ESP_MTZ	7	0/38760.42/2494670	0.06/210.34/1196.37
ESP_Drexl	99.36	0/253.53/11565	0/18.93/1186.68
ESP_FltC	99.93	0/124.58/15453	0/14.18/1181.18

Table 8 Computational results over all 15,000 instances

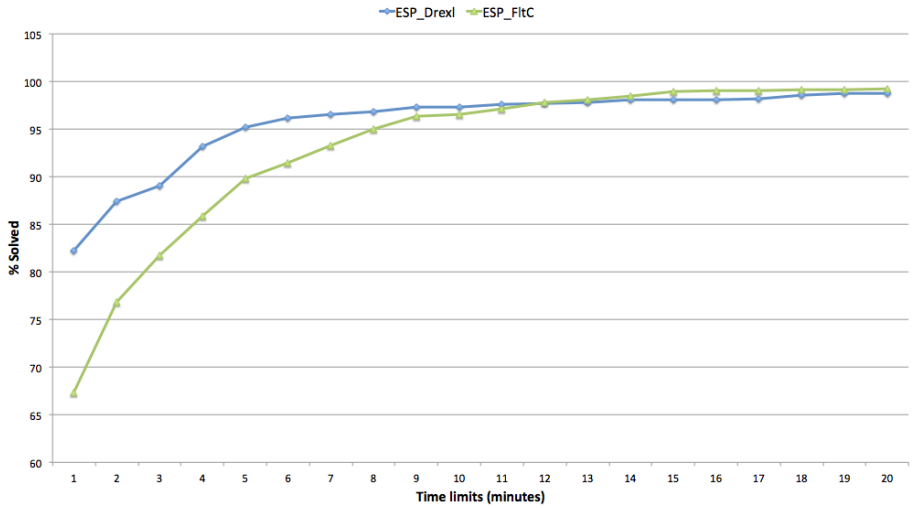

Fig. 10 Comparing two algorithms in terms of the number of solved instances of the group *P_first_100* in given times

nodes with a large computation time and could not solve any instance of 50 and 100 nodes. A weakness of *ESP_MTZ* is the number of nodes in its branch-and-bound tree. While *ESP_ComFlow* can solve some instances of 50 nodes, it cannot solve any instance of 100 nodes because the number of variables in Model 2 (used in *ESP_ComFlow*) is too large.

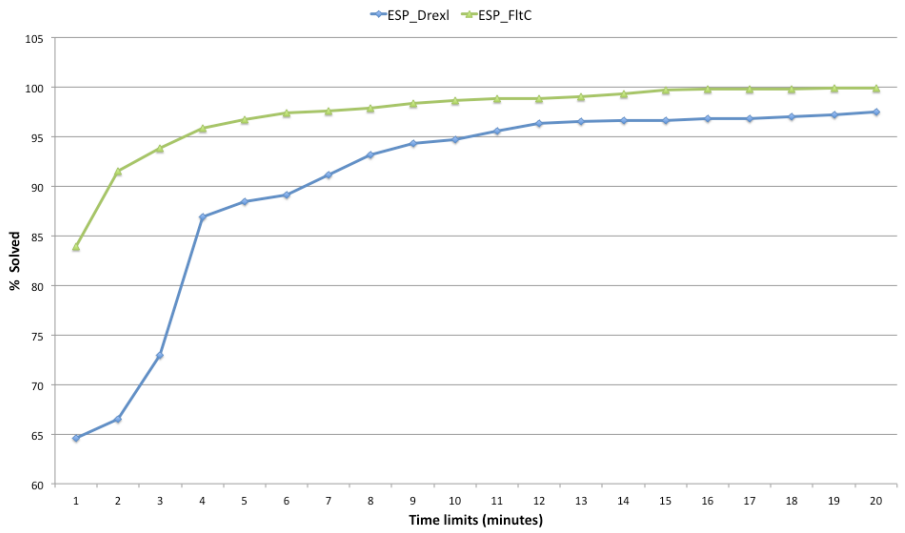

Fig. 11 Comparing two algorithms in terms of the number of solved instances of the group *P_last_but_one_100* in given times

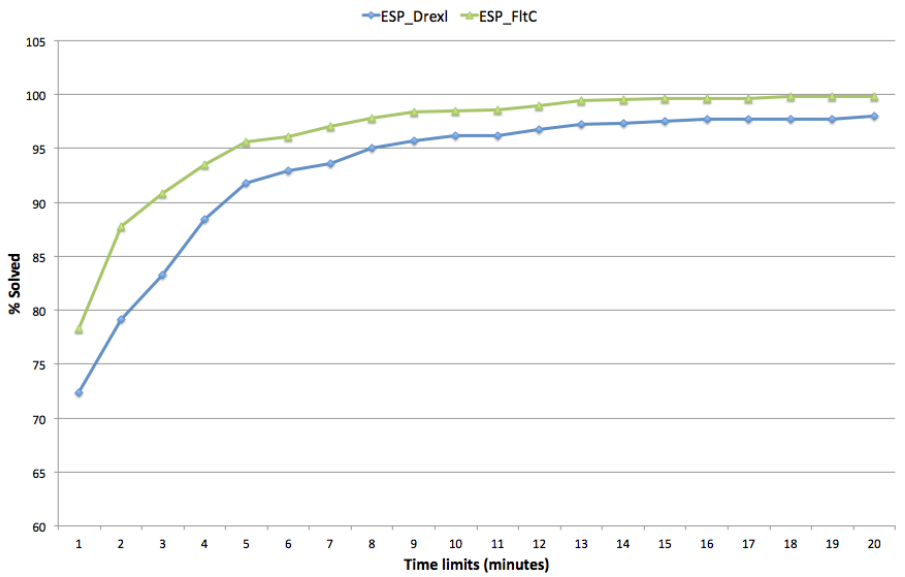

Fig. 12 Comparing *ESP_Drexl* and *ESP_FltC* in terms of the number of solved instances of the group *P_last_100* in given times

Next, Tables (3–7) and Figures (10–12) show that *ESP_FltC* outperforms *ESP_Drexl* in solving 15,000 instances of the class **SI1**. Over all 15,000 instances, within a time limit of 20 minutes, *ESP_FltC* solved 99.93% of all instances, while *ESP_Drexl* solved 99.36 % of all instances. The difference is not significant. However, when solving the 3,000 difficult instances with 100 nodes of the three groups

P_first_100, P_last_but_one_100 and *P_last_100*, our algorithm is faster than the state of the art algorithm. For example, in 1,000 instances of the group *P_last_100*, *ESP_FltC* solved 998 instances with an average computation time of 58.62 seconds, while *ESP_Drexl* solved just 956 instances, with an average computation time of 114.62 seconds. Moreover, Table 10 shows that *ESP_FltC* is much better than *ESP_Drexl* when the time limit is short.

6.3.2 Solving 5,000 instances of the class SI2

We here compare the algorithms *ESP_Drexl*, *ESP_FltC* and *ESP_FltC_BB* in solving 5,000 instances of the class **SI2**. All instances were created from connected directed graphs with many bridges. The computational results are given in Table 9. Note that as *ESP_FltC_BB* computes many times the elementary shortest path in blocks, the number of nodes in the branch-and-bound tree is meaningless.

Instance group	$\|V\|$	$\|A\|$	$\|B\|$	Algorithm	% optimal	B&B nodes (min./avg./max.)	Computation time (min./avg./max.)
g_bb_500	500	6274	25	*ESP_Drexl*	100	0/2.31/86	102.8/272.92/1140.66
				ESP_FltC	100	0/2.49/121	0.07/0.75/9.55
				ESP_FltC_BB	100	na/na/na	0.05/0.33/8.25
g_bb_1000	1000	12549	50	*ESP_Drexl*	65	0/5.98/21	71.42/534.76/1193.88
				ESP_FltC	100	0/9.26/242	0.14/1.29/20.68
				ESP_FltC_BB	100	na/na/na	0.09/0.52/3.56
g_bb_1500	1500	18824	75	*ESP_Drexl*	2	0/1/2	632.17/879.6/1127.02
				ESP_FltC	100	0/9.56/411	0.25/6.47/81.33
				ESP_FltC_BB	100	na/na/na	0.15/8.65/152.08
g_bb_2000	2000	25099	100	*ESP_Drexl*	0	na/na/na	na/na/na
				ESP_FltC	100	0/9.26/210	0.39/9.87/86.22
				ESP_FltC_BB	100	na/na/na	0.25/8.41/165.56
g_bb_2500	2500	31374	125	*ESP_Drexl*	0	na/na/na	na/na/na
				ESP_FltC	100	0/7.93/114	0.51/7.85/109.72
				ESP_FltC_BB	100	na/na/na	0.3/15.88/185.05
g_bb_200	200	2009	10	*ESP_Drexl*	100	0/2.55/39	2.57/8.37/41.4
				ESP_FltC	100	0/2.64/56	0.03/0.41/6.81
				ESP_FltC_BB	100	na/na/na	0.03/0.62/74.5
g_bb_400	400	4009	10	*ESP_Drexl*	100	0/27.42/426	5.15/144.82/1148.39
				ESP_FltC	100	0/13.24/447	0.06/1.97/53.13
				ESP_FltC_BB	100	na/na/na	0.06/1.26/14.08
g_bb_600	600	6009	10	*ESP_Drexl*	79.2	0/18.79/98	17.28/383.29/1191.95
				ESP_FltC	99.4	0/132.52/3147	0.12/32.83/1188.2
				ESP_FltC_BB	100	na/na/na	0.09/8.65/89.19
g_bb_800	800	8009	10	*ESP_Drexl*	64	0/9.55/35	38.66/554.99/1199.02
				ESP_FltC	99.6	0/42.33/555	0.18/22.09/1142.31
				ESP_FltC_BB	100	na/na/na	0.15/9.85/199.1
g_bb_1000	1000	10009	10	*ESP_Drexl*	19.8	0/6.15/21	72.3/504.25/1174.62
				ESP_FltC	85.8	0/115.64/1434	0.39/141.9/1189.99
				ESP_FltC_BB	100	na/na/na	0.45/49.12/664.65

Table 9 Computational results for instances of the class **SI2**

From the computational results we conclude that among the three algorithms, *ESP_FltC_BB* is the best and *ESP_Drexl* is the worst in solving 5,000 instances of the class **SI2** in terms of computation time. For instance,

- As to the 500 instances of the group *g_bb_2500*, *ESP_Drexl* could not solve any instances because of an 'out of memory' error, while *ESP_FltC* and *ESP_FltC_BB* solved all 500 instances with average computation times of 7.85 seconds and 15.88 seconds, respectively.
- As to the 500 instances of the group *g_bb_2500*, *ESP_Drexl* solved 99 instances (19.8%) with an average computation time of 504.25 seconds, while *ESP_FltC* solved 429 instances with an average computation time of 141.9 seconds and *ESP_FltC_BB* solved all 500 instances, with an average computation time of 49.12 seconds.

The efficiency of *ESP_FltC_BB* is analyzed as follows. It is a dynamic programming algorithm that benefits from the properties of the instances where graphs have many bridges. Firstly it computes many elementary shortest paths in blocks. These computations are very fast because the number of nodes in each block is very small, for instance, 20 nodes per one block. Finally it merges appropriately computed paths into a complete solution path.

The computation time for the instances in the group *g_bb_2000* (instances of 2,000 nodes) is much smaller than that of instances in the group *g_bb_1000* (instances of 1,000 nodes), 8.41 seconds compared to 49.12 seconds. The reason is that the number of nodes per block for the instances of 2,000 nodes is only 20, while there are 100 nodes in each block of one of the instances of 1,000 nodes.

In this experiment, *ESP_FltC* clearly outperforms *ESP_Drexl*. For example, *ESP_Drexl* cannot solve any instances of the groups *g_bb_2000* and *g_bb_2,500* while *ESP_FltC* solved all 1,000 instances in these two classes. This efficiency mainly comes from the preprocessing carried out in *ESP_FltC*. Table 10 shows the efficiency of this preprocessing. For some instances of the group *g_bb_2500*, a graph of 2,500 nodes can be reduced to a graph of 20 nodes, as both the source and destination nodes are in the same block of 20 nodes. For some other instances of the group *g_bb_1000*, the reduction does not reduce the number of nodes. On average, the size of the graphs is greatly reduced, as illustrated in Table 10 for the 500 instances of the group *g_bb_2500*.

	Before		After	
Instance group	$\|V\|$	$\|A\|$	$\|V\|$ (min./avg./max.)	$\|A\|$ (min./avg./max.)
g_bb_500	500	6274	20/58.35/400	219/706.5/4993
g_bb_1000	1000	12549	20/57.14/300	220/691.32/3739
g_bb_1500	1500	18824	20/75.81/380	220/925.71/4742
g_bb_2000	2000	25099	20/75.72/520	218/924.57/6499
g_bb_2500	2500	31374	20/86.82/380	220/1063.71/4745
g_bb_200	200	2009	20/51.9/200	172/500.35/1994
g_bb_400	400	4009	39/104.87/399	369/1032.39/3992
g_bb_600	600	6009	60/139.94/600	567/1380.37/5993
g_bb_800	800	8009	80/259.25/800	768/2574.26/7995
g_bb_1000	1000	10009	100/306.02/1000	967/3041.99/9993

Table 10 Reducing the graph size of the instances in the class **SI2** by using the preprocessing by the algorithm *ESP_FltC*

6.3.3 Solving 3,000 instances of the class SI3

In this section, we compare the algorithms *ESP_Drexl*, *ESP_FltC* and *ESP_FltC_SCC* in solving 3,000 instances of the class **SI3** that are directed graphs with many strongly connected components. The computational results are given in Table 11.

From the computational results, we see that *ESP_FltC_SCC* is much more efficient than the other two algorithms in terms of computation time. To reach this efficiency, the dynamic programming algorithm *ESP_FltC_SCC* exploits the special property of instances where the graphs have many strongly connected components.

Instance class	$\|V\|$	Algorithm	% optimal	B&B nodes (min./avg./max.)	Computation time (min./avg./max.)
g_scc_100	100	*ESP_Drexl*	100	0/10.7/296	0.15/2.63/226.45
		ESP_FltC	100	0/14/287	0.05/0.84/18.15
		ESP_FltC_SCC	100	-/-/-	0.04/0.83/6.88
g_scc_500	500	*ESP_Drexl*	75.6	0/22.82/167	8.11/230.04/1195.56
		ESP_FltC	94.1	0/124.59/4258	0.1/55.68/1197.87
		ESP_FltC_SCC	99.3	-/-/-	0.06/27.58/1196.81
g_scc_1000	1000	*ESP_Drexl*	7.2	0/0.19/12	53.02/118.48/1040.27
		ESP_FltC	77.1	0/184.82/54438	0.16/96.16/1195.28
		ESP_FltC_SCC	91.5	-/-/-	0.09/84.21/1122.21

Table 11 Computational results for instances of the class **SI3**

As for solving instances of the class **SI2**, we also see that *ESP_FltC* is much better than *ESP_Drexl* at solving the 3,000 instances of the class **SI3**. This mainly comes from the preprocessing by *ESP_FltC*, as shown in Table 12.

Graph class	Before		After	
	$\|V\|$	$\|A\|$	$\|V\|$ (min./avg./max.)	$\|A\|$ (min./avg./max.)
g_scc_100	100	1903-2005	50/66.95/100	898/1276.77/1977
g_scc_500	500	10051-10068	50/218.445/500	948/4347.46/10035
g_scc_1000	1000	20130-20148	47/283.29/1000	846/5656.28/20109

Table 12 Reducing the size of graphs of the instances in the class **SI3** by the use of preprocessing by the algorithm *ESP_FltC*

6.4 Solving ELPP(G)

We showed that the two problems ESPP(s, t, G) and ELPP(G) are equivalent to each other. Based on this equivalence, we slightly adapt the two algorithms *ESP_FltC* and *ESP_Drexl* to solve ELPP(G). We obtain two new algorithms for solving ELPP(G): *ELP_FltC* and *ELP_Drexl*.

In addition, we propose in this paper the algorithm *ELP_FltC_BB* to solve ELPP(G) on directed graphs with many bridges. This algorithm uses some different MIP search methods (solveELPP($\{r_i\}, S_i \setminus Exit(r_i), G(S_i)$), solveELPP($S_i \setminus Exit(r_i), \{r_i\}, G(S_i)$) and solveELPP($G(S_i), f^*$)). We replace the MIP search methods by CP search methods in *ELP_FltC_BB* to obtain a new algorithm denoted by *ELP_CP2_BB*.

In the literature, for solving ELPP(G), there exist two state of the art algorithms, both based on CP, which were proposed in our previous paper, [42]. The one that solves ELPP(G) on general undirected graphs is denoted by *ELP_CP*. The algorithm that solves ELPP(G) on undirected graphs with many bridges is denoted by *ELP_CP_BB*. Both are implemented in the COMET programming language [1].

6.4.1 Solving 9 instances of the class LI1

We here compare three algorithms in solving 9 instances of the class **LI1**. Two of them are based on MIP: they are *ELP_FltC* and *ELP_Drexl*. The last one is based on CP: it is *ELP_CP*. The time limit has been set here to 30 minutes to meet the experimental settings of *ELP_CP* in [42]. The computation time is given in Table 13. The two branch-and-cut algorithms solved very successfully only the first instance. *ELP_FltC* solved 5 instances, while two others could not solve any other instance. Although the graph size of the instances is not too large, they are very difficult because the costs of the edges are very similar.

Instances	$\|V\|$	$\|A\|$	$\|B\|$	*ELP_FltC*	*ELP_Drexl*	*ELP_CP*
planar-n100-m285	100	285	1	0.8	3.3	-
planar-n150-m432	150	432	1	1660.42	-	-
planar-n200-m583	200	583	1	1761.42	-	-
planar-n250-m731	250	731	1	1763.04	-	-
planar-n300-m880	300	880	1	1760.9	-	-
planar-n350-m1031	350	1031	1	-	-	-
planar-n400-m1182	400	1182	1	-	-	-
planar-n450-m1329	450	1329	1	-	-	-
planar-n500-m1477	500	1477	1	-	-	-

Table 13 Computation times in seconds of three algorithms in solving 9 instances of the class **LI1**

*6.4.2 Solving 10 instances of the class **LI2***

In this section, we compare 5 algorithms at solving ELPP(G) on undirected graphs with many bridges. The results are shown in Table 14. The following analysis can be made.

Instances	$\|V\|$	$\|A\|$	$\|B\|$	*ELP_CP_BB*	*ELP_CP2_BB*	*ELP_Drexl*	*ELP_FltC*	*ELP_FltC_BB*
planar-n100-m216	100	216	10	4.0	1.95	3.34	2.43	1.09
planar-n200-m434	200	434	20	21.32	10.17	12.73	28.23	5.21
planar-n300-m655	300	655	30	54.73	25.28	68.06	192.25	92.5
planar-n400-m870	400	870	40	121.15	59.76	95.65	790.52	1.12
planar-n500-m1089	500	1089	50	235.73	111.19	267.59	-	5.8
planar-n600-m1301	600	1301	60	364.77	169.95	626.89	-	93.85
planar-n700-m1526	700	1526	70	569.15	257.78	-	-	131.3
planar-n800-m1747	800	1747	80	825.12	421.87	-	-	131.88
planar-n900-m1959	900	1959	90	1161.53	554.71	-	-	15.7
planar-n1000-m2177	1000	2177	100	-	186.58	-	-	16.05

Table 14 Computation times in seconds of 5 algorithms at solving 10 instances of the class **LI2**

– *ELP_CP2_BB* solved the instances much more quickly than *ELP_CP_BB*, for example, to solve the instance *planar-n1000-m2177*, *ELP_CP_BB* needed 1649.42 seconds while *ELP_CP2_BB* spent only 186.58 seconds. Both algorithms are based on CP and were implemented in the COMET programming language. The only difference between the two algorithms is that ELP_CP_BB executes only once the methods solveELPP($\{r_i\}, S_i \backslash Exit(r_i), G(S_i)$), and solveELPP($S_i \backslash$

$Exit(r_i), \{r_i\}, G(S_i))$, while the algorithm *ELP_CP_BB* must execute many times the method solveELPP($\{u\}$, $\{v\}, G(S_i)$). Therefore, we conclude that this speeds up computations by *ELP_CP2_BB*.

- *ELP_FltC_BB* is much faster than *ELP_CP2_BB*. For example, to solve the instance *planar-n900-m1959*, *ELP_CP2_BB* needed 554.71 seconds while *ELP_FltC_BB* needed only 15.7 seconds. Here, too, there is only one difference between the two: *ELP_FltC_BB* uses MIP search methods while *ELP_CP2_BB* uses CP search methods. In short, the MIP approach is more efficient than the CP approach in this problem and in this experiment.
- When we compare three dynamic algorithms, *ELP_CP_BB, ELP_CP2 _BB*, and *ELP_FltC_BB*, with the two algorithms that do not exploit the special properties of the graph (*ELP_Drexl* and *ELP_FltC*), we conclude that our proposed dynamic algorithms for solving ELPP(G) on graphs with many bridges is very efficient.

6.4.3 Solving 10 instances of the class LI3

Three algorithms were tested in this experiment: *ELP_Drexl*, *ELP_FltC* and *ELP_FltC_BB*. The task is to solve ELPP(G) on a directed graph with many bridges. The results in computation time are given in Table 15.

Instances	$\|V\|$	$\|A\|$	$\|B\|$	*ELP_Drexl*	*ELP_FltC*	*ELP_FltC_BB*
g_bb_500	500	6274	25	302	131.57	10.53
g_bb_1000	1000	12549	50	-	-	24.13
g_bb_1500	1500	18824	75	-	-	25.76
g_bb_2000	2000	25099	100	-	-	33.74
g_bb_2500	2500	31374	125	-	-	44.11
g_bb_200	200	2009	10	99.21	7.22	4.56
g_bb_400	400	4009	10	-	220.59	14.04
g_bb_600	600	6009	10	-	-	53.15
g_bb_800	800	8009	80	-	-	80.6
g_bb_1000	1000	10009	100	-	-	86.19

Table 15 Comparing three algorithms in solving 10 instances of the class **LI3** in terms of computation time

It is easy to see that *ELP_FltC_BB* clearly dominates the two other algorithms *ELP_Drexl* and *ELP_FltC* as it solves all instances with short computation times while the two other algorithms only solve, respectively, two and three instances.

In short, our proposed algorithm *ELP_FltC_BB* solves efficiently ELPP(G) on directed graphs with many bridges.

7 Conclusion

In this paper, we focus on solving two problems: the elementary shortest path problem between two specified nodes ESPP(s, t, G) and the elementary longest path problem ELPP(G). We also demonstrated some equivalences: one between these two problems and some between other relevant problems.

For ESPP(s, t, G), we presented three integer programming formulations. We slightly adapted a few classes of inequalities that are valid for the asymmetric traveling salesman problem and its variants, to obtain new classes of inequalities that are valid for ESPP(s, t, G). We also proposed a class of inequalities called 'maximum outflow inequalities'. The central point of this paper is that we proposed an exact algorithm based on mixed integer programming. The salient fact about this algorithm is that it generates lots of inequalities of different classes but it uses only the most violated inequalities as cuts to tighten relaxed solutions until it obtains an optimal solution. Moreover, we proposed two dynamic programming algorithms that solve the problem on directed graphs with many bridges and many strongly connected components. The experiments showed that all our proposed algorithms are more interesting than the state of the art algorithms.

For ELPP(G), based on the equivalence between the two problems ESPP(s, t, G) and ELPP(G), we proposed algorithms for ELPP(G) by adapting an algorithm for ESPP(s, t, G). We also extended the dynamic programming algorithm from our previous paper, which was for solving ELPP(G) on undirected graphs with many bridges, to solve ELPP(G) on directed graphs with many bridges. An experiment showed that all our proposed algorithms are more efficient than the state of the art algorithms.

In the future, we would like to apply the technique that uses only the most violated inequalities as cuts, to solve other problems, for example, the elementary shortest path problem with resource constraints.

Acknowledgements We thanks the anonymous reviewers for their helpful and constructive comments. This research was partially sponsored by Vietnamese National Foundation for Science and Technology Development (project FWO.102.2013.04), and by the UCLouvain Action de Recherche Concertee ICTM22C1.

References

1. *Comet Tutorial*. Dynamic Decision Technologies, Inc, 2010.
2. Achterberg, T. and Berthold, T.: Hybrid branching, In *Proceeding of the 6th international conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 309 – 311, 2009.
3. Achterberg, T., Koch, T. and Martin, A.: Branching rules revisited, *Operations Research Letters*, 33(1):42 – 54, 2005.
4. Aráoz, J., Fernández, E. and Meza, O.: A simple exact separation algorithm for 2-matching inequalities. *Research Report DR-2007/13, EIO Departement, Technical University of Catalonia (Spain). http : //www.optimization-online.org/DBHTML/2007/11/1827.html*, 2007.
5. Ascheuer, N., Jünger, M. and Reinelt, G.: A branch & cut algorithm for the asymmetric traveling salesman problem with precedence constraints. *Comput. Optim. Appl.*, 17(1):61–84, 2000.
6. Balas, E., Ceria, S. and Cornuéjols, G.: Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. *Management Science*, 42(9):1229–1246, 1996.
7. Balas, E. and Fischetti, M.: A lifting procedure for the asymmetric traveling salesman polytope and a large new class of facets. *Mathematical Programming*, 58:325–352, 1993.
8. Baldacci, R., Mingozzi, A. and Roberti, R.: Recent exact algorithms for solving the vehicle routing problem under capacity and time window constraints. *European Journal of Operational Research*, 218(1):1 – 6, 2012.
9. Beasley, J.E. and Christofides, N.: An algorithm for the resource constrained shortest path problem. *Network*, 19:379–394, 1989.
10. Belotti, P., et al.: Mixed-integer nonlinear optimization. *Acta Numerica*, 22:1–131, 2013.

11. Bienstock, D., et al.: A note on the prize collecting traveling salesman problem. *Mathematical Programming*, (59):413–420, 1993.
12. Boland, N., Dethridge, J. and Dumitrescu, I.: Accelerated label setting algorithms for the elementary resource constrained shortest path problem. *Operations Research Letters*, 34(1):58 – 68, 2006.
13. Cormen, T.H., et al.: *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
14. Costa, A.M., Cordeau, J.F and Laporte, G.: Steiner tree problems with profits, *INFOR: information systems and operational research*, 44:99–116, 2006.
15. Dell'Amico, M., Maffioli, F. and Värbrand, P.: On prize-collecting tours and the asymmetric travelling salesman problem. *International Transactions in Operational Research*, (38):1073– 1079, 1995.
16. Drexl, M. and Irnich, S.: Solving elementary shortest-path problems as mixed-integer programs. *OR Spectrum*, 1–16, 2012.
17. Dror, M.: Note on the complexity of the shortest path models for column generation in WRPTW. *Operations Research*, 42(5):977–978, 1994.
18. Edmonds, J.: Maximum matching and a polyhedron with 0, 1-vertices. *Journal of Research of the National Bureau of Standards B*, 69:125–130, 1965.
19. Feillet, D., Dejax, P. and Gendreau, M.: Traveling salesman problems with profits. *Transportation Science*, 39:188–205, 2005.
20. Feillet, D., et al.: An exact algorithm for the elementary shortest path problem with resource constraints: Application to some vehicle routing problems. *Networks*, 216–229, 2004.
21. Fischetti, M.: Facets of the asymmetric traveling salesman polytope. *Mathematics of Operations Research*, 16(1):pp. 42–56, 1991.
22. Fischetti, M., Lodi, A. and Toth, P.: Exact methods for the asymmetric traveling salesman problem. In Gregory Gutin and AbrahamP. Punnen, editors, *The Traveling Salesman Problem and Its Variations*, 169–205, 2004.
23. Goemans, M.X. and Williamson, D.P.: A general approximation technique for constrained forest problems. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, 307–316, 1992.
24. Gondran, M. and Minoux, M.: *Graphes et algorithmes, 3e édition revue et augmentée*. Eyrolles, 1995.
25. Grötschel, M. and Padberg, M.: Polyhedral theory. In In E. L. Lawler, Jan Karel Lenstra and A. H. G. Rinnooy Kan, editors, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, 251–305, 1985.
26. Ibrahim, M. S., Maculan, N. and Minoux, M.: A strong flow-based formulation for the shortest path problem in digraphs with negative cycles. *International Transactions in Operational Research*, 16:361–369, 2009.
27. Irnich, S. and Desaulniers, G.: Shortest Path Problems with Resource Constraints. In Guy Desaulniers, Jacques Desrosiers and Marius M. Solomon, editors, *Column Generation*, 33–65, 2005.
28. Jepsen, M.K., Petersen, B. and Spoorendonk, S.: A branch-and-cut algorithm for the elementary shortest path problem with a capacity constraint. *Technical report, Department of Computer Science, University of Copenhagen*, 2008.
29. Karger, D., Motwani, R. and Ramkumar, G.: On approximating the longest path in a graph. *Algorithmica*, 18:82–98, 1997.
30. Koch, T. and Martin, A.: Solving Steiner tree problems in graphs to optimality. *Networks*, 32:207–232, 1998.
31. Kulkarni, R.V. and Bhave, P.R.: Integer programming formulations of vehicle routing problems. *European Journal of Operational Research*, 20(1):58 – 67, 1985.
32. Little, John D.C., et al.: An algorithm for the traveling salesman problem, *Operations Research*, 11:972–989, 1963.
33. Ljubić, I., et al.: An algorithmic framework for the exact solution of the prize-collecting Steiner tree problem. *Mathematical Programming*, 105:427–449, 2006.
34. Ljubić, I., et al.: An algorithmic framework for the exact solution of the prize-collecting Steiner tree problem. *Mathematical Programming*,427-449, 2006.
35. Lucena, A. and Beasley, J.E.: A branch and cut algorithm for the Steiner problem in graphs. *Networks*, 31:39–59, 1998.
36. Miller, C.E., Tucker, A.W. and Zemlin, R.A.: Integer programming formulation of traveling salesman problems. *J. ACM*, 7(4):326–329, 1960.

37. Nguyen, V.H. and Nguyen, T.T.T.: Approximating the asymmetric profitable tour. *Electronic Notes in Discrete Mathematics*, 36:907 – 914, 2010.
38. Padberg, M. and Rinaldi, G.: Optimization of a 532-city symmetric traveling salesman problem by branch and cut. *Operations Research Letters*, 6(1):1 – 7, 1987.
39. Padberg, M. and Rinaldi, G.: Facet identification for the symmetric traveling salesman polytope. *Mathematical Programming*, 47:219–257, 1990.
40. Padberg, M.W. and Rao, M.R.: Odd minimum cut-sets and b-matchings. *Mathematics of Operations Research*, 7:67–80, 1982.
41. Padberg, M.W. and Grötschel, M.: Polyhedral computation. In Lawler, E. L., Lenstra, J.K. and Rinnooy Kan, A. H. G. editors, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, 251–305, 1985.
42. Pham, Q.D and Deville, Y.: Solving the longest simple path problem with constrains-based techniques In *Proceedings of the 9th international conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 292–306, 2012.
43. Portugal, D., Antunes, C.H. and Rocha, R.P.: A study of genetic algorithms for approximating the longest path in generic graphs. In *Systems Man and Cybernetics (SMC), 2010 IEEE International Conference on*, 2539 –2544, 2010.
44. Portugal, D. and Rocha, R.P.: Msp algorithm: multi-robot patrolling based on territory allocation using balanced graph partitioning. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, 1271–1276, 2010.
45. Punnen, A.P.: The traveling salesman problem: Applications, formulations and variations. In Gutin, G. and Punnen, A.P., editors, *The Traveling Salesman Problem and Its Variations*, 1–28, 2004.
46. Radulescu, A., López-Ibáñez, M. and Stützle, T.: Automatically improving the anytime behaviour of multiobjective evolutionary algorithms. *Technical Report TR/IRIDIA/2012-019, IRIDIA, Université Libre de Bruxelles, Belgium*, 2012.
47. Righini, G. and Salani, M.: Symmetry helps: Bounded bi-directional dynamic programming for the elementary shortest path problem with resource constraints. *Discrete Optimization*, 3:255–273, 2006.
48. Righini, G. and Salani, M.: New dynamic programming algorithms for the resource constrained elementary shortest path problem. *Network*, 51:155–170, 2008.
49. Schmidt, K. and Schmidt, E.G.: A longest-path problem for evaluating the worst-case packet delay of switched ethernet. In *Industrial Embedded Systems (SIES), 2010 International Symposium on*, 205–208, 2010.
50. Tarjan, R.: Depth-first search and linear graph algorithm. *SIAM J. Comput*, 146–160, 1972.
51. Toth, P. and Vigo, D.: The vehicle routing problem. *SIAM Monographs on Discrete Mathematics and Applications*, 2002.
52. Tseng, I.L., Chen, H.W. and Lee, C.I.: Obstacle-aware longest-path routing with parallel MILP solvers. In *Proceedings of the World Congress on Engineering and Computer Science*, 827-831, 2010.
53. Uchoa, E.: Reduction tests for the prize-collecting Steiner problem. *Operations Research Letters*, 34(4):437 – 444, 2006.
54. Volgenant, T. and Jonker, R.: On some generalizations of the travelling-salesman problem. *Journal of the Operational Research Society*, (3):297 – 308, 1987.
55. Wong, W.Y. Information retrieval in p2p networks using genetic algorithm. In *Proceedings of the 14th International World Wide Web Conference*, 922-923, 2005.