European Journal of Operational Research 236 (2014) 695–705

Contents lists available at ScienceDirect

European Journal of Operational Research

journal homepage: www.elsevier.com/locate/ejor

Innovative Applications of O.R.

Column-generation based bounds for the Homogeneous Areas Problem

Fabio Colombo, Roberto Cordone *, Marco Trubian

Università degli Studi di Milano, Dipartimento di Informatica, Via Comelico 39, 20135 Milano, Italy

ARTICLE INFO

Article history:
Received 2 May 2012
Accepted 23 December 2013
Available online 31 December 2013

Keywords:
Graph partitioning
Column Generation
Tabu Search

ABSTRACT

Given an undirected graph and a collection of vertex subsets with suitable costs, we consider the problem of partitioning the graph into subgraphs of limited cost, splitting as little as possible the given subsets among different subgraphs. This problem originates from the organization of a region (the graph) including several towns (the vertices) into administrative areas (the subgraphs). The officers assigned to each area take care of activities which involve several towns at a time (the subsets). An activity involving towns from more areas engages the officers of all those areas, leading to redundancies which must be minimized.

This paper introduces a column generation approach to compute a lower bound for the problem. Since the pricing subproblem is $\mathcal{NP}$-hard, we solve it with a Tabu Search algorithm, before applying a suitably strengthened multi-commodity flow formulation. Moreover, we also compute an upper bound for the overall problem with a primal heuristic based on the idea of diving and limited discrepancy search. The computational results refer to two real-world instances, a class of realistic instances derived from them, and two different classes of random instances.

1. Introduction

The *Homogeneous Areas Problem* (*HAP*) can be formulated as follows. Let $G = (V, E)$ be an undirected graph with $V = \{1, \ldots, n\}$, $\mathcal{S} \subseteq 2^V$ a collection of subsets of vertices, $q : \mathcal{S} \to \mathbb{R}^+$ a cost function defined on $\mathcal{S}$ and Q a cost threshold. Finally, let k be an integer positive number. Given any subset of vertices $U \subseteq V$, we denote the cost of the subgraph induced by U as $\sum_{S \in \mathcal{S}: S \cap U \neq \emptyset} q_S$, i.e. the sum of the values q_S for all subsets S intersecting U. The problem requires to partition graph G into at most k vertex-disjoint connected subgraphs $G_i = (U_i, E_i)$ such that the cost of G_i does not exceed Q for all i and the total cost

$$\phi = \sum_i \sum_{S \in \mathcal{S}: S \cap U_i \neq \emptyset} q_S \tag{1}$$

is minimum. The *HAP* is strongly $\mathcal{NP}$-hard (Ceselli, Colombo, Cordone, & Trubian, in press).

This problem derives from a practical requirement, concerning the partitioning of two administrative regions in Northern Italy (the provinces of Milan and Monza) into "homogeneous areas". In that case, the vertices correspond to towns, the edges to pairs of adjacent towns, each subset $S \in \mathcal{S}$ represents an activity involving a subset of towns and requiring from the officers of the province administration a known amount of working hours, q_S. The aim of the problem is to divide the province into connected areas (subgraphs) and to assign a team of officers to each area, in such a way that each activity is split as little as possible among different areas. This is due to the fact that the officers in charge of an area need to be trained on all the activities involving the towns of the area and, therefore, splitting an activity implies a redundancy (more officers trained on the same topics). The cost of a subgraph expresses the number of working hours required from the officers in charge of the corresponding area. The limited number of working hours available for each officer imposes an upper threshold on the cost of each area. The value of this threshold can also be tuned to improve fairness among the teams.

In Fig. 1 we report a sample instance and some of its solutions. Fig. 1(a) provides a graph G with 7 vertices and 9 edges, three subsets with costs $q_{S_1} = 10, q_{S_2} = 9$ and $q_{S_3} = 11$, a cost threshold $Q = 25$ and a maximum number of subgraphs $k = 3$. If the nodes could be partitioned so as to keep all subsets in $\mathcal{S}$ unsplitted, the overall cost would hit the theoretical lower bound $\sum_{S \in \mathcal{S}} q_S = 30$. This value, however, cannot be reached due to the cost threshold imposed on each subgraph. Fig. 1(b) shows an optimal solution, with two subgraphs and a total cost equal to $q_{S_1} + 2q_{S_2} + q_{S_3} = 39$ (subset S_2 intersects both the subgraphs). Fig. 1(c) shows a suboptimal solution with three subgraphs, in which both S_1 and S_2 intersect two subgraphs, so that the overall cost is $2q_{S_1} + 2q_{S_2} + q_{S_3} = 49$. Finally, the solution in Fig. 1(d) is unfeasible since the subgraph induced by $U = \{1, 2, 3, 4, 5\}$ intersects all three subsets and its cost $q_{S_1} + q_{S_2} + q_{S_3} = 30$ exceeds the threshold $Q = 25$.

* Corresponding author. Tel.: +39 0250316235; fax: +39 0250316373.
E-mail addresses: fabio.colombo2@unimi.it (F. Colombo), roberto.cordone@unimi.it (R. Cordone), marco.trubian@unimi.it (M. Trubian).

0377-2217/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.ejor.2013.12.030

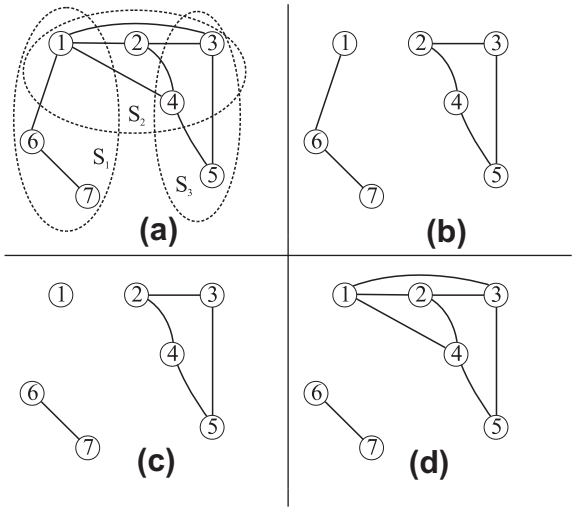

Fig. 1. (a) Sample instance having $q_{S_1} = 10, q_{S_2} = 9, q_{S_3} = 11$ and $Q = 25$; (b) Optimal solution with cost $q_{S_1} + 2q_{S_2} + q_{S_3} = 39$; (c) Sub-optimal solution with cost $2q_{S_1} + 2q_{S_2} + q_{S_3} = 49$; (d) Unfeasible solution: the subgraph induced by $U = \{1, 2, 3, 4, 5\}$ has cost $q_{S_1} + q_{S_2} + q_{S_3} = 30 > Q$.

This work proposes a column generation approach to the *HAP*. Since in our decomposition the pricing subproblem is itself $\mathcal{NP}$-hard, we apply a customized heuristic before solving it exactly with an Integer Linear Programming (*ILP*) formulation, for which we introduce some valid inequalities. The heuristic is a Tabu Search algorithm, while the exact approach exploits a multicommodity flow formulation. Section 2 introduces a compact formulation of the problem, and an extended one, solved by a column generation approach. Section 3 deals with the pricing subproblem, discussing its formulation with some strengthenings, its computational complexity and a heuristic approach to solve it. Section 4 presents a primal heuristic for the *HAP*. It is based on the column generation framework and exploits the concepts of diving and limited discrepancy search. The final section presents the computational results.

1.1. On the relationship with graph partitioning problems

The *HAP* can be seen as a variant of the Graph Partitioning Problem (*GPP*). This section provides some references to the huge literature on the *GPP*, a small example to illustrate the specific features of the *HAP* and a discussion of the differences it exhibits with respect to the other related models. A more detailed discussion, with counterexamples to the possibility of easily reducing the *HAP* to a standard *GPP*, can be found in Ceselli et al. (in press).

Given an undirected edge-weighted graph $G = (V, E)$, the most common versions of the *GPP* ask to divide the vertex set V into a given number k of nonempty, pairwise disjoint subsets, such that the edge-cut, i.e. the total weight of the edges that connect vertices in different subsets, is minimized. This basic problem admits a number of variations; see, e.g., the survey in Fjällström (1998). Several different approaches have been proposed to solve them, such as hierarchic multi-level heuristics (Sanders & Schulz, 2011), geometry-based and flow-based methods (Arora, Rao, & Vazirani, 2008), genetic approaches (Kim, Hwang, Kim, & Moon, 2011), spectral methods (Donath & Hoffman, 1973), mathematical programming approaches (Fan & Pardalos, 2010), local search metaheuristics and integrated approaches (Osipov, Sanders, & Schulz, 2012). The *HAP* differs from these classical *GPP*s both in the constraints and in the objective function, which pose specific challenges to a solving algorithm.

1.1.1. Cardinality constraint

In the *GPP*, the number k of vertex-disjoint subsets is usually given, and the subsets are required to be nonempty, since their cardinalities, $n_1, \ldots, n_k$, with $\sum_{j=1}^{k} n_j = |V|$ are explicitly imposed (Guttmann-Beck & Hassin, 2000) or constrained to be approximately of the same size, see e.g. Osipov et al. (2012). In the *HAP*, k is just an upper threshold, so that *the subsets of vertices U_i are allowed to be empty*. In fact, merging two subsets into a single one is always profitable, and only the cost threshold Q possibly forbids to do it.

1.1.2. Cost threshold

The *HAP* is related to the *Node-capacitated Graph Partitioning Problem* (Ferreira, Martin, de Souza, Weismantel, & Wolsey, 1998), in which the total weight of each subset in the partition is limited by a threshold. However, the threshold is managed differently in the *HAP*: since it is not associated to single vertices, the cost of a subset U does not increase linearly as new vertices are included, but stepwise as new subsets $S \in \mathcal{S}$ intersect U. Such a nonlinear dependence is much harder to handle.

1.1.3. Connectivity constraint

The connectivity constraint is usually not imposed in *GPP*s, where the edges of the graph are taken into account only when computing the objective function. Quite commonly, the edge costs model a proximity measure, and the subsets end up naturally to be connected in the optimal solution. In the *HAP*, on the contrary, the edges determine the feasibility of the solutions, since each subset must induce a connected subgraph on G, but they have no relation with the objective function. In fact, even considering the smaller benchmark instances, which can be solved exactly, the optimal result obtained relaxing the connectivity constraints is on average 35% lower than the one obtained respecting them (Ceselli et al., in press). This suggests that neglecting the connectivity constraint would not provide meaningful information on the original problem and that classical methods ignoring this constraint would not provide useful solutions.

1.1.4. Objective function

The objective function of the classical *GPP*s depends linearly on the cost of the edges whose extreme vertices belong to different subgraphs. Sometimes, this cost is tuned by a function of the cardinality of the subgraphs; see, e.g., Matula and Shahrokhi (1990). The objective function of the *HAP* is completely independent from the edge set E, and depends nonlinearly on the intersections between the subsets in $\mathcal{S}$ and the subsets of vertices of the subgraphs.

These remarks on the difference between the constraints and the objective function of the *HAP* with respect to other *GPP*s have moved us to develop *ad hoc* methods, instead of straightforwardly adapting algorithms drawn from the literature.

2. Mathematical programming formulations

Hereafter, we present two different formulations of the *HAP*. The first one is a compact multicommodity flow formulation that can be directly solved using a commercial *ILP* solver. The second one is an extended formulation which associates a variable to each feasible subgraph. At the end of the section, we describe the column generation approach used to solve the continuous relaxation of the extended formulation.

2.1. Compact formulation

The *HAP* admits a multicommodity flow formulation based on an auxiliary directed graph $G' = (V, E')$, derived from G replacing

each edge in E with two opposite arcs. This allows to enforce the connectivity requirement by representing each subgraph as a rooted arborescence. There are at most k roots, i.e. nodes generating flow. Each other node receives flow generated from exactly one root, absorbs a single unit of flow and transmits the rest. The index ℓ of the root denotes in the model both the corresponding commodity and the arborescence whose nodes receive the flow. To avoid multiple equivalent solutions, without loss of generality, we require the root of each arborescence to be the node with minimum index. Therefore, any arborescence rooted in ℓ is restricted to the subgraph $G^\ell = (V^\ell, E^\ell)$, where $V^\ell = \{v \in V : v \geqslant \ell\}$ and $E^\ell = \{(u,v) \in E' : u \geqslant \ell, v \geqslant \ell\}$. For example, referring to Fig. 1a, subgraph G^4 consists of four vertices ($V^4 = \{4,5,6,7\}$) and two edges ($E^4 = \{(4,5),(6,7)\}$). Finally, for each node $v \in V$ we denote the collection of subsets from $\mathscr{S}$ which contain v as $\mathscr{S}_v = \{S \in \mathscr{S} : v \in S\}$. For example, $\mathscr{S}_4 = \{S_2, S_3\}$.

Given the following decision variables:

- $x_v^\ell = 1$ if an arborescence rooted in ℓ includes $v \in V^\ell, x_v^\ell = 0$ otherwise;
- $z_S^\ell = 1$ if an arborescence rooted in ℓ intersects $S \in \mathscr{S}, z_S^\ell = 0$ otherwise;
- f_{uv}^ℓ is the flow on directed arc $(u,v) \in E^\ell$ generated by node $\ell \in V$

the *HAP* can be formulated as:

$$\min \phi = \sum_{S \in \mathscr{S}} \sum_{\ell \in V} q_S z_S^\ell \tag{2a}$$

$$\sum_{\ell \in V : v \in V^\ell} x_v^\ell = 1 \quad v \in V \tag{2b}$$

$$\sum_{\ell \in V} x_\ell^\ell \leqslant k \tag{2c}$$

$$\sum_{S \in \mathscr{S}} q_S z_S^\ell \leqslant Q x_\ell^\ell \quad \ell \in V \tag{2d}$$

$$x_v^\ell \leqslant z_S^\ell \quad \ell \in V, v \in V^\ell, S \in \mathscr{S}_v \tag{2e}$$

$$x_v^\ell \leqslant x_\ell^\ell \quad \ell \in V, v \in V^\ell \tag{2f}$$

$$\sum_{(\ell,v) \in E^\ell} f_{\ell v}^\ell = \sum_{v \in V^\ell \setminus \{\ell\}} x_v^\ell \quad \ell \in V \tag{2g}$$

$$\sum_{(u,v) \in E^\ell} f_{uv}^\ell \leqslant \left(|V^\ell| - 1\right) x_v^\ell \quad \ell \in V, v \in V^\ell \tag{2h}$$

$$\sum_{(u,v) \in E^\ell} f_{uv}^\ell - \sum_{(v,w) \in E^\ell} f_{vw}^\ell = x_v^\ell \quad \ell \in V, v \in V^\ell \setminus \{\ell\} \tag{2i}$$

$$x_v^\ell \in \{0,1\} \quad \ell \in V, v \in V^\ell \tag{2j}$$

$$0 \leqslant z_S^\ell \leqslant 1 \quad S \in \mathscr{S}, \ell \in V \tag{2k}$$

$$f_{uv}^\ell \geqslant 0 \quad \ell \in V, (u,v) \in E^\ell \tag{2l}$$

Eq. (2a) defines the objective function ϕ. Constraints (2b) state that each vertex belongs to exactly one arborescence. Constraint (2c) imposes the correct number of arborescences, since $x_\ell^\ell = 1$ if and only if the solution contains an arborescence rooted in ℓ. Constraints (2d) state that, if ℓ is the index of an arborescence, its cost should not exceed the threshold. Constraints (2e) state that, if a node v is assigned to an arborescence, all subsets $S \in \mathscr{S}$ which contain v contribute to the cost of the arborescence. Constraints (2f) state that if a node belongs to arborescence ℓ, node ℓ is the root of the arborescence. Constraints (2g) state that the root of an arborescence generates one unit of flow for each other node of the arborescence. Constraints (2h) state that, if a node receives flow ℓ, it belongs to arborescence ℓ. Coefficient $|V^\ell|$ can be replaced by any upper bound on the number of the nodes in arborescence ℓ, since each node absorbs exactly one unit of flow. As discussed in the following, tighter values should be preferred because they strengthen the continuous relaxation of the model. Constraints (2i) guarantee the conservation of flow, while the following ones impose integrality or nonnegativity on the decision variables. Notice that, if the x_v^ℓ variables are binary, constraints (2e) and the objective function trivially guarantee that also the z_S^ℓ variables are binary. Formulation (2) can be strengthened by fixing the values of some variables and introducing additional logical constraints, as described in Ceselli et al. (in press).

2.2. Extended formulation

It is possible to derive from Formulation (2) an alternative one, which implies a column generation approach. Let $\mathscr{G}^\ell$ be the collection of all connected subgraphs of G which have ℓ as the node with minimum index and whose cost is not larger than Q. We define a binary variable y_i^ℓ for each element of $\mathscr{G}^\ell$, with $i = 1, \ldots, |\mathscr{G}^\ell|$, such that $y_i^\ell = 1$ if the ith subgraph is used in the solution, $y_i^\ell = 0$ otherwise.

$$\min \phi = \sum_{\ell \in V} \sum_{i=1}^{|\mathscr{G}^\ell|} \phi_i^\ell y_i^\ell \tag{3a}$$

$$\sum_{\ell \in V} \sum_{i=1}^{|\mathscr{G}^\ell|} a_{iv} y_i^\ell = 1 \quad v \in V \quad (\lambda_v \text{ free}) \tag{3b}$$

$$\sum_{\ell \in V} \sum_{i=1}^{|\mathscr{G}^\ell|} y_i^\ell \leqslant k \qquad (\mu \leqslant 0) \tag{3c}$$

$$y_i^\ell \in \{0,1\} \quad \ell \in V, i = 1, \ldots, |\mathscr{G}^\ell| \tag{3d}$$

where ϕ_i^ℓ is the cost of the ith subgraph and a_{iv} indicates whether node v belongs to the ith subgraph or not.

Eq. (3a) defines the objective function ϕ. Constraints (3b) state that each node should belong to exactly one subgraph; each λ_v represents the dual variable of the corresponding constraint in the continuous relaxation of the problem. Constraint (3c) imposes the correct number of subgraphs; μ represents the corresponding nonpositive dual variable.

2.3. Computing the linear relaxation of the extended formulation

To solve Formulation (3) we start by relaxing the integrality conditions on the binary variables y_i^ℓ, generating the so called master problem (MP). Since the MP has an exponential number of variables, we apply a column generation approach (Desrosiers & Lübbecke, 2005). In particular we define the reduced master problem (RMP) that, at the beginning, considers only a small collection of feasible subgraphs, $\bar{\mathscr{G}}^\ell$, for each $\ell \in V$. Then we solve the RMP and we use the optimal dual values either to generate non-basic variables with negative reduced costs or to prove that the current basic solution is optimal also for the full MP. This requires to solve, for each $\ell \in V$, the so called pricing problem (PP$^\ell$), as described in Section 3. When the full MP is solved to optimality, its optimum provides a lower bound for the original problem.

3. On solving the pricing problem

Hereafter, we present two different approaches to solve each pricing problem PP$^\ell$. The first one is a flow formulation which can be solved exactly by a commercial *ILP* solver. The second one is a Tabu Search heuristic, which is used to quickly identify variables with negative reduced costs. Section 5.3 describes the strategy used to combine these two approaches to efficiently obtain a lower bound for the *HAP*. Section 4 describes the role of the Tabu Search heuristic as a component of a primal heuristic for the *HAP*.

3.1. A formulation of the pricing subproblem

The pricing subproblem for each node $\ell \in V$ looks for a connected subgraph in $\mathscr{G}^\ell$ with the minimum reduced cost. We represent each subgraph as a rooted arborescence to enforce the connectivity requirement, as already done for the compact formulation. The pricing subproblem PP^ℓ can be formulated introducing the following decision variables:

- $x_v = 1$ if the arborescence includes node $v \in V^\ell$, and $x_v = 0$ otherwise;
- $z_S = 1$ if the arborescence intersects subset $S \in \mathscr{S}$, and $z_S = 0$ otherwise;
- f_{uv} is the flow on directed arc $(u, v) \in E^\ell$

$$\min \bar{\phi}^\ell = \sum_{S\in\mathscr{S}} q_S z_S - \sum_{v\in V^\ell} \lambda_v x_v - \mu \tag{4a}$$

$$\sum_{S\in\mathscr{S}} q_S z_S \leqslant Q \tag{4b}$$

$$x_v \leqslant z_S \quad v \in V^\ell, S \in \mathscr{S}_v \tag{4c}$$

$$x_\ell = 1 \tag{4d}$$

$$\sum_{(\ell,v)\in E^\ell} f_{\ell v} = \sum_{v\in V^\ell\setminus\{\ell\}} x_v \tag{4e}$$

$$\sum_{(u,v)\in E^\ell} f_{uv} \leqslant \left(|V^\ell| - 1\right) x_v \quad v \in V^\ell \tag{4f}$$

$$\sum_{(u,v)\in E^\ell} f_{uv} - \sum_{(v,w)\in E^\ell} f_{vw} = x_v \quad v \in V^\ell \setminus \{\ell\} \tag{4g}$$

$$x_v \in \{0,1\} \quad v \in V^\ell \tag{4h}$$

$$0 \leqslant z_S \leqslant 1 \quad S \in \mathscr{S} \tag{4i}$$

$$f_{uv} \geqslant 0 \quad (u,v) \in E^\ell \tag{4j}$$

The objective function $\bar{\phi}^\ell$, given by Eq. (4a), is the reduced cost of a column (of index ℓ) of Formulation (3). It can be also interpreted as the difference between the cost of the column, on one hand, and the sum of the prizes λ_v of the nodes included in the corresponding subgraph plus the constant term μ, on the other hand. Hereafter, for any feasible solution of Formulation (4), we will denote the term $\sum_{S\in\mathscr{S}} q_S z_S$ as the *cost* of the corresponding arborescence, as opposed to its reduced cost, which is the objective of the pricing subproblem. Constraint (4b) limits the cost of the arborescence. Constraints (4c) state that, if a node v is assigned to the arborescence, all subsets $S \in \mathscr{S}$ which contain v contribute to the cost of the arborescence. Constraint (4d) guarantees that the root node belongs to the arborescence. Constraint (4e) sets the amount of flow generated by the root of the arborescence equal to the number of the other nodes belonging to it. Constraints (4f) state that, if a node receives one unit of flow, it must belong to the arborescence. Constraints (4g) guarantee the conservation of flow, while the following ones impose integrality or nonnegativity on the decision variables.

3.1.1. Computational complexity

The problem PP^ℓ is $\mathscr{NP}$-hard. Let us consider the problem PP which requires to identify simultaneously both the root ℓ and the corresponding minimum reduced cost arborescence among the optimal solutions of all the PP^ℓ for each $\ell = 1, \ldots, n$. It is straightforward to observe that, in order to solve PP, it is sufficient to solve all the PP^ℓ problems and to identify the solution with minimum cost. Hence, if we can solve each PP^ℓ in polynomial time, we can also solve PP in polynomial time. The following proposition proves that PP is $\mathscr{NP}$-hard, thus implying that PP^ℓ is $\mathscr{NP}$-hard, too.

Proposition 1. *The PP problem is $\mathscr{NP}$-hard, even if $\lambda_v \geqslant 0$ for each $v \in V$ and $Q = +\infty$.*

Proof. The proof is based on a reduction from the *Maximum Weight Connected Subgraph* (*MWCS*) problem, which is $\mathscr{NP}$-hard (Ideker, Ozier, Schwikowski, & Siegel, 2002) and is defined as follows. Given a graph $\widetilde{G} = (\widetilde{V}, \widetilde{E})$, and a weight function defined on the vertices, $w : \widetilde{V} \to \mathbb{R}$, find a connected subgraph $\widetilde{G}' = (\widetilde{V}', \widetilde{E}')$ of $\widetilde{G}$ with maximum total weight $w_{\widetilde{G}'} = \sum_{v\in\widetilde{V}'} w_v$.

We now show that from any given instance of the *MWCS* problem, it is possible to build an instance of problem PP, such that their optimal solutions correspond one-to-one. Graph $G = (V, E)$ coincides with $\widetilde{G}$. We set $Q = +\infty$ and $\mu = 0$. For each vertex $v \in \widetilde{V}$ such that $w_v < 0$, we define a singleton subset $S = \{v\} \in \mathscr{S}$ with cost $q_S = -w_v$ and we set $\lambda_v = 0$. For each vertex $v \in \widetilde{V}$ such that $w_v \geqslant 0$, we set $\lambda_v = w_v$.

Each feasible solution $G' = (V', E')$ of problem PP is a connected subgraph, and therefore is also feasible for the *MWCS* problem. Its cost is equal to $\sum_{v\in V':w_v<0}(-w_v) - \sum_{v\in V':w_v\geqslant 0} w_v = -\sum_{v\in V'} w_v$, which is the opposite of the objective function of the *MWCS*. Thus, minimizing the objective of PP corresponds to maximizing the objective of the *MWCS*. □

3.1.2. Formulation strengthening

Formulation (4) can be strengthened by adapting the improvements introduced in Ceselli et al. (in press) for the compact formulation (2). These improvements introduce new constraints and fix the value of some variables. First of all, the cost constraint (4b) induces logical constraints on the node variables x_v. These constraints derive from the computation of lower bounds on the cost of the solution, that is on the left-hand side of constraint (4b), under the assumption that one or two nodes different from the given root ℓ are imposed or forbidden in the solution. If any such lower bound exceeds Q, the constraint is certainly violated and the assumption must be reversed. Since all feasible solutions are connected subgraphs, lower bounds can be obtained computing the minimum cost of simple paths on auxiliary node-weighted graphs. In particular, given the subgraph G^ℓ defined above, we introduce the auxiliary cost function

$$p^{\ell u}(t) = \begin{cases} \sum_{S\in\mathscr{S}_t\setminus(\mathscr{S}_\ell\cup\mathscr{S}_u)} \frac{q_S}{|V_S|} & t \in V^\ell \setminus \{\ell, u\} \\ 0 & t \in \{\ell, u\} \end{cases}$$

whose value $p^{\ell u}(t)$ represents a lower bound on the contribution of node t to the cost of any arborescence rooted in ℓ and including node u. In addition, the cost of such an arborescence must necessarily include the cost of all subsets in $\mathscr{S}_\ell$ and $\mathscr{S}_u$.

Proposition 2. *Given a root $\ell \in V$ and a node $u \in V^\ell$, let $\pi^{\ell u}$ be the minimum cost of a path on subgraph G^ℓ between nodes ℓ and u, with respect to the cost function $p^{\ell u}$. If $\pi^{\ell u} + \sum_{S\in(\mathscr{S}_\ell\cup\mathscr{S}_u)} q_S > Q$, then $x_u = 0$ in any feasible solution of Formulation (4).*

Proof. Any feasible solution of the pricing problem PP^ℓ is a connected subgraph including vertex ℓ. Let us assume, by contradiction, that the solution also includes vertex u. The minimum possible left-hand side of constraint (4b) is associated with a path connecting ℓ and u. Its cost is underestimated by $\pi^{\ell u}$ plus the total cost of the subsets containing ℓ and u. If this underestimate exceeds the cost threshold Q, the assumption must be reversed, fixing $x_u = 0$. □

Proposition 3. *Given a root $\ell \in V$ and two nodes $u, v \in V^\ell$ with $u \neq v$, let $\pi^{\ell u}_{\neg v}$ be the minimum cost of a path on subgraph G^ℓ between nodes ℓ and u which does not use v, with respect to the cost function $p^{\ell u}$. If $\pi^{\ell u}_{\neg v} + \sum_{S \in (\mathscr{S}_\ell \cup \mathscr{S}_u)} q_S > Q$, then any feasible solution of Formulation (4) satisfies the following binding constraint*

$$x_u \leqslant x_v$$

Proof. Let us now assume, by contradiction, that the solution includes vertex u and does not include vertex v. The minimum possible left-hand side of constraint (4b) is associated with a path connecting ℓ and u, and not visiting v. The cost of such a path can be underestimated by $\pi^{\ell u}_{\neg v}$, plus the total cost of the subsets containing ℓ and u. If the underestimate exceeds Q, the assumption must be reversed, stating that if the solution includes u, it necessarily includes also v. □

A complementary implication can be derived by defining the following auxiliary cost function

$$p^{\ell u v}(t) = \begin{cases} \sum_{S \in \mathscr{S}_t \setminus (\mathscr{S}_\ell \cup \mathscr{S}_u \cup \mathscr{S}_v)} \frac{q_S}{|V_S|} & t \in V^\ell \setminus \{u, v\} \\ 0 & t \in \{\ell, u, v\} \end{cases}$$

whose value $p^{\ell u v}(t)$ represents a lower bound on the contribution of node t to the cost of any arborescence rooted in ℓ and including u and v.

Proposition 4. *Given a root $\ell \in V$ and two nodes $u, v \in V^\ell \setminus \{\ell\}$ with $u < v$, the minimum cost of a path between ℓ and u (ℓ and v, or u and v) with respect to the cost function $p^{\ell u v}$, plus the constant term $\sum_{S \in (\mathscr{S}_\ell \cup \mathscr{S}_u \cup \mathscr{S}_v)} q_S$ is a lower bound on the cost of any arborescence rooted in ℓ including both u and v. If any of these three lower bounds exceeds Q, then any feasible solution of Formulation (4) satisfies the following incompatibility constraint*

$$x_u + x_v \leqslant 1$$

Proof. By contradiction, we assume that the solution includes both vertex u and vertex v, besides the root ℓ. We underestimate the left-hand side of constraint (4b) by requiring the solution to connect two of the three vertices, instead of all of them. This yields three possible lower bounds, given by the total cost of the subsets containing ℓ, u and v, plus the minimum cost of a path between two of the three vertices. If any of these underestimates exceeds Q, the assumption must be reversed, stating that u and v cannot be both included in the solution. □

The $(|V^\ell| - 1)$ coefficient of Constraints (4f) can be eventually reduced by replacing $|V^\ell|$ with any upper bound M^ℓ on the number of nodes which can belong to a feasible arborescence rooted in ℓ.

Proposition 5. *The optimum of the following problem provides an upper bound on the number of nodes of a feasible arborescence rooted in ℓ.*

$$\max M^\ell = \sum_{v \in V^\ell} x_v \tag{5a}$$

$$\sum_{S \in \mathscr{S}} q_S z_S \leqslant Q \tag{5b}$$

$$x_\ell = 1 \tag{5c}$$

$$x_v \leqslant z_S \quad \in V^\ell, S \in \mathscr{S}_v \tag{5d}$$

$$x_v \in \{0, 1\} \quad v \in V^\ell \tag{5e}$$

$$0 \leqslant z_S \leqslant 1 \quad S \in \mathscr{S} \tag{5f}$$

Proof. The problem can be obtained from the pricing subproblem PP^ℓ by relaxing all constraints concerning the flow variables, that is by neglecting the connection requirement. The objective function maximizes the number of vertices in the subgraph. This necessarily provides an upper bound on the number of vertices of each feasible solution of PP^ℓ. □

Of course, the bound improves if one includes in Formulation (2) the fixings, the binding and the incompatibility constraints described in Propositions 2–4.

It is worth noticing that all the bounds and strengthenings described above depend on the labels assigned to the nodes when defining set V as $\{1, \ldots, n\}$. In fact, they depend on subgraph G^ℓ, which strictly depends on these labels. Consequently, different labellings of the nodes could yield more or less effective strengthenings. According to our experience, a good heuristic labelling can be obtained by computing for each node v the number of incident arcs δ_v and the total cost of the subsets which include v, i.e. $\gamma_v = \sum_{S \in \mathscr{S}_v} q_S$, and sorting the nodes, first by nonincreasing δ_v and then by nonincreasing γ_v.

3.2. A Tabu Search heuristic for the pricing problem

Since the pricing subproblem is $\mathscr{NP}$-hard and since the systematic application of an *ILP* solver proved rather inefficient, we developed a Tabu Search heuristic, *PrTS*, to quickly identify negative reduced cost columns. Hence, we limit the use of the *ILP* solver to the cases in which the heuristic *PrTS* fails to provide any.

Tabu Search is a well-known local search metaheuristic approach which allows the visit of nonimproving solutions. It is controlled by memory mechanisms to avoid the insurgence of cyclic behaviours. It was introduced by Glover (1986) and the interested reader can find in Glover and Laguna (1997) a detailed treatment of its applications and variants. In the following, we mainly focus on the specific aspects of our implementation.

First of all, the heuristic *PrTS* does not impose a fixed root node, thus implicitly solving the overall pricing problem PP instead of the single subproblems PP^ℓ for each $\ell \in V$. *PrTS* starts from a given feasible solution $G' = (U, E')$, which corresponds to a subgraph of G, possibly empty. We defined two simple moves: the addition (removal) of a node to (from) the current set of nodes U. At each iteration, the heuristic evaluates all nodes $v \in V$, one at a time: if $v \in U$, it computes the value of the reduced cost of the graph induced by $U \setminus \{v\}$; if $v \in V \setminus U$, it computes the value of the reduced cost of the graph induced by $U \cup \{v\}$. In either case, the move is forbidden if the resulting subset of nodes does not induce a connected subgraph or if its cost exceeds the threshold Q. The neighbourhood of each solution, therefore, is the set of all the solutions which can be obtained by applying one of the two kinds of move, and it contains at most n members.

3.2.1. A nonstandard tabu mechanism

As for any Tabu Search method, *PrTS* classifies the solutions in the current neighbourhood either as tabu or non-tabu. The presence of a tabu mechanism has the purpose to avoid visiting solutions obtained previously by forbidding to reverse the effect of moves performed too recently. The common way to implement this mechanism is to maintain, explicitly or implicitly, a list of attributes of performed moves and to forbid the execution of moves whose attributes are in the list. The list has a limited length, say *tt*, usually called *tabu tenure*, and it is managed as a FIFO list. This implies that, after lasting *tt* iterations in the list, an attribute is removed from it and all the moves which have that attribute can now be performed.

In our algorithm, for each node $v \in V$, we save in I_v the last iteration in which v changed status, either entering or leaving the solution. In technical terms, the *attribute* of a move is the index v of the node which is removed or added. As a consequence, the solution obtained adding (removing) a node v to (from) the current set of

nodes U is tabu if the value of the current iteration counter is smaller than $I_v + tt$, meaning that v was moved for the last time less than tt iterations ago.

In the Tabu Search literature there are two mainstreams, respectively adopting a fixed and a variable tabu tenure. In the latter case, the tenure is usually updated depending either on the quality of the last move performed or on the cardinality of the neighbourhood. More specifically, it is common to decrease the value of the tenure when the last move performed is improving and to increase it in the opposite case; as well, it is common to decrease the value of the tenure when the neighbourhood becomes smaller and to increase it when it becomes larger (Glover & Laguna, 1997). The purpose of these adaptive mechanisms is to favour the exploration of more promising regions of the solution space and to drive the search away from less promising ones.

Since we visit only feasible solutions, which correspond to connected subgraphs of G, and since in general graph G is not complete, the size of the neighbourhood defined above can vary significantly from iteration to iteration. For this reason, the use of a fixed tabu tenure proved very ineffective, and even the standard adaptive mechanisms, based on the quality of the last move or on the size of the current neighbourhood, failed. In fact, we frequently observed that the value of the tabu tenure could not keep pace with the current situation. For example, the moves whose attributes were saved in the tabu list were quite often nearly all unfeasible, and therefore unnecessarily tabu. On the other hand, the insurgence of a cyclic behaviour triggered the standard anti-cycling mechanism of increasing the tabu tenure, until nearly all feasible moves became tabu. This worsened the quality of the available solutions. The result was that the search moved alternatively between cycles and bad solutions.

In order to solve this problem, we decided to get rid of the tabu tenure, while preserving the basic idea of Tabu Search. At each iteration, we compute the number k of feasible moves and we consider tabu the $\lfloor \epsilon k \rfloor$ moves with the most recent attribute I_v. Parameter $\epsilon \in (0;1)$ is defined by the user. Please notice that, due to the above rounding and since $\epsilon < 1$, at least one move is always non-tabu.

In general, the move selected at each step is the one which produces the non-tabu solution with the minimum reduced cost in the neighbourhood, but we also apply the standard *aspiration criterion*: if a solution is tabu, but its reduced cost is the smallest one found so far, the tabu status is overridden.

We also apply the following *anti-cycling mechanism*: if for a given number of consecutive iterations K_{acm} the same sequence of moves generates the same sequence of objective function values, we assume this as a hint that a cyclic behaviour is occurring, and consequently increase ϵ to $\epsilon' \in (\epsilon;1)$ for other K_{acm} iterations, in an attempt to break the cycle.

Finally, we adopt a *frequency-based diversification strategy*. We save the number n_v of visited solutions which contain node $v \in V$ and the number n_S of visited solutions which contain subset $S \in \mathscr{S}$. If the objective function does not improve for K_{ni} consecutive iterations, we start a diversification phase, which lasts for K_{div} consecutive iterations. During this phase, we replace the objective function with the following one

$$\tilde{\phi} = \sum_{S \in \mathscr{S}} \frac{n_S}{\max\limits_{S \in \mathscr{S}} n_S} q_S z_S - \sum_{v \in V} \left(1 - \frac{n_v}{\max\limits_{v \in V} n_v}\right) \pi_v x_v$$

The aim of this change is to decrease the cost of the subsets and to increase the prize of the nodes which have occurred less frequently in the visited solutions.

3.2.2. Initialization

Heuristic *PrTS* requires a starting solution. As we embed it in a column generation approach, we restart *PrTS* from the subgraphs associated with all the k' basic variables y_i^ℓ which have a strictly positive value in the current optimal solution of the RMP. These solutions are promising starting points because, by definition, they have a zero reduced cost. Our experiments show that, using this *warm start* strategy, the overall column generation algorithm requires less computing time.

3.2.3. Solution pool

To improve the column generation convergence rate, instead of the best solution, we save all the negative reduced cost columns found by *PrTS*, checking them so as to avoid duplicates (in general, the limited memory mechanism of Tabu Search allows to visit the same solution more than once). When the heuristic terminates, we add all the saved columns to the RMP.

3.2.4. Stopping criteria

PrTS has three stopping criteria. First of all, it stops as soon as it has found $C_{\max}$ columns. In fact, adding several columns in each iteration decreases the number of iterations required to obtain the optimal solution, but also increases the time required to solve the RMP. So, we need to find a trade-off between these two effects. Second, for each of the k' starting solutions, *PrTS* performs at least $I_{\min}/k'$ iterations. If during this search it finds at least one negative reduced cost column, it moves to the next starting solution. Otherwise, it proceeds until either it finds a negative reduced cost column or it performs $I_{\max}/k'$ iterations, and moves to the next starting solution.

4. A column generation based heuristic for the *HAP*

Besides a lower bound, the column generation approach also provides useful information to build good heuristic solutions. In the literature, there are different strategies to exploit such information. For example, Cacchiani, Hemmelmayr, and Tricoire (2014) propose an effective heuristic for the Periodic Vehicle Routing Problem, based on the combination of column generation with Tabu Search, while Prescott-Gagnon, Desaulniers, and Rousseau (2009) combine branch and price with Large Neighbourhood Search (Pisinger & Ropke, 2010) to obtain quasi optimal solutions for the Vehicle Routing Problem with Time Windows. In this paper, we develop a heuristic for the *HAP* following another strategy, proposed in Joncour, Michel, Sadykov, Sverdlov, and Vanderbeck (2010).

This algorithm, denoted in the following as *HAP-LDS*, can be seen as a truncated exploration of a branching tree in which the branching variables are columns of the extended formulation. Since the pricing problem is $\mathscr{NP}$-hard, when we process a node of the branching tree, the associated MP is solved only heuristically. In other words, when the Tabu Search procedure fails to find variables with negative reduced cost, instead of applying the *ILP* solver to Formulation (4), the column generation algorithm terminates. Hence, the final value of the RMP is not guaranteed to be a lower bound for the original problem, and the final solution of the RMP is not guaranteed to be optimal for the MP. However, this solution probably includes useful information.

HAP-LDS maintains a list L of tabu columns, which is empty at the root node. At each node of the branching tree, the algorithm selects the column with the largest value in the solution of the current RMP among those which are not tabu. Then, it fixes the selected column to 1 and updates the RMP accordingly: the right-hand side of Constraints (3b) turns from 1 to 0 for the nodes belonging to the subgraphs associated to the fixed columns and the right-hand side of Constraint (3c) decreases by the number of fixed columns. In the end, the algorithm reoptimizes the RMP with column generation, always applying only the heuristic pricing

procedure. In practice, it is very easy to take into account the variable fixing when solving heuristically the pricing subproblems: we just remove from graph G all nodes belonging to the columns which have been fixed to 1 so far, and solve the pricing subproblem on the remaining subgraph. The columns fixed to 0 are not forbidden in the Tabu Search procedure, but they are simply not returned to the RMP. In fact, the same check which avoids introducing duplicates in the solution pool of the pricing subproblem (see the end of Section 3.2) allows to identify these columns and to avoid returning them. The process of fixing columns and reoptimizing the RMP is called *diving*. It terminates either when the current RMP has an integer solution or when it becomes unfeasible. The best integer feasible solution is saved. Then, the search backtracks, guided by two parameters: the maximum depth $D_{\max}$ and the maximum length $L_{\max}$ of the tabu list. In detail, the backtracking stops when the current depth becomes $< D_{\max}$ or when the length of the current tabu list becomes $< L_{\max}$. When it is no longer possible to backtrack, *HAP-LDS* terminates. Otherwise, it creates a new child node, whose tabu list includes all the columns which were tabu in the parent node plus those that have been fixed in the previous sibling nodes.

An example of branching tree is illustrated in Fig. 2, for $D_{\max} = 2$, $L_{\max} = 2$. The labels of the branching nodes indicate the order in which they are visited. The label of each arc reports in round parenthesis the tabu list L which constrains the choice of the next fixed column and the column which is fixed into the solution. Let the candidate columns at the root node be y_a, y_b and y_c, in nonincreasing order of value. After fixing column y_a and reoptimizing, let the candidate columns at node 1 be y_d, y_e and y_f. Since list L is empty, algorithm *HAP-LDS* chooses y_d and reoptimizes the RMP. Then, it "dives", i.e. it keeps fixing other columns until the current RMP has an integer solution or becomes unfeasible. It backtracks up to the first level whose depth is $< D_{\max} = 2$, i.e. up to node 1, inserts column y_d into list L and creates a new child node 3, fixing column y_e. From there, the algorithm dives again, and backtracks once more up to 1. Now, it inserts column y_e into list L and creates a new child node 4, fixing column y_f. From node 4, *HAP-LDS* first dives and then backtracks up to node 0; node 1, in fact, cannot generate other children, because the length of list L has grown equal to $L_{\max} = 2$. Back at the root node, the algorithm puts column y_a into L and fixes column y_b. Then, it proceeds as reported in Fig. 2. In particular, notice how parameter $L_{\max}$ limits the number of children at node 5 and directly imposes to dive at node 8.

The best feasible integer solution found during the diving phases is saved. Since the *HAP* is a partitioning problem and imposes an upper limit k on the number of subgraphs, the algorithm does not guarantee to always obtain a feasible solution. At the end of the branching process, however, the columns generated in all branching nodes (including the root) form an *ILP* problem, which is a reduced instance of Formulation (3). We solve it by means of a general-purpose solver, possibly obtaining a feasible solution, which cannot be worse than the best one found (if any) during the diving phases.

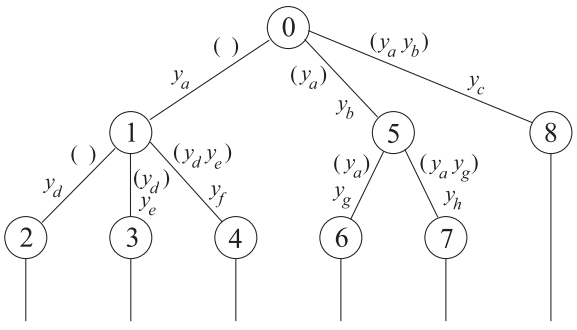

Fig. 2. An example of branching tree for heuristic *HAP-LDS*.

5. Computational results

In this section, we first describe the benchmark instances used, then we report the comparison between the lower bounds computed with the column generation approach and with the multicommodity flow formulation (2). In the end, we compare the upper bounds obtained by the primal heuristic of Section 4 with those obtained by the alternative heuristic *VLNS-TS* proposed in Ceselli et al. (in press) and with the best known lower bounds. All the algorithms presented above have been implemented in C++ language and run on an Intel Pentium Core 2 Duo E6700 2.6 gihahertz with 3 gigabytes of RAM. The *LP* and *ILP* problems have been solved with CPLEX 12.2.

5.1. Benchmark instances

We tested our algorithms on two real-world instances, on a benchmark set A of 25 instances extracted from the real ones and on two benchmark sets B and C of, respectively, 72 and 60 randomly generated instances. All the tested instances are available at http://homes.di.unimi.it/cordone/research/hap.html.

The real-world instances correspond to the Italian provinces of Milan and Monza: the former has 134 vertices and 774 subsets, the latter 55 vertices and 426 subsets. The adjacency graph G derives from geographical data. The number k of areas, the subsets $\mathscr{S}$, the cost q_S of each subset and the threshold Q have been provided by the staff of the two provinces. In particular, $k = 9$ for Milan and $k = 3$ for Monza.

The benchmark set A was generated merging the graphs of Milan and Monza, which are geographically adjacent, and extracting subgraphs from their union, according to the following strategy. First, we selected the five towns involved in the most costly activities, that is with the largest value of $\sum_{S \in \mathscr{S}_v} q_S$. Starting from each seed town, we extracted the first n vertices found during a breadth-first visit of the adjacency graph, with $n = 50, 60, 70, 80$ and 90. This allowed to produce five different instances with a shape reasonably similar to a standard province and centred on a reasonable main town. We included in each instance all the subsets containing the extracted towns. We set the cost threshold Q to the same value used for the whole province, and identified reasonable values for the maximum number of subgraphs, k, running algorithm *VLNS-TS*. This produced $5 \cdot 5 = 25$ instances overall.

In the random benchmarks B and C, the structure of graph G, and the cardinality of the subsets $S \in \mathscr{S}$ mirror those of the real-world instances, but the vertices composing each subset are randomly distributed. The graph G is a random planar graph built as follows. First, we uniformly generate n points in a Euclidean squared plane. Then we build a triangulation of these points: we consider, in turn, all pairs of points by non-decreasing distances, and we draw the corresponding segment if and only if it does not cross the previous ones. Each point turns into a node and each segment into an edge of graph G. Given the number of nodes, n, the number of subsets is $|\mathscr{S}| = 2n$, and the number of subgraphs is $k = n/5$. The collection of subsets $\mathscr{S}$ is randomly generated, devoting special care to guarantee that each node belongs to at least one subset and that no subset is empty. The average cardinality of the subsets S is set to $\alpha|\mathscr{S}|$, with $\alpha \in \{0.05; 0.1\}$.

The two random benchmark sets differ with respect to the distribution of the values of function q_S. In B, the distribution mirrors the rather involved way in which costs are assigned both to subsets and to single vertices in the real-world instances (see

Ceselli et al. (in press) for more details). In C, the cost of each subset directly depends on the cost of the vertices it contains. In fact, the values q_S are computed as $q_S = \sum_{v \in S} w_v$, where w_v is either fixed to 1 or randomly extracted (with a uniform distribution) from $\{1, \ldots, 10\}$ or $\{1, \ldots, 100\}$.

As for the cost threshold Q, this is defined in both benchmark sets as

$$Q = \beta \frac{\sum_{S \in \mathscr{S}} q_S \min(k, |V_S|)}{k}$$

which is a rough estimate of the average cost of each subgraph, where coefficient $\beta \in \{1.00, 1.15\}$ allows to produce tight instances or loose ones.

Benchmark B combines 9 sizes (n ranges from 30 to 70 by steps of 5), two average cardinalities for the subsets, two distributions for q_S and two cost thresholds Q, and therefore includes $9 \cdot 2 \cdot 2 \cdot 2 = 72$ instances overall. Set C combines 5 sizes (n ranges from 30 to 70 by steps of 10), three ranges for w_v (i.e. three alternative distributions for q_S) and two cost thresholds Q, and therefore includes $5 \cdot 2 \cdot 3 \cdot 2 = 60$ instances.

5.2. A note on the objective function

The objective function ϕ as defined in Eq. (1) is equivalent to the following one:

$$\psi = \phi - \sum_{S \in \mathscr{S}} q_S$$

since we are subtracting a constant term to ϕ. Function ψ measures the excess cost with respect to the ideal situation in which no subset is split among different subgraphs. Moreover, function ψ removes an offset which unduly reduces the gap between upper and lower bounds, or between different heuristic solutions. In fact, given two solutions A and B with the corresponding objective function values ϕ_A and ϕ_B, or ψ_A and ψ_B, the following relation is always true:

$$\frac{|\phi_A - \phi_B|}{\min\{\phi_A, \phi_B\}} \leqslant \frac{|(\phi_A - \sum_{S \in \mathscr{S}} q_S) - (\phi_B - \sum_{S \in \mathscr{S}} q_S)|}{\min\{\phi_A, \phi_B\} - \sum_{S \in \mathscr{S}} q_S} = \frac{|\psi_A - \psi_B|}{\min\{\psi_A, \psi_B\}}$$

In all the following tables we report gaps referring to function ψ.

5.3. Lower bound comparison

This section compares the lower bound obtained by solving with CPLEX the multicommodity flow formulation (2) with the lower bound achieved by solving the extended formulation (3) with our column generation approach. Formulation (2) has been strengthened as in Ceselli et al. (in press) with inequalities and parameter settings corresponding to those given by Propositions 2–5 for the pricing subproblem.

We adopt the following strategy to speed up the column generation process. At first, we apply heuristic *PrTS* with the parameter setting reported in Table 1. As long as *PrTS* finds negative reduced cost columns, we add them to the RMP, and reoptimize it. After that, for each $\ell \in V$ in turn, we invoke CPLEX to solve problem PP^ℓ. As soon as CPLEX finds a negative reduced cost column (no matter if it is an optimal solution or not), we add it to the RMP and reoptimize, as we do with the Tabu Search heuristic. When CPLEX proves that PP^ℓ admits no such column, we select the next root ℓ and proceed with the associated pricing problem. When CPLEX fails to identify a negative reduced cost column for all $\ell \in V$, the optimal solution of the RMP is optimal also for the MP and provides a lower bound for the extended formulation. The whole process benefits by considering the roots ℓ in increasing order, because in this way we solve first the larger subproblems PP^ℓ, which are more likely to provide negative reduced cost columns.

Notice that in our first experiments we directly applied CPLEX to solve the pricing subproblems, because we had not yet implemented the Tabu Search heuristic. With that configuration, we could not compute the continuous relaxation of the extended formulation in a reasonable amount of time even for small instances. Only after developing the Tabu Search heuristic, it became possible to solve the MP. Moreover, our analysis of the computational experiments shows that the time required to solve the smallest pricing subproblems is negligible w.r.t. the time required to solve the biggest ones. As a consequence, we did not enumerate the smallest solutions, and we did not develop any *ad hoc* exhaustive search procedure, but we decided to simply apply the commercial *ILP* solver.

Tables 2 and 3 report the results obtained, respectively, on the benchmarks B and C of random instances, and on the instances A extracted from the real ones. The first columns of the three tables identify each tested instance, while column BK contains the best known value of the objective function. Under label CF_{root} (2) we report the percentage gap between the best known value and the lower bound provided by the continuous relaxation of the compact formulation (2) (column Δ (%)). The gap is computed as $(BK - LB)/LB$. This bound can be computed in less than one second for all the considered instances of benchmarks B and C. For the instances of benchmark A, Table 4 includes an additional column, labelled CPU, which reports the time required, in seconds. As the percentage gaps are huge (the upper bound is often several times larger than the lower bound), the following two columns, labelled CF (2) report the percentage gap and the computational time obtained by CPLEX, with a time limit of one hour. If an instance can be solved to optimality, the gap column reports a "–" label and the CPU column reports a value lower than 3600; otherwise, the gap column reports the residual gap and the CPU column reports the label "TL". Finally, the columns labelled EF_{root} (3) report the percentage gap and the computational time required to solve the continuous relaxation of the extended formulation (3).

Notice that in these tables we report only the results obtained on the smallest instances in the considered benchmarks. On the larger instances, in fact, even the best lower bounds found by CPLEX in one hour exhibited very large gaps w.r.t. the best known upper bounds.

The results for the multicommodity flow formulation describe what can be achieved directly applying a general-purpose solver in a reasonable time limit. In all cases, the gap at the root node is very large. Anyway, the branching process allows to solve to optimality within one hour some instances with 30 vertices and a single instance with 35 vertices of the random benchmarks B and C, considered in Tables 2,3. By contrast, for the instances A extracted from the real ones, considered in Table 4, CPLEX is able to solve in one hour all the instances up to $n = 60$ and some instances with 70 and 80 nodes. The gap increases steeply when passing from 30 to 35–40 vertices for benchmarks B and C and from 70 to 80 vertices for benchmark A, and the computation requires to analyze hundreds of thousands of branching nodes, with a non-negligible memory consumption.

The column generation approach, on the contrary, does not require any branching operation and its results are obtained in a matter of a few seconds (few minutes for the instances of benchmark A). The gap is quite stable with respect to the size of

Table 1
Values of the parameters.

Anti-cycling			Diversification		Stopping criteria		
K_{acm}	ϵ	ϵ'	K_{ni}	K_{div}	$C_{\max}$	$I_{\min}$	$I_{\max}$
3	0.7	0.9	25	30	1000	2000	20,000

Table 2
Comparison of the percentage gaps between the best known upper bound and the lower bound achieved by the compact multicommodity flow formulation and the extended formulation solved with column generation on the random instances of benchmark B within a time limit of one hour.

Instance				BK	CF_{root} (2)	CF (2)		EF_{root} (3)	
n	α	q_{max}	β		%Δ	%Δ	CPU	%Δ	CPU
30	0.05	10	1.00	721.04	330.0	–	181	3.02	3
30	0.05	10	1.15	548.03	539.7	–	312	3.64	3
30	0.05	100	1.00	804.36	231.4	–	328	3.01	4
30	0.05	100	1.15	602.80	266.5	–	186	2.78	2
30	0.10	10	1.00	2586.06	723.8	–	3139	2.02	7
30	0.10	10	1.15	2318.93	961.2	13.24	TL	14.85	7
30	0.10	100	1.00	3207.94	277.7	13.05	TL	1.91	5
30	0.10	100	1.15	2886.68	378.1	13.96	TL	4.50	7
35	0.05	10	1.00	1586.08	221.9	9.72	TL	5.66	4
35	0.05	10	1.15	1394.39	237.0	26.89	TL	4.92	8
35	0.05	100	1.00	1725.72	123.9	–	2621	3.59	4
35	0.05	100	1.15	1561.41	179.1	18.72	TL	5.97	4
35	0.10	10	1.00	4270.00	961.4	64.58	TL	6.79	13
35	0.10	10	1.15	3698.28	1249.7	63.25	TL	10.54	16
35	0.10	100	1.00	5242.28	399.9	32.73	TL	3.18	10
35	0.10	100	1.15	4692.87	465.5	56.02	TL	5.77	14

Table 3
Comparison of the percentage gaps between the best known upper bound and the lower bound achieved by the compact multicommodity flow formulation and the extended formulation solved with column generation on the random instances of benchmark C within a time limit of one hour.

Instance				BK	CF_{root} (2)	CF (2)		EF_{root} (3)	
n	α	w_{max}	β		%Δ	%Δ	CPU	%Δ	CPU
30	0.05	1	1	39	963.0	–	399	6.68	3
30	0.05	1	1.15	36	1457.0	–	2100	12.79	10
30	0.05	10	1	265	1427.9	–	927	10.78	5
30	0.05	10	1.15	234	1996.0	–	1733	17.28	6
30	0.05	100	1	2487	1416.2	–	702	11.13	3
30	0.05	100	1.15	2187	1966.1	–	1136	17.08	6
30	0.10	1	1	179	2295.2	10.05	TL	8.01	12
30	0.10	1	1.15	145	3189.8	–	1274	21.44	15
30	0.10	10	1	993	2996.5	22.36	TL	10.23	10
30	0.10	10	1.15	798	4057.9	–	2579	24.42	24
30	0.10	100	1	9343	2992.8	23.21	TL	12.36	12
30	0.10	100	1.15	7368	3950.1	–	1577	22.22	13

the instance and mainly depends on the value of the other parameters: the hardest instances for benchmark B are those with $\alpha = 0.10, q_{max} = 10$ and $\beta = 1.15$; the hardest ones for benchmark C have $\alpha = 0.10$ and $\beta = 1.15$. For these reasons, in the following experiments we take into account only the lower bound provided by column generation.

5.4. Experiments on the heuristic algorithm

This section compares the results of heuristic *HAP-LDS*, described in Section 4, with those obtained by a local search metaheuristic, denoted as *VLNS-TS*, which combines the Tabu Search and the Very Large Neighbourhood Search approaches (Ceselli et al., in press). Both heuristics have been executed on the same machine described at the beginning of Section 5.

In order to solve the pricing subproblem, algorithm *HAP-LDS* uses the Tabu Search procedure *PrTS* with the same parameters reported in Table 1, except for the stopping criteria, which have been modified setting $I_{min} = I_{max} = 10,000$. In this case, in fact, we are mainly interested in finding a large number of columns for the final *ILP* model. For the other parameters of *HAP-LDS*, we have set $L_{max} = 2$ and $D_{max} = 4$. As discussed in Section 4, the aim of column

Table 4
Comparison of the percentage gaps between the best known upper bound and the lower bound achieved by the compact multicommodity flow formulation and the extended formulation solved with column generation on the instances of benchmark A extracted from the provinces of Milan and Monza within a time limit of one hour.

Instance	BK	CF_{root} (2)		CF (2)		EF_{root} (3)	
		%Δ	CPU	%Δ	CPU	%Δ	CPU
50-1	3864.21	38.12	0	–	117	14.73	76
50-2	5035.11	104.07	1	–	92	0.35	73
50-3	5847.16	151.34	1	–	409	3.73	53
50-4	5136.93	155.46	1	–	162	14.37	122
50-5	5460.25	104.40	0	–	77	4.70	68
60-1	4438.98	42.57	1	–	1894	9.76	71
60-2	6036.54	112.48	1	–	240	9.73	515
60-3	6696.81	138.95	1	–	900	10.54	95
60-4	5703.16	76.20	0	–	393	–	395
60-5	6121.01	80.03	1	–	327	4.68	220
70-1	4617.64	34.40	2	2.55	TL	2.94	66
70-2	6111.93	77.81	2	–	411	4.50	445
70-3	7091.37	105.09	1	–	3315	9.06	368
70-4	7058.42	94.16	1	3.80	TL	12.95	284
70-5	7100.00	76.27	2	7.54	TL	6.43	352
80-1	5434.92	60.69	4	20.30	TL	16.62	114
80-2	6816.78	68.41	4	–	1228	3.67	524
80-3	7830.83	94.00	2	15.90	TL	8.25	334
80-4	8111.31	85.10	5	10.46	TL	4.92	141
80-5	7660.55	77.00	4	16.39	TL	8.97	255

generation here is to quickly obtain heuristic solutions, and not to prove their optimality through the computation of a tight lower bound. Hence, we do not apply CPLEX to solve the pricing subproblems, but we terminate the generation process as soon as *PrTS* proves unable to identify negative reduced cost columns. For the same reason, we impose a time limit of 5000 seconds on the execution of *HAP-LDS*, because for a few large instances the resolution of the final *ILP* problem requires a high amount of computational time, most of which is spent in proving the optimality of a solution found much earlier.

Tables 5–7 report the results obtained by the two heuristics, respectively, on the random benchmarks B and C and on benchmark A. The first column of each table reports the number of vertices n. Column CPU reports the computational time in seconds required by *HAP-LDS*; only on one of the largest instances of benchmark A, the computation reached the time limit of 5000 seconds. For the sake of fairness, the same time was assigned, instance by instance, to algorithm *VLNS-TS*. The following two columns report the percentage gap between the result of *HAP-LDS* and the lower bound obtained by column generation, and the number of columns of the final *ILP* problem. The gap is computed as $(UB - LB)/LB$, where *UB* stands for the heuristic value obtained. The last two columns report the percentage gap achieved and the time required by *VLNS-TS* to reach its best solution. All the reported values are averaged over the instances of the same size.

The performance of *HAP-LDS* on the random benchmark B (see Table 5) is similar to that of *VLNS-TS*: the average gap obtained is slightly lower and the result is better in 22 cases out of 80 versus 15 better results obtained by the competing algorithm. In order to evaluate whether these differences are statistically significant, we have applied *Wilcoxon's matched-pairs signed-ranks test* (Wilcoxon, 1945), which estimates a probability $\leqslant 0.1065$ that such differences are due to random fluctuations. Hence, the two algorithms have a nearly equivalent performance, when given the same amount of time.

The random benchmark C (see Table 6) appears to be harder than B. In this benchmark the cost function does not mimic the distribution of the original real-world instances, as in the first one; in fact, the costs are generated with a more straightforward random process. The performance of *HAP-LDS* is still similar to that of

Table 5
Comparison of the results obtained by the *HAP-LDS* (percentage gap and number of columns generated) and by *VLNS-TS* (percentage gap and time to reach the best solution found) on benchmark B.

n	CPU	HAP-LDS		VLNS-TS	
		%Δ	C_{tot}	%Δ	CPU_{opt}
30	40.1	4.47	9424.63	4.47	0.50
35	65.1	5.80	12377.25	5.80	0.67
40	97.1	5.53	18161.00	5.41	8.54
45	115.9	3.62	18597.50	3.73	10.38
50	151.9	3.25	28464.00	3.35	26.49
55	188.7	3.79	29070.25	3.73	14.14
60	230.2	2.53	31410.25	2.54	21.28
65	254.1	2.74	28521.63	2.91	65.21
70	309.0	2.67	37594.25	2.82	65.56
75	474.0	2.90	43743.88	2.82	146.25
Avg.	192.6	3.73	25736.46	3.76	35.90

Table 6
Comparison of the results obtained by the *HAP-LDS* (percentage gap and number of columns generated) and by *VLNS-TS* (percentage gap and time to reach the best solution found) on benchmark C.

n	CPU	HAP-LDS		VLNS-TS	
		%Δ	C_{tot}	%Δ	CPU_{opt}
30	39.7	14.74	10132.75	14.27	0.25
40	120.9	11.20	23422.58	10.89	6.84
50	202.6	7.34	34826.75	6.88	32.72
60	430.4	5.87	44376.42	5.37	103.77
70	501.9	4.47	54987.33	4.40	66.31
Avg.	259.1	8.72	33549.17	8.36	41.98

Table 7
Comparison of the results obtained by the *HAP-LDS* (percentage gap and number of columns generated) and by *VLNS-TS* (percentage gap and time to reach the best solution found) on benchmark A.

n	CPU	HAP-LDS		VLNS-TS	
		%Δ	C_{tot}	%Δ	CPU_{opt}
50	172.0	7.59	60258.00	7.57	11.00
60	338.4	7.06	61498.60	6.94	8.20
70	564.0	8.05	120587.20	7.18	6.20
80	1534.0	9.49	181234.60	8.49	14.80
90	2190.0	5.51	214633.60	5.10	58.40
Avg.	959.7	7.54	127642.40	7.06	19.72

VLNS-TS, but the average gap obtained is larger and *HAP-LDS* obtains a better result in 5 cases out of 60 versus 17 better results obtained by *VLNS-TS*. Wilcoxon's test (Wilcoxon, 1945) estimates a probability $\leqslant 0.005$ that such differences are due to random fluctuations. Hence, the primal heuristic is slightly worse than *VLNS-TS*, when given the same amount of time. A similar relation can be observed on benchmark A, whose instances are extracted from the real ones. Wilcoxon's test (Wilcoxon, 1945) suggests that the performance of *VLNS-TS* is better than that of *HAP-LDS* with a probability $\leqslant 0.00003$ of random fluctuations.

An interesting remark on the percentage gap of both heuristics is that it almost consistently tends to decrease as the size of the instances increases. Since the trend affects both heuristics and since we do not know the optimal solution of most instances, this probably means that the quality of the lower bound provided by column generation improves as size increases, and is in general rather tight. There is also a clear dependence on other parameters: the gap tends to increase moving from benchmark B to C to A, when comparing instances of the same size. Focusing on benchmarks B and C, the instances with $\alpha = 0.10$ and $\beta = 1.15$, i.e. with a looser cost threshold and subsets of higher cardinality tend to have larger gaps.

Finally, it is worth noticing that in a majority of cases the result of the final *ILP* problem improves upon that of the diving phase (18 out of 25 for benchmark A, 64 out of 80 for benchmark B and 44 out of 60 for benchmark C). For two instances of benchmark B and one of C the diving phase is unable to find feasible solutions. In other words, the combination of columns obtained in different stages of the diversification mechanism, allowed by the final *ILP* problem, is a crucial step and significantly improves the results of *HAP-LDS*.

5.5. Results on the real-world instances

The real-world instance concerning the province of Monza proves easy to solve: the optimum can be found in 98 s by applying CPLEX to the multicommodity flow formulation. The province of Milan, on the contrary, is challenging: one hour of computation approximately corresponds to solving the continuous relaxation of the multicommodity flow formulation at the root node and provides a lower bound equal to 4609.81. By contrast, the column generation approach provides a lower bound equal to 9668.84 in 1918 seconds.

We then applied *HAP-LDS* imposing a time limit of 5000 seconds on the diving phase. During this phase the algorithm found the best known solution having a cost equal to 9785.58, with a 1.21% gap. The final *ILP* problem was solved to optimality in about 300 seconds, but without improving the best known solution. By contrast, Algorithm *VLNS-TS* found in the same time a solution with a cost equal to 9968.19, with a 3.10% gap. This solution is found after 1043 seconds, but never improved in the following.

6. Conclusions

This paper considers a graph partitioning problem which models the partition of an organization into administrative areas. It proposes a column generation approach which obtains tight lower bounds (within a few percent units from the best known results) also for realistic size instances (from 70 to 90 vertices), as opposed to a multicommodity flow formulation which solves instances up to 30 vertices for random instances and 70 vertices for realistic ones, but yields large gaps as soon as the number of vertices exceeds these limits. We also propose a primal heuristic, based on the column generation approach, which combines the generation of promising columns based on their reduced cost, a heuristic limited discrepancy search mechanism and the solution of a final set partitioning problem on the whole set of columns generated. This approach, though its computational burden is intrinsically heavy, proves competitive with a refined local search metaheuristic drawn from our previous research on the topic. On random instances which mimic the structure of realistic problems, it achieves slightly better heuristic solutions, whereas on more general random instances and on instances drawn from the real ones its performance is slightly worse. When applied to the real-world instance of Milan (133 vertices), the gap between the result of the primal heuristic and the bound provided by column generation is as small as 1.21%.

References

Arora, S., Rao, S., & Vazirani, U. (2008). Geometry, flows, and graph-partitioning algorithms. *Communications of the ACM, 51*(10), 96–105. http://dx.doi.org/10.1145/1400181.1400204. ISSN 0001-078.

Cacchiani, V., Hemmelmayr, V. C., & Tricoire, F. (2014). A set-covering based heuristic algorithm for the periodic vehicle routing problem. *Discrete Applied Mathematics, 163*(1), 53–64. http://dx.doi.org/10.1016/j.dam.2012.08.032. ISSN 0166-218X.

Ceselli, A., Colombo, F., Cordone, R., Trubian, M. (in press), Employee workload balancing by graph partitioning. Discrete Applied Mathematics doi: 10.1016/j.dam.2013.02.014.

Desrosiers, J., & Lübbecke, M. E. (2005). A primer in column generation. In G. Desaulniers, J. Desrosiers, & M. M. Solomon (Eds.), *Column generation* (pp. 1–32). Springer.

Donath, W. E., & Hoffman, A. J. (1973). Lower bounds for the partitioning of graphs. *IBM Journal of Research and Development, 17*(5), 420–425.

Fan, N., & Pardalos, P. (2010). Linear and quadratic programming approaches for the general graph partitioning problem. *Journal of Global Optimization, 48*, 57–71. ISSN 0925-5001.

Ferreira, C. E., Martin, A., de Souza, C. C., Weismantel, R., & Wolsey, L. A. (1998). The node capacitated graph partitioning problem: A computational study. *Mathematical Programming, 81*, 229–256. ISSN 0025-5610.

Fjällström, P.-O. (1998). Algorithms for graph partitioning: A survey. *Linköping Electronic Articles in Computer and Information Science, 10*.

Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research, 13*, 533–549.

Glover, F., & Laguna, M. (1997). *Tabu search*. Kluwer Academic Publishers.

Guttmann-Beck, N., & Hassin, R. (2000). Approximation algorithms for minimum *k*-cut. *Algorithmica, 27*(2), 198–207.

Ideker, T., Ozier, O., Schwikowski, B., & Siegel, A. F. (2002). Discovering regulatory and signalling circuits in molecular interaction networks. *Bioinformatics, 18*(suppl 1), S233–S240. http://dx.doi.org/10.1093/bioinformatics/18.suppl_1.S233.

Joncour, C., Michel, S., Sadykov, R., Sverdlov, D., & Vanderbeck, F. (2010). Column generation based primal heuristics. In M. Haouari & A. R. Mahjoub (Eds.), *Proceedings of International Symposium on Combinatorial Optimization (ISCO 2010). Electronic notes in discrete mathematics* (Vol. 36, pp. 695–702). Tunisia: Hammamet.

Kim, J., Hwang, I., Kim, Y.-H., & Moon, B.-R. (2011). Genetic approaches for graph partitioning a survey. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation. GECCO '11* (pp. 473–480). New York: ACM. http://dx.doi.org/10.1145/2001576.2001642. ISBN 978-1-4503-0557-0.

Matula, D. W., & Shahrokhi, F. (1990). Sparsest cuts and bottlenecks in graphs. *Discrete Applied Mathematics, 27*(1-2), 113–123. http://dx.doi.org/10.1016/0166-218X(90)90133-W. ISSN 0166-218X.

Osipov, V., Sanders, P., & Schulz, C. (2012). Engineering graph partitioning algorithms. In R. Klasing (Ed.), *Experimental algorithms. Lecture notes in computer science* (Vol. 7276, pp. 18–26). Berlin/Heidelberg: Springer. ISBN 978-3-642-30849-9.

Pisinger, D., & Ropke, S. (2010). Large neighborhood search. In M. Gendreau & J.-Y. Potvin (Eds.), *Handbook of metaheuristics. International series in operations research & management science* (Vol. 146, pp. 399–419). US: Springer. http://dx.doi.org/10.1007/978-1-4419-1665-5_13. ISBN 978-1-4419-1663-1.

Prescott-Gagnon, E., Desaulniers, G., & Rousseau, L.-M. (2009). A branch-and-price-based large neighborhood search algorithm for the vehicle routing problem with time windows. *Networks, 54*(4), 190–204.

Sanders, P., & Schulz, C. (2011). Engineering multilevel graph partitioning algorithms. In C. Demetrescu & M. Halldórsson (Eds.), *Algorithms – ESA 2011. Lecture notes in computer science* (Vol. 6942, pp. 469–480). Berlin/Heidelberg: Springer. ISBN 978-3-642-23718-8.

Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics, 1*, 80–83.