Constraints
DOI 10.1007/s10601-012-9124-0

LS(Graph): a constraint-based local search for constraint optimization on trees and paths

Quang Dung Pham · Yves Deville · Pascal Van Hentenryck

Abstract Constrained optimum tree (COT) and constrained optimum path (COP) problems arise in many real-life applications and are ubiquitous in communication networks. They have been traditionally approached by dedicated algorithms, which are often hard to extend with side constraints and to apply widely. This paper proposes a constraint-based local search framework for COT/COP applications, bringing the compositionality, reuse, and extensibility at the core of constraint-based local search and constraint programming systems. The modeling contribution is the ability to express compositional models for various COT/COP applications at a high level of abstraction, while cleanly separating the model and the search procedure. The main technical contribution is a connected neighborhood based on rooted spanning trees to find high-quality solutions to COP problems. This framework is applied to some COT/COP problems, e.g., the quorumcast routing problem, the edge-disjoint paths problem, and the routing and wavelength assignment with delay side constraints problem. Computational results show the potential importance of the approach.

Keywords Combinatorial optimization · Constraint-based local search · Graphs · Constrained optimum trees · Constrained optimum paths · Quorumcast routing · Edge-disjoint paths · Routing and wavelength assignment with delay constraints

Q. D. Pham (✉)
School of Information and Communication Technology,
Hanoi University of Science and Technology, Hanoi, Vietnam
e-mail: dungpq@soict.hut.edu.vn

Y. Deville
Université catholique de Louvain, 1348 Louvain-la-Neuve, Belgium
e-mail: yves.deville@uclouvain.be

P. Van Hentenryck
Optimization Research Group, NICTA, Victoria Research Laboratory,
Electrical and Electronic Engineering, The University of Melbourne,
Melbourne, VIC 3010, Australia
e-mail: pvh@nicta.com.au

Published online: 12 July 2012

1 Introduction

Constrained optimum tree (COT) and constrained optimum path (COP) problems appear in various real-life applications such as telecommunication and transportation networks. These problems consist of finding one or more trees (or paths) on a given graph satisfying some given constraints while minimizing or maximizing an objective function. Some COT problems have been considered and solved in the literature, e.g., Degree Constrained Minimum Spanning Tree (DCMST) [7, 45], Bounded Diameter Minimum Spanning Tree (BDMST) [35], Capacitated Minimum Spanning Tree problem (CMST) [3, 56], Minimum Diameter Spanning Tree (MDST) [50], Edge-Weighted *k*-Cardinality Tree (KCT), [20, 25], Steiner Minimal Tree (SMT) [28, 66], Optimum Communication Spanning Tree problems (OCST) [32], etc. We also see many COP problems which have been studied and solved in the literature. For instance, in telecommunication networks, routing problems supporting multiple services involve the computation of paths minimizing transmission costs while satisfying bandwidth and delay constraints [15, 27, 30]. Similarly, the problem of establishing routes for connection requests between network nodes is one of the basic operations in communication networks and it is typically required that no two routes interfere with each other due to quality-of-service and survivability requirements. This problem can be modeled as an edge-disjoint paths problem [18]. Most of these COT/COP problems are NP-hard. They are often approached by dedicated algorithms including exact methods, such as the Lagrangian-based heuristic [7], the ILP-based algorithm using directed cuts [25], the Lagrangian-based branch and bound in [15], and the vertex labeling algorithm from [30]; there are also meta-heuristic algorithms such as a hybrid evolutionary algorithm [19], ant colony optimization [21], and local search [20]. These techniques exploit the structure of the constraints and the objective functions but are often difficult to extend or reuse.

This paper[1] proposes a constraint-based local search (CBLS) [62] framework for COT/COP applications to support the compositionality, reuse, and extensibility at the core of CBLS and CP systems. It follows the trend of defining domain-specific CBLS frameworks, capturing modeling abstractions and neighborhoods for classes of applications exhibiting significant structures. As is traditional for CBLS, the resulting `LS(Graph)` framework allows the model to be compositional and easy to extend, and provides a clean separation of concerns between the model and the search procedure. Moreover, the framework captures structural moves that are fundamental in obtaining high-quality solutions for COT/COP applications. The key technical contribution underlying this COP framework is a novel connected neighborhood for COP problems based on rooted spanning trees. More precisely, this COP framework incrementally maintains, for each desired elementary path, a rooted spanning tree that specifies the current path and provides an efficient data structure to obtain its neighboring paths and their evaluations.

The availability of high-level abstractions (the "what") and the underlying connected neighborhood for elementary paths (the "how") make the `LS(Graph)` framework particularly appealing for modeling and solving complex COP applications.

[1] This paper is an extended version of [54] and is based on the PhD thesis [53].

The LS(Graph) framework, implemented in COMET, was evaluated experimentally on two classes of applications: COT with the quorumcast routing (QR) problem and COP with the edge-disjoint path (EDP) problems and the routing and wavelength assignment problem with side constraints (RWA-D). In [37], we present another application in the domain of traffic engineering in switched ethernet networks. The experimental results show the potential of the approach.

1.1 Case studies

We first describe three problems that will be modeled and solved by the LS(Graph) framework.

1.1.1 The quorumcast routing (QR) problem

The quorumcast routing (QR) problem arises in distributed applications [24, 29, 48, 63]. Given a weighted undirected graph $G = (V, E)$, to each edge $e \in E$ there is associated a cost $w(e)$. Given a source node $r \in V$, an integral value q, and a set $S \subseteq V$ of *multicast* nodes, the quorumcast routing problem consists in finding a minimum cost tree $T = (V', E')$ of G spanning r and q nodes of S. $T = (V', E')$ is a graph satisfying the following properties:

1. $V' \subseteq V \wedge E' \subseteq E$.
2. T is connected.
3. $\exists Q \subseteq S$ such that $\sharp Q = q \wedge Q \cup \{r\} \subseteq V'$.
4. The cost of

$$T = \sum_{e \in E'} w(e)$$

is minimal over all subgraphs of G with properties 1–3.

An exact algorithm [48] has also been proposed for solving the QR problem but experiments were performed on small graphs (e.g., graph with 30 nodes). Three heuristics have been proposed in [24] including Minimal Cost Path Heuristic (MPH), Improved Minimum Path Heuristic (IMP), and Modified Average Distance Heuristic (MAD). Experimental results in that paper show that, among these heuristics, the IMP heuristic produces the best solutions. In [29], a multispace search heuristic has been proposed for solving this problem which gives better results than the IMP and the MAD heuristics on 12-node networks and 100-node networks.

In [63], the authors considered the QR problem with additional constraints imposed on the total cumulative delay along the path from s to any destination node of Q, and proposed a distributed heuristic algorithm for solving it. Experiments were conducted on graphs of up to 200 nodes.

In Section 6.1, we propose a simple model in LS(Graph) for this problem using a tabu search. This example illustrates the expressive power of LS(Graph) where a simple but efficient model can be designed in a few lines. Experimental results show that our LS(Graph) model gives better results than the standard IMP heuristic.

1.1.2 The edge-disjoint paths (EDP) problem

We are given an undirected graph $G = (V, E)$ and a set $T = \{\langle s_i, t_i\rangle \mid i = 1, 2, ..., \sharp T; s_i \neq t_i \in V\}$ representing a list of commodities. A subset $T' \subseteq T$, $T' = \{\langle s_{i_1}, t_{i_1}\rangle, ..., \langle s_{i_k}, t_{i_k}\rangle\}$ is called *edp*-feasible if there exist mutually edge-disjoint paths from s_{i_j} to t_{i_j} on G, $\forall j = 1, 2, .., k$. The EDP problem consists in finding a *edp*-feasible subset of T with maximal cardinality. In other words,

$$\begin{array}{lll} max & \sharp T' & (1) \\ s.t. & T' \subseteq T & (2) \\ & T' \text{ is } edp\text{-feasible} & (3) \end{array}$$

This problem appears in many applications such as real-time communication, VLSI-design, routing, and admission control in modern networks [8, 23]. The existing techniques for solving this problem include approximation algorithms [13, 22, 42, 43], greedy approaches [42, 44], and an ant colony optimization (ACO) metaheuristic [18]. It has been shown in [18] that ACO is the start-of-the-art algorithm for this problem. In that paper, the ACO algorithm were compared with a simple greedy algorithm in [42](the multi-start version).

In Section 6.2, we propose two heuristic algorithms applying `LS(Graph)`. We experimentally show competitive results compared with the ACO algorithm in [18]. This example illustrates how `LS(Graph)` can be used to implement more complex heuristics.

1.1.3 The routing and wavelength assignment problem with a delay side constraint (RWA-D)

Wavelength division multiplexing (WDM) optical networks [49] provide high bandwidth communications. The routing and wavelength assignment (RWA) problem is an essential problem on WDM optical networks. The RWA problem can be described as follows. Given a set of requests for all-optical connections, the RWA problem consists of finding routes from the source nodes to their respective destination nodes and assigning wavelengths to these routes. A condition that must be satisfied is that two routes sharing common edges must be assigned different wavelengths. Normally, the number of available wavelengths is limited and the number of requests is high. Two variants of this problem have been studied extensively in the literature: the *minRWA* problem aims at minimizing the number of wavelength used for satisfying all requests, and the *maxRWA* aims at maximizing the number of requests with a given number of wavelengths. Both variants are NP-Hard [26].

In the literature, there have been different techniques proposed for solving these problems, e.g.: exact methods based on the ILP formulation [23, 40, 46, 47, 52, 55, 61, 65]; heuristic algorithms [11, 12, 31, 67]; and metaheuristics, including tabu search [39, 51] and Genetic [4, 10, 38]. These techniques have been tried on realistic networks of small size (networks up to 27 nodes and 70 edges) but involving a large number of connection requests. RWA with additional constraints has also been considered, e.g., in [5, 64].

In order to show the interest of the modeling framework, we consider the *minRWA* problem with a side constraint (e.g., a delay constraint) specifying that the cost of each route must be less than or equal to a given value. The point here is not to study a model competitive in comparison with state-of-the-art techniques for classical RWA problems. Rather, we show the flexibility of this modeling framework, one which enables a combination of `VarGraph` of `LS(Graph)` with `var{int}` of `COMET`.

The formal definition of the problem (called RWA-D) is the following. Given an undirected weighted graph $G = (V, E)$, each edge e of G has cost $c(e)$ (e.g., the delay in traversing e). We suppose given a set of connection requests $R = \{\langle s_1, t_1\rangle, \langle s_2, t_2\rangle, ..., \langle s_k, t_k\rangle\}$ and a value D. The RWA-D problem consists of finding routes p_i from s_i to t_i and their wavelengths for all $i = 1, 2, ..., k$ such that:

1. the wavelengths of p_i and p_j are different if they have common edges, $\forall i \neq j \in \{1, 2, ..., k\}$ (wavelength constraint),
2. $\sum_{e \in p_i} c(e) \leq D, \forall i = 1, 2, ..., k$ (delay constraint)
3. the number of different wavelengths is minimized (objective function).

In Section 6.3, a local search algorithm and its implementation in `LS(Graph)` will be proposed for solving the RWA-D problem.

1.2 Contribution

The contributions of this paper are the following:

1. We design and implement a constraint-based local search (CBLS) [62] framework, called `LS(Graph)`, for COT/COP applications. It supports the compositionality, reuse, and extensibility at the core of CBLS and CP systems. The proposed framework can be used as either a black box or a glass box. The black box is exploited in the sense that users only need to state the model in a declarative way, with variables, constraints, and an objective function to be optimized. Built-in search components (e.g., tabu search) are then performed automatically. The glass box allows users to extend the framework by designing and implementing their own components (e.g., invariants, constraints, objective functions, and search heuristics) and integrating them with the system.
2. The `LS(Graph)` combines graph variables (i.e., `VarTree, VarPath` for modeling trees and paths in a high-level way) with standard `var{int}` of `COMET`, which enables the modeling of various COT/COP applications on graphs for which both the topology and scalar values must be determined.
3. A key technical contribution of the paper is a novel connected neighborhood for COP problems based on rooted spanning trees. More precisely, the COP framework incrementally maintains, for each desired elementary path, a rooted spanning tree that specifies the current path and provides an efficient data structure to obtain its neighboring paths and their evaluations.
4. We propose incremental algorithms for implementing some fundamental abstractions of the framework. We show that the incrementality does not improve the theoretical complexity but is efficient in practice.
5. We apply the constructed framework to a COT problems: the quorumcast routing problem and two COP problems: the edge-disjoint paths problem and the routing and wavelength assignment problem with delay side constraints on optical networks. Experimental results show the potential significance of our approach from both the programming and the computation stand points. For

the first two problems, we show competitive results in comparison with existing techniques and for the third problem, we show how to solve complex problems flexibly and easily.

The `LS(Graph)` framework is open source. The `COMET` code of `LS(Graph)` and applications as well as instances experimented in this paper are available at http://becool.info.ucl.ac.be/lsgraph.

1.3 Outline

The rest of this paper is organized as follows. Section 2 gives the basic definitions and notations. Section 3 specifies neighborhoods for COT applications and proposes our novel neighborhoods for COP applications. Section 4 gives an overview of data structures and algorithms for implementing two fundamental and non-trivial abstractions of the framework. The implementation of the framework in `COMET` programming language will be introduced in Section 5. Sections 6 presents the application of the framework to the resolution of the QR, EDP and RWA-D problems. Finally, Section 7 concludes the paper and gives some future work.

2 Definitions and notations

Graphs Given an undirected graph g, we denote the set of nodes and the set of edges of g by $V(g)$, $E(g)$ respectively. The degree of a node v (denoted $\deg_g(v)$) is the number of incident edges to this edge: $\deg_g(v) = \sharp\{u \mid (v, u) \in V(g)\}$.

A graph sg is called subgraph of a graph g if $V(sg) \subseteq V(g)$ and $E(sg) \subseteq E(g)$ and we denote $sg \subseteq g$.

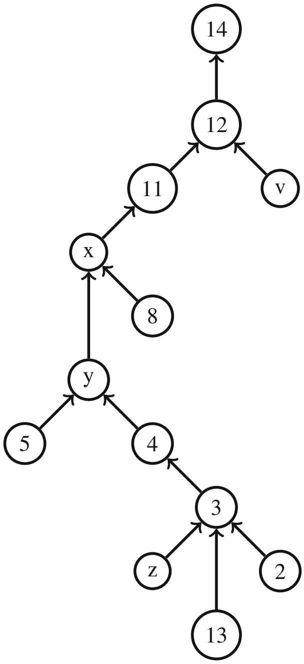
A path on g is a sequence of nodes $\langle v_1, v_2, ..., v_k\rangle$ $(k > 1)$ in which $v_i \in V(g)$ and $(v_i, v_{i+1}) \in E(g), \forall i = 1, \ldots, k-1$. The nodes v_1 and v_k are the origin and the destination of the path. A path is called *simple* if there is no repeated edge and *elementary* if there is no repeated node. A cycle is a path in which the origin and the destination are the same. This paper only considers elementary paths and hence we use "path" and "elementary path" interchangeably if there is no ambiguity. A graph is connected if and only if there exists a path from u to v for all $u, v \in V(g)$.

Given two paths $px = \langle x_1, x_2, ..., x_k\rangle$ and $py = \langle y_1, y_2, ..., y_q\rangle$, we denote $px + py$ the concatenation of these two paths: $px + py = \langle x_1, x_2, ..., x_k, y_1, y_2, ...y_q\rangle$ if $x_k \neq y_1$ and $px + py = \langle x_1, x_2, ..., x_k = y_1, y_2, ..., y_q\rangle$ if $x_k = y_1$.

Given paths p, p_1, p_2, and q,

- $V(p)$ is the set of nodes of p
- $p_1 \cup p_2$ $(p_1 \cap p_2)$ is the set $V(p_1) \cup V(p_2)$ $(V(p_1) \cap V(p_2))$.
- $x \in P$ is the predicate $x \in V(p)$.
- $s(p), t(p)$ are, respectively, the starting and terminating nodes of p.
- $p(u, v)$ is the subpath of p starting from u and terminating at v ($u, v \in p$ and u is not located after v on p).
- $sp_p(x), tp_p(x)$ is the subpath of p from $s(p)$ to x and from x to $t(p)$.
- $repl(p, q) = sp_p(s(q)) + q + tp_p(t(q))$ with $s(q), t(q) \in p$. Intuitively, $repl(p, q)$ is the path generated by replacing the subpath of p from $s(q)$ to $t(q)$ by q.

Fig. 1 Illustrating Property 1



Trees A tree is an undirected connected graph containing no cycles. A spanning tree tr of an undirected connected graph g is a tree spanning all the nodes of g: $V(tr) = V(g)$ and $E(tr) \subseteq E(g)$. A tree tr is called a rooted tree at r if the node r has been designated the root. Each edge of tr is implicitly oriented towards the root. If the edge (u, v) is oriented from u to v, we call v the father of u in tr, which is denoted by $fa_{tr}(u)$. Given a rooted tree tr and a node $s \in V(tr)$,

- $root(tr)$ denotes the root of tr,
- $path_{tr}(v)$ denotes the path from v to $root(tr)$ on tr. For each node u of $path_{tr}(v)$, we say that u *dominates* v in tr (alternatively, u is a *dominator* of v, v is a *descendant* of u) which we denote by $u\ Dom_{tr}\ v$. If u does not dominates v on tr, we write $u\ \overline{Dom_{tr}}\ v$.
- $path_{tr}(u, v)$ denotes the path from u to v in tr $(u, v \in V(tr))$.
- $nca_{tr}(u, v)$ denotes the nearest common ancestor of two nodes u and v. In other words, $nca_{tr}(u, v)$ is the common dominator of u and v such that there is no other common dominator of u and v that is a descendant of $nca_{tr}(u, v)$.
- Given a node $v \in V(tr)$, we denote by $T_{tr}(v)$ the subtree of tr rooted at v. If $v \neq root(tr)$, we denote by $\overline{T_{tr}}(v)$ the subtree of tr generated by removing $T_{tr}(v)$ and the edge $(v, fa_{tr}(v))$ from tr: $V(\overline{T_{tr}}(v)) = V(tr) \setminus V(T_{tr}(v))$ and $E(\overline{T_{tr}}(v)) = E(tr) \setminus (E(T_{tr}(v)) \cup \{(v, fa_{tr}(v))\})$.

Property 1 Suppose given a rooted tree tr.

1. Suppose given a node $x \in V(tr)$. We have $x\ Dom_{tr}\ y, \forall y \in V(T_{tr}(x))$. In other words, a vertex x of a rooted tree tr dominates all vertices of the subtree of tr rooted at x.

2. Suppose given two nodes $x, y \in V(tr)$ such that $x = fa_{tr}(y)$ and two nodes z, v such that $z \in V(T_{tr}(y))$, $v \in V(\overline{T_{tr}(y)})$. We have $nca_{tr}(v, z) = nca_{tr}(v, x)$. This property is illustrated in Fig. 1: $nca_{tr}(v, z) = nca_{tr}(v, x) = 12$.

3 Neighborhoods

This section defines neighborhoods for COT and COP problems. The neighborhood for COT applications is based on traditional modification actions on dynamic trees (i.e., trees which can be modified): add, remove, and replace over edges. Our main technical contribution for COP applications is to propose a neighborhood structure based on spanning trees. We first present neighborhoods for COT applications.

3.1 COT neighborhood

A neighborhood of a tree is a set of trees generated by performing modification actions on the given tree. Given an undirected graph g and a dynamic tree tr of g (tr can be modified such that $tr \subseteq g$), we specify a set of basic modifications conserving the tree property. We consider in this framework the following basic modifications.

1. **add edge action** An edge $e = (u, v) \in E(g) \setminus E(tr)$ can be added to tr if tr is empty, or if there is exactly one node u or v in the tree tr: $u \in V(tr)$ `XOR` $v \in V(tr)$. This edge is called an *insertable* edge. The insertion of this edge implicitly adds its endpoints to tr if they do not exist in tr. The set of insertable edges of tr is denoted by $Inst(tr)$ and this insertion action is denoted by $addEdge(tr, e)$. We also use $addEdge(tr, e)$ to denote the resulting tree. The first basic neighborhood is the following:

$$NT_1(tr) = \{addEdge(tr, e) \mid e \in Inst(tr)\}$$

2. **remove edge action** An edge $e = (u, v) \in E(tr)$ can be removed from tr if one node u or v is a leaf of tr: $deg_{tr}(u) = 1 \vee deg_{tr}(v) = 1$. This edge is called a *removable* edge. The removal of this edge thus also removes its endpoints if they are the leaves of tr. The set of removable edges of tr is denoted by $Remv(tr)$ and this removal action is denoted by $removeEdge(tr, e)$. We also use $removeEdge(tr, e)$ to denote the resulting tree. The second basic neighborhood is defined as follows:

$$NT_2(tr) = \{removeEdge(tr, e) \mid e \in Remv(tr)\}$$

3. **replace cycle edge action** [2] An edge e' of tr can be replaced by another edge $e = (u, v) \in E(g) \setminus E(tr)$ with $u, v \in V(tr)$ conserving the tree property in the following case: the insertion of e creates a fundamental cycle containing e' and the removal of e' removes the cycle and restores the tree property. The edge e is called a *replacing* edge, and e' is called a *replaceable* edge of e. The set of nodes of tr is unchanged by this replacement. We denote by $Repl(tr)$ the set of replacing edges of tr and $Repl(tr, e)$ the set of replaceable edges of the replacing edge e. We use $replaceEdge(tr, e', e)$ to denote both the replacement action and the resulting tree. The third basic neighborhood is defined as follows:

$$NT_3(tr) = \{replaceEdge(tr, e', e) \mid e \in Repl(tr) \wedge e' \in Repl(tr, e)\}$$

In practice, we can combine the above basic moves to perform more complex moves. For instance, we take $addEdge(tr, e_1)$ and $removeEdge(tr, e_2)$ at hand where $e_1 \in Remov(tr)$ and $e_2 \in Inst(tr)$ and e_1 and e_2 do not have common endpoint that is the leaf tr.[2] The set of such pairs of $\langle e_1, e_2 \rangle$ is denoted by $RemvInst(tr)$. This kind of neighborhood has been considered in the tabu search algorithm of [20]. The formal definition of this neighborhood is

$$NT_{1+2}(tr) = \{addEdge(removeEdge(tr, e_2), e_1) \mid \langle e_1, e_2 \rangle \in RemvInst(tr)\}$$

In the following section, we introduce a novel neighborhood for COP applications.

3.2 COP neighborhood

We consider in this paper only elementary paths, i.e., paths having no repeated vertices. These are those which appear in most COP applications. Our constructed framework also supports the modeling of paths where vertices or edges can be repeated, but this will not be presented here (see more details in [53]).

For COP problems, a neighborhood of a path defines a set of paths that can be reached from the current path. The most general neighborhood of a path p on a given graph g is defined as the set of paths generated by replacing a subpath of the current path by another path on the given graph conserving the path property: $\mathcal{N}(p) = \{repl(p, q) \mid q \in \mathfrak{R}(p)\}$ in which $\mathfrak{R}(p)$ is the set of paths q satisfying followings conditions:

(1) $q \in g$
(2) $s(q), t(q) \in p$
(3) $sp_p(s(q)) \cap q = \{s(q)\}$
(4) $tp_p(t(q)) \cap q = \{t(q)\}$

Conditions (3) and (4) ensure the path property of all elements of $\mathcal{N}(p)$ (no repeated vertices are allowed in a path except starting and terminating vertices).[3]

Unfortunately, such a neighborhood is too large and does not allow being explored in a generic way. To overcome this difficulty, in this section, we propose a restricted neighborhood based on rooted spanning trees. This notion can be widely applied and allows users to perform efficient neighborhood explorations.

Related work As far as we know, there exist only a few local search approaches for COP applications on general graphs. Moreover, these local search algorithms do not explicitly describe neighborhood structures. Rather, the authors talk about the moves, which are very specific and sophisticated. Such moves do not enable the compositionality, modularity, and reuse of the local search programs.

On complete graphs, some local search algorithms have been applied for solving the traveling salesman problem [41] or the vehicle routing problem [9, 34]. In these approaches, a path is explicitly represented by a sequence of vertices and the neighborhood consists of paths generated by changing some vertices of this sequence (e.g., by removing, inserting, exchanging, or changing the position of some vertices). These

[2] This condition ensures the preservation of the tree property under the modification action.

[3] By some authors, walks with no repeated vertices are referred to as elementary paths.

neighborhood structures cannot be applied to general graphs because a sequence of vertices can not be guaranteed to always form a path on the given graph.

To obtain a reasonable efficiency, a local search algorithm must maintain incremental data structures that allow a fast exploration of this neighborhood and a fast evaluation of the impact of the moves (differentiation). The key novel contribution of our COP framework is to use a rooted spanning tree to represent the current solution and its neighborhood. It is based on the observation that, given a spanning tree *tr* whose root is *t*, the path from a given node *s* to *t* in *tr* is unique. Moreover, the spanning tree implicitly specifies a set of paths that can be reached from the induced path and provides a data structure for evaluating their desirability. The rest of this section describes the neighborhood in detail. Our COP framework considers both directed and undirected graphs, but, to simplify the presentation, only undirected graphs are treated.

3.2.1 Rooted spanning trees

Given an undirected graph g and a target node $t \in V(g)$, our COP neighborhood maintains a spanning tree of g rooted at t. Moreover, since we are interested in elementary paths between a source s and a target t, the data structure also maintains the source node s and is called a rooted spanning tree (RST) over (g, s, t). An RST tr over (g, s, t) specifies a unique path from s to t in g: $path_{tr}(s) = \langle v_1, v_2, ..., v_k \rangle$ in which $s = v_1, t = v_k$ and $v_{i+1} = fa_{tr}(v_i)$, $\forall i = 1, \ldots, k-1$. By maintaining RSTs for COP problems, our framework avoids an explicit representation of the paths and enables the definition of a connected neighborhood that can be explored efficiently. Indeed, the tree structure directly captures the path structure from a node s to the root; simple updates to the RST (e.g., an edge replacement) will induce a new path from s to the root. In this framework, we also consider COP applications in which the sources and the destinations of the paths are not fixed. Hence, the source s and the destination (or root) of the RST (g, s, t) can also be changed (but this will not be presented in this paper, interested readers can refer to the PhD thesis [53]).

Given an RST tr over (g, s, t), we denote by $path(tr)$ the path $path_{tr}(s)$ which is the path induced by tr from s to the root t of tr. Given an undirected graph g and a path p on g, we denote by *RSTInduce(g,p)* the set of RSTs of g, rooted at $t(p)$, which induce p.

We define in the following section the neighborhood structure based on edge replacements. In COP applications, generally, a candidate solution is a set of paths. Each path has its own neighborhood. A neighborhood of a candidate solution is the set of candidate solutions generated by changing some paths of the current candidate solution with their neighbors. Hence, we present only neighborhoods of one path.

3.2.2 The edge-replacement based neighborhood

We first show in this section how to update an RST tr over (g, s, t) based on edge replacements to generate a new rooted spanning tree tr' over (g, s, t) which induces a new path from s to t in g: $path_{tr'}(s) \neq path_{tr}(s)$.

Let tr be an RST over (g, s, t), we consider the third basic neighborhood of tr (see Section 3.1):

$$NT_3(tr) = \{replaceEdge(tr, e', e) \mid e \in Repl(tr) \wedge e' \in Repl(tr, e)\}$$

which is the set of RST of (g, s, t). It is easy to observe that two RSTs tr_1 and tr_2 over (g, s, t) may induce the same path from s to t. For this reason, we now show how to compute a subset $ERNP_1(tr) \subseteq NT_3(tr)$ such that $path_{tr'}(s) \neq path_{tr}(s), \forall tr' \in ERNP_1(tr)$.

We first fix some notations to be used in the following presentation. Given an RST tr over (g, s, t) and a replacing edge $e = (u, v)$, the nearest common ancestors of s and the two endpoints u, v of e are both located on the path from s to t. We denote by $lownca_{tr}(e, s)$ and $upnca_{tr}(e, s)$ the nearest common ancestors of s on the one hand and one of the two endpoints of e on the other hand, with the condition that $upnca_{tr}(e, s)$ dominates $lownca_{tr}(e, s)$. We denote by $low_{tr}(e, s)$, $up_{tr}(e, s)$ the endpoints of e such that $nca_{tr}(s, low_{tr}(e, s)) = lownca_{tr}(e, s)$ and $nca_{tr}(s, up_{tr}(e, s)) = upnca_{tr}(e, s)$. Figure 2 illustrates these concepts. The left part of the figure depicts the graph g and the right side depicts an RST tr over (g, s, r). Edge (8,10) is a replacing edge of tr; $nca_{tr}(s, 10) = 12$ since 12 is the common ancestor of s and 10. $nca_{tr}(s, 8) = 7$ since 7 is the common ancestor of s and 8. $lownca_{tr}((8, 10), s) = 7$ and $upnca_{tr}((8, 10), s) = 12$ because 12 Dom_{tr} 7; $low_{tr}((8, 10), s) = 8$; $up_{tr}((8, 10), s) = 10$.

We now specify the replacements that induce a new path from s to t.

Proposition 1 *Let tr be an RST over (g, s, t), $e = (u, v)$ be a replacing edge of tr, let e' be a replaceable edge of e, and let $tr' = rep(tr, e', e)$. Let $su = upnca_{tr}(e, s)$ and $sv = lownca_{tr}(e, s)$. We have that $path_{tr'}(s) \neq path_{tr}(s)$ if and only if*

(1) $su \neq sv$ *and*
(2) $e' \in path_{tr}(sv, su)$

A replacing edge e of tr satisfying the condition (1) is called a *preferred replacing* edge and a replaceable edge e' of e in tr satisfying condition (2) is called a *preferred*

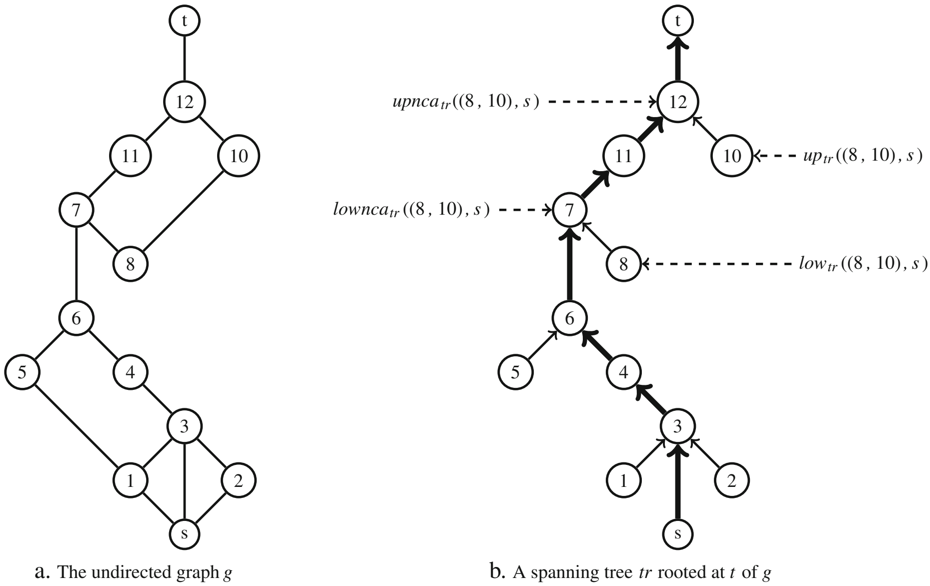


Fig. 2 An example of rooted spanning tree

replaceable edge of e. We denote by $prefRepl(tr)$ the set of preferred replacing edges of tr and by $prefRepl(tr, e)$ the set of preferred replaceable edges of the preferred replacing edge e on tr. We also denote by $rep(tr, e', e)$ the action and the resulting RST of replacing a preferred replaceable edge e' by a preferred replacing edge e on the RST tr. The edge-replacement based neighborhood (called ER-neighborhood) of an RST tr is defined by

$$ERNP_1(tr) = \{tr' = rep(tr, e', e) \mid e \in prefRepl(tr), e' \in prefRepl(tr, e)\}.$$

The action $rep(tr, e', e)$ is called an ER-move and is illustrated in Fig. 3. In the current tree tr (see Fig. 3a), the edge (8,10) is a preferred replacing edge, $nca_{tr}(s, 8) = 7$, $nca_{tr}(s, 10) = 12$, $lownca_{tr}((8, 10), s) = 7$, $upnca_{tr}((8, 10), s) = 12$, $low_{tr}((8, 10), s) = 8$ and $up_{tr}((8, 10), s) = 10$. The edges (7,11) and (11,12) are preferred replaceable edges of (8,10) because these edges belong to $path_{tr}(7, 12)$. The path induced by tr is $\langle s, 3, 4, 6, 7, 11, 12, t\rangle$. The path induced by tr' is $\langle s, 3, 4, 6, 7, 8, 10, 12, t\rangle$ (see Fig. 3b).

ER-moves ensure that the neighborhood is connected, which is explained in detail in Proposition 2.

Proposition 2 *Let tr^0 be an RST over (g, s, t) and $\mathcal{P}$ be a path from s to t. An RST inducing $\mathcal{P}$ can be reached from tr^0 in $k \leq l$ basic moves, where l is the length of $\mathcal{P}$.*

3.2.3 Neighborhood of independent ER-moves

It is possible to consider more complex moves by applying a set of independent ER-moves. Two ER-moves are independent if the execution of the first one does not affect the second one and vice versa. The sequence of ER-moves $\langle rep(tr, e'_1, e_1), \ldots, rep(tr, e'_k, e_k)\rangle$, denoted by $rep(tr, e'_1, e_1, e'_2, e_2, ..., e'_k, e_k)$, is defined as the application of the sequence of actions $\langle rep(tr_1, e'_1, e_1), rep(tr_2, e'_2, e_2), \ldots,$

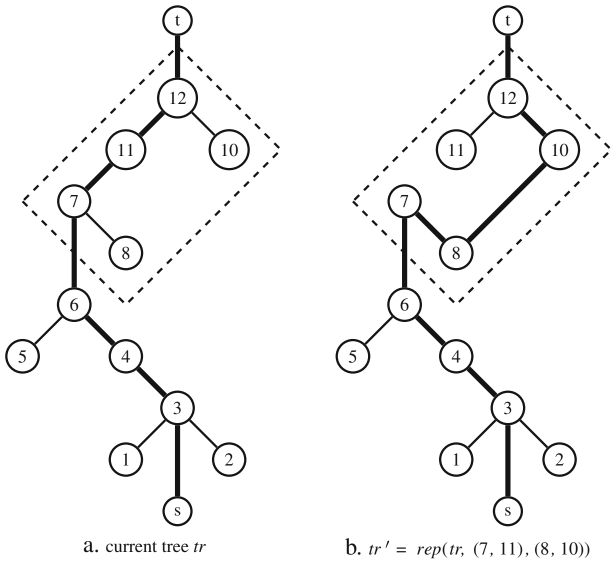


Fig. 3 Illustrating a basic move

$rep(tr_k, e'_k, e_k)\rangle$, where $tr_1 = tr$ and $tr_{j+1} = rep(tr_j, e'_j, e_j)$, $\forall j = 1, \ldots, k-1$. It is feasible if the ER-moves are feasible, i.e., $e_j \in prefRpl(tr_j)$ and $e'_j \in prefRpl(tr_j, e_j)$.

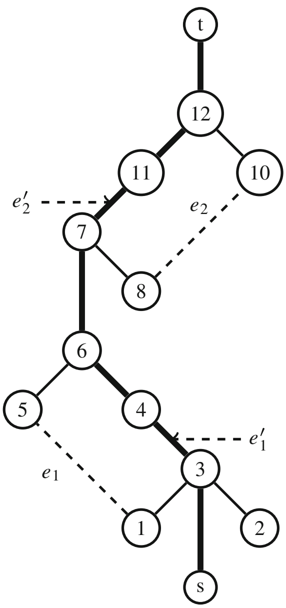
Proposition 3 *Consider k ER-moves $rep(tr, e'_1, e_1), \ldots, rep(tr, e'_k, e_k)$. If all possible execution sequences of these basic moves are feasible and the edges $e'_1, e_1, e'_2, e_2, \ldots, e'_k, e_k$ are all different, then these k ER-moves are independent.*

We denote by $ERNP_k(tr)$ the set of neighbors of tr obtained by applying k independent ER-moves. The action of taking a neighbor in $ERNP_k(tr)$ is called an ER-k-move.

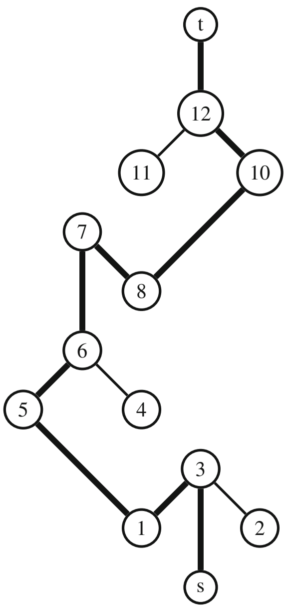
It remains to find some criterion for whether two ER-moves are independent. Given an RST tr over (g, s, t) and two preferred replacing edges e_1, e_2, we say that e_1 *dominates* e_2 *in* tr, written e_1 Dom_{tr} e_2, if $lownca_{tr}(e_1, s)$ dominates $upnca_{tr}(e_2, s)$. Then, two preferred replacing edges e_1 and e_2 are independent w.r.t. tr if e_1 dominates e_2 in tr or e_2 dominates e_1 in tr.

Proposition 4 *Let tr be an RST over (g, s, t), e_1 and e_2 be two preferred replacing edges such that e_2 Dom_{tr} e_1, $e'_1 \in prefRpl(tr, e_1)$, and $e'_2 \in prefRpl(tr, e_2)$. Then $rep(tr, e'_1, e_1)$ and $rep(tr, e'_2, e_2)$ are independent and the path induced by $rep(tr,e'_1,e_1,e'_2,e_2)$ is $path_{tr}(s, v_1) + path_{tr}(u_1, v_2) + path_{tr}(u_2, t)$, where the addition sign denotes path concatenation and $v_1 = low_{tr}(e_1, s)$, $u_1 = up_{tr}(e_1, s)$, $v_2 = low_{tr}(e_2, s)$, and $u_2 = up_{tr}(e_2, s)$.*

Figure 4 illustrates a complex move. In tr, the two preferred replacing edges $e_1 = (1, 5)$ and $e_2 = (8, 10)$ are independent because $lownca_{tr}((8, 10), s) = 7$, which



a. The Current Tree tr (dashed edges are not included)



b. $tr' = rep(tr, (7, 11), (8, 10), (3, 4), (1, 5))$

Fig. 4 Illustrating a Complex Move

dominates $upnca_{tr}((1,5),s) = 6$ in tr. The new path induced by tr' is $\langle s, 3, 1, 5, 6, 7, 8, 10, 12, t\rangle$, which is actually the path $path_{tr}(s,1) + path_{tr}(5,8) + path_{tr}(10,t)$.

4 Data structure and algorithms

In this section, we briefly describe the implementation of some fundamental and non-trivial abstractions and then analyze their complexities.

4.1 VarTree and nearest common ancestors

VarTree(g) is an abstraction representing a dynamic tree over an undirected graph g that can be modified by removing, inserting an edge, or replacing an edge by another edge. It also allows querying information about the tree. For facilitating manipulations on dynamic trees, the trees are implicitly stored as rooted trees. Several well-known data structures have been proposed for representing dynamic trees, for instance, ST-trees [57, 58], topology trees [33], ET-trees [36], top trees [6, 59], and RC-trees [1] (and the references therein). These data structures maintain a forest of dynamic rooted trees, supporting update actions (e.g., link and cut) and some queries (e.g., minimum (maximum) cost edge, node on a path, nearest common ancestors of two nodes, medians, centers of a tree) in $\mathcal{O}(\log n)$ time per operation where n is the number of vertices of the given graph. These data structures have been experimentally studied in [60]. These data structures are dedicated to implementing specific network algorithms, for instance the maximum flow problem.

In the `LS(Graph)` framework, it is required to maintain a dynamic rooted tree supporting update actions (i.e., add, remove, replace edges) and different basic queries such as nearest common ancestors of two nodes, the father of a node, the set of nodes, edges, the set of adjacent edges of a given node. At each step of the local search process, the system explores a neighborhood, queries the quality of all neighbors, and chooses one neighbor to move. Usually, the neighborhood is large and the neighborhood exploration should be as quick as possible. This exploration requires frequent performances of the above queries over dynamic rooted trees. Queries over dynamic trees should thus be as fast as possible. For this purpose, we use a direct data structure for the tree by maintaining the father of each node, the sets for storing nodes, and the edges and the adjacent edges of each node of the tree. So the time complexity for each update action is $\mathcal{O}(n)$ and the above queries (except for that for the nearest common ancestors) take $\mathcal{O}(1)$ instead of $\mathcal{O}(\log n)$.

Concerning the nearest common ancestors problem, Bender et al. [16] presented a simple optimal algorithm for trees which is a sequentialized version of the more complicated PRAM algorithm of Berkman and Vishkin [17]. An intermediate data structure is precomputed in $\mathcal{O}(n)$; each query $nca(u,v)$ is then computed in $\mathcal{O}(1)$ time. The data structure is based on Euler Tour and the data structure for the range minimum query (RMQ) problem. We apply the data structure of [16] with an incremental implementation. This means we partially update the data structure whenever the tree is modified (i.e., by adding, removing, or replacing edges) instead of recomputing it from scratch. This incremental implementation does not improve the time complexity in the worst case ($\mathcal{O}(n)$ for each update action) but it is more efficient in practice. We have tested this implementation on dynamic trees of size

98, 198, 498, 998, of complete graphs of size 100, 200, 500, 1000. For each graph, we generate randomly 20 sequences of 10,000 update actions (adding, removing, replacing edges) conserving the size of the tree. The experimental results show that this incremental implementation is about 1.6 times faster than recomputing from scratch.

4.2 Maintaining weighted distances between vertices on dynamic trees

NodeDistances(vt) is a graph invariant which maintains the weighted distances between all pairs of vertices of a *VarTree vt*. This invariant allows querying the cost of the path between any pair of nodes in $\mathcal{O}(1)$, and thus allows querying the differentiations in $\mathcal{O}(1)$ in some cases, for instance, querying the change in the cost of a path under edge replacement actions. To implement this graph invariant, we use a direct 2-dimensional data structure *dis*: $dis(u, v)$ represents the cost of the path from u to v on the current RST *tr*. The size of this data structure is $\mathcal{O}(n^2)$ but at any time of computation, it is maintained and used partially: only those $dis(u, v)$ such that v dominates u on the current tree *tr* are considered.

The cost of any two nodes x and y on *tr* can be queried by Algorithm 1 in $\mathcal{O}(1)$ where line 1 can be queried in $\mathcal{O}(1)$.

Algorithm 1: distance(x, y)

Input:
Output:
1 $r \leftarrow nca_{tr}(x, y)$;
2 **return** $dis(x, r) + dis(r, y)$;

We now show how to update the $dis(x, y)$ data structure under a local move on *tr*, viz., $rep(tr, (u_1, v_1), (u_2, v_2))$. Without loss of generality, suppose that v_1 Dom_{tr} v_2 and u_1 Dom_{tr} v_1 (see an example in Fig. 5). We put $S = \{x \in V(tr) \mid v_1\ Dom_{tr}\ x\}$. The following elements of the data structure should be updated: $dis(x, y), \forall x \in S, y \in path_{tr}(v_2, nca_{tr}(x, v_2)) \cup path_{tr}(u_2)$. The update schema is given in Algorithm 2, in which $c(u_2, v_2)$ is the weighted distance between u_2 and v_2 in the given graph (see line 6).

Algorithm 2: updateDistances

Input:
Output:
1 **foreach** $x \in S$ **do**
2     $rx \leftarrow nca_{tr}(v_2, x)$;
3   **foreach** $y \in path_{tr}(v_2, rx)$ **do**
4         $dis(x, y) \leftarrow dis(x, rx) + dis(y, rx)$;
5   **foreach** $y \in path_{tr}(u_2)$ **do**
6         $dis(x, y) \leftarrow dis(x, rx) + dis(v_2, rx) + c(u_2, v_2) + dis(u_2, y)$;

The worst case time complexity is $\mathcal{O}(n^2)$ but it performs more efficiently in practice. We now experimentally analyze the efficiency of incrementality in comparison with recomputation from scratch. To do so, we analyze the ratio $r_i = \frac{s_{i-1}}{S_i}$ of data

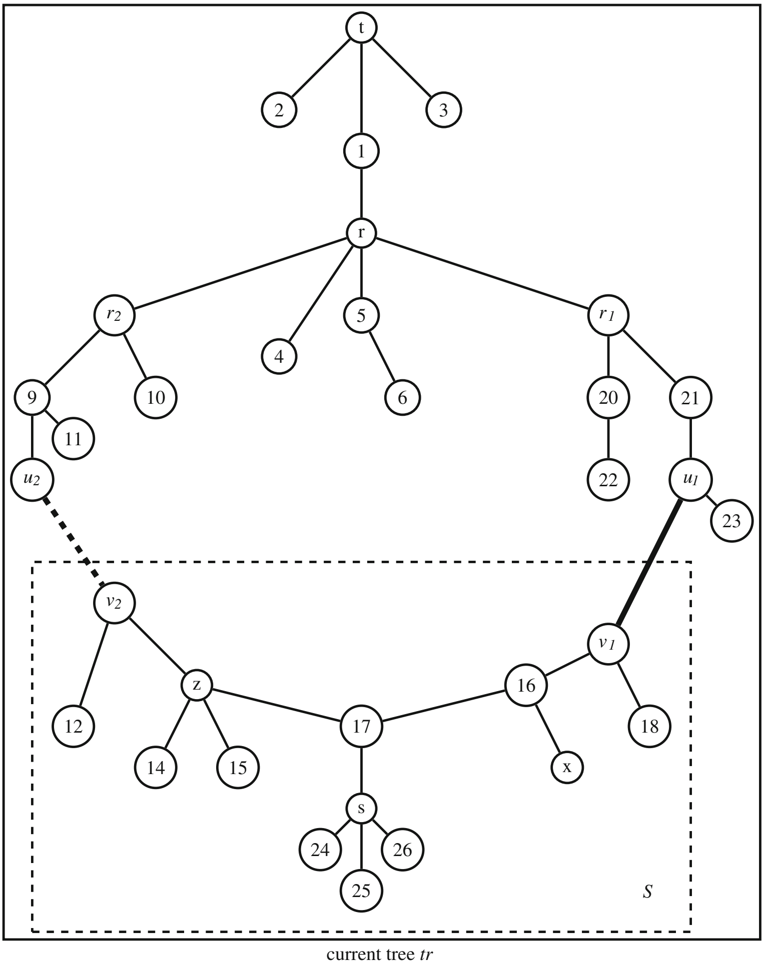


Fig. 5 Ilustrating the update of $dis(u, v)$ under the $replaceEdge(tr, (u1, v1), (u2, v2))$ action

structures to be updated (i.e., $dis(u, v)$) where S_i is the number of elements of *dis* to be maintained at each step i of the computation:

$$S_i = \sum_{v \in V(tr^i)} c_{tr^i}(v)$$

where tr^i is the tree at step i and $c_{tr^i}(v)$ is the number of nodes on the path from v to the root of tr^i; s_i is the number of elements of *dis* to be changed at step i by the incremental version. We look at dynamic trees of size 98, 198, 498, 998 on complete graphs of size 100, 200, 500, 1,000. For each graph, we randomly generate 20 sequences of 10,000 moves. The experimental results show that the average value of r_i is about $\frac{1}{10}$. Figures 6 and 7 show the number of elements to be updated and the number of total elements to be maintained in the last 20 iterations: each iteration is

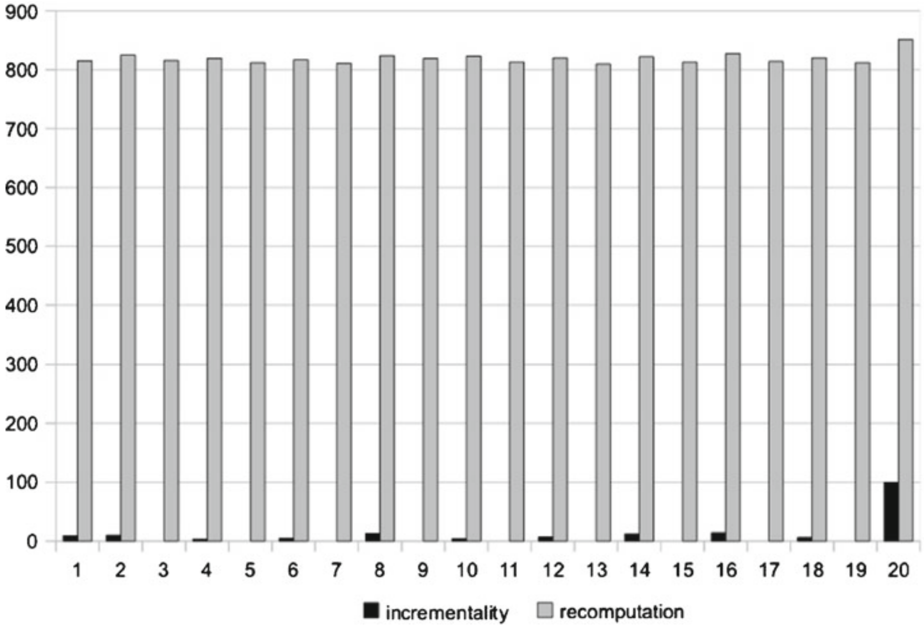


Fig. 6 20 last iterations for a complete graph of size 100

a replace edge action or a sequence of two actions (add and remove edge). It is clear that in the remove edge action, we do not need to update the data structures, so the number of elements to be updated in this action is zero.

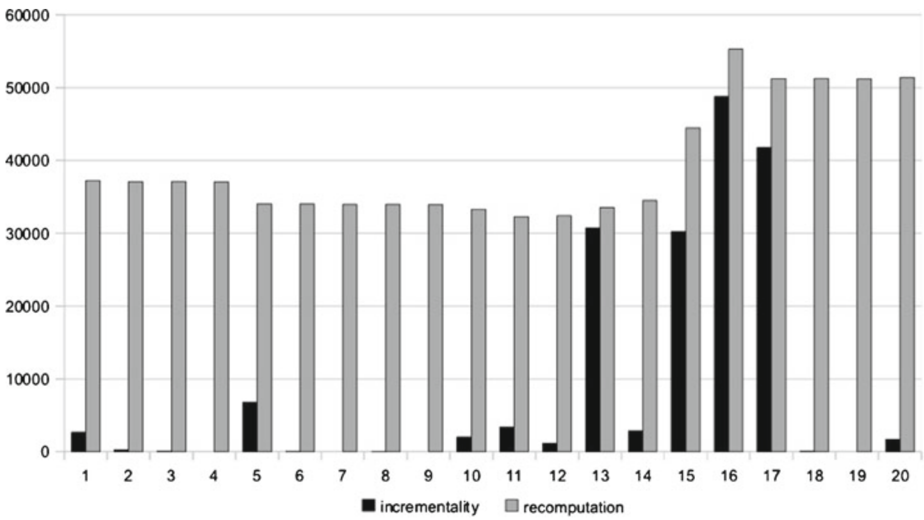


Fig. 7 20 last iterations for a complete graph of size 1,000

```
1 interface Invariant<LSGraph> extends Invariant<LS>{
2   Solver<LSGraph>    getLSGraphSolver ();
3   VarGraph[]     getVarGraphs();

5   bool propagateAddEdge(VarTree vt, Edge ei);
6   bool propagateRemoveEdge(VarTree vt, Edge eo);
7   bool propagateReplaceEdge(VarTree vt, Edge eo, Edge ei);
8   bool propagateReplaceEdge(VarPath vp, Edge eo, Edge ei);
9 }
```

Fig. 8 Interface of graph invariants (partial description)

5 Implementation in COMET

The LS(Graph) framework is implemented in COMET [62]. That is an extension (about 25,000 lines of COMET code) of the COMET system. The core of the framework is the graph variables (e.g., `VarTree`, `VarPath` objects representing dynamic trees, paths which can be changed) over which are defined the graph invariants, graph constraints, and graph functions. The graph invariants maintain the properties of dynamic trees and paths such as the set of insertable, removable, or replacing edges of a `VarTree`, the sum of weights of all the edges of a path, and the diameter of a tree. The graph constraints and graph functions are differentiable objects which not only maintain the properties of dynamic trees, paths (for instance, the number of violations of a constraint or the value of an objective function), but also allow determining the impact of local moves on these properties, a feature known as differentiation.

5.1 Interfaces

Figure 8 depicts part of the interface concerning the graph invariants. Line 2 returns a `Solver<LSGraph>` object which manages all graph variables and graph invariants, and maintains a precedence graph relating these graph variables and graph invariants of the model. A local move (modification action) over a graph variable (`VarTree`, `VarPath`) induces a propagation which updates all graph invariants, constraints, and functions that are defined over these variables thanks to the precedence graph. This means that one does not have to call procedures to update graph invariants, constraints, or functions. Rather, the update is automatically performed whenever users apply local moves. Line 3 returns the list of graph variables[4] over which the graph invariant is defined. Lines 5–8 are some propagation methods corresponding to different local moves.

The differentiation interface is depicted in Fig. 9. The differentiation methods evaluate the impact of various local moves, for instance, `getAddEdgeDelta-(VarTree vt, Edge e)` in line 2 computes the change in the value of the function when the edge `e` is added to the tree `vt`; the method in line 6 returns the change in the value of the function when the replacing edge `e` is applied.[5] The method in line

[4] `VarGraph` is an abstract class from which `VarTree`, `VarPath` are derived.

[5] When a local move $replaceEdge(tr, e', e)$ is applied with the neighborhood $ERNP_1$ (see Section 3.2), the resulting path depends only on the replacing edge e used, not on the replaceable edge e'.

```
1  interface Differentiation<LSGraph>{
2     float getAddEdgeDelta(VarTree t, Edge e);
3     float getRemoveEdgeDelta(VarTree t, Edge e);
4     float getReplaceEdgeDelta(VarTree t, Edge eo, Edge ei);

6     float getDeltaWhenUseReplacingEdge(VarPath vg, Edge e);
7     float getDeltaWhenUseReplacingPath(VarPath vp, Vertex v, Vertex
          x, Vertex y);
8  }
```

Fig. 9 Differentiation interface (partial description)

```
1  interface Constraint<LSGraph> extends Invariant<LSGraph>,
       Differentiation<LSGraph>{
2     var{float} violations();
3     float violations(VarGraph vg);
4  }
```

Fig. 10 Interface of graph constraints (partial description)

7 is generic and computes the impact of moves where the subpath of `vp` between two endpoints of x and y is replaced by the path $\langle \mathtt{x}, \mathtt{v}, \mathtt{y} \rangle$ (see the definition of the most general COP neighborhood $\mathcal{N}$ at the beginning of Section 3.2). It enables the exploration of neighborhoods other than the $ERNP_1$.

Figure 10 depicts the interface of graph constraints in which the method in line 2 returns the violations of the constraint. Line 3 returns the violations of the constraint attributed to `VarGraph vg`. If the graph variable does not appear directly in the definition of the constraint, it does not contribute any violations. This information may be useful when applying multistage heuristics.

All graph invariants, functions, and constraints in the system must implement these interfaces. This enables the compositionality of model. Moreover, one can design and implement one's own functions and constraints, respecting these interfaces, and integrate them into the system.

5.2 Abstractions

The `Solver<LS>` of `COMET` does not support specific operations on user-defined objects (i.e., edge replacement on dynamic trees). So in this framework, we designed and implemented a `Solver<LSGraph>` which maintains a precedence graph representing the dependence of graph invariants, graph functions, and graph constraints on the graph variables and performs the propagations for updating the graph invariants, graph functions, and graph constraints under different modification actions over the graph variables. The implementation of `Solver<LSGraph>` extends `Solver<LS>`, enabling combinations between the two solvers (e.g., we can combine standard invariants of `COMET` with graph invariants of `LS(Graph)` by arithmetic operators). Table 1 partially presents some abstractions[6] available in the framework

[6]For a full description of the abstractions, see the PhD thesis [53].

Table 1 Some graph invariants, functions and constraints of the framework (partial description)

Type	Name	Description
Variables	VarTree(Solver<LSGraph> ls, UndirectedGraph g)	represents dynamic subtree of the graph g
	VarSpanningTree(Solver<LSGraph> ls, UndirectedGraph g)	represents dynamic spanning tree of the graph g
	VarPath(Solver<LSGraph> ls, UndirectedGraph g, Vertex s, Vertex t)	represents dynamic path from s to t on the undirected graph g
	VarPath(Solver<LSGraph> ls, DirectedGraph g, Vertex s, Vertex t)	represents dynamic path from s to t on the directed graph g
Invariants	InsertableEdges(VarTree vt)	set of *insertable* edges of vt
	RemovableEdges(VarTree vt)	set of *removable* edges of vt
	ReplacingEdges(VarTree vt)	set of *replacing* edges of vt
	ReplacingEdgesMaintainPath(VarPath vp)	set of *preferred replacing* edges of vp
	NodeDistances(VarTree vt, int[] indW)	weighted distances w.r.t the indices in W of weights on edges between all pairs of two nodes of vt
Functions	Weight(VarTree vt, int k)	total weights indexed k of all edges of vt
	LongestPath(VarTree vt, int k)	weight indexed k of longest path on vt
	PathCostOnEdge(VarPath vp, int k)	total weights indexed k of all edges of the path vp
	NBVisitedVerticesTree(VarTree[] vts, set{Vertex} S)	number of vertices of S visited by the list of trees vts
	NBVisitedEdgesTree(VarTree[] vts, set{Edge} S)	number of edges of S visited by the list of trees vts
	NBVisitsVertexTree(VarTree[] vts, Vertex v)	number of times the vertex v is visited by the list of trees vts
	NBVisitsEdgeTree(VarTree[] vts, Edge e)	number of times the edge e is visited by the list of trees vts
	NBVisitedVerticesPath(VarPath[] vps, set{Vertex} S)	number of vertices of S visited by the list of paths vps
	NBVisitedEdgesPath(VarPath[] vps, set{Edge} S)	number of edges of S visited by the list of paths vps
	NBVisitsVertexPath(VarPath[] vps, Vertex v)	number of times the vertex v is visited by the list of paths vps
	NBVisitsEdgePath(VarPath[] vps, Edge e)	number of times the edge e is visited by the list of paths vps
	+, -, *	arithmetic operators over graph functions
	GraphFunctionCombinator(Solver<LSGraph> ls)	differentiable object which combines graph functions
Constraints	DiameterAtMost(VarTree vt, int k, float maxD)	the longest path w.r.t index k on weight cannot exceed maxD
	DegreeAtMost(VarTree vt, float maxD)	degree of each vertex of vt cannot exceed maxD
	TreesEdgeDisjoint(VarTree[] vts)	edge-disjoint constraint over the list of trees vps
	TreesVertexDisjoint(VarTree[] vts)	node-disjoint constraint over the list of trees vps
	TreesContainVertices(VarTree[] vts, set{Vertex} S)	the list of trees vps must visit all vertices of S
	TreesContainEdges(VarTree[] vts, set{Edge} S)	the list of trees vps must visit all edges of S
	PathsEdgeDisjoint(VarPath[] vps)	edge-disjoint constraint over the list of paths vps
	PathsVertexDisjoint(VarPath[] vps)	node-disjoint constraint over the list of paths vps
	PathsContainVertices(VarPath[] vps, set{Vertex} S)	the list of paths vps must visit all vertices of S
	PathsContainEdges(VarPath[] vps, set{Edge} S)	the list of paths vps must visit all edges of S
	>=, <=, ==	relation operators over graph functions
	ConstraintSystem<LSGraph>(Solver<LSGraph> ls)	differentiable object which combines graph constraints
solver	Solver<LSGraph>	solver of the framework
search	TabuSearch<LSGraph>(Model<LSGraph> mod)	generic tabu search component
	GreedyLocalSearch<LSGraph>(Model<LSGraph> mod)	generic greedy local search component

including some graph variables, invariants, functions, and constraints which are used to model various COT/COP problems: create a solver Solver<LSGraph>, declare variables VarTree, VarPath, and state functions and constraints. Different search procedures can then be performed over the model. Fundamental functions representing relations between the trees, paths, nodes, and edges have been designed and implemented, e.g., NBVisitedVerticesTree(VarTree[] vts, set{Vertex} S) represents the number of vertices of S which are visited by the list of trees vts, and NBVisitsVertexTree(VarTree[] vts, Vertex v) represents the number of times the list of trees vts visit it. Weight(VarTree vt, int k) represents the weight of a tree vt, and PathCostOnEdges(VarPath vp, int k) represents the cost of a path vp.[7] These functions can be combined by traditional arithmetic or relation operators to state more complex functions or constraints. Various fundamental constraints on graphs can be stated by using these functions and traditional relation operators. For achieving a more efficient performance, some global constraints have been designed and implemented, for instance, PathsEdgeDisjoint(VarPath[] vps) specifies that the list of paths vps must be edge-disjoint, and PathsContainVertices(VarPath[] vps, set{Vertex} S) specifies that the list of paths vps must visit the set of vertices S.

FunctionCombinator<LSGraph> is a graph function that combines several functions, constraints of the model by the "+" operator with a weight. This object strengthens the modeling of the framework when there are a number of functions proportional to the size of the problem to be stated.

ConstraintSystem<LSGraph> is a graph constraint which combines all constraints appearing in the considered problem by the post method. By using this object, one can add or remove some constraints from the model without having to change the search procedure.

The LS(Graph) framework is open in that it allows users to design and implement their own invariants, constraints, and functions respecting predefined interfaces and integrate them into the system.

5.3 Search procedures

In order to illustrate the modeling and the search component, we give an example in Fig. 11 in which we solve the problem of finding a spanning tree of a given undirected graph g such that the degree of each node does not exceed maxDe and the diameter of the spanning tree does not exceed maxDia.

The model is given in lines 1–15, in which line 2 creates a Solver<LSGraph> ls and lines 3–4 randomly initialize a spanning tree variable vt of a given undirected graph g associated with ls. Line 5 initializes a graph invariant rpl (line 4) representing the set of replacing edges of vt. Lines 7–13 state and post constraints on the degree and diameter of the spanning tree vt to a graph constraint system gcs which is declared in line 10. Whenever the model is closed (line 15), the initPropagation methods of all graph invariants are called to initialize the values and internal data structures of these objects.

[7] k is the index of the considered weight on edges.

```
1    // The Modeling
2    Solver<LSGraph> ls();
3    int k = g.numberOfVertices()-1;
4    VarTree vt(ls,g,k); // tree variable
5    ReplacingEdgesVarTree rpl(ls,vt); // invariant representing the
         set of replacing edges of vt

7    DegreeAtMost degreeC(vt,maxDe); // constraint on degrees of
         vertices of vt
8    DiameterAtMost diameterC(vt,0,maxDia);// constraint on the
         diameter of vt

10   ConstraintSystem<LSGraph> gcs(ls); // constraint system
11   gcs.post(diameterC); // posting the constraint on degrees
12   gcs.post(degreeC); // posting the constraint on diameter
13   gcs.close();

15   ls.close();

17   // The Search
18   int it = 1;
19   while(it < 1000 && gcs.violations() > 0){
20     selectMin(ei in rpl.getSet(),
21       eo in getReplacableEdges(vt,ei))
22         (gcs.getReplaceEdgeDelta(vt,eo,ei)){
23       vt.replaceEdge(eo,ei); // perform the move
24     }
25     it++;
26   }
```

Fig. 11 Model for bounded diameter and degree constrained spanning tree

The search is given in lines 17–26, which is a simple greedy search. At each iteration, we explore the NT_3 neighborhood and choose the best neighbor w.r.t. the graph constraint system `gcs`: we choose a replacing edge `ei` and a replaceable edge `eo` of `ei` such that the number of violations of `gcs` is most reduced when `eo` is replaced by `ei` (see method `getReplaceEdgeDelta(vt,eo,ei)`). Line 23 is the local move which induces automatically a propagation to update all graph invariants and constraints defined over it (e.g., `rpl`, `degreeC, diameterC`) thanks to the precedence graph maintained in `ls`.

We can see in this example that the model and the search are independent. On the one hand, we can state and post other constraints to the graph constraint system `gcs` without having to change the search. On the other hand, we can apply different heuristic local searches in the search component without changing the model.

We now describe one of generic neighborhood explorations. Figure 12 explore the basic COP neighborhood $ERNP_1$. The quality of a solution is evaluated in terms of the number of violations of the `Constraint<LSGraph> c`. Variables `it` and `fgb` represent the current iteration of the local search and the smallest value of the number of violations of the constraint `c` found so far. All `VarPath vps[j]` are scanned (lines 7–8). Line 9 retrieves the `Invariant<LSGraph> repl` representing the set of preferred replacing edges of `vps[j]`. All preferred replacing edges `e` are scanned in line 10 and line 11 evaluates the quality of the move when

```
void exploreTabuMinReplace1Move1VarPath(Neighborhood N, VarPath[] vps, dict{VarPath->ReplacingEdgesMaintainPath} mapReVarPath, Constraint<LSGraph> c, GTabuEdge[] tbIn, GTabuEdge[] tbOut, int it, float fgb, bool firstImprovement){

  Edge sel_ei = null; // the selected replacing edge for the move
  int ind = -1; // the index of the selected VarTree for the move
  float eval = System.getMAXINT(); // the minimum evaluation

  forall(j in vps.rng()){
    VarPath vp = vps[j]; // considered VarPath
    ReplacingEdgesMaintainPath repl = mapReVarPath{vp}; // invariant representing the set of preferred replacing edges of vp
    forall(e in repl.getSet()){ // scan all preferred replacing edges
      float d = c.getDeltaWhenUseReplacingEdge(vp,e); // evaluation of using the preferred replacing edge e

      if(!tbIn[j].isTabu(e,it) || d + c.violations() < fgb){ // check the tabu condition or the aspiration criterion
        if(d < eval){ // update the information of the chosen move
          eval = d;
          ind = j;
          sel_ei = e;
        }
        if(firstImprovement && eval < 0)
          break; // stop the neighborhood exploration if a first improving neighbor is found
      }
    }
    if(firstImprovement && eval < 0)
      break; // stop the neighborhood exploration if a first improving neighbor is found
  }

  if(ind > -1){
    VarPath vp = vps[ind];
    Edge sel_eo = null;

    select(eo in getPreferredReplacableEdges(vp,sel_ei)){
      sel_eo = eo;
    }

    if(sel_eo != null)
      neighbor(eval,N){// submit the chosen move
        tbIn[ind].makeTabu(sel_eo,it); // make the selected preferred replacable edge tabu
        tbOut[ind].makeTabu(sel_ei,it); // make the selected preferred replacing edge tabu

        vp.replaceEdge(sel_eo,sel_ei); // perform the move
      }
  }
}
```

Fig. 12 Exploring the $ERNP_1$ neighborhood applying the replacing edge `e` in term of the variation of the number of violations of `c`. Line 13 checks whether `e` is tabu or the aspiration criterion is reached (i.e., the move is tabu but it improves the best solution found so far). Lines 31–33 choose a preferred replaceable `sel_eo`. Lines 36–41 submit a move (lines 36–41) and its evaluation `eval` to a `Neighborhood N` and it will be called later.

Components for a generic tabu search, `TabuSearch<LSGraph>`, and a greedy local search, `GreedyLocalSearch<LSGraph>`, have been implemented for COT/COP applications. This tabu search component features aspiration criteria with adaptive tabu length (the tabu length can be changed within *tb Min* and *tb Max*, depending on the behavior of the search). A full description of the abstractions and generic search components can be found in [53].

6 Applications

In this section, we present the application of the `LS(Graph)` framework to the resolution of three COT/COP problems: the quorumcast routing (QR) problem, the edge-disjoint paths (EDP) problem, and the routing and wavelength assignment with side constraint (RWA-D) problem.

For the first and the third applications (QR and RWA-D), we apply tabu search. Two parameters of tabu search are the length *tbl* of the tabu lists and *maxStable*: if the best-restart solution[8] does not improve in *maxStable* successive local moves, then the search is restarted.

Experiments were performed on XEN virtual machines with 1 core of a CPU Intel Core2 Quad Q6600 @2.40 GHz and 1 GB of RAM.

6.1 The quorumcast routing (QR) problem

6.1.1 Problem statement

Given a weighted undirected graph $G = (V, E)$, each edge $e \in E$ is associated with a cost $w(e)$. Given a source node $r \in V$, an integral value q, and a set $S \subseteq V$ of multicast nodes, the quorumcast routing problem is to find a minimum cost tree $T = (V', E')$ of G spanning r and q nodes of S. $T = (V', E')$ is a graph satisfying

1. $V' \subseteq V \wedge E' \subseteq E$,
2. T is connected,
3. $\exists Q \subseteq S$ such that $\sharp Q = q \wedge Q \cup \{r\} \subseteq V'$,
4. The cost of

$$T = \sum_{e \in E'} w(e)$$

is minimum over all subgraphs of G with properties 1–3.

In this section, we present a local search model for solving the QR problem with `LS(Graph)`.

[8] The best-restart solution is the best solution found for each restart.

6.1.2 The model

We propose a tabu search model in `LS(Graph)` exploring different neighborhoods for solving this problem. The model is given in Fig. 13, in which line 1 creates a `Solver<LSGraph>` and line 2 declares a `VarTree tr` associated with `ls`. Lines 4–7 state the constraints of the problem where `NBVisitedVertices(tr,S)` is a `Function<LSGraph>` representing the number of vertices of `S` which are in the tree `tr`. The constraint posted in line 5 says that the tree `tr` must contain at least `q` vertices of `S` and the constraint posted in line 6 says that `tr` must contain the vertex `s`. Line 9 creates a `Model<LSGraph> mod` with only one variable `tr`, the constraint `gcs`, the objective function to be minimized is the total weight of `tr`. Line 11 initializes a search component which extends `TabuSearch<LSGraph>` (see Fig. 14). Lines 12–14 set parameters for the search and line 16 calls the search procedure. We now describe the search component in Fig. 14. The variables `_card` and `_root` represent the number of edges of the initial tree and its root computed in the `initSolution` method. The overriding `initSolution` method (lines 17–31) constructs the tree in a greedy random way. It clears the tree `tr` (line 22) and selects randomly a first edge containing `_root` (lines 23–25). It then iteratively selects an edge with minimal weight for adding to the constructed tree `tr` (lines 27–30). The `exploreNeighborhood` method of `TabuSearch<LSGraph>` is also overriden (lines 34–39) with different neighborhoods: NT_1 (line 35), NT_2 (line 36), NT_{1+2} (line 37), and NT_3 (line 38).

```
1   Solver<LSGraph> ls(); // create a solver
2   VarTree tr(ls,g); // initialize a tree variable, g is the given
        graph

4   ConstraintSystem<LSGraph> gcs(ls); // constraint system
5   gcs.post(q <= NBVisitedVerticesTree(tr,S)); // posting the
        constraint specifying that tr must contain at least q vertices
        of S
6   gcs.post(NBVisitedVerticesTree(tr,s) == 1); // the tree tr must
        contain the vertex s
7   gcs.close();

9   Model<LSGraph>
        mod(tr,gcs,Weight<Tree>(tr,1),NonSpanningTree,MINIMIZATION);
        // encapsulate variables, constraints, and objective function
        into a model object

11  QRSearch se(mod); // create a search object which extends the
        built-in generic search
12  se.setMaxIter(1000);
13  se.setCard(q);
14  se.setRoot(s);

16  se.search(); // perform the search
```

Fig. 13 Tabu search model for the QR problem

```
include "LS(Graph)";

class QRSearch extends TabuSearch<LSGraph>{
  Vertex  root;
  int     _card;
  QRSearch(Model<LSGraph> mod): TabuSearch<LSGraph>(mod){
  }
  void setCard(int ca){
    _card = ca;
  }
  void setRoot(Vertex r){
    root = r;
  }
  void restartSolution(){ // restart the search by using the initial solution generation procedure
    initSolution();
  }
  void initSolution(){// generate the initial solution
    Solver<LSGraph> ls = getLSGraphSolver(); // get the solver
    VarTree tr = getFirstVarTree(); // retrieve the tree variable tr
    InsertableEdgesVarTree inst = getInsertableEdges(tr); // retrieve the invariant representing the set of insertable edge of tr

    tr.clear(); // clear the tree
    select(e in inst.getSet():e.contains(root)){ // choose randomly a first edge to be added to tr
      tr.addEdge(e);
    }

    forall(i in 1.._card-1) // repeat adding an edge until the tree tr has _card edges
      selectMin(e in inst.getSet())(e.weight()){ // select an insertable edge having smallest weight
        tr.addEdge(e); // add the selected edge to the tree
      }
  }


  void exploreNeighborhood(Neighborhood N){ // explore all four neighborhoods of VarTree
    exploreTabuMinAdd1VarTree(N,true); // explore the neighborhood NT_1
    exploreTabuMinRemove1VarTree(N,true); // explore the neighborhood NT_2
    exploreTabuMinAddRemove1VarTree(N,true); // explore the neighborhood NT_{1+2}
    exploreTabuMinReplace1VarTree(N,true); // explore the neighborhood NT_3
  }
}
```

Fig. 14 The search component for the QR problem

6.1.3 Experiments

We compare our tabu model in LS(Graph) with the IMP heuristic, which is the best heuristic among the three heuristic algorithms in [24]. The original instances and the implementation of the IMP algorithm are not available. We thus re-implemented the IMP algorithm in COMET and generated new benchmarks.

Problem instances We take six graphs from the benchmark of the KCT problem [20] which are 4-regular graphs of sizes from 50 to 1,000 nodes and six graphs from the Steiner tree instances. For each graph of size n, we generate randomly $n * tau_1$ nodes for the set S, the value for q is set to $n * tau_1 * tau_2$ with $tau_1, tau_2 \in \{0.2, 0.5\}$, and the root is set to be node 1.

Results The IMP algorithm and our model in LS(Graph) are executed 20 times for each problem instance. The time limit for our model is 30 min. From our preliminary results, we set *tbl* to 5 and *maxStable* to 200. The experimental results are shown in Tables 2 and 3. Columns 3–6 present the average, the minimal, the maximal, and the standard deviation of the best objective value found in 20 executions. The same information for our model is presented in columns 8–11. Column 7 is the average execution time (in seconds) of the IMP algorithm over 20 executions, while column 12 presents the average time (in seconds) for finding the best solutions over 20 executions of our tabu search model. Table 2 shows that for KCT instances, our LS(Graph) model finds better solutions than the IMP on average. Moreover, the worst solutions found by our model are, in most cases, even better than the best solution found by the IMP (among 20 executions). Table 3 shows that the results found by our model are better than those found by the IMP algorithm on average except for the last four instances (45–48). A comparison of the two algorithms in terms of box-and-whiskers plots (see their template presentation in Fig. 15) can be found in Figs. 16, 17, 18, and 19. Two consecutive bars present the results computed by the IMP and the tabu search algorithms on a given instance. The figures show that for each algorithm, the variance of the results among the 20 executions is small. It also shows that, in most instances, the solutions found by our tabu search are better than those found by the IMP algorithm.

6.2 The edge-disjoint paths problem

6.2.1 Problem statement

We are given an undirected graph $G = (V, E)$ and a set $T = \{\langle s_i, t_i\rangle \mid i = 1, 2, ..., \sharp T; s_i \neq t_i \in V\}$ representing a list of commodities. A subset $T' \subseteq T$, $T' = \{\langle s_{i_1}, t_{i_1}\rangle, ..., \langle s_{i_k}, t_{i_k}\rangle\}$ is called *edp*-feasible if there exist mutually edge-disjoint paths from s_{i_j} to t_{i_j} on $G, \forall j = 1, 2, .., k$. The EDP problem consists in finding a maximal cardinality *edp*-feasible subset of T. In other words,

$$\begin{array}{lll} max & \sharp T' & (1) \\ s.t. & T' \subseteq T & (2) \\ & T' \text{ is } edp\text{-feasible} & (3) \end{array}$$

Table 2 Experimental results on KCT instances

Index	Instances	IMP					LS(Graph)				
		avg	min	max	std_dev	avg_t	avg	min	max	std_dev	avg_t
1	g50_20_20	111	111	111	0	0.78	111	111	111	0	0.06
2	g50_20_50	251	251	251	0	0.78	248	248	248	0	0.08
3	g50_50_20	169	169	169	0	0.8	169	169	169	0	0.09
4	g50_50_50	386	386	386	0	0.76	369	369	369	0	0.22
5	g75_20_20	93	93	93	0	1.06	93	93	93	0	0.02
6	g75_20_50	358	358	358	0	1.06	328	328	328	0	1.12
7	g75_50_20	207	207	207	0	1.05	175	175	175	0	0.08
8	g75_50_50	630	630	630	0	1.05	564.6	560	568	3.56	181.81
9	g100_20_20	178	178	178	0	1.44	178	178	178	0	0.23
10	g100_20_50	526	526	526	0	1.45	524	524	524	0	2.31
11	g100_50_20	294	294	294	0	1.51	273	273	273	0	0.28
12	g100_50_50	948	948	948	0	1.53	854	854	854	0	36.84
13	g200_20_20	428	428	428	0	7.13	402	402	402	0	32.6
14	g200_20_50	926	926	926	0	7.07	849.6	849	851	0.92	342.49
15	g200_50_20	483	483	483	0	7.26	468	468	468	0	8.09
16	g200_50_50	1,499	1,499	1,499	0	7.35	1,411.45	1,403	1,424	5.82	816.43
17	g400_20_20	599	599	599	0	52.18	556.6	551	560	1.88	240.46
18	g400_20_50	1,724.05	1,702	1,739	13.92	51.43	1,610.95	1,600	1,626	7.05	689.33
19	g400_50_20	1,154.55	1,140	1,166	7.75	51.6	1,010.65	1,005	1,018	4.3	584.08
20	g400_50_50	3,040	3,040	3,040	0	52.19	2,829.15	2,799	2,856	17.25	845.52
21	g1000_20_20	1,832.1	1,810	1,836	9.28	812.61	1,568.65	1,505	1,621	27.67	584.75
22	g1000_20_50	4,762.2	4,755	4,771	7.96	795.64	4,493.55	4,406	4,599	49.9	869.09
23	g1000_50_20	2,743	2,733	2,746	4.29	801.06	2,487.2	2,429	2,533	27.04	697.85
24	g1000_50_50	7,293.85	7,229	7,361	36.28	817.82	7,098.95	6,891	7,372	117.06	1,330.21

Table 3 Experimental results on steiner instances

Index	Instances	IMP					LS(Graph)				
		avg	min	max	std_dev	avg_t	avg	min	max	std_dev	avg_t
25	steinb4_20_20	11	11	11	0	0.72	11	11	11	0	0
26	steinb4_20_50	32	32	32	0	0.75	32	32	32	0	0.12
27	steinb4_50_20	20.35	20	21	0.48	0.74	20	20	20	0	0.08
28	steinb4_50_50	52.25	51	53	0.77	0.74	41	41	41	0	0.27
29	steinb10_20_20	19	19	19	0	1	19	19	19	0	0.12
30	steinb10_20_50	29	29	29	0	0.98	29	29	29	0	0.32
31	steinb10_50_20	27.8	26	29	1.25	1.03	22	22	22	0	0.16
32	steinb10_50_50	65	65	65	0	0.99	65	65	65	0	2.76
33	steinb16_20_20	10	10	10	0	1.46	10	10	10	0	0.01
34	steinb16_20_50	73.35	69	76	2.33	1.55	61	61	61	0	9.46
35	steinb16_50_20	32.85	32	37	1.19	1.47	31	31	31	0	0.93
36	steinb16_50_50	87.25	85	92	2.09	1.48	82	82	82	0	12.6
37	steinc6_20_20	92.35	81	98	4.75	100.58	69.7	69	72	0.9	514.06
38	steinc6_20_50	234.15	229	240	3.61	101.77	221.9	218	225	1.73	614.34
39	steinc6_50_20	130.95	122	147	6.16	99.82	115.9	113	118	1.09	372.27
40	steinc6_50_50	399.55	395	407	3.17	103.59	381.55	374	387	3.28	866.87
41	steinc11_20_20	43.95	40	47	1.99	102.08	38.7	38	39	0.46	481.52
42	steinc11_20_50	116	113	119	1.55	102.6	107.4	107	109	0.58	803.34
43	steinc11_50_20	75.05	70	79	2.48	101.89	67.75	67	69	0.62	455
44	steinc11_50_50	207.25	201	213	2.9	102.04	202.45	199	208	2.31	1,000.79
45	steinc16_20_20	22.25	21	25	1.13	100.2	23.6	22	24	0.66	200.7
46	steinc16_20_50	54.45	52	59	1.66	100.22	54.95	53	56	0.74	267.01
47	steinc16_50_20	50	50	50	0	99.34	50.3	50	52	0.56	451
48	steinc16_50_50	125	125	125	0	104.98	140.25	133	148	4.09	1,567.46

Fig. 15 Box-and-Whiskers plot: the *X-axis* represents the algorithm and the instance (**A** denotes the algorithm and **ins** denotes the instance) and the *Y-axis* represents the value of the objective function

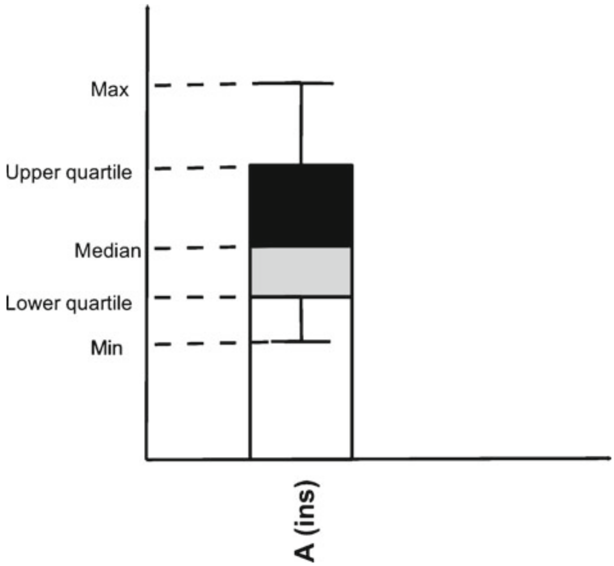


Fig. 16 Comparison between IMP and LS(Graph) on KCT instances

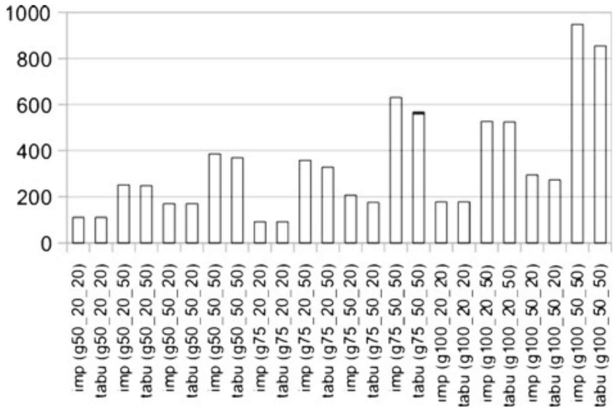
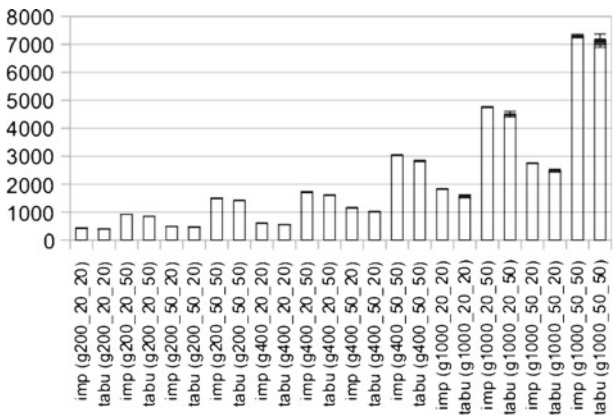


Fig. 17 Comparison between IMP and LS(Graph) on KCT instances



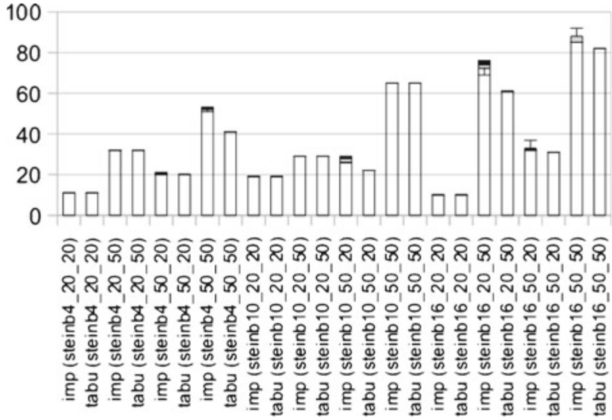


Fig. 18 Comparison between IMP and LS(Graph) on steiner instances

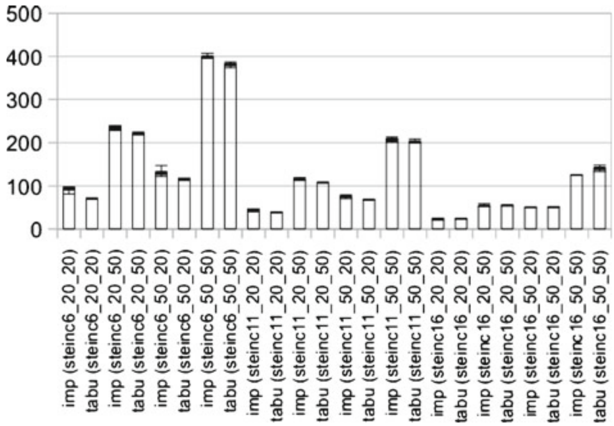


Fig. 19 Comparison between IMP and LS(Graph) on steiner instances

In this section, we propose two algorithms based on neighborhood search for solving the EDP problem by `LS(Graph)`. They are complex heuristics which make use of local search in `LS(Graph)` as sub-routines. We first describe the simple greedy algorithm SGA [42] because one of our algorithms (detailed later) will apply this as sub-procedure (see Algorithm 3).

Algorithm 3: SGA(G,T)

Input: Problem instance $\langle G = (V, E), T \rangle$ consist of a graph G and a commodity list T
Output: Set of edge-disjoint paths on G connecting endpoints in T

$S \leftarrow \oslash$;
$E_1 \leftarrow E$;
foreach $T_j = \langle s_j, t_j \rangle \in T$ **do**
    if *s_j and t_j can be connected by a path in $G_1 = (V, E_1)$* **then**
        $P_j \leftarrow$ shortest path from s_j to t_j in $G_1 = (V, E_1)$;
        $S \leftarrow S \cup \{P_j\}$;
        $E_1 \leftarrow E_1 \setminus \{e \mid e \in P_j\}$;
return S;

6.2.2 The simple greedy algorithm

This algorithm starts with an empty solution S (line 1). At each iteration j (line 3), it selects a pair $T_j = \langle s_j, t_j \rangle$ and tries to find the shortest path P_j from s_j to t_j in the graph $G_1 = (V, E_1)$, initializing the set of edges E_1 to be E (line 2). If such a path exists, it is inserted into S and the set E_1 is updated for the next step by removing all edges of the path P_j.

Obviously, the SGA algorithm depends strongly on the order of commodities T_j considered. The multi-start version of SGA (called MSGA) performs SGA iteratively with different orders of T_j to be scanned in T.

In the ACO algorithm of [18], the following criterion is introduced, which quantifies the degree of non-disjointness of a solution. $S = \{P_1, P_2, ... P_k\}$ (P_j is a path from s_j to t_j):

$$C(S) = \sum_{e \in E} \left(\max\{0, \sum_{P_j \in S} \rho^j(S, e) - 1\} \right),$$

where $\rho^j(S, e) = 1$ if $e \in P_j$, and $\rho^j(S, e) = 0$ otherwise. From a solution constructed by ANTs, a solution to the EDP problem is extracted by iteratively removing the path which has the most edges in common with other paths, until all remaining paths are mutually edge-disjoint (see Algorithm 4).

Algorithm 4: Extract(S)

```
Input: set S of paths
Output: subset of edges-disjoint paths of S
S0 ← S;
while C(S0) > 0 do
    foreach p ∈ S0 do
        c(p) ← number of edges of the path p in common with other paths of S0;
    p* ← argMax_{p∈S0} c(p);
    S0 ← S0 \ {p*};
return S0;
```

In this section, we propose two algorithms based on local search for solving this problem: the LS-SGA and the LS-R algorithms. These algorithms perform a local search procedure applying the `LS(Graph)` framework combined with the extraction method (Algorithm 4) and the simple greedy algorithm. These algorithms make use of the $PathsEdgeDisjoint(P_1, P_2, ..., P_k)$ constraint of the `LS(Graph)` framework saying that the set of paths $\{P_1, P_2, ..., P_k\}$ must be edge-disjoint. The number of violations of the $PathsEdgeDisjoint(P_1, P_2, ..., P_k)$ constraint is defined to be $C(\{P_1, P_2, ..., P_k\})$ and the local search algorithms used in our heuristics try to minimize this number.

6.2.3 The LS-SGA algorithm

The LS-SGA algorithm has been proposed in our paper [54]. The main idea of the LS-SGA algorithm (given in detail in Algorithm 5) is to perform a local search algorithm aiming at minimizing the number of violations of the $PathsEdgeDisjoint(P_1, P_2, ..., P_k)$ constraint. The variable S (line 2) stores a set of paths $\{P_1, P_2, ..., P_k\}$ connecting all commodities. It is initialized randomly (lines

3–5). At each step, we perform a local move. The LocalMove method (line 7) returns true if it finds a move that decreases the number of violations of the *PathsEdgeDisjoint*$(P_1, P_2, ..., P_k)$ constraint. If no such move exists, we make some random moves (line 22). From a candidate solution S found by the local search, a solution S_1 to the EDP problem will be extracted by applying the Extract algorithm (line 9) combined with the SGA algorithm (line 15) on the remaining graph G'' (the graph G'' is obtained by removing all edges E' (line 12) of the paths extracted by the Extract algorithm) and the remaining commodities T'' (lines 10 and 11). The best solution is updated in line 17 and lines 18–20 update some paths of S by the new found paths of S_2.

Algorithm 5: LS-SGA(G,T)

Input: Problem instance $\langle G = (V, E), T\rangle$ consist of a graph G and a commodity list T
Output: Set of edge-disjoint paths on G connecting endpoints in T

1 $S_{best} \leftarrow \oslash$;
2 $S \leftarrow \oslash$;
3 **foreach** $\langle s_i, t_i\rangle \in T$ **do**
4     $p_i \leftarrow$ random path from s_i to t_i on G;
5     $S \leftarrow S \cup \{p_i\}$;
6 **while** *termination criterion is not reached* **do**
7     $hasMove \leftarrow$ LocalMove(S);
8   **if** $hasMove$ **then**
9         $S_1 \leftarrow$ Extract(S);
10         $T' \leftarrow$ set of commodities that are connected by paths in S_1;
11         $T'' \leftarrow T \setminus T'$;
12         $E' \leftarrow$ set of edges of paths of S_1;
13         $E'' \leftarrow E \setminus E'$;
14         $G'' \leftarrow (V, E'')$;
15         $S_2 \leftarrow$ SGA(G'', T'');
16   **if** $\sharp S_1 + \sharp S_2 > \sharp S_{best}$ **then**
17             $S_{best} \leftarrow S_1 \cup S_2$;
18   **foreach** $p_i \in S_2$ **do**
19   p is a path of $S \setminus S_1$ such that starting point of $p \equiv$ starting point of p_i and terminating point of $p \equiv$ terminating point of p_i;
20                 $p \leftarrow p_i$;
21   **else**
22   RandomMoves(S);
23 **return** S_{best};

6.2.4 The LS-R algorithm

The idea is to connect recursively as much as possible the commodities of T (see Algorithm 6). The core is the recursive method LS-Recursive in Algorithm 7, which receives a graph G and a list of commodities T as input and computes a set of maximally edge-disjoint paths connecting the commodities of T. This paths set is then accumulated in the solution *Sol* (*Sol* is a global variable) and all edges visited by these paths are removed from G for the next recursive call. Line 1 computes a set of edge-disjoint paths by a greedy local search method, GreedyLocalSearch. Lines 2–3 update the solution by adding the new found edge-disjoint paths of S_i. Lines 3–4 compute the set of connected components CC of the graph generated from the current graph by removing all edges E' of paths of S_i. For each graph G_i of these connected components and each set of commodities T_i that belong to G_i, we perform recursively the LS-Recursive method (see lines 6–8).

Algorithm 6: LS-R(G, T)

Input: Problem instance $\langle G = (V, E), T \rangle$ consist of a graph G and a commodity list T
Output: Set of edge-disjoint paths on G connecting endpoints in T

$S_{best} \leftarrow \oslash$;
while *termination criterion is not reached* **do**
  $Sol \leftarrow \oslash$;
  LS-Recursive(G, T);
  if $\sharp Sol > \sharp S_{best}$ **then**
    $S_{best} \leftarrow Sol$;

The implementation of these algorithms in `LS(Graph)` is given in the PhD thesis [53]. It is more complicated than that of the QR problem: it requires some processing (e.g., removing edges and vertices from a graph, and computing the connected components of a graph) other than just stating the model and performing the search.

Algorithm 7: LS-Recursive(G, T)

Input: Problem instance $\langle G = (V, E), T \rangle$ consist of a graph G and a commodity list T; Sol is a global variable that stores a set of edges-disjoint paths under construction
Output: Update Sol

1 $S_i \leftarrow$ GreedyLocalSearch(G, T);
2 **foreach** $p \in S_i$ **do**
3   $Sol \leftarrow Sol \cup \{p\}$;
4 $E' \leftarrow$ set of edges of paths of S_i;
5 $CC \leftarrow$ set of connected components of the graph $(V, E \setminus E')$;
6 **foreach** $G_i \in CC$ **do**
7   $T_i \leftarrow$ set of commodities that are not connected by any path of S_i such that their endpoints belong to G_i;
8   LS-Recursive(G_i, T_i);

6.2.5 Experiments

Problem instances We tried the two proposed algorithms on three types of benchmark. The first benchmark contains instances on four graphs provided by Blesa [18]. The second benchmark contains instances on some graphs of the Steiner benchmark from the Or-Library [14]. The third benchmark consists of instances on random planar graphs. Table 4 gives a description of these graphs.

An instance of the EDP problem consists of a graph and a set of commodities. The instances in the original paper [18] are not available. As a result, we base our trial on the instance generator described in [18] and generate new instances as follows. For each graph of the first set, we generate randomly different sets of commodities with different sizes, depending on the size of the graph: for each graph of size n, we generate randomly two instances[9] with 0.10*n, 0.25*n, and 0.40*n commodities. We do the same for each Steiner and planar graph but we generate only one instance for

[9]This is different from what we did in [54], where we randomly generated 20 instances for each rate of commodity. For each instance, the algorithm was executed only once.

Table 4 Description of graphs of the benchmarks

Name	$\|V\|$	$\|E\|$	Degree avg.
bl-wr2-wht2.10-50.rand	500	1,020	4.08
bl-wr2-wht2.10-50.sdeg	500	1,020	4.08
mesh15×15	225	420	3.73
mesh25×25	625	1,200	3.84
steinb4.txt	50	100	4.00
steinb10.txt	75	150	4.00
steinb16.txt	100	200	4.00
steinc6.txt	500	1,000	4.00
steinc11.txt	500	2,500	10.00
steinc16.txt	500	12,500	50.00
planar-n50	50	135	5.4
planar-n100	100	285	5.7
planar-n200	200	583	5.83
planar-n500	500	1,477	5.91

each rate of commodity instead of two. Table 5 describes the instances generated, including their numbers of vertices, edges, and the sizes of the commodity sets T.

For comparison, we have reimplemented the ACO algorithm described in [18] in the COMET programming language. For each problem instance, the three algorithms ACO, LS-SGA, and LS-R are executed 20 times each. Due to the high complexity of the problem, we set the time limit to 30 min for each execution. In total, we have 54 problem instances and 1,080 executions.

Results The experimental results are shown in Tables 6, 7 and 8. These tables have the same structure, which is described in what follows. The first column presents the instance name. Columns 2–5 present the results of the ACO algorithm [18], including the average, the minimal and the maximal of the best objective values found in 20 executions, and the average time for finding these best objective values. The same information for LS-SGA and LS-R are presented in columns 6–9 and columns 11–14. Column 10 compares the ACO and LS-SGA algorithms in the format a/b where a is the number of times the ACO algorithm found better solutions than the LS-SGA algorithm and b is the number of time the LS-SGA found better solutions than the ACO algorithm in 20 executions. Column 15 presents the same information as column 10 but for the comparison between the ACO and the LS-R algorithms. A comparison of the two algorithms in terms of box-and-whiskers plots (see their template presentation in Fig. 15) can be found in Figs. 20, 21, 22, 23, 24, and 25. Three consecutive bars present the results computed by the ACO, LS-SGA, and the LS-R algorithms on a given instance. The figures show that for each algorithm, the

Table 5 Description of instances

Index	Name	$\sharp V$	$\sharp E$	$\sharp T$
1	bl-wr2-wht2.10-50.rand.bb_com10_ins1	500	1,020	50
2	bl-wr2-wht2.10-50.rand.bb_com25_ins1	500	1,020	125
3	bl-wr2-wht2.10-50.rand.bb_com40_ins1	500	1,020	200
4	bl-wr2-wht2.10-50.rand.bb_com10_ins2	500	1,020	50
5	bl-wr2-wht2.10-50.rand.bb_com25_ins2	500	1,020	125
6	bl-wr2-wht2.10-50.rand.bb_com40_ins2	500	1,020	200
7	bl-wr2-wht2.10-50.sdeg.bb_com10_ins1	500	1,020	50

Table 5 (continued)

Index	Name	$\sharp V$	$\sharp E$	$\sharp T$
8	bl-wr2-wht2.10-50.sdeg.bb_com25_ins1	500	1,020	125
9	bl-wr2-wht2.10-50.sdeg.bb_com40_ins1	500	1,020	200
10	bl-wr2-wht2.10-50.sdeg.bb_com10_ins2	500	1,020	50
11	bl-wr2-wht2.10-50.sdeg.bb_com25_ins2	500	1,020	125
12	bl-wr2-wht2.10-50.sdeg.bb_com40_ins2	500	1,020	200
13	mesh15×15.bb_com10_ins1	225	420	22
14	mesh15×15.bb_com25_ins1	225	420	56
15	mesh15×15.bb_com40_ins1	225	420	90
16	mesh15×15.bb_com10_ins2	225	420	22
17	mesh15×15.bb_com25_ins2	225	420	56
18	mesh15×15.bb_com40_ins2	225	420	90
19	mesh25×25.bb_com10_ins1	625	1,200	62
20	mesh25×25.bb_com25_ins1	625	1,200	156
21	mesh25×25.bb_com40_ins1	625	1,200	250
22	mesh25×25.bb_com10_ins2	625	1,200	62
23	mesh25×25.bb_com25_ins2	625	1,200	156
24	mesh25×25.bb_com40_ins2	625	1,200	250
25	steinb4.txt_com10_ins1	50	100	5
26	steinb4.txt_com25_ins1	50	100	12
27	steinb4.txt_com40_ins1	50	100	20
28	steinb10.txt_com10_ins1	75	150	7
29	steinb10.txt_com25_ins1	75	150	18
30	steinb10.txt_com40_ins1	75	150	30
31	steinb16.txt_com10_ins1	100	200	10
32	steinb16.txt_com25_ins1	100	200	25
33	steinb16.txt_com40_ins1	100	200	40
34	steinc6.txt_com10_ins1	500	1,000	50
35	steinc6.txt_com25_ins1	500	1,000	125
36	steinc6.txt_com40_ins1	500	1,000	200
37	steinc11.txt_com10_ins1	500	2,500	50
38	steinc11.txt_com25_ins1	500	2,500	125
39	steinc11.txt_com40_ins1	500	2,500	200
40	steinc16.txt_com10_ins1	500	12,500	50
41	steinc16.txt_com25_ins1	500	12,500	125
42	steinc16.txt_com40_ins1	500	12,500	200
43	planar-n50.ins1.txt_com10_ins1	50	135	5
44	planar-n50.ins1.txt_com25_ins1	50	135	12
45	planar-n50.ins1.txt_com40_ins1	50	135	20
46	planar-n100.ins1.txt_com10_ins1	100	285	10
47	planar-n100.ins1.txt_com25_ins1	100	285	25
48	planar-n100.ins1.txt_com40_ins1	100	285	40
49	planar-n200.ins1.txt_com10_ins1	200	583	20
50	planar-n200.ins1.txt_com25_ins1	200	583	50
51	planar-n200.ins1.txt_com40_ins1	200	583	80
52	planar-n500.ins1.txt_com10_ins1	500	1,477	50
53	planar-n500.ins1.txt_com25_ins1	500	1,477	125
54	planar-n500.ins1.txt_com40_ins1	500	1,477	200

variance of the results among the 20 executions is small. It also shows that, in most instances, the solutions found by LS-SGA and LS-R are better than those found by the ACO algorithm.

Table 6 Experimental results of the first graphs set

Ins.	ACO				LS-SGA					LS-R				
	$\overline{f}$	m	M	$\bar{t}$	$\overline{f}$	m	M	$\bar{t}$	τ_1	$\overline{f}$	m	M	$\bar{t}$	τ_2
1	14.8	14	16	131.6	15.6	14	16	410.76	2/13	16	16	16	194.71	0/19
2	31.85	31	32	165.22	31.4	30	32	564.99	10/3	32	32	32	263.71	0/3
3	37.85	37	38	219.56	37.6	36	38	322.96	6/2	37.9	37	38	230.29	1/2
4	25.25	25	26	95.41	25.9	25	26	434.43	0/13	26	26	26	151.09	0/15
5	34.75	34	35	97.33	34.4	32	35	544.02	8/4	34.95	34	35	303.26	0/4
6	36.95	36	37	185.14	36.05	34	37	422.18	13/0	36.95	36	37	293.27	1/1
7	15.95	15	16	89.24	16.25	16	17	529.03	0/6	17	17	17	430.99	0/20
8	35.8	35	36	67.08	35.45	34	36	536.4	8/3	36	36	36	423.68	0/4
9	33.65	33	34	169.9	33.1	31	34	472.56	11/4	34	34	34	557.57	0/7
10	19.2	19	20	401.19	19.65	19	20	522.22	2/11	20	20	20	448.59	0/16
11	32.95	32	34	365.09	32.9	31	34	880.04	9/7	33.9	33	34	516.21	1/14
12	36.5	35	37	133	35.7	35	37	936.57	11/2	37	37	37	583.99	0/9
13	19.65	19	21	457.46	21.75	21	22	644.11	0/20	21.55	21	22	360.53	0/19
14	27.7	26	29	470.98	29.8	29	31	335.51	0/19	32	31	33	887.93	0/20
15	35.3	32	38	871.22	35.8	33	39	763.25	6/10	38.8	37	40	960.97	0/20
16	17.5	17	19	479.89	19.4	19	20	515.65	0/19	19.45	19	20	568.74	0/18
17	29.2	28	31	1,010.52	31.7	30	33	480.98	0/20	33.05	32	34	592.96	0/20
18	34	33	36	750.55	34.6	33	37	649.26	4/13	37.6	36	39	910.63	0/20
19	32.85	29	36	996.96	39.15	36	41	864.6	0/20	41	39	43	946.47	0/20
20	45	42	49	1,104.82	51.95	49	56	1,053.57	0/20	55.55	54	59	1,111.8	0/20
21	57.7	53	61	797.14	65.3	60	69	950.87	0/20	69.3	67	72	1,520.61	0/20
22	30.1	28	33	944.36	35.7	34	37	875.12	0/20	37.9	36	40	945.08	0/20
23	45.6	44	48	1,015.84	51.35	47	54	673.59	0/20	54.7	52	59	1,042.11	0/20
24	57.75	54	61	939.82	65.05	62	68	1,409.13	0/20	68.85	66	71	1,040.24	0/20

Table 7 Experimental results of the steiner graphs set

Ins.	ACO				LS-SGA					LS-R				
	$\overline{f}$	m	M	$\bar{t}$	$\overline{f}$	m	M	$\bar{t}$	τ_1	$\overline{f}$	m	M	$\bar{t}$	τ_2
25	5	5	5	0.01	5	5	5	1.12	0/0	5	5	5	1.09	0/0
26	12	12	12	0.44	12	12	12	1.22	0/0	12	12	12	1.4	0/0
27	20	20	20	51.11	20	20	20	5.45	0/0	19.9	19	20	2.8	2/0
28	7	7	7	0.02	7	7	7	1.35	0/0	7	7	7	1.16	0/0
29	17.85	17	18	96.48	18	18	18	13.42	0/3	18	18	18	5.2	0/3
30	24.35	23	26	242.04	26.25	26	27	682.84	0/20	27.3	27	28	505.22	0/20
31	10	10	10	0.25	10	10	10	1.46	0/0	10	10	10	1.52	0/0
32	24.35	24	25	364.99	25	25	25	93.53	0/13	25	25	25	8.86	0/13
33	32.45	32	34	658.25	34.1	33	36	747.34	0/19	35.95	35	37	646.19	0/20
34	49.1	47	50	572.75	50	50	50	184.44	0/12	50	50	50	240.75	0/12
35	89.9	85	94	728.76	92.2	87	100	734.88	4/12	104.95	102	108	1,370.88	0/20
36	109.8	106	117	924.1	112.05	106	118	971.4	2/14	121.4	119	125	1,372.37	0/20
37	50	50	50	23.59	50	50	50	42.19	0/0	50	50	50	37.64	0/0
38	123.3	122	125	521.8	125	125	125	128.49	0/17	125	125	125	262.38	0/17
39	194.25	190	198	494.64	200	200	200	395.54	0/20	200	200	200	473.81	0/20
40	50	50	50	6.89	50	50	50	55.12	0/0	50	50	50	46.01	0/0
41	125	125	125	17.13	125	125	125	194.36	0/0	125	125	125	113.83	0/0
42	200	200	200	45.32	200	200	200	366.69	0/0	200	200	200	183.32	0/0

Table 8 Experimental results of the planar graphs set

Ins.	ACO				LS-SGA					LS-R				
	$\overline{f}$	m	M	$\bar{t}$	$\overline{f}$	m	M	$\bar{t}$	τ_1	$\overline{f}$	m	M	$\bar{t}$	τ_2
43	5	5	5	0.03	5	5	5	0.86	0/0	5	5	5	0.8	0/0
44	12	12	12	0.16	12	12	12	0.96	0/0	12	12	12	0.97	0/0
45	20	20	20	36.38	20	20	20	25.12	0/0	19.9	19	20	31.18	2/0
46	10	10	10	0.12	10	10	10	1.14	0/0	10	10	10	1.07	0/0
47	25	25	25	20.22	25	25	25	7.05	0/0	25	25	25	5.33	0/0
48	34	33	36	680.72	35.3	34	37	813.56	0/16	36	35	37	698.88	0/18
49	20	20	20	13.46	20	20	20	4.06	0/0	20	20	20	5.23	0/0
50	41.8	39	43	889.07	44.85	43	47	988.81	0/20	45.95	45	48	853.18	0/20
51	49.35	47	51	790.65	53.35	51	56	1,033.97	0/19	55.7	54	58	901.74	0/20
52	44.95	42	47	1,100.41	49.95	49	50	484.84	0/20	50	50	50	309.24	0/20
53	60.95	57	65	954.35	73.85	70	77	1,345.74	0/20	78.2	77	80	1,044.03	0/20
54	82.85	78	86	1235.13	93.95	91	99	1,366.27	0/20	100.15	97	102	1,455.43	0/20

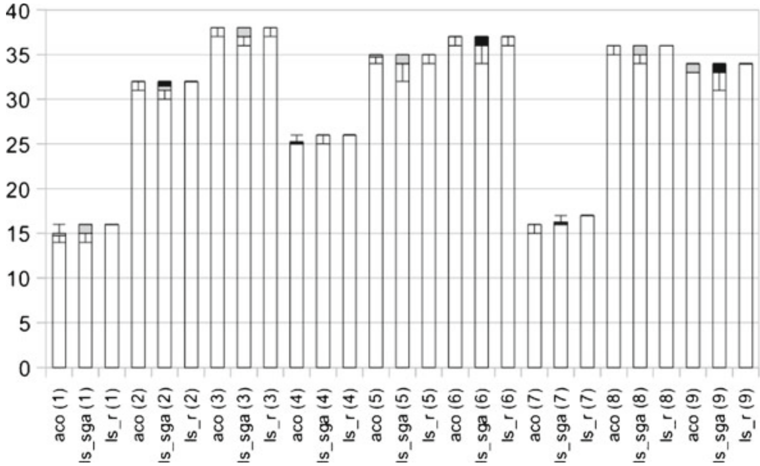


Fig. 20 Comparison between the ACO, LS-SGA and LS-R algorithms (part I)

The experiments results show that on average, the LS-R algorithm is better than two other algorithms. The LS-SGA algorithm is better than the ACO algorithm. The LS-SGA finds better solutions than the ACO algorithm in 534 out of 1,080 executions while the ACO algorithm finds better solutions in 96 out of 1,080 executions. LS-R finds better solutions than ACO in 614 out of 1,080 executions while the ACO algorithm finds better solutions than LS-R in 7 out of 1,080 executions.

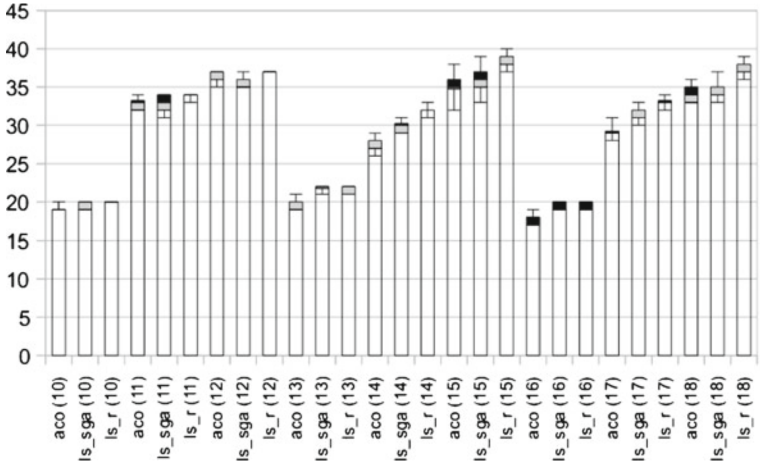


Fig. 21 Comparison between the ACO, LS-SGA and LS-R algorithms (part II)

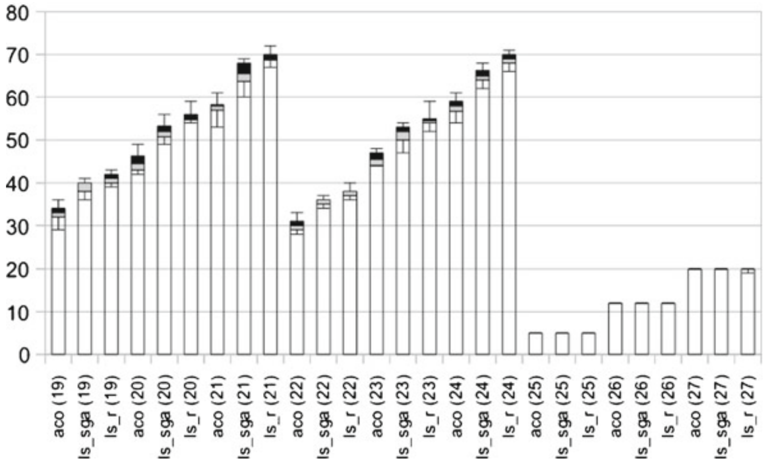


Fig. 22 Comparison between the ACO, LS-SGA and LS-R algorithms (part III)

6.3 The routing and wavelength assignment problem with delay side constraint (RWA-D)

The last application demonstrates that `VarPath` variables of `LS`(`Graph`) and `var{int}` of `COMET` can easily be combined.

6.3.1 Problem statement

Given an undirected weighted graph $G = (V, E)$, each edge e of G has cost $c(e)$ (e.g., the delay in traversing e). Suppose given a set of connection requests

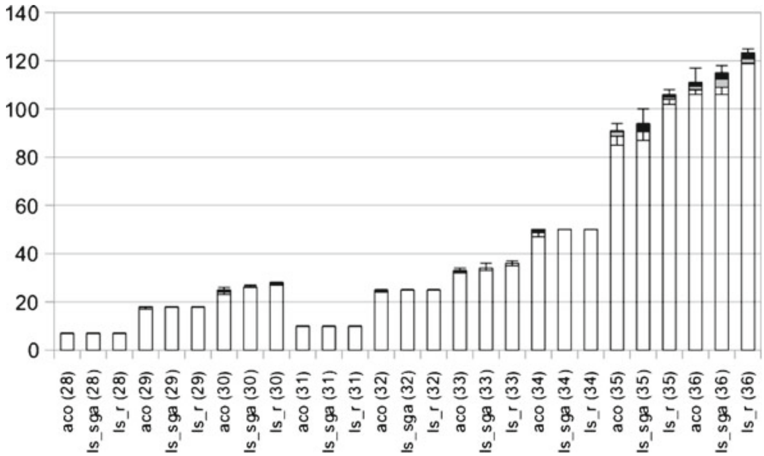


Fig. 23 Comparison between the ACO, LS-SGA and LS-R algorithms (part IV)

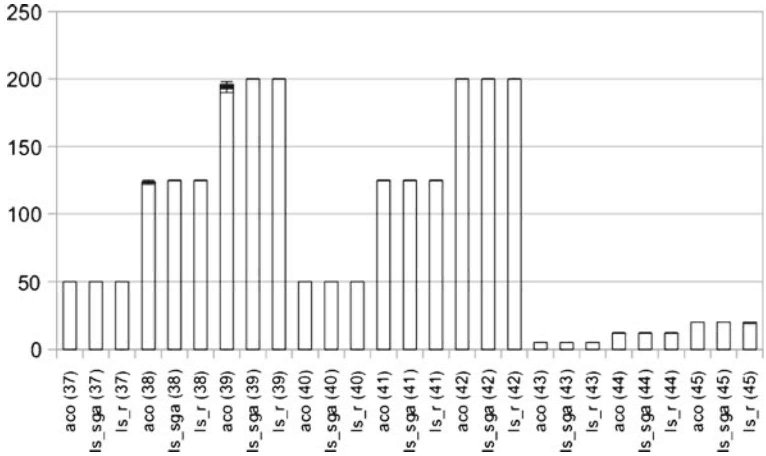


Fig. 24 Comparison between the ACO, LS-SGA and LS-R algorithms (part V)

$R = \{\langle s_1, t_1\rangle, \langle s_2, t_2\rangle, ..., \langle s_k, t_k\rangle\}$ and a value D. The RWA-D problem consists of finding routes p_i from s_i to t_i and their wavelengths for all $i = 1, 2, ..., k$ such that:

1. the wavelengths of p_i and p_j are different if they have common edges, $\forall i \neq j \in \{1, 2, ..., k\}$ (wavelength constraint),
2. $\sum_{e \in p_i} c(e) \leq D, \forall i = 1, 2, ..., k$ (delay constraint),
3. the number of different wavelengths is minimized (objective function).

6.3.2 The model

The idea of the proposed algorithm is simple. We iteratively perform a local search algorithm for finding a feasible solution to the RWA-D problem with W wavelengths ($W = 1, 2, 3, ...$) until the first feasible solution is discovered.

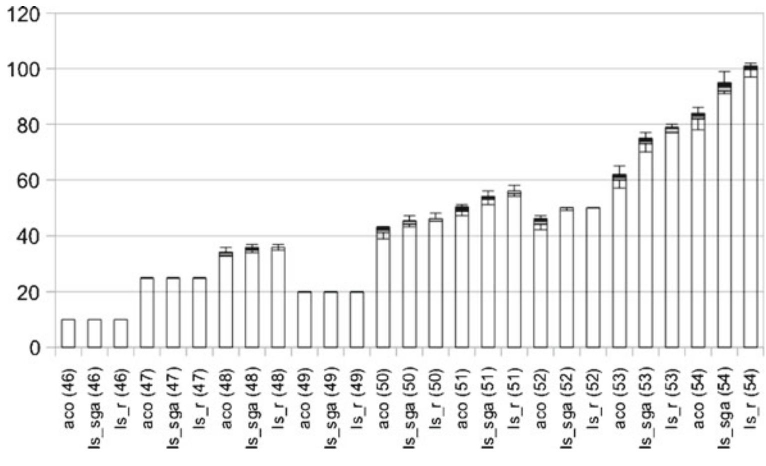


Fig. 25 Comparison between the ACO, LS-SGA and LS-R algorithms (part VI)

The model is given in Fig. 26. Lines 4–10 initialize all `VarPath vps[i]` from `s[i]` to `t[i]` with the shortest version. Line 11 initializes an array `vw` where `vw[i]` stores the wavelength value for the path `vps[i]`. The search starts with one wavelength (see line 14). At each step, we try to find a feasible solution to the RWA-D problem by a `localsearch` procedure (line 16). The search terminates (line 17) if a feasible solution to the RWA-D problem is discovered, otherwise, we increase `W` by one (line 19).

The `localsearch` procedure described in Fig. 27 receives an array of `VarPath`, a value `W` of the number of wavelengths, and local search parameters `maxIt` and `maxT` as input. Line 2 creates a `Solver<LSGraph> ls` and lines 4–6 post all `VarPath` to it. Line 8 initializes an array `var{int} xw`, where `xw[i]` represents the wavelength assigned to the path `vps[i]` and is initialized with the value `vw[i]`. The domain of `xw[i]` is `1..W`. Line 10 initializes a `ConstraintSystem <LSGraph> CS`. The first constraint of the RWA-D problem is stated and posted in line 12. Lines 14 and 15 state and post all side constraints (the delay constraint) to `CS` and line 17 closes the constraint system `CS`. Line 19 groups all variables `vps`, `xw`, and the constraint `CS`, into a model `mod`. Line 20 creates a search component which will be given in detail in Fig. 28. Lines 22 and 23 set parameters for the search and line 25 performs the search. The value of `xw` is stored in `vw` for the next iteration (see lines 27 and 28): all paths `vps[i]` and their wavelengths `xw[i]` are conserved for the next `localsearch`. The `localsearch` returns `true` if a feasible solution to the RWA-D problem is discovered (lines 30–32).

The search component is given in Fig. 28. It extends the `TabuSearch<LSGraph>` and receives `Lmax` (line 3) as parameters for the solution initialization when restarting the tabu search. The `restartSolution` is overriden (lines 13–24) in which we initialize the value for the `VarPath vps[i]` with the shortest version if its

```
1   void minRWA(int maxIt, float maxT){
2     range Size = 1..ca;

4     vps = new VarPath[Size];
5     // init VarPaths with the shortest version
6     LSGraphPath p(g);
7     forall(i in Size){
8       p.dijkstra(g,s[i],t[i]);
9       vps[i] = new VarPath(g,p);
10    }
11    vw = new int[Size] = 1; // the wavelengths of all paths are
          all initialized by 1

13    bool finished = false;
14    int W = 1;
15    while(!finished){// iteratively search with 1, 2, ...
          wavelengths until a feasible solution is found
16      if(localsearch(vps,W,maxIt,maxT)){
17        finished = true;
18      }else{
19        W++;
20      }
21    }
22  }
```

Fig. 26 Model for the RWA-D problem

```
bool localsearch(VarPath[] vps, int W, int maxIt, float maxT){ //
    try to find a feasible solution with W wavelengths
  Solver<LSGraph>  ls(); // create a new solver

  forall(i in vps.rng()){
    ls.post(vps[i]);
  }

  xw = new var{int}[i in vps.rng()](ls,1..W) := vw[i]; // initial
      wavelengths of paths (decision variables) using the
      values computed at the previous iteration. At this point,
      the domains of wavelengths are extended from 1..W-1 (at
      the previous iteration) to 1..W

  ConstraintSystem<LSGraph> CS(ls); // constraint system

  CS.post(AllDistinctLightPaths(vps,xw)); // posting the
      constraint specifying that two paths vps[i] and vps[j]
      sharing a link must have different wavelengths xw[i] and
      xw[j].

  forall(i in vps.rng())
    CS.post(PathCostOnEdges(vps[i]) <= Lmax); // posting the
        delay constraint

  CS.close();

  Model<LSGraph> mod(vps,xw,CS); // encapsulate variables,
      constraints into a model object
  RWASearch se(mod,Lmax); // create the search object which
      extends the generic built-in tabu search component

  se.setMaxIter(maxIt);
  se.setMaxTime(maxT);

  se.search(); // perform the local search

  forall(i in xw.rng())
    vw[i] = xw[i]; // store the wavelengths of paths computed
        for the next search iteration with higher number of
        wavelengths

  if(CS.violations() == 0){
    return true;
  }
  return false;
}
```

Fig. 27 The local search procedure for the RWA-D problem

cost is greater than `Lmax`. This aims at quickly satisfying the delay constraint. The `initSolution` is also overriden, which does nothing in order not to change the value of the variables computed in the previous step of the search. The search explores two neighborhoods (lines 7–10) (see [53] for details about these neighborhood explorations).

6.3.3 Naive greedy algorithm

As far as we know, the RWA-D problem has not been considered before. In order to assess the efficiency of our local search, we implement a simple greedy heuristic algorithm for the RWA-D problem (see Algorithm 8). The main idea of

```
class RWASearch extends TabuSearch<LSGraph>{
   float   _Lmax;
   RWASearch(Model<LSGraph> mod, float Lmax):
        TabuSearch<LSGraph>(mod){
     _Lmax = Lmax;
   }

   void exploreNeighborhood(Neighborhood N){
     exploreTabuMinMultiStageAssign(N,true); // explore the
          neighborhood based on changing the wavelengths
     exploreTabuMinMultiStageReplace1Move1VarPath(N,true); //
          explore the neighborhood based on changing the paths
   }


   void restartSolution(){
     // init paths with shortest versions for paths whose current
          cost greater than Lmax
     forall(k in _vps.rng()){
        VarPath vp = _vps[k];
        float d = sum(e in vp.getEdges())(e.weight());
        if(d > _Lmax){ // update the path vp if its delay is greater
            than Lmax
          LSGraphPath pa(vp.getLUB()); // initialize a path object
          pa.dijkstra(vp.getSource(),vp.getDestination()); //
               compute the shortest path from the source of vp to
               the destination of vp
          vp.assign(pa); // assign the shortest path to vp
        }
     }
   }
   void initSolution(){// do nothing, use the values a computed at
        the previous iteration
   }
}
```

Fig. 28 The search component

this greedy heuristic is to find the shortest path[10] for each connection request and assigns a wavelength to this connection request in a greedy way without violating the wavelength constraint. Variable *Sol* in line 1 represents the set of paths under construction. Variable W (line 2) contains the set of wavelengths used for the paths which have already been constructed. Variable *nb Wavelengths* (line 3) is the number of wavelengths used. For each connection request $\langle s_i, t_i \rangle$ (line 4), we assign the shortest path P_i to it (line 6). Variable W_i in line 5 represents the candidate wavelengths for P_i. Lines 7–9 remove all impossible wavelengths for P_i from W_i. If no wavelength already used is possible for P_i (line 10), then we have to find a new wavelength w_i for P_i (lines 11 and 12). If the candidate set W_i is not null, we select

[10]The shortest path best ensures satisfaction of the delay constraint.

randomly a wavelength from W_i and assign it to P_i (line 15). Lines 16 and 17 update the solution.

Algorithm 8: RWADGreedy

Input: $G = (V, E), T = \{\langle s_i, t_i \rangle\}$ representing connection requests
Output: Number of wavelengths used for satisfying all requests
1 $Sol \leftarrow \emptyset$;
2 $W \leftarrow \emptyset$;
3 $nbWavelengths \leftarrow 0$;
4 **foreach** $\langle s_i, t_i \rangle \in T$ **do**
5     $W_i \leftarrow W$;
6     $P_i \leftarrow$ shortest path from s_i to t_i in G;
7   **foreach** $P_j \in Sol$ **do**
8   **if** *P_i and P_j have common edges* **then**
9   Remove from W_i the wavelength assigned for P_j;
10   **if** $W_i = \emptyset$ **then**
11         $nbWavelengths \leftarrow nbWavelengths + 1$;
12         $w_i \leftarrow nbWavelengths$;
13         $W \leftarrow W \cup \{w_i\}$;
14   **else**
15         $w_i \leftarrow$ select random an element of W_i;
16   Assign the wavelength w_i to the path P_i;
17     $Sol \leftarrow Sol \cup \{P_i\}$;
18   **return** $nbWavelengths$;

6.3.4 Experiments

We compare our local search model with the naive greedy algorithm described in Algorithm 8 (multistart version with 1,000 different orders of $\langle s_i, t_i \rangle$ to be considered).

The two algorithms have been tried on different instances (graphs from 16 nodes and 33 edges to 100 nodes and 261 edges and with 10, 20, and 50 connection requests for each graph). Due to the complexity of the problem, we set the number of iterations for the tabu search (the value of `maxIt` in line 16 of Fig. 26) to 200. For each problem instance, the model is executed 20 times. From our preliminary results, we set the length of the tabu lists *tbl* to 5 and the value of *maxStable* to 20.

Table 9 shows the experimental results. Column 2 presents the objective values found by the naive greedy algorithm. Columns 3–6 show the minimal, the maximal, and the average of the best objective value found, and the average execution time (in seconds) over 20 runs. The experimental results show that the local search gives better solutions than the naive greedy algorithm. Especially when the number of connection requests increases (i.e., with 50 connection requests), the results found by the local search are two or three times better than those found by the naive greedy algorithm. We can see that the number of wavelengths used increases when the number of connection requests increases. Given a number of connection requests, if the size of the graph increases, then the number of wavelengths used decreases due to the fact that on larger graphs, each link is shared by fewer paths of the solution found by the local search and if two paths are completely edge-disjoint, they can be assigned the same wavelength. For instance, with 50 connection requests, on the graph of 100 vertices, we needed to use only four wavelengths (line 18), while on the graph of 16 vertices, we had to use eight wavelengths (line 6).

Table 9 Experimental results for the RWA-D problem

Instances	Greedy	$f\_\min$	$f\_\max$	$\overline{f}$	$\bar{t}$
arpanet_ca10.ins1	5	2	2	2	2.15
arpanet_ca20.ins1	9	6	7	6.05	11.36
arpanet_ca50.ins1	16	8	9	8.3	68.35
grid_ext_4×4_ca10.ins1	3	2	2	2	1.62
grid_ext_4×4_ca20.ins1	9	4	5	4.3	7.16
grid_ext_4×4_ca50.ins1	24	8	10	8.55	55.04
grid_ext_5×5_ca10.ins1	4	2	2	2	2.43
grid_ext_5×5_ca20.ins1	7	2	3	2.95	6.21
grid_ext_5×5_ca50.ins1	21	5	8	6.45	54.15
grid_ext_6×6_ca10.ins1	3	2	2	2	2.58
grid_ext_6×6_ca20.ins1	4	2	3	2.1	6.66
grid_ext_6×6_ca50.ins1	20	5	6	5.35	58.99
grid_ext_8×8_ca10.ins1	4	2	2	2	3.81
grid_ext_8×8_ca20.ins1	9	3	4	3.3	14.61
grid_ext_8×8_ca50.ins1	11	4	6	5.15	73.97
grid_ext_10×10_ca10.ins1	4	2	4	2.7	9.71
grid_ext_10×10_ca20.ins1	8	3	5	4	24.22
grid_ext_10×10_ca50.ins1	13	4	6	4.75	105.2

Once again, in the above model, we notice that it is easy to state and post various built-in `COMET` constraints over `var{int}` to the graph constraint system `CS`, which shows the flexibility and compositionality of the framework.

7 Conclusion

This paper considered constrained optimum trees and paths (COT/COP) problems which arise in many real-life applications. It proposed a domain-specific constraint-based local search (CBLS) framework (called `LS(Graph)`) for solving COT/COP applications, enabling models to be high level, compositional, and extensible, and allowing for a clear separation between model and search. The key technical contribution to support the COP framework is a novel neighborhood based on a rooted spanning tree that implicitly defines a path between the source and the target and its neighbors, and provides an efficient data structure for differentiation. The paper proved that the neighborhood obtained by swapping edges in this tree is connected and presented a larger neighborhood involving multiple independent moves. The `LS(Graph)` framework, implemented in `COMET`, was applied to the quorumcast routing problem, the edge-disjoint paths problem, and the routing and wavelength assignment problem with side constraints on optical networks. Computational results showed the potential significance of the approach, both from a modeling and a computational standpoint.

Our future work will focus on the construction of a generic constraint programming (CP) framework and a hybrid system combining CP and CBLS for modeling and solving COT/COP problems.

Acknowledgements We would like to thank the reviewers for their helpful comments and suggestions. This research is partially supported by the Interuniversity Attraction Poles Programme (Belgian State, Belgian Science Policy) and the FRFC project 2.4504.10 of the Belgian FNRS (National Fund for Scientific Research).

Appendix

This appendix presents the proofs of above propositions.

Proof of Proposition 1

Proof The proof is divided into two phases:

1. We show that if the conditions (1) and (2) are satisfied, then $path_{tr'}(s) \neq path_{tr}(s)$.
   The condition (1) ensures that the selected edge e' satisfying the condition (2) always exist. It is easy to see that e' belongs to $path_{tr}(s)$ and this edge is removed from that path after taking $rep(tr, e', e)$. That means e' does not belong to $path_{tr'}(s)$. Hence, $path_{tr'}(s) \neq path_{tr}(s)$.
2. We now show that if $path_{tr'}(s) \neq path_{tr}(s)$, then the conditions (1) and (2) are satisfied.
   We prove this by refutation. Suppose that $su = sv$. We denote $r = su = sv$ and $r_1 = nca_{tr}(u, v)$. Because $r\ Dom_{tr}\ u$ and $r\ Dom_{tr}\ v$, we have $r\ Dom_{tr}\ nca_{tr}(u, v) = r_1$ (3).
   We now show that $path_{tr}(u, v)$ does not contain any edges that belong to $path_{tr}(s)$.

   – If $path_{tr}(u, r_1)$ contains an edge (x, y) (where $y = fa_{tr}(x)$) of $path_{tr}(s)$, then we have $x\ Dom_{tr}\ u$ and $x\ Dom_{tr}\ s$. Hence, $x\ Dom_{tr}\ nca_t r(s, u) = r$ (4). Otherwise, $(x, y) \in path_{tr}(u, r_1)$, so $r_1\ Dom_{tr}\ y$, and we have $r\ Dom_{tr}\ y$ (because $r\ Dom_{tr}\ r_1$) that means $r\ Dom_{tr}\ fa_{tr}(x)$ (5). We see that (4) conflicts with (5). From that, we have the fact that $path_{tr}(u, r_1)$ does not contain any edges of $path_{tr}(s)$.
   – In the same way we can show that $path_{tr}(v, r_1)$ does not contain any edges of $path_{tr}(s)$.

   From that, we have $path_{tr}(u, v)$ which is actually the concatenation of $path_{tr}(u, r_1)$ and $path_{tr}(v, r_1)$ does not contain any edges of $path_{tr}(s)$.
   e' is a replacable edge that belongs to $path_{tr}(u, v)$. So after the replacement is taken, no edge of $path_{tr}(s)$ is removed. Hence, the path from s to the root of the tree does not change, that means $path_{tr'}(s) = path_{tr}(s)$ (this conflicts with the hypothesis that $path_{tr'}(s) \neq path_{tr}(s)$). So we have $su \neq sv$.
   We now suppose that e' (the edge to be removed) does not belong to $path_{tr}(su, sv)$. We can see easily that the path from u to v on tr ($path_{tr}(u, v)$) is composed by the path from u to su, the path from su to sv and the path from sv to v on tr. So after the replacement is taken, no edge of $path_{tr}(s)$ is removed. Hence, $path_{tr'}(s) = path_{tr}(s)$ (this conflicts with the hypothesis). So we have $e' \in path_{tr}(su, sv)$. □

Proof of Proposition 2

Proof The proposition is proved by showing how to generate that instance tr^k. This can be done by Algorithm 9. The idea is to travel the sequence of nodes of $\mathcal{P}$ on the current tree tr. Whenever we get stuck (we cannot go from the current node x to the next node y of $\mathcal{P}$ by an edge (x, y) on tr because (x, y) is not in tr), we change tr by replacing (x, y) by a replacable edge of (x, y) that is not traversed. The edge (x, y) in line 7 is a *replacing* edge of tr because this edge is not in tr but it is an edge of g. Line 8 chooses a *replacable* edge eo of ei that is not in S. This choice is always successfully done because the set of *replacable* edges of ei that are not in S is not null (at least an edge $(y, fa_{tr}(y))$ belongs to this set). Line 9 performs the move that replaces the edge eo by the edge ei on tr. So Algorithm 9 always terminates and returns a rooted spanning tree tr inducing $\mathcal{P}$. Variable S (line 1) stores the set of traversed edges.

Algorithm 9: Moves

Input: An instance tr^0 of *RST* on (g, s, t) and a path $\mathcal{P}$ on g, s = firstNode($\mathcal{P}$), t = lastNode($\mathcal{P}$)
Output: A tree inducing $\mathcal{P}$ computed by taking $k \leq l$ basic moves (l is the length of $\mathcal{P}$)

```
1  S ← ∅;
2  tr ← tr^0;
3  x ← firstNode(P);
4  while x ≠ lastNode(P) do
5      y ← nextNode(x, P);
6      if (x, y) ∉ E(tr) then
7          ei ← (x, y);
8          eo ← replacable edge of ei that is not in S;
9          tr ← replaceEdge(tr, eo, ei);
10     S ← S ∪ {(x, y)};
11     x ← y ;
12 return tr;
```

□

Proof of Proposition 3

Proof All sequences of these basic moves are executable and the final results have the same set of edges $E(tr) \setminus \{eo_1, eo_2, ..., eo_k\} \cup \{ei_1, ei_2, ..., ei_k\}$. Thus the result trees of all execution sequences are the same. □

Proof of Proposition 4

Proof Let $x = nca_{tr}(u_1, v_2)$, $sv_1 = nca_{tr}(s, v_1)$, $su_1 = nca_{tr}(s, u_1)$, $sv_2 = nca_{tr}(s, v_2)$, $su_2 = nca_{tr}(s, u_2)$. Because $su_1\ Dom_{tr}\ sv_1$, $sv_2\ Dom_{tr}\ su_1$, $su_2\ Dom_{tr}\ sv_2$, e'_1 belongs to $path_{tr}(sv_1, su_1)$ and e'_2 belongs to $path_{tr}(sv_2, su_2)$, we have $e'_1 \neq e'_2$. Otherwise, $e_2\ Dom_{(tr)}\ e_1$ and these two edges are not in tr, whereas e'_1 and e'_2 are in tr. So e_1, e'_1, e_2, e'_2 are all different.We will show that the sequence: $rep(tr, e'_1, e_1), rep(tr, e'_2, e_2)$ is feasible as follows:

Suppose that v'_1, u'_1 are endpoints of e'_1 such that $u'_1 = fa_{tr}(v'_1)$ and let $tr_1 = rep(tr, e'_1, e_1)$. We have that:

(1) $su_1\ Dom_{tr}\ u'_1$
(2) $su_2\ Dom_{tr}\ u'_1$
(3) $sv_2\ Dom_{tr}\ u'_1$

It is straightfoward to find that $\overline{T_{tr}}(v'_1)$ does not change after taking $rep(tr, e'_1, e_1)$. We can also find that u_1, v_2, u_2 must belong to $\overline{T_{tr}}(v'_1)$ (if not, u_1, v_2, u_2 must belong to $T_{tr}(v'_1)$, thus $nca_{tr}(s, u_1)$, $nca_{tr}(s, v_2)$, $nca_{tr}(s, u_2)$ are dominated by v'_1, hence this conflicts with (1)–(3)). Thus $nca_{tr_1}(s, u_2) = nca_{tr}(s, u_2) = su_2$

and $nca_{tr_1}(u_1, v_2) = nca_{tr}(u_1, v_2) = x$. Moreover, from the Property 1, we have $nca_{tr_1}(s, v_2) = nca_{tr_1}(u_1, v_2) = x$ (4).

Due to the fact that sv_2 Dom_{tr_1} su_1 and su_1 Dom_{tr_1} u_1, we have sv_2 Dom_{tr_1} u_1 (5). From the fact that sv_2 Dom_{tr_1} v_2 and sv_2 Dom_{tr_1} u_1, we have sv_2 Dom_{tr_1} $nca_{tr_1}(u_1, v_2) = x$ (6). We have e'_2 belongs to $path_{tr}(sv_2, su_2)$ (7). From (6) and (7) we have that e'_2 belongs to $path_{tr_1}(x, su_2)$ (8). From (4) and (8), we have $e'_2 \in path_{tr^1}(nca_{tr^1}(s, v_2), nca_{tr^1}(s, u_2))$. That means e'_2 is still a *preffered replacable* edge of e_2 on tr_1. So the sequence $rep(tr, e'_1, e_1)$, $rep(tr, e'_2, e_2)$ is feasible.

In similar way, we can prove that the sequence $rep(tr, e'_2, e_2)$, $rep(tr, e'_1, e_1)$ is also feasible. Hence, two basic moves $rep(tr, e'_1, e_1)$, $rep(tr, e'_2, e_2)$ are independent. □

References

1. Acar, U., Blelloch, G., Harper, R., Vittes, J., & Woo, S. (2004). An experimental analysis of change propagation in dynamic trees. In *Proc 15th SODA* (pp. 524–533).
2. Ahuja, R. K., Magnanti, T. L., & Orlin, J. B. (1993). *Network flows: Theory, algorithms, and applications*. Englewood Cliffs: Prentice Hall.
3. Ahuja, R. K., Orlinb, J. B., & Sharma, D. (2003). A composite very large-scale neighborhood structure for the capacitated minimum spanning tree problem. *Operations Research Letters, 31*, 185–194.
4. Ali, M., Ramamurthy, B., & Deogun, J. (1999). Genetic algorithm for routing in wdm optical networks with power considerations. Part I: The unicast case. In *Proceedings of the 8th IEEE ICCCN'99* (pp. 335–340). Boston-Natick, MA, USA.
5. Ali, M., Ramamurthy, B., & Deogun, J. (2000). Routing and wavelength assignment with power considerations in optical networks. *Computer Networks, 32*, 539–555.
6. Alstrup, S., Holm, J., Lichtenberg, K. D., & Thorup, M. (2005). Maintaining information in fully dynamic trees with top trees. *ACM Transactions on Algorithms, 1*(2), 243–264. doi:10.1145/1103963.1103966.
7. Andrade, R., Lucena, A., & Maculan, N. (2006). Using lagrangian dual information to generate degree constrained spanning trees. *Discrete Applied Mathematics, 154*(5), 703–717.
8. Awerbuch, B., Gawlick, R., Leighton, T., & Rabani, Y. (1994). On-line admission control and circuit routing for high performance computing and communication. In *35th IEEE symposium on foundations of computer science (FOCS1994)* (pp. 412–423).
9. Badeau, P., Gendreau, M., Guertin, F., Potvin, J. Y., & Taillard, E. (1997). A parallel tabu search heuristic for the vehicle routing problem with time windows. *Transportation Research - C, 5*, 109–122.
10. Banerjee, D., Mehta, V., & Pandey, S. (2004). A genetic algorithm approach for solving the routing and wavelength assignment problem in wdm networks. In *3rd IEEE/IEE international conference on networking, ICN'2004* (pp. 70–78). Paris.
11. Banerjee, D., & Mukherjee, B. (2000). Wavelength-routed optical networks: Linear formulation, resource budgeting tradeoffs, and a reconfiguration study. *IEEE/ACM Transactions on Networking, 8*, 598–607.
12. Banerjee, S., Yoo, J., & Chen, C. (1997). Design of wavelength routed optical networks for packet switched traffic. *IEEE Journal of Lightware Technology, 15*(9), 1636–1646.
13. Baveja, A., & Srinivasan, A. (2000). Approximation algorithms for disjoint paths and related routing and packing problems. *Mathematics of Operations Research, 25*(2), 255–280.
14. Beasley, J. E. (2012). *Or-library*. http://people.brunel.ac.uk/~mastjjb/jeb/info.html. Accessed 15 Nov 2009.
15. Beasley, J. E., & Christofides, N. (1989). An algorithm for the resource constrained shortest path problem. *Network, 19*, 379–394.
16. Bender, M. A., Farach-Colton, M., Pemmasani, G., Skiena, S., & Sumazin, P. (2005). Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms, 57*, 75–94.
17. Berkman, O., & Vishkin, U. (1993). Recursive star-tree parallel data structure. *SIAM Journal On Computing, 22*(2), 221–242.

18. Blesa, M., & Blum, C. (2007). Finding edge-disjoint paths in networks: An ant colony optimization algorithm. *Journal of Mathematical Modelling and Algorithms, 6*(3), 361–391.
19. Blum, C. (2006). A new hybrid evolutionary algorithm for the huge k-cardinality tree problem. In *Proceedings of the 8th annual conference on genetic and evolutionary computation* (pp. 515–522).
20. Blum, C., & Blesa, M. (2005). New metaheuristic approaches for the edge-weighted k-cardinality tree problem. *Computers and Operations Research, 32*(6), 1355–1377.
21. Bui, T., & Sundarraj, G. (2004). Ant system for the k-cardinality tree problem. In K. Deb, et al. (Ed.), *Proceedings of the genetic and evolutionary computation conference (GECCO 2004)* (Vol. 3102, pp. 36–47).
22. Chekuri, C., & Khanna, S. (2003). Edge disjoint paths revisited. In *Proceedings of the 14th ACM-SIAM symposium on discrete algorithms (SODA2003)* (pp. 628–637).
23. Chen, C., & Banerjee, S. (1996). A new model for optimal routing and wavelength assignment in wavelength division multiplexed optical networks. In *INFOCOM 1996* (pp. 164–171).
24. Cheung, S. Y., & Kumar, A. (1994). Efficient quorumcast routing algorithms. In *Proceedings of INFOCOM'94* (pp. 840–847).
25. Chimani, M., Kandyba, M., Ljubic, I., & Mutzel, P. (2009). Obtaining optimal k-cardinality trees fast. *ACM Journal of Experimental Algorithmics, 14*(2), 5.1–5.23.
26. Chlamtac, I., Ganz, A., & Karmi, G. (1992). Lightpath communications: An approach to high bandwidth optical WANS. *IEEE Transactions on Communications, 40*(7), 1171–1182.
27. Clímaco, J. C. N., Craveirinha, J. M. F., & Pascoal, M. M. B. (2003). A bicriterion approach for routing problems in multimedia networks. *Networks, 41*, 206–220.
28. de Aragão, M., Uchoa, E., & Werneck, R. (2001). Dual heuristics on the exact solution of large Steiner problems. In *Proceedings of the Brazilian symposium on graphs, algorithms and combinatorics GRACO'01*. Fortaleza.
29. Du, B., Gu, J., Tsang, D., & Wang, W. (1996). Quorumcast routing by multispace search. In *Proceedings of IEEE Globecom1996* (pp. 1069–1073).
30. Dumitrescu, I., & Boland, N. (2003). Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem. *Networks, 42*, 135–153.
31. Dutta, R., & Rouskas, G. N. (2000). A survey of virtual topology design algorithms for wavelength routed optical networks. *Optical Networks, 1*(1), 73–89.
32. Fischer, T. (2007). Improved local search for large optimum communication spanning tree problems. In *MIC'2007—7th metaheuristics international conference*.
33. Frederickson, G. N. (1997). A data structure for dynamically maintaining rooted trees. *Journal of Algorithms, 24*(1), 37–65.
34. Funke, B., Grünert, T., & Irnich, S. (2005). Local search for vehicle routing and scheduling problems: Review and conceptual integration. *Journal of Heuristics, 11*(4), 267–306.
35. Gruber, M., van Hemert, J., & Raidl, G. (2006). Neighborhood searches for the bounded diameter minimum spanning tree problem embedded in a VNS, EA, and ACO. In *Proceedings of the genetic and evolutionary computation conference* (pp. 1187–1194).
36. Henzinger, M. R., & King, V. (1999). Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM, 46*(4), 502–516.
37. Ho, V., Francois, P., Deville, Y., Pham, D., & Bonaventure, O. (2010). Using local search for traffic engineering in switched ethernet networks. In *Proceedings of 22nd international teletraffic congress (ITC-22)*. Amsterdam, Netherlands.
38. Hyytia, E. (2004). *Resource allocation and performance analysis problems in optical networks.* Ph.d. thesis, Dpt. of Electrical and Communications Engineering, Helsinki University of Technology, Helsinki, Sweden.
39. Jaumard, B., Meyer, C., & Yu, X. (2006). How much wavelength conversion allows a reduction in the blocking rate? *Journal of Optical Networking, 5*(12), 881–900.
40. Jaumard, B., Meyer, C., & Thiongane, B. (2007). Comparison of ILP formulations for the rwa problem. *Optical Switching and Networking, 4*, 157–172.
41. Kanellakis, P. C., & Papadimitriou, C. H. (1980). Local search for the asymmetric traveling salesman problem. *Operations Research, 28*(5), 1086–1099.
42. Kleinberg, J. (1996). *Approximation algorithms for disjoint-paths problems*. PhD thesis. Cambridge: MIT Press.
43. Kolliopoulos, S. G., & Stein, C. (2004). Approximating disjoint-path problems using packing integer programs. *Mathematical Programming, 99*(1), 63–87.
44. Kolman, P., & Scheideler, C. (2001). Simple on-line algorithms for the maximum disjoint paths problem. In *13th ACM symposium on parallel algorithms and architectures (SPAA'01)* (pp. 38–47).

45. Krishnamoorthy, M., Ernst, A. T., & Sharaiha, Y. M. (2001). Comparison of algorithms for the degree constrained minimum spanning tree. *Journal of Heuristics, 7*(6), 587–611.
46. Krishnaswamy, R. M., & Sivarajan, K. N. (2001). Algorithms for routing and wavelength assignment based on solutions of LP-relaxations. *IEEE Communications Letters, 5*(10), 435–437.
47. Lee, K., Kang, K., Lee, T., & Park, S. (2002). An optimization approach to routing and wavelength assignment in wdm all-optical mesh networks without wavelength conversion. *ETRI Journal, 24*(2), 131–141.
48. Low, C. P. (1998). A fast search algorithm for the quorumcast routing problem. *Information Processing Letters, 66*, 87–92.
49. Mukherjee, B. (2006). *Optical WDM networks*. Springer.
50. Nardelli, E., & Proietti, G. (2001). Finding all the best swaps of a minimum diameter spanning tree under transient edge failures. *Journal of Graph Algorithms and Applications, 5*(5), 39–57.
51. Noronha, T., & Ribeiro, C. (2006). Routing and wavelength assignment by partition coloring. *European Journal of Operational Research, 171*(3), 797–810.
52. Ozdaglar, A., & Bersekas, D. (2003). Routing and wavelength assignment in optical networks. *IEEE/ACM Transactions on Networking, 11*(2), 259–272.
53. Pham, Q. D. (2011) *LS(Graph): A constrained-based local search framework for constrained optimum tree and path problems on graphs.* PhD thesis, Université catholique de Louvain.
54. Pham, Q. D., Deville, Y., & Hentenryck, P. V. (2010). Constraint-based local search for constrained optimum paths problems. In *Proceedings of the seventh international conference on integration of artificial intelligence and operations research techniques in constraint programming, CPAIOR'2010* (pp. 267–281).
55. Ramaswami, R., & Sivarajan, K. (1995). Routing and wavelength assignment in all-optical networks. *IEEE/ACM Trans Network, 3*(5), 489–500.
56. Reimann, M., & Laumanns, M. (2004) A hybrid aco algorithm for the capacitated minimum spanning tree problem. In *Proceedings of first international workshop on hybrid metaheuristics* (pp. 1–10).
57. Sleator, D. D., & Tarjan, R. E. (1983). A data structure for dynamic trees. *Journal of Computer and System Sciences, 26*(3), 362–391.
58. Sleator, D. D., & Tarjan, R. E. (1985). Self-adjusting binary search trees. *Journal of the Association for Computing Machinery, 32*(3), 652–686.
59. Tarjan, R. E., & Werneck, R. F. (2005). Self-adjusting top trees. In *Proceedings of the 16th annual ACM-SIAM symposium on discrete algorithms (SODA)* (pp. 813–822).
60. Tarjan, R. E., & Werneck, R. F. (2009). Dynamic trees in practice. *Journal of Experimental Algorithmics (JEA), 14*, 4.5:1–4.5:21.
61. Tornatore, M., Maier. G., & Pattavina, A. (2002). Wdm network optimization by ilp based on source formulation. In *Proceedings of IEEE INFOCOM* (pp. 1813–1821).
62. Van Hentenryck, P., & Michel, L. (2005). *Constraint-based local search*. London: MIT Press.
63. Wang, B., & Hou, J. C. (2004). An efficient qos routing algorithm for quorumcast communication. *Computer Networks Journal, 44*(1), 43–61.
64. Ye, Y., Chai, T., Cheng, T., & Lu, C. (2003). Algorithms for the design of wdm translucent optical networks. *Optics Express, 11*(22), 2917–2926.
65. Yetginer, E., Liu, Z., & Rouskas, G. N. (2010). RWA in WDM rings: An efficient formulation based on maximal independent set decomposition. In *The 17th IEEE workshop on local and metropolitan area networks (IEEE LANMAN 2010)*.
66. Zachariasen, M. (1999). Local search for the steiner tree problem in the Euclidean plane. *European Journal of Operational Research, 119*, 282–300. doi:10.1016/S0377-2217(99)00131-9.
67. Zang, H., Jue, J. P., & Mukherjee, B. (2000). A review of routing and wavelength assignment approaches for wavelength-routed optical wdm networks. *Optical Networks Magazine, 1*(1), 47–60.