**ICTEAM, pôle d'Ingénierie Informatique**
**Ecole polytechnique de Louvain**
**Université catholique de Louvain**
**Louvain-la-Neuve, Belgium**

# LS(Graph): A constraint-based local search framework for constrained optimum trees and paths problems on graphs

Quang Dung PHAM

February 2011

Dissertation présentée en vue de
l'obtention du grade de Docteur en
Sciences de l'Engénieur

**Composition du jury:**

| | |
|---|---|
| Pr. Y. Deville (Promoteur) | ICTEAM, UCL, Belgium |
| Pr. P. Van Hentenryck (Examinateur) | Brown University, RI, USA |
| Pr. C. Pêcheur (Examinateur) | ICTEAM, UCL, Belgium |
| Pr. C. Solnon (Examinateur) | LIRIS CNRS UMR, France |
| Pr. O. Bonaventure (President) | ICTEAM, UCL, Belgium |

# ABSTRACT

Constrained Optimum Tree (COT) and Constrained Optimum Path (COP) are two classes of problems which arise in many real-life applications and are ubiquitous in communication networks, transportations, very large scale integration (VLSI) and distributed systems. Most of these problems are computationally very hard to solve. They have been traditionally approached by dedicated algorithms including heuristics and exact algorithms, which are often hard to extend with side constraints and to apply widely because they depend strongly on the problem structures. Moreover, it is required huge research and programming efforts for solving new problems.

In this thesis, we construct a constraint-based local search (CBLS) framework, called `LS(Graph)`, for solving COT/COP applications, bringing the compositionality, reuse, and extensibility at the core of CBLS and CP 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 `LS(Graph)` framework will strengthen the modeling benefits of CBLS. By using `LS(Graph)`, users can quickly develop a local search algorithm for a new problem which gives, in most of cases, an acceptable solution while waiting for experts who do research with huge efforts for dedicated algorithms. Moreover, this solution can be used as the initial solution in more complex and hybrid algorithms.

The main technical contribution is a connected neighborhood based on rooted spanning trees. The idea behind is to use rooted spanning tree for representing solutions which are paths and their neighborhoods.This approach enables the genericity of the framework from both modeling and computation standpoints.

The constructed framework is applied to some three COT (i.e., the edge-weighted $k$-cardinality tree problem, the quorumcast routing problem, the problem of minimizing congestions on ethernet networks) and four COP problems (i.e., the resource constrained shortest path problem, the edge-disjoint paths problem, the routing and wavelength assignment with delay side constraint problem, and the routing for network covering problem). Experimental results show the potential benefits of the approach. On the one hand, we show the facility and the genericity of the resolution of the COT/COP applications which can be extended with side constraints. On the other hand, for the quorumcast routing and the edge-disjoint paths problems, we show competitive results in comparing with existing techniques.

# ACKNOWLEDGEMENTS

This thesis could not be completed without the help of following persons.

First, I would like to express my deep gratitude to my advisor, Professor Yves Deville, who has motivated me at the beginning and during my four years of research. His significant ideas and suggestions have been of such great benefit for improvement of my work. I learnt a lot from his methodology in doing research.

I would like to thank Professor Pascal Van Hentenryck for his interesting ideas and significant comments in writing scientific papers. His huge expertise in optimization was beneficial in this work.

I thank also Ho Tuong Vinh, Alain Boucher and all members of the MSI research group for helping me during my working visit at the MSI laboratory of the Institut de la Francophonie pour l'Informatique in Hanoi, Vietnam.

I would like to thank to Professors Christine Solnon, Charles Pêcheur, and Olivier Bonaventure for their significant comments and suggestions on the first manuscript of my thesis.

Many thanks to all members of the BeCool team (Stéphane Zampelli, Pierre Schaus, Jean-Noël Monette, Sébastien Mouthuy, Ho Trong Viet, Julien Dupuis, Vianney le Clément, Florence Massen, Jean-Baptiste Mairy, Duong Khanh Chuong, Nguyen Thi Hong Hiep) for listening and giving interesting comments during my seminars. Friendly atmosphere of the team during conferences in foreign countries is unforgettable.

Special thank to Stéphanie Landrain, Chantal Poncin for welcoming and helping me during the first and difficult period of my stay in Belgium. I also thank the whole INGI department, especially people of the volley-ball team. They help me to realize that the research life at the INGI department is not boring at all.

I devote my warm gratitude to my parents for their supports during my study in Belgium. I also thank all Vietnamese students in Louvain-la-Neuve for interesting life out of research with sports and delicious meals. Thanks also to the UCL "coopération au développement" for their financial support.

And, last but not least, the most special thanks belong to two special women who were always beside me during my study in Belgium. My wife Ninh and my daughter Hang often motivate me in my work. Their supports were essential for completing my PhD thesis.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# 1 INTRODUCTION

The goal of this dissertation is to provide a generic and efficient framework for solving constrained optimum paths (COP) and trees (COT) problems on graphs by constraint-based local search.

## 1.1 Context

In telecommunication networks, a routing problem supporting multiple services involves the computation of paths minimizing transmission costs while satisfying bandwidth and delay constraints. Also, we consider the problem of establishing routes for connection requests between network terminals and it is typically required that no two routes interfere with each other due to quality-of-service and survivability requirements. In the telecommunication network design, we need to interconnect some network terminals in order to transmit information satisfying some criteria, for instance, the traffic congestion or connection cost is minimized. These problems can be modeled by constrained optimum paths (COP) and trees (COT) problems on graphs which consist of finding one or several paths, trees of a given graph optimizing an objective function while satisfying some given constraints. Such problems have been extensively studied and solved in the literature, for instance, Degree Constrained Minimum Spanning Tree (DCMST) [KES01, ALM06], Bounded Diameter Minimum Spanning Tree (BDMST) [GvHR06], Capacitated Minimum Spanning Tree (CMST) [RL04, AOS03], Minimum Diameter Spanning Tree (MDST) [NP01], Edge-Weighted $k$-Cardinality Tree (KCT), [BB05, CKIL09], Steiner Minimal Tree (SMT) [Zac99, dAUW01], Optimum Communication Spanning Tree Problems (OCST) [Fis07], Resource Constrained Shortest Path [BC89, DB03], Edge- and Vertex-Disjoint Paths [BB07, Kle96, KS04, KS01a, CK03], Minimum Edge-disjoint paths with different path costs [GCJ09, LMSL92], Multiple Choice Constrained Shortest Path [Smi06], Path cover problems [NH79], Shortest path with mandatory nodes [DDD05], Routing and Wavelength Assignment problems on optical networks [CB96, JMT07, ZJM00],

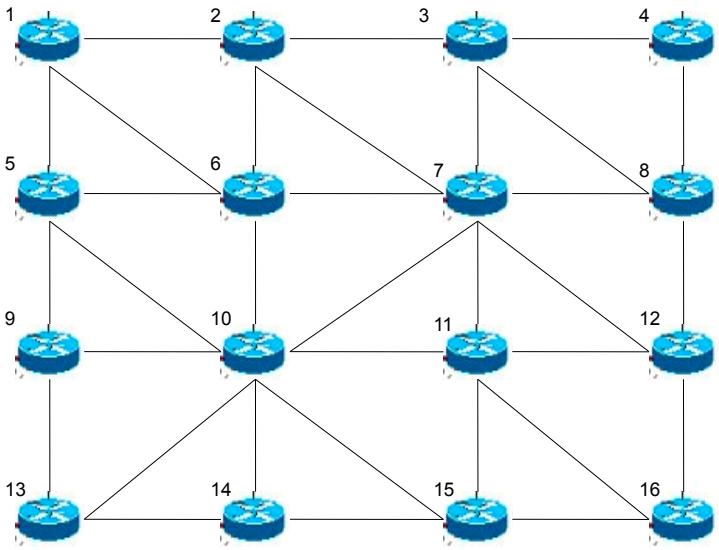



Figure 1.1: A WDM optical network

Chinese Postman Problem [CMS02, YC02], Arc Routing Problem [GW81, MTY09, KY10, HM01], etc.

**Example** In order to bring an intuition of applications to be tackled by this thesis, we give an example of the routing and wavelength assignment problem on wavelength-division-multiplexed (WDM) optical networks [CB96, JMT07, ZJM00]. The network consists of a set of wavelength routers interconnected by point-to-point optical links (see Figure 1.1).

We are given a set of connection requests represented by a set of pairs of vertices on the given graph. For each pair $(a, b)$, we need to find a path from $a$ to $b$ and assign a wavelength to this path satisfying the condition that two paths sharing a link must be assigned with different wavelengths. The objective is to use the minimal number of wavelengths.

Suppose the connection requests have 10 pairs of vertices $R = \{\langle 15, 16\rangle, \langle 9, 10\rangle, \langle 4, 16\rangle, \langle 7, 10\rangle, \langle 7, 10\rangle, \langle 6, 14\rangle, \langle 4, 11\rangle, \langle 3, 12\rangle, \langle 13, 15\rangle, \langle 2, 8\rangle\}$, the best solution for this problem is illustrated in Figure 1.2 with two wavelengths used: dashed lines are paths with wavelength number 1 and continuous lines are paths with wavelength number 2.

- route 1: 15 -> 16 with wavelength = 1
- route 2: 9 -> 10 with wavelength = 2
- route 3: 4 -> 8 -> 12 -> 16 with wavelength = 1

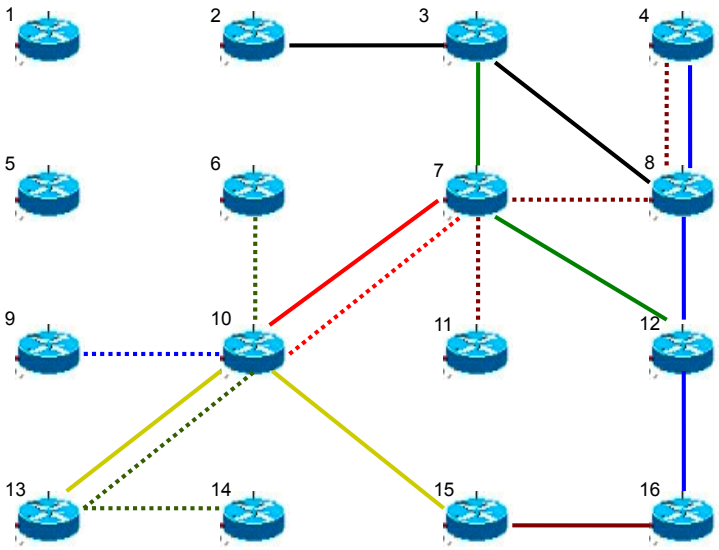



Figure 1.2: A solution to the routing and wavelength assignment problem

- route 4: 7 -> 10 with wavelength = 1
- route 5: 7 -> 10 with wavelength = 2
- route 6: 6 -> 10 -> 13 -> 14 with wavelength = 2
- route 7: 4 -> 8 -> 7 -> 11 with wavelength = 2
- route 8: 3 -> 7 -> 12 with wavelength = 1
- route 9: 13 -> 10 -> 15 with wavelength = 1
- route 10: 2 -> 3 -> 8 with wavelength = 1

## 1.2 Challenges

Most of these COT/COP problems are NP-hard. This means that exact methods for solving them induce an exponential computation time in the worst case. They are often approached by dedicated algorithms including exact methods, such as the Lagrangian-based heuristic [ALM06], the ILP-based algorithm using directed cuts [CKIL09], the Lagrangian-based branch and bound in [BC89] and the vertex labeling algorithm from [DB03] and meta-heuristic algorithms like hybrid evolutionary algorithm [Blu06], ant colony optimization [BS04], local search [BB05] and approximation algorithms

[Kle96, KS04]. These techniques exploit the structure of constraints and objective functions but are often difficult to extend and reuse. On the one hand, it is required huge programming effort when new problems need to be solved. On the other hand, it is too complicated to extend existing algorithms when new constraints need to be added.

## 1.3 Objective

The objective of this thesis is to design and implement a generic and efficient framework, called `LS(Graph)`, which allows to model in a flexible way and solve some COT/COP problems on graphs by constraint-based local search. The `LS(Graph)` framework allows users to solve different COT/COP problems declaratively (black box). On the one hand, users model problem by declaring variables, stating constraints and objective functions in high-level way and use built-in search components to find high-quality solutions. On the other hand, users can also easily perform different (meta-)heuristic search strategies. The `LS(Graph)` framework is open (white box) that allows users to design and implement their own abstractions and integrate them to the framework. The proposed framework features the modeling benefits of CBLS [VM05] (i.e., compositionality, modularity and reuse) and strengthens the modeling capacity of COMET programming language for applications on graphs. It will also be evaluated on various COT/COP applications and compared with state-of-the-art techniques.

## 1.4 Contributions

The contributions of this thesis are the following:

1. We design and implement a constraint-based local search (CBLS) [VM05] framework, called `LS(Graph)`, for some COT/COP applications to support the compositionality, reuse, and extensibility at the core of CBLS and CP systems (the current version of `LS(Graph)` is about 25,000 lines of COMET code). The proposed framework can be used as both black box and white 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 objective function to be optimized. Built-in search components (e.g., tabu search) are then performed automatically. The white box allows users to extend the framework by design and implementation of their own components (e.g., invariants, constraints and objective functions) and integrate them to the system.

2. The `LS(Graph)` combines graph variables (i.e., `VarTree, VarPath` for modeling trees, paths in a high-level way) with standard `var{int}` of COMET which enable the modeling of various COT/COP applications on graphs in which both topology and scalar values must be determined.

3. There exist few local search approaches for COP applications on general graphs. On complete graphs, some local search algorithms have been applied for solv-

ing the traveling salesman problem or the vehicle routing problem. In these approaches, a path is 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 position of vertices). These neighborhood structures cannot be applied for general graphs because a sequence of vertices can not be sured to always form a path on the given graph. A key technical contribution of this thesis 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. The proposed neighborhood can be applied widely on COP problems in general graphs.

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 two COT problems: the edge-weighted $k$-cardinalty tree problem, the quorumcast routing problem and four COP problems: the resource constrained shortest path problem, the edge-disjoint paths problem, the routing and wavelength assignment problem on optical networks and the routing for network covering problem. Experimental results show the facility and flexibility for modeling and solving these problems. The proposed models provide good results in comparison with state-of-the-art techniques. In particular, for the edge-disjoint paths problem, we show the proposed model finds better results than the state-of-the-art ACO algorithm. Moreover, for the quorumcast routing problem, we proposed a tabu search model which gives better results than the IMP heuristic algorithm which is, in our best knowledge, the best existing heuristic for this problem.

## 1.5 Outline

Chapter 2 presents the background related to the work of this thesis. We first present an overview of graph theory and give some definitions and notations which will be used throughout the thesis. In Section 2.2, we present the definition of constraint satisfaction problem. Constraint-based local search technique and the COMET programming language will be presented in Section 2.3. In Section 2.4, we give an overview of related works. Finally, we present the problem examples which will be solved by `LS(Graph)`.

Chapter 3 introduces the `LS(Graph)` framework in which the architecture will be presented in Section 3.1. Sections 3.2 and 3.3 present the framework for COT and COP applications in which we present the concept of path and tree variables, as well as different neighborhood structures for these applications. For COT applications, we apply traditional modification actions (e.g., edges insertions, removals, replacements over dynamic trees) for defining neighborhoods. For COP applications, we

propose a novel connected neighborhood (called COP neighborhood) which is based on rooted spanning tree. An overview of abstractions including common interfaces, graph invariants, graph functions, graph constraints and generic search components will be introduced in Section 3.5. Section 3.6 gives a description of data structures and algorithms and their complexities for implementing fundamental abstractions of the framework. The subsection 3.6.1 present the data structure for representing dynamic trees and an incremental algorithm for maintaining nearest common ancestors of all pairs of two vertices on dynamic trees. An incremental algorithm for maintaining a set of edges which are used for COP neighborhood is given in the subsection 3.6.2. Finally, the subsection 3.6.3 presents an incremental algorithm for maintaining distances between vertices of a dynamic tree.

Chapter 4 describes the implementation in COMET for some fundamental abstractions of the `LS(Graph)` framework. Only main methods and data structures for each class are depicted. The aim is to give an implementation principle. Based on this, users can extend the `LS(Graph)` by designing and implementing their own components and integrate them to the system.

Chapter 5 presents the applications of the constructed framework to the modeling of some COT/COP problems including the edge-weighted $k$-cardinality tree problem, the quorumcast routing problem, the resource constrained shortest path problem, the edge-disjoint paths problem, the routing and wavelength assignment in optical networks and the problem of routing for network covering. For each of these problems, we propose a local search algorithm and implement it in `LS(Graph)` using the constructed framework. These algorithms will be experimented and compared with existing algorithms.

Chapter 6 concludes the thesis and gives various research directions.

## 1.6 Publications

Here are our major publications during my doctoral research.

- Q. D. Pham, Y. Deville, P. Van Hentenryck. LS(Graph): A Local Search Framework for Constraint Optimization on Graphs and Trees. Proceedings of the 2009 ACM Symposium on Applied Computing (SAC'09), March 8-12, pages 1402-1407, 2009.

- Q. D. Pham, Y. Deville, P. Van Hentenryck. Constraint-Based Local Search for Constrained Optimum Paths Problems, 7th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2010), Lecture Notes in Computer Science, Springer, pages 267-281, 2010.

- Q. D. Pham, P. T. Do, Y. Deville, T. V. Ho. Constraint-based local search for solving non-simple paths problems on graphs: Application to the Routing for Network Covering Problem. In Proceedings of Symposium on Information and Communication Technology, SoICT2010, Hanoi, Vietnam, 2010.

The `LS(Graph)` framework is the open source and is available at

http://becool.info.ucl.ac.be/lsgraph

# 2

# BACKGROUND

This chapter aims at presenting the background of this thesis. We first present the graph theory which is based on [GY03] and give some basic definitions and notations. Then, we give the definition of Constraint Satisfaction Problems (CSP), Constraint Satisfaction Optimization Problems (CSOP) and present two alternative approaches for solving CSPs/CSOPs: Local Search and Constraint Programming. Constraint-Based Local Search (CBLS) and COMET [VM05], a new concept and system for solving CSPs are then presented. We conclude the chapter with a summary of existing systems for solving CSPs/CSOPs.

## 2.1 Graph theory

This section gives graph theory notions necessary to present the rest of this thesis.

**Graphs** Graph is a mathematical object involving points and binary connections between them. Most of theoretical graph theory considers simple graphs (i.e., graphs in which there are at most one connection between two points and there is no connection from one point to itself) and most of practical problems regarding general graphs can be reduced to problems on simple graphs. In this thesis, we consider only simple graphs and use the word "graph" instead of simple graph.

A graph $G = (V, E)$ is formed by a set of vertices (or nodes) $V$ and a set $E$ of connections (called edges) between vertices: $E \subseteq \{(u, v) \mid u \neq v \in V\}$. Given a graph $G$, we denote $V(G)$ and $E(G)$ respectively the set of vertices and edges of $G$ when $G$ is not the only graph under consideration. A graph can be directed or undirected. In directed graphs, each edge $e = (u, v)$ (also called arc) is oriented from $u$ to $v$: $u$, $v$ are respectively the head and the tail of $e$. In undirected graphs, we do not consider the orientation on edges: $(u, v) \equiv (v, u)$. In other words, for each undirected graph $G = (V, E)$, there does not exist two vertices $u, v \in V$ such that $(u, v) \in E \wedge (v, u) \in E$.

Given two graphs $G_1$ and $G_2$, we denote $G_1 + G_2$ the graph $G$ in which $V(G) = V(G_1) \cup V(G_2)$ and $E(G) = E(G_1) \cup E(G_2)$.

**Subgraphs** A graph $G_1$ is called subgraph of a graph $G_2$ if $V(G_1) \subseteq V(G_2) \wedge E(G_1) \subseteq E(G_2)$. A subgraph $G_1$ is called spanning subgraph of $G_2$ if it is subgraph of $G_2$ and $V(G_1) = V(G_2)$.

**Degree** For each edge $e = (u, v)$ of a graph $G$, $u$ and $v$ are called endpoints of $e$; $e$ is called incident on $u, v$; $u$ is called adjacent to $v$ (and vice versa).

The degree of a vertex $v$ on an undirected graph $G$, denoted by $deg_G(v)$, is the number of incident edges on $v$: $deg_G(v) = \sharp\{u \mid (u, v) \in E(G)\}$.

The in degree of a vertex $v$ on a directed graph $G$, denoted by $deg^-_G(v)$, is the number of arcs of $G$ entering $v$: $deg^-_G(v) = \sharp\{u \mid (u, v) \in E(G)\}$

The out degree of a vertex $v$ on a directed graph $G$, denoted by $deg^+_G(v)$, is the number of arcs of $G$ leaving $v$: $deg^+_G(v) = \sharp\{u \mid (v, u) \in E(G)\}$

**Walks, Paths** A walk $w$ in a graph $G$ is a sequence of vertices $\langle s = v_1, v_2, ..., v_k = t\rangle$ such that $(v_i, v_{i+1}) \in E(G), \forall i = 1, ..., k-1$. We denote $w(i)$ the $i^{th}$ vertex[1] of the sequence (i.e., the vertex $v_i$) and *len*$(w)$ the length of $w$ (i.e., the number of edges of the sequence).

A path in a graph $G$ is a walk in $G$ which contains no repeated vertices except starting and terminating vertices. A cycle is a path such that the starting and terminating vertices are the same.

Given two walks $w_1 = \langle x_1, x_2, ..., x_k\rangle$ in $G_1$ and $w_2 = \langle y_1, y_2, ..., y_q\rangle$ in $G_2$, we denote $w_1 + w_2$ the walk $w$ in $G_1 + G_2$ in which $w = \langle x_1, x_2, ..., x_k = y_1, y_2, ..., y_q\rangle$ if $x_k = y_1$ and $w = \langle x_1, x_2, ..., x_k, y_1, y_2, ..., y_q\rangle$ if $x_k \neq y_1 \wedge (x_k, y_1) \in E(G_1 + G_2)$.

Given a path $p$, we denote:

- $V(p)$ and $E(p)$ respectively the set of vertices and the set of edges of $p$. Suppose that $p = \langle v_1, v_2, ..., v_k\rangle$, then $V(p) = \{v_1, v_2, ..., v_k\}$ and $E(p) = \{(v_1, v_2),$-$(v_2, v_3), ..., (v_{k-1}, v_k)\}$. For short, given two paths $p_1, p_2$, we denote $p_1 \cup p_2$ ($p_1 \cap p_2$) the set $V(p_1) \cup V(p_2)$ ($V(p_1) \cap V(p_2)$) and if $x$ is a vertex of a path $p$, we denote $x \in p$. We also denote $G(p)$ the graph in which $V(p)$ and $E(p)$ are respectively the set of vertices and the set of edges.
- $s(p), t(p)$ respectively the starting and terminating vertices of $p$.
- $p(u, v)$ the subpath of $w$ starting from $u$ and terminating at $v$ ($u, v \in p$ and $u$ is not located after $v$ on $p$).
- $sp_p(x), tp_p(x)$ the subpath of $p$ from $s(p)$ to $x$ and from $x$ to $t(p)$.
- $repl(p, q) = sp_p(s(q)) + q + tp_p(t(q))$ where $q$ is a path and $s(q), t(q) \in p$. Intuitively, $repl(p, q)$ is the path obtained by replacing the subpath of $p$ from $s(q)$ to $t(q)$ by $q$.

[1] Vertices are indexed from 1, 2, ... in the sequence.

If $p$ is a walk (path) on a graph $g$, we denote $p \in g$. Given a graph $g$ and an integral value $k$, we denote *Paths(g,k)* the set of paths on $g$ having $k$ edges.

**Connectedness** A graph is called connected if between every pair of vertices, there exists a walk.

**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 $s \in V(tr)$, we use the following notations:

- *root*(*tr*) for the root of *tr*,
- $path_{tr}(v)$ for the path from $v$ to $root(tr)$ on $tr$. For each node $u$ of $path_{tr}(v)$, we say that $u$ *dominates* $v$ in $tr$ ($u$ 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 denote $u\ \overline{Dom_{tr}}\ v$.
- $path_{tr}(u, v)$ for the path from $u$ to $v$ in $tr$ ($u, v \in V(tr)$). This path does not take into account the orientation on edges.
- $nca_{tr}(u, v)$ for the nearest common ancestor of two nodes $u$ and $v$ on $tr$. 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 $T_{tr}(v)$ the subtree of $tr$ rooted at $v$. If $v \neq root(tr)$, we denote $\overline{T_{tr}}(v)$ the subtree of $tr$ generated by removing $T_{tr}(v)$ and 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** *Given a rooted tree $tr$.*

1. *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. *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 Figure 2.1: $nca_{tr}(v, z) = nca_{tr}(v, x) = 12$.*

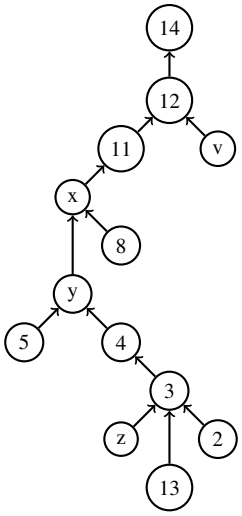



Figure 2.1: Illustrating property 1

## 2.2 Constraint Satisfaction Problem

A Constraint Satisfaction Problem (CSP) is a triple $\langle X, D, C\rangle$ in which $X = \{x_1, x_2,$-$..., x_n\}$ is a set of variables, $D$ is a set of domains of variables ($D_i$ is the domain of the variable $x_i$) and $C$ is a set of constraints (i.e., relations imposed over variables) defined over variables: $C = \{C_1(S_1), C_2(S_2), ..., .C_k(S_k)\}$ in which each $S_i$ is a set of variables. A constraint $C_i(S_i)$ is a combination of valid values for the variables $S_i$.

A solution to a CSP is an assignment of all variables such that all constraints are satisfied. Deciding the satisfiability of a CSP (i.e., the existence of a solution) is a NP-complete problem in the general case.

A Constraint Satisfaction Optimization Problem (CSOP) $\langle X, D, C, f\rangle$ is a CSP $\langle X, D, C\rangle$ with an objective function $f : X \rightarrow \mathbb{R}$ to be optimized. The objective is to find a solution to the $\langle X, D, C\rangle$ that optimizes (minimizes or maximizes) $f$.

In some cases, constraints in CSPs/CSOPs can be divided into hard constraints and soft constraints. Hard constraints must be satisfied by the solution while soft constraints are allowed to have a number of violations.

**Example** Given a CSP=$\langle X, D, C\rangle$ where:

- $X = \{x_1, x_2, x_3\}$,
- $D_1 = \{1, 2\}$, $D_2 = \{1, 2, 3\}$, $D_3 = \{2, 3\}$,
- $C = \{C_1, C_2, C_3\}$ with:
  - $C_1$ (hard constraint): $x_1 \leq x_2$,
  - $C_2$ (hard constraint): $x_1 + x_2 = x_3$,
  - $C_3$ (soft constraint): $(x_1 + x_3)$ mod $x_2 = 1$.

Clearly, the assignment $x_1 = 1$, $x_2 = 2$, $x_3 = 3$ is a solution to this CSP because the hard constraints $C_1$ and $C_2$ are satisfied.

### 2.2.1 Constraint Programming

Constraint Programming is a technique for solving CSPs by combining constraint propagation with search. The set of constraints is used to filter the domains of the variables (i.e., prune values of variables from their domains that do not belong to any solution). This filtering is called propagation. In general, the propagation is not sufficient to deduce a solution. Instead, the current CSP is simplified by splitting a domain of a variable which creates two or more CSPs. This splitting is called branching. The propagation can then remove further values from the domains of the variables. The interleaving of branching and propagation is called search. This search can be viewed as a tree (also called search tree). The branching specifies the forms of the search tree while exploring it is the way the tree is traversed. Constraint Programming is a complete technique in the sense that it ensures to find a solution to a CSP if any or prove that the given CSP does not have solutions. Its time complexity is exponential in general. Interested readers can consult the handbook of Constraint Programming [RvBW06] for a detailed description.

### 2.2.2 Local Search

Local search [MAK07] is an alternative technique that aims at finding a high-quality solution to a CSP or CSOP in polynomial time. Local search algorithms start with an initial solution that is constructed by some heuristic algorithm and searches through the solution space by continually moving from a candidate solution[2] to one of its neighbors until some criteria are reached. The key problem of a local search algorithm is the definition of a neighborhood and its exploration for selecting the next candidate solution. The action of moving from a current candidate solution to another candidate solution is called local move. For a short, we use the word "solution" instead of "candidate solution" if there is no ambiguity. A solution satisfies all the constraints is called feasible solution.

We describe in this section the basic local search template and sample some fundamental heuristics, metaheuristics which will be applied in later sections of the thesis. Interested readers can consults the books [MAK07, VM05] for other metaheuristics like simulated annealing, iterated local search, etc. The presentation is mostly based on [VM05].

#### Basic local search template

Algorithm 1 depicts a basic local search template which is parameterized by the function $f : X \rightarrow \mathbb{R}$ to be minimized, the neighborhood function $N$, the neighborhood restriction function $L$ and the selection operator $S$. The function $f$ measures the quality of solutions to the problem under consideration. In the CSPs, it represents the

[2] A candidate solution is an assignment of values to the variables satisfying all hard constraints.

distance between the current solution and a feasible solution. In a pure optimization problem (without constraints to be satisfied), it expresses the objective of the problem or a function of finer granularity that differentiates between solutions having the same objective value. In problems having both constraints to be satisfied and an objective function to be optimized, the function $f$ may combine the feasibility and the optimality measures appropriate for the problem at hand. This section simply assumes the availability of a function $f$ to be minimized. If we denote $\mathcal{S}$ the set of all possible solutions and $2^{\mathcal{S}}$ the set of all possible subsets of $\mathcal{S}$, then the functions $N$, $L$, and $S$ are defined to be:

- $N : \mathcal{S} \rightarrow 2^{\mathcal{S}}$,
- $L : 2^{\mathcal{S}} \times \mathcal{S} \rightarrow 2^{\mathcal{S}}$,
- $S : 2^{\mathcal{S}} \times \mathcal{S} \rightarrow 2^{\mathcal{S}}$.

The local search starts from an initial solution $s$ (line 1) and iteratively moves from the current solution to one of its neighbors (lines 3 and 6) based on $N$, $L$, and $S$ . Among solutions of the given neighborhood, only legal solutions (identified by the function $L$) are considered[3]. The operation $S$ selects one of these legal solutions. Lines 4-5 simply keep the best solution $s^*$ encountered so far.

**Algorithm 1**: LocalSearch($f, N, L, S$)

**Input**: Problem instance $\langle X, D, C \rangle$, functions $f, N, L, S$ with

- $f : X \rightarrow \mathbb{R}$
- $N : \mathcal{S} \rightarrow 2^{\mathcal{S}}$,
- $L : 2^{\mathcal{S}} \times \mathcal{S} \rightarrow 2^{\mathcal{S}}$,
- $S : 2^{\mathcal{S}} \times \mathcal{S} \rightarrow 2^{\mathcal{S}}$.

**Output**: A solution

```
1 s ← GenerateInitialSolution();
2 s* ← s;
3 forall k ∈ 1..MaxTrials do
4     if satisfiable(s) ∧ f(s) < f(s*) then
5         s* ← s;
6     s ← S(L(N(s), s), s);
7 return s*;
```

### Heurisitics

Typically, heuristics select the next solution based on local information (e.g., the current solution and its neighborhood) and they drive the search toward local optima.

[3]The definition of legality depends on the given problem and local search algorithm.

Heuristics are typically specified by instantiation of the selection operator $S$ and the restriction function $L$. For instance, improvement heuristic can be formulated by instantiating the $L$ function with L-Improvement described in Algorithm 2.

**Algorithm 2**: L-Improvement($N, s$)

**Input**: A set of solutions $N$ and the current solution $s$
**Output**: A subset of solutions of $N$ which are better than $s$
1 **return** $\{x \in N \mid f(x) < f(s)\}$;

**Best Neighbor** The selection function $S$ can be instantiated by S-Best or S-First. The S-Best aims at selecting the best neighboring solution (see Algorithm 3). A best-improvement heuristic can thus be specified by the instantiation of $S$ and $L$ with S-Best and L-Improvement (see Algorithm 4)

**Algorithm 3**: S-Best($N, s$)

**Input**: A set of solutions $N$ and the current solution $s$
**Output**: A solution of $N$ with smallest value of $f$
1 $N^* \leftarrow \{x \in N \mid f(x) = min_{z \in N} f(z)\}$;
2 **return** $x \in N^*$ with probability $\frac{1}{\sharp N^*}$;

**Algorithm 4**: BestImprovement

1 **return** LocalSearch($f$,$N$,L-Improvement,S-Best);

**First Neighbor** The best-improvement heuristic scans completely the neighborhood which may be too costly when the neighborhood is large. The first-improvement simply chooses the first neighbor that improves the current solution (see Algorithm 6 where the selection function is instantiated by S-First). The selection function S-First is depicted in Algorithm 5 which selects the first improving neighboring solution when scanning the neighborhood. In this algorithm, we assume, without loss of generality, a function $lex(n)$ that specifies the lexicographic order of a neighbor $n$ when scanning the neighborhood.

### Metaheuristics

Contrary to heuristics, metaheuristics choose the next solution based on global information (e.g., the execution sequence) and aim at escaping from local optima. We now describe the generic tabu search algorithm which is one of the most popular metaheuristics for local search and which will be applied in later sections of the thesis. We generalize slightly the generic local search presented earlier (described in Algorithm

**Algorithm 5**: S-First($N, s$)

**Input**: A set of solutions $N$ and the current solution $s$
**Output**: The first solution of $N$

1 **return** $n \in N$ minimizing $lex(n)$;

**Algorithm 6**: FirstImprovement

1 **return** LocalSearch($f, N$,L-Improvement,S-First);

1). The new generic local search algorithm depicted in Algorithm 7 maintains a sequence of solutions explored so far $\tau = \langle s_0, s_1, ..., s_k \rangle$. $\tau$ is initialized by the initial solution $s_0$ (line 2) and extended after each local move (line 7).

**Algorithm 7**: LocalSearch($f, N, L, S, s$)

**Input**: Problem instance $\langle X, D, C \rangle$, an initial solution $s$, and functions $f, N, L, S$ with

- $f : X \to \mathbb{R}$
- $N : \mathcal{S} \to 2^{\mathcal{S}}$,
- $L : 2^{\mathcal{S}} \times \mathcal{S} \to 2^{\mathcal{S}}$,
- $S : 2^{\mathcal{S}} \times \mathcal{S} \to 2^{\mathcal{S}}$.

**Output**: A solution

1 $s_0 \leftarrow s$;
2 $\tau \leftarrow \langle s_0 \rangle$;
3 **forall** $k \in 0..MaxTrials$ **do**
4  **if** $satisfiable(s_k) \wedge f(s_k) < f(s^*)$ **then**
5  $s^* \leftarrow s_k$;
6  $s_{k+1} \leftarrow S(L(N(s_k), \tau, s_k), \tau)$;
7  $\tau \leftarrow \tau :: s_{k+1}$;
8 **return** $s^*$;

The tabu search is now specified by instantiating the restriction function $L$ with L-NotTabu (see Algorithm 8) and the selection $S$ with S-Best (see Algorithm 9).

There are two interesting features of the tabu search described here. First, the local search is allowed to choose moves that degrade the quality of the current solution because the definition of the legality function $L$ does not impose any constraints on the objective value of the solutions. This helps to escape local optima. Second, the application of S-Best ensures that the objective function does not reduce too much at any step, since the best neighbor is always selected.

**Algorithm 8**: L-NotTabu($N, \tau, s$)

1 **return** $\{x \in N \mid x \notin \tau\}$;

**Algorithm 9**: TabuSearch($f, N, s$)

**Input**: Problem instance $\langle X, D, C \rangle$, an initial solution $s$, and functions $f, N$ with

- $f : X \to \mathbb{R}$
- $N : \mathcal{S} \to 2^{\mathcal{S}}$,

**Output**: A solution

1 **return** LocalSearch($f$, $N$,L-NotTabu,S-Best,$s$);

**Short-Term Memory** The two main difficulties of the tabu search algorithm described above is the need of keeping track of all visited solutions and checking efficiently whether or not a solution is revisited. To overcome this, the tabu search can maintain a list of most recently move abstractions performed (called tabu list and denoted by $\tilde{\tau}$), for example, each move abstraction can be represented by a pair $\langle x, v \rangle$ when the move is the assignment of the variable $x$ by a new value $v$. The number of most recently move abstractions maintained is called tabu length. This strategy is called short-term memory. As a consequence, the set of legal moves can be specified in Algorithm 10 where, informally speaking, $tabu(x, \tilde{\tau}, s)$ hold if the solution $x$ is

**Algorithm 10**: L-NotTabu($N, \tilde{\tau}, s$)

1 **return** $\{x \in N \mid \neg tabu(x, \tilde{\tau}, s)\}$;

generated from the current solution $s$ by performing a move whose abstraction is in $\tilde{\tau}$. Obviously, such short-term memory may not prevent the local search from revisiting solution entirely and it may forbid moves which should be allowed.

**Aspiration** Since the tabu search maintain a list of move abstractions, not a list of solutions, it may forbid solutions which have not been visited before. Some of these solutions may be very desirable, (i.e., better than the best solution found so far). To remedy this limitation, tabu search algorithms often features an aspiration criterion: we accept the move whose abstraction is tabu but the new solution is better than the best solution found so far (see Algorithm 11 which instantiates the legality function $L$)

**Long-Term Memory** The short-term memory strategy cannot capture long-term information. Hence, it cannot prevent the search from spending too much time in the same region of the solutions space and bring low-quality solutions. Many tabu search

**Algorithm 11**: L-NotTabu-Asp($N, \tau, s$)

**return** $\{x \in N \mid \neg tabu(x, \tilde{\tau}, s) \vee f(x) < f(s^*)\}$;

algorithms are thus enhanced with additional long-term memory structures to intensify and diversify the search. The intensification stores high-quality solutions during the search and returns to these solutions periodically. It allows the search to explore the region of the solutions space where the best solutions so far have been discovered (see Algorithm 12). The diversification directs the search toward other regions of the solutions space. This can be achieve by iterative local search to perturb or to restart the local search.

**Algorithm 12**: IntensifiedLocalSearch($f, N, L, S$)

**Input**: Problem instance $\langle X, D, C \rangle$, and functions $f, N, L, S$ with

- $f : X \to \mathbb{R}$
- $N : \mathcal{S} \to 2^{\mathcal{S}}$,
- $L : 2^{\mathcal{S}} \times \mathcal{S} \to 2^{\mathcal{S}}$,
- $S : 2^{\mathcal{S}} \times \mathcal{S} \to 2^{\mathcal{S}}$.

**Output**: A solution

```
s ← GenerateInitialSolution();
s* ← s;
forall k ∈ 1..MaxSearches do
    s ←LocalSearch(f, N, L, S, s);
    if f(s) < f(s*) then
        s* ← s;
    s ← s*;
return s*;
```

### 2.2.3 Ant Colony Optimization

Ant Colony Optimization (ACO) is a metaheuristic method for solving combinatorial optimization problems which was initially proposed by Marco Dorigo in his PhD thesis in 1992. The idea is inspired from natural behavior of ants when seeking a path between their colony and a source of food. Ants initially wander in a random way to find food and return to their colony. During its itinerary, an ant lays a pheromone trail. These pheromone trails are then followed by other ants which reinforce the trails when come back to the colony with food. As time goes by, the source food becomes exhausted, the pheromone trails start to evaporate.

The basic idea when applying ACO for solving combinatorial optimization problems is to iteratively build solutions in a greedy random way (see Algorithm 13). More precisely, at each cycle, each artificial ant builds a solution from an empty solution by iteratively adding solution components until the solution is completed. At each step of this construction, the next solution component to be added is selected with respect to a probability which depends on two factors:

- The pheromone factor which reflects the experience of ants regarding the selection of this component.
- The heuristic factor which measures how good this component is with respect to the objective function. This factor is problem-dependent.

Let $C$ be the set of candidates for solution components and $S$ be the partial solution under construction. Each solution component is associated with a pheromone trail $\tau_S(c_k)$ which depends on the partial solution $S$ and which is updated over time. The probability for choosing a solution component $c_k$ is:

$$p(c_k) = \frac{\tau_S(c_k)^\alpha . \eta_S(c_k)^\beta}{\sum_{c_i \in C} \tau_S(c_i)^\alpha . \eta_S(c_i)^\beta}$$

where $\tau_S(c_k)^\alpha$ is the pheromone factor (weighted by a parameter $\alpha$) and $\eta_S(c_k)$ is the heuristic factor (weighted by a parameter$\beta$).

**Algorithm 13**: GenericACO

```
initialize pheromone trails;
while termination criteria not reach do
    each ant builds a solution;
    update pheromone trails;
```

Once each ant has constructed a solution, pheromone trails are updated. First, all pheromone trails are decreased (or evaporated) by multiplying them by a factor (1-$\rho$) where $\rho \in [0; 1]$ is called evaporation rate. This evaporation process allows ants to progressively forget older constructions and to focus on more recent constructions. In a second step, some solutions are rewarded by laying pheromone trails on their solution components. This allows to increase the probability of selecting the solution components of these solutions during the following constructions. We may reward all the solutions that have been constructed during the last cycle, or we may select the best solution of the last cycle for rewarding. These different strategies have a strong influence on the intensification and the diversification of the search.

Recently, ACO has been widely applied for solving subset selection, sequencing and paths finding problems. For more detail about Ant Colomy Optimization, interested readers are referred to [DS04] and ACO website (http://iridia.ulb.ac.be/ mdorigo/ACO/ACO.html).

## 2.3 Constraint-Based Local Search and the COMET programming language

In constraint-based local search (CBLS) [VM05], constraints are used to describe and control local search. This enables compositionality, modularity, reuse, and separation of concerns. Invariants and differentiable objects are two main concepts of the constraint-based local search architecture.

### 2.3.1 Invariant

Invariant is a concept describing objects that maintain a number of properties of the application structure. For example, the snippet:

```
1  Solver<LS> m();
2  var{int} x[1..5](m,1..5);
3  var{int} s(m) <- sum(i in 1..5)x[i];
```

initializes in line 3 an invariant `s` which maintains the sum of all decision variables `x[i]`. The value of `s` change under the modification of decision variables `x[1..5]`.

### 2.3.2 Differentiable objects

Differentiable objects not only maintain a number of properties of the applications structure but also allow to query the impact of various local moves over these properties. For instance, constraints are differentiable objects that maintain a number of violations of the constraint and allow to query the variation of this property under different local moves. The snippet:

```
1  int delta = S.getSwapDelta(x[i],x[j]);
```

computes the variation of the number of violations of the constraint `S` when the two decision variables `x[i]` and `x[j]` are exchanged.

The availability of invariants and differentiable objects enables the facility of programming. On the one hand, this allows users to state in a flexible way various constraints and objective functions. On the other hand, it reduces the programming effort. Users do not have to maintain sophisticated data structures. Rather, they can focus on the modeling and the exploration of different (meta)-heuristic search strategies.

### 2.3.3 COMET programming language

COMET [VM05] is a high-level modeling language with a number of innovative control abstractions for local search. The COMET project was started in 2001 and it now becomes a hybrid platform supporting both constraint programming, local search (with CBLS architecture) and mathematical programming. Figure 2.2 illustrates the high-level modeling and local search on the $n$-queens problem which entails placing $n$ queens on a chessboard of size $n \times n$ so that no two queens lie on the same row, column, or diagonal. We can see in this example, the COMET program is compact

and high-level which consists of two main parts: the model (lines 1-17) and the search (lines 19-25). Line 3 creates a local solver `m` which is a container that stores incremental variable, invariants, constraints and objective function and maintains a precedency graph relating these objects. Line 7 declares decision variables `queen[1..n]` associated with the local solver `m` and are initialized randomly by a uniform distribution `distr` (line 6) in which `queen[i]` represents the row of the queen at column `i`. Line 9 initializes a constraint system `S` and lines 11-12 state and post all constraints of the problem to `S`. Line 15 initializes an invariant `mostViolatedQueens` which maintains the set of indices of most violating variables (`varVlts[i]` in line 14 represents the number of violations of the constraint system `S` of the decision variable `queen[i]`). The model is closed in line 17. This instruction builds a dependency graph used to update the constraint `S` based on changes to the incremental variables. The search is depicted in lines 19-25. At each step, we choose randomly a most violating variable `queen[i]` (line 21), and then we select a value `v` such that the number of violations of the constraint reduces much (line 22). `S` is differentiable object supporting the method `getAssignDelta(queen[i],v)` which computes and returns the variation of the number of violations of the constraint `S` when the value `v` is assigned to the variable `queen[i]`. Line 23 performs the local move which assigns the chosen value `v` to the chosen variable `queen[i]`. This assignment induces a propagation that updates all the invariants, constraints of the model (i.e., `S`) thanks to the dependency graph. This update is generally very efficiently computed by incremental algorithms. It is clear that the program features separation of concerns: the model and the search are independent. We can easily state and post new constraints to the constraint system without changing the search. Moreover, we can perform different heuristic and meta-heuristic strategies without having to modify the model. We can also see the genericity and reuse feature of the program. The search is essentially generic which relies only on decision variables and the constraint system. It could thus be reused without changes in other contexts.

This thesis extends the COMET system by designing and implementing abstractions for modeling and solving COT/COP problems on graphs. The constructed framework features modeling benefits of CBLS enabling compositionality, modularity and reuse.

## 2.4 Related work

A number of generic systems have been developed for modeling and solving CSPs by Constraint Programming including COMET (http://dynadec.com/), ILOG solver, Mozart Programming System (http://www.mozart-oz.org/), Gecode (http://www.gecode.org), etc. More specifically, Grégoire Dooms [DDD05] introduces a `CP(Graph)` computation domain in Constraint Programming. Graph variables have been introduced over which constraints are defined including kernel constraints like *Arcs* constraint, *Nodes* constraint, *ArcNode* constraint and global constraints like *SubGraph* constraint, *Connected* constraint, etc. Some pruning techniques have been proposed for achieving bound consistency. The framework implemented in C++ and integrated in the Gecode

```
1  import cotls;

3  Solver<LS> m();
4  int n = 8;
5  range Size = 1..n;
6  UniformDistribution distr(Size);
7  var{int} queen[Size](m,Size) := distr.get();

9  ConstraintSystem<LS> S(m);
10 S.post(alldifferent(queen));
11 S.post(alldifferent(all(i in Size) queen[i] + i));
12 S.post(alldifferent(all(i in Size) queen[i] - i));

14 var{int} varVlts[i in Size] = S.violations(queen[i]);
15 var{set{int}} mostViolatedQueens(m) <- argMax(i in Size)(varVlts[i]);

17 m.close();

19 int it = 0;
20 while (S.violations() > 0 && it < 100000){
21   select(i in mostViolatedQueens)
22     selectMin(v in Size)(S.getAssignDelta(queen[i],v))
23       queen[i] := v;
24   it++;
25 }
```

Figure 2.2: CBLS model for n-queens problem

framework which is dedicated for modeling and solving constraint satisfaction problems on graphs. The `CP(Graph)` framework has also been applied to the resolution of a constrained path finding application in the biochemical analyses.

Related work on the use of trees and paths in local search will be presented in the `LS(Graph)` framework chapter (Sections 3.2 and 3.3). Related works on specific applications will be presented in Section 2.5.

When implementing the `LS(Graph)` framework, we exploit sophisticated data structure and incremental algorithms, for instance, data structure for maintaing nearest common ancestor of all pairs of two vertices on dynamic trees. Related work will be presented in the `LS(Graph)` framework chapter (Section 3.6).

## 2.5 Problem examples

In this thesis, we apply the `LS(Graph)` framework to the resolution of two COT problems and four COP problems. This section describes these problems and presents related works. Most of them are studied and solved in the literature except the last one, the routing for network covering (RNC) problem, which is inspired from the AROUND project [MSI06]. We introduce in Section 2.5.6 this new problem which is closely related to some other classical problems and also discuss its challenge.

`LS(Graph)` has also been successfully applied on a Traffic Engineering in Switched Ethernet Networks problem which consists of finding a spanning tree on a given network minimizing the traffic congestion which is reported in [HFD+10].

### 2.5.1 The edge-weighted $k$-cardinality tree (KCT) problem

Given an undirected weighted graph $G = (V, E)$ and an integral value $k$, the KCT[4] problem consists of finding a connected and acyclic subgraph (i.e., a tree) of $G$ having exactly $k$ edges such that the sum of weights of edges is minimal. A tree having $k$ edges is called $k$-cardinality tree. This problem appears in various applications such as oil-field leasing, facility layout, open pit mining, matrix decomposition, quorumcast routing and telecommunications (see [BB05] and references therein).

The KCT problem has been solved by both metaheuristic and exact methods. Different metaheuristic algorithms including tabu search [BB05], ant colony optimization algorithms [BS04, BB05] and evolutionary computation algorithms [BB05, Blu06] have been proposed and experimented on benchmark on [BB]. Among these, no algorithm gives the best results on all instances but the hybrid evolutionary algorithm [Blu06] find the best results on most instances. Most recently in 2009, Chimani has proposed an exact ILP-based algorithm using directed cuts [CKIL09] which outperforms the metaheuristic approaches. Garg in [GH97] considered the problem on euclidean graph and proposed an approximation algorithm for solving it. Additional references for this problem can be found in [EFHM97, MJ96, BMX01, BBXG00, MU01, Blu02, BE03].

[4] Also referred to as the k-minimum spanning tree (k-MST) problem, or just the $k$-tree problem.

The tabu search algorithm in [BB05] considers the neighborhood which consists of all $k$-cardinality trees generated by removing a leaf edge $e$ (i.e., the edge which has one endpoint of degree 1), which results in a $(k-1)$-cardinality tree $T_{k-1}$, and adding an edge from the set $IE(T_{k-1}) \setminus \{e\}$, where $IE(T_{k-1})$ is the set of edges that do not belong to $T_{k-1}$ and that have exactly one endpoint in $T_{k-1}$. The tabu search algorithm maintains two tabu lists which store the edges that participate in the moves: one list stores the added edge and the other stores the removed edge. The length of each tabu list *tbl* varies from *tbMin* and *tbMax* with an increment value *tinc*. At the beginning of each restart phase, *tbl* is set to *tbMin*. If the restart-best solution is not improved for a maximum number of specified iterations, the tabu length is increased by *tinc*. Whenever the restart-best solution is improved, the tabu length *tbl* is set back to *tbMin*. When *tbl* is greater than *tbMax*, the restart is performed.

We implement in Section 5.2 a tabu search (denoted by KCT_MTABU) using the `LS(Graph)` framework and applying the same local search schema of [BB05] but also exploiting an additional neighborhood based on so-called cyclic moves. A cyclic move adds an edge having two endpoints in the current $k$-cardinality tree which produces a cycle and removes another edge from this cycle, which results in a new $k$-cardinality tree. This additional neighborhood has been considered in [BBXG00]. We experimentally show how easily, shortly and flexibly to implement a local search algorithm for solving the KCT problem which gives good results.

### 2.5.2 The quorumcast routing (QR) problem

The quorumcast routing (QR) problem arises in distributed applications [CA94, DGTW96, WH04, Low98]. Given a weighted undirected graph $G = (V, E)$, each edge $e \in E$ associates with a cost $c(e)$. Given a source node $s \in V$, an integral value $q$ and a set $S \subseteq V$, the quorumcast routing problem consists in finding a minimum cost tree $T = (V', E')$ of $G$ spanning $s$ and $q$ nodes $S$. $T = (V', E')$ 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 $|Q| = q \wedge Q \cup \{s\} \subseteq V'$
4. The cost of $T = \sum_{e \in E'} c(e)$ is minimum over all subgraphs of $G$ with properties 1, 2, and 3

An exact [Low98] algorithm has been proposed for solving the QR problem but experiments were performed on small graphs (e.g., graphs with 30 nodes). Three heuristics have been proposed in [CA94] including Minimal Cost Path Heuristic (MPH), Improved Minimum Path Heuristic (IMP) and Modified Average Distance Heuristic (MAD). The idea of the MPH heuristic is to construct the solution in a greedy way. It starts from a partial solution (a tree under construction) containing only the source node $s$. At each step (called selection step), it selects the closest node $v$ of S that does not belong to the partial solution and inserts to the current partial solution all the

nodes of the corresponding shortest path from $v$ to the partial solution until the partial solution contains $q$ nodes of $S$. The main idea of the IMP algorithm is to repeat the MPH several times but at each selection step, it does not consider the nodes of $S$ that have been selected in any in previous MPH calls. Experimental results in this paper show that, among three heuristics, the IMP heuristic produces the best solutions. In [DGTW96], 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. This multispace search algorithm consider the QR problem where $S \equiv V$.

In [WH04], the authors considered the QR problem with additional constraint 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 have been conducted over graphs up to 200 nodes.

We implement in Section 5.3 a tabu search algorithm using the `LS(Graph)` framework and compare it with the IMP heuristic algorithm of [CA94]. 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.

### 2.5.3 The Edge-Disjoint Paths (EDP) problem

We are given an undirected graph $G = (V, E)$ and a set $T = \{\langle s_i, t_i\rangle \mid i = 1, 2, ..., |T|; s_i \neq t_i \in V\}$ 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 exists multually 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, the formulation of the EDP problem is given by:

| | | |
|---|---|---|
| *min* | $\sharp T'$ | (1) |
| *s.t.* | $T' \subseteq T$ | (2) |
| | $T'$ is *edp*-feasible | (3) |

This problem appears in many applications such as real-time communication, VLSI-design, routing and admission control in modern networks [AGLR94, CB96]. Existing techniques for solving this problem includes approximation algorithms [KS04, BS00, Kle96, CK03], greedy approaches [Kle96, KS01a] and Ant Colony Optimization (ACO) metaheuristic [BB07]. It has been shown in [BB07] that the ACO is the state-of-the-art algorithm for this problem. In that paper, the ACO algorithm has been compared with a Simple Greedy Algorithm (SGA) in [Kle96](the multi-start version).

The main idea of SGA is described as follows. It starts with an empty solution $S$ and it proceeds through the commodities in the order that is given as input. At each step $i$ ($\forall i = 1, 2, ..., |T|$), it considers the commodity $T_i = \langle s_i, t_i\rangle$ on the graph $G_i$ which is generated from $G$ by removing all the edges that are already in the paths of the solution $S$ under construction. If $s_i$ can be connected with $t_i$ in the graph, then we connect $s_i$ to $t_i$ by the shortest path $P_i$ and add $P_i$ to the solution $S$. The multi-start version of SGA (called MSGA) performs several calls to SGA but with different orders of the commodities to be processed.

The main idea of the ACO algorithm is summarized as follows. The construction of a solution consists of each ant $i$ building a path $P_i$ between the two endpoints of its commodity $T_i$. The solution constructed by this procedure is called ant solution which is a set of not necessarily edge-disjoint paths. From each ant solution, a *edp*-feasible solution can be extracted by iteratively removing the path which has most edges in common with other paths, until the remaining paths are mutually edge-disjoint. In this ACO algorithm, each pheromone model $\tau^i$ is used for building each path $P_i$, and each pheromone model $\tau^i$ consists of a pheromone value $\tau^i_e$ for each edge $e \in E$. In the path construction, each ant $i$ iteratively extends the partial path $P^j_i = \langle s_i = v_1, v_2, ..., v_k \rangle$ by selecting and adding an edge $e = (v_k, v_{k+1})$ based on a probability expression (see Section 2.2.3). The heuristic factor of this probability expression consists of two terms: $p(D_e)$ and $p(U_e)$ in which $p(D_e)$ specifies the influence of the distance from $v_k$ via $v_{k+1}$ to the goal $t_i$ and $p(U_e)$ determines the influence of the overall usage of the edge $e$ (it provides the information whether $e$ is already used in the path of another ant for the same solution).

In Section 5.5, we propose two models in `LS(Graph)`. We experimentally show competitive results compared with the ACO algorithm in [BB07]. This example illustrates how `LS(Graph)` can be used to implement more complex heuristics.

### 2.5.4 The resource constrained shortest path (RCSP) problem

The resource constrained shortest path problem (RCSP) [BC89] is the problem of finding the shortest path between two vertices on a network satisfying the constraints over resources consumed along the path. There are some variations of this problem, but we first consider a simplified version introduced and evaluated in [BC89] over instances from the OR-Library [Bea]. Given a directed graph $G = (V, E)$, each arc $e$ is associated with a length $c(e) \geq 0$ and a vector $r(e) \geq 0$ of resources consumed in traversing the arc $e$. Given a source node $s$, a destination node $t$ and two vectors $L$, $U$ of resources corresponding to the minimum and maximum amount that can be used on the chosen path (i.e., a lower and an upper limit on the resources consumed on the path). The length of a path $\mathcal{P}$ is defined as $f(\mathcal{P}) = \sum_{e \in \mathcal{P}} c(e)$. The resources consumed in traversing $\mathcal{P}$ is defined as $r(\mathcal{P}) = \sum_{e \in \mathcal{P}} r(e)$ The formulation of RCSP is then given by:

$$
\begin{array}{lll}
\textit{min} & f(\mathcal{P}) & (1) \\
\textit{s.t.} & L \leq r(\mathcal{P}) & (2) \\
 & r(\mathcal{P}) \leq U & (3) \\
 & \mathcal{P} \text{ is a path from } s \text{ to } t \text{ on } G & (4)
\end{array}
$$

In the paper [BC89], Beasley and Christofides describe a model for solving the RCSP problem considering constraint over both minimum (2) and maximum (3) resources consumed, but only (3) is considered in the experimentation (the vertor $L$ is set to 0). The constraint (2) appears when each edge associates with a benefit which represents the gain obtained when traversing this edge.

The RCSP problem with only constraints on the maximum resources consumed is also considered in [LHH07, KK01b, KK01a, KKT02, JSMR01, MK04, KKKM04, CRW08, DB03] among which, [DB03] and [CRW08] are most recent techniques. Du-

mitrescu and Boland [DB03] describe a label-setting algorithm combined with several preprocessing techniques in [DB03] while Carlyle et al. [CRW08] propose a Lagrangian relaxation with near-shortest-paths enumeration (LRE) techniques which gives better results on extensive data set including instances from OR-library [Bea], road networks of Maryland, Virginia, Washington D.C., grid graphs up to 135002 vertices and 404850 edges. The algorithm of [CRW08] lagrangianizes the side constraints, optimizes the lagrangian function by bisection search which provides dual lower bound of the objective function, identifies a feasible solution (often while optimizing the lagrangian function) which gives primal upper bound of the objective function, and closes optimality gap by enumerating near-shortest paths [CW03]. Notice that we cannot extend these techniques when the constraint (2) need to be added or it is required substantial programming effort.

In this thesis, we apply the `LS(Graph)` to the resolution of the RCSP problem with both constraint over lower bound and upper bound of resource consumed which will be presented in Section 5.4. This shows that by using the `LS(Graph)`, we can easily model and solve COP problems with arbitrary weights (negative or positive) on edges (vertices) while the non-negative weights on edges (vertices) condition is required for many dedicated algorithms.

### 2.5.5 The Routing and Wavelength Assignment with delay side constraint (RWA-D) problem

The WDM (Wavelength Division Multiplexing) optical networks [Muk06] provide the high bandwidth of communications. The routing and wavelength assignment (RWA) problem appears as essential problem on the 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 source nodes to their respective destination nodes and assigning wavelengths to these routes. A condition that must be satisfied is that two non edge-disjoint routes must be assigned different wavelengths. Normally, the number of available wavelength is limited and the number of requests is high. Two variants of the problem have been considered and 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 of two variants are NP-Hard [CGK92].

Scientific papers proposed different techniques for solving these problems e.g., exact method based on ILP formulation [CB96, KS01b, TMP02, OB03, RS95, LKLP02, JMT07, YLR10], heuristic algorithms [DR00, ZJM00, BM00, BYC97], metaheuristics including tabu search [JMY06, NR06] and Genetic [ARD99, BMP04, Hyy04]. These techniques have been experimented on realistic networks of small size (networks up to 27 nodes and 70 edges) but considered high number of connection requests. RWA with additional constraint has also been considered, for instance, [YCCL, ARD00].

In order to show the interest of the modeling framework, we consider the minRWA problem with side constraint (e.g., delay constraint) specifying that the cost of each route must be less than or equal to a given value. The objective here is not to show a

competitive model in comparison with state-of-the-art techniques for classical RWA problems. Rather, we show the flexibility of the modeling framework enabling the combination between `VarGraph` 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 $c : E \rightarrow \mathbb{R}$, $c(e)$ represents the delay in traversing $e$. Given a set of connection requests $R \subseteq V \times V$, $R = \{\langle s_1, t_1\rangle, \langle s_2, t_2\rangle, ..., \langle s_k, t_k\rangle\}$ and a value $D$. The RWA-D problem consists in finding routes $p_i$ from $s_i$ to $t_i$ and its wavelength 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 wavelength is minimized (objective function).

### 2.5.6 The Routing for Network Covering (RNC) problem

Above sections consider constrained path finding problems where vertices and edges cannot be repeated on a path. This section introduces a constrained walk (path where vertices and edges can be repeated) finding problem: the Routing for Network Covering (RNC) problem.

#### Context

The AROUND project [MSI06] being carried out at the MSI/IFI laboratory aims at designing and implementing a real-time decision support system for the rescue after natural disasters in urban areas. A team of autonomous robots that are capable of auto-organization explores the urban area in order to capture informations from the disaster. Rescue teams such as ambulances or firefighters are distributed to take care of victims, to extinguish fires, etc.

Historically, we have seen huge efforts for solving routing problems on networks in which the vehicle routing problem (VRP) [DR59] and capacitated arc routing problem (CARP) [GW81, hlk08] appears as a central problem in the fields of transportation, distribution and logistics. In the VRP problem, we have to route a fleet of identical vehicles from one or several depots in order to deliver goods for a set of $n$ customers. Each customer has a demand $q_i$ of goods ($i = 1, 2, ..., n$). Each vehicle has a capacity to deliver at most $Q$ quantity of goods for each tour, so it has to periodically return to the depot for reloading. The objective of the VRP problem is to determine a set of tours of minimum total travel time where each tour starts from and terminates at the depot, each customer must be visited exactly once, and the quantity of goods delivered on each tour must not exceed the vehicle capacity $Q$. In the CARP problem, a fleet of identical vehicles must be routed to serve a set of edges of a given urban network. More precisely, we are given a graph $G = (V, E)$, a node $v_0 \in V$ represents the depot, each edge $e$ of $G$ has a demand $d(e)$, a cost of serving $sc(e)$ and a cost of traversing without serving $dc(e)$. We are given a set of vehicle with a limited capacity

$Q$, CARP is the problem of finding a set of walks (each walk for one vehicle) starting and terminating at the depot of minimal total cost such that each edge is served by exactly one vehicle and the total demands of edges served by each vehicle do not exceed its capacity.

In this section, we inspire from the AROUND project and propose a problem relating to the VRP and the CARP problems, called routing for network covering (RNC) problem, which consists of routing a fleet of identical vehicles with limited capacity (e.g. the capacity of energy) from one or several depots on a transportation network in order to visit a set of specified edges of the network.

Three important constraints are taken into account:

1. Each vehicle travels a walk (or itinerary) which starts from and terminates at the depot,

2. Each vehicle cannot travel a walk whose cost is greater than a given value (its capacity),

3. A given set of streets of the urban area must be visited.

Both VRP and RNC problems have a common mission that is to route a fleet of vehicles with limited capacity on a transportation network to carry out some works. But the main differences between these problems are:

- In the RNC problem, vertices and edges can be repeated on each path while this is not allowed in the VRP problem.

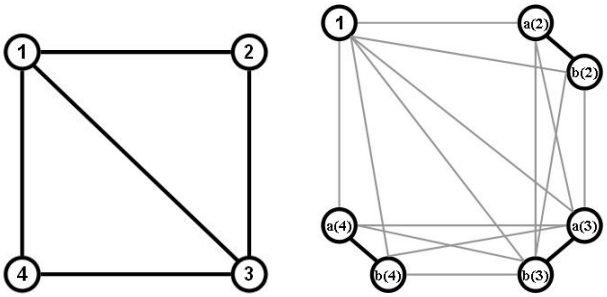
- In the VRP problem, constraints are defined over vertices of the graphs while in the RNC problem, constraints are specified over edges of the graphs.

Both CARP and RNC problems seek a set of closed walks with capacity constraints but in the RNC problem, the total demand for each walk is accumulated on all edges of the walk while in the CARP, the total demand for each walk is accumulated only on edges which are served by the walk.

The RNC problem is also related to the chinese postman problem [GJ79] where we have to find a cycle in a mixed graph (i.e., a graph that includes both directed and undirected edges) whose length does not exceed a given value and which visits each edge of the given graph at least once.

Several objective functions can be considered. In case where each vehicle has a given cost, we prefer to minimize the number of walks in order to minimize the total budget used. On the other hand, in urgent situations, the process of collecting information need to be performed as fast as possible. In such case, all paths of vehicles are carried out in parallel and we need to minimize the length of the longest walk.

There may exist different side constraints in the real-world situation but we consider in this section the most basic version of RNC problems. To our best knowledge, the RNC problem has not been considered before and there are thus no previous works for solving this problem.




a. hamilton graph b. 1-RNC graph

Figure 2.3: Example of reducing from a hamilton graph to a 1-RNC graph

**Problem definition**

Given an undirected weighted graph $G = (V, E)$ representing a transportation network and a vertex $d \in V$ representing the depot. A length function $c : E \rightarrow \mathbb{R}$ is defined on edges of $G$, $c(e)$ is the length of $e$.

The length of an itinerary $I$ (denoted by $c(I)$) is defined to be the sum of lengths of all edges of $I$:

$$c(I) = \sum_{e \in E(I)} c(e)$$

Given a vertex $d \in V$ representing the depot, a set $S$ of edges of $G$ ($S \subseteq E$) and a value $L$, a feasible itinerary is defined to be an itinerary starting from and terminating at $d$ whose length is less than or equal to $L$. A feasible solution to the RNC problem is a set of feasible itineraries that visit all edges of $S$. A feasible solution that has $k$ feasible itineraries is called $k-$feasible solution. RNC is the problem of determining a feasible solution having the smallest cardinality. An RNC problem instance is denoted by $\langle G, c, d, L, S \rangle$. The problem of determining whether or not there exists a $k$-feasible solution is called $k-$RNC problem.

**Complexity**

**Theorem 1** 1-*RNC problem is NP-complete.*

**Proof** We show that the hamilton cycle problem [GJ79] which is NP-complete can be reduced to this problem. Given an instance of hamilton problem which is an undirected unweighted graph $G = (V, E)$ ($V = \{v_1, v_2, ..., v_n\}$), we transform $G$ to an instance $G' = (V', E')$ of 1-RCN problem as follows:

1. We choose a node $v_1 \in V$, for each node $u \in V \setminus \{v_1\}$, we create two nodes $a(u), b(u)$. The set $V'$ is specified as follows: $V' = \{a(u), b(u) \mid u \in V \setminus \{v_1\}\} \cup \{v_1\}$. The set $V'$ is partitioned into $n$ partitions: $P(v_1) = \{v_1\}$, $P(v_i) = \{a(v_i), b(v_i)\}, \forall i = 2, 3, ..., n$.

2. The set $E'$ is specified as follows: $E' = A \cup B$ where $A = \{(x, y) \mid \exists (u, v) \in E \wedge x \in P(u) \wedge y \in P(v)\}$, $B = \{(x, y) \mid \exists u \in V \setminus \{v_1\} \wedge x, y \in P(u)\}$. We set $c(e) = 0, \forall e \in B$ and $c(e) = 1, \forall e \in A$. Value $L$ is set to be $n$ and the set $S$ is set to be $B$. Each edge that belongs to $B$ is called 0-edge and each edge that belongs to $A$ is called 1-edge.

This transformation can obviously be done in polynomial time. Figure 2.3 gives an example of the problem reduction. Figure 2.3a is the hamilton cycle instance and Figure 2.3b is the 1-RCN instance where $L = 4$ and $S = \{(a(2), b(2)), (a(3), b(3)), (a(4), b(4))\}$.

Suppose that there exists a hamilton cycle $\langle v_1, v_2, ..., v_n, v_1\rangle$ in $G$, we show there exists a cycle starting from and terminating at $v_1$ whose length is less than or equals to $n$ which passes all edges of $B$. The desired cycle on $G'$ is $\langle v_1, a(v_2), b(v_2), a(v_3), b(v_3), ..., a(v_n), b(v_n), v_1\rangle$.

On the other hand, suppose that there exists a cycle $\mathcal{C}$ starting from and terminating at $v_1$ on $G'$ such that:

1. $c(\mathcal{C}) \leq n$ (1),
2. $\mathcal{C}$ passes all edges $(a(v_2), b(v_2))$, $a(v(_3), b(v_3))$,..., $a(v_n)$, $b(v_n))$

we show there exists a hamilton cycle on $G$. Without generality, suppose that the cycle $\mathcal{C}$ is constituted by the concatenation:

1. itinerary $I_1$ from $v_1$ to $a(v_2)$
2. edge $(a(v_2), b(v_2))$
3. itinerary $I_2$ from $b(v_2)$ to $a(v_3)$
4. edge $(a(v_3), b(v_3))$
5. ...
6. edge $(a(v_n), b(v_n))$
7. itinerary $I_n$ from $b(v_n)$ to $v_1$

It is clear that $c(I_i) \geq 1, \forall i = 1, 2, ..., n$ because it starts from a partition and terminates at different partitions. Hence $c(\mathcal{C}) = \sum_{i=1}^{n} c(I_i) + \sum_{i=2}^{n} c(a(v_i), b(v_i)) \geq n$. In combining with the condition (1), we have $I_i = 1, \forall i = 1, 2, ..., n$. There is thus only one 1-edge on $I_i$ which is from a node of $P(v_i)$ to a node of $P(v_{i+1})$, that means there exists an edge from $v_i$ to $v_{i+1}$. Hence, we have a hamilton cycle $\langle v_1, v_2, v_3, ..., v_n, v_1\rangle$ on $G$. ∎

**Corollary 2** *Given an undirected weighted graph $G = (V, E)$, two vertices $u, v \in V$, a value $L$ and a set of edges $S \subseteq E$, the problem of determining whether or not there exists an itinerary from $u$ to $v$ whose cost is less than or equal to $L$ which passes all edges of $S$ is NP-complete.*

| Problems | Existing approaches |
|---|---|
| KCT | Tabu Search<br>Evolutionary Computation<br>ACO<br>Variable Neighborhood Search<br>ILP |
| QR | Heuristics (MPH, IMP, MAD)<br>Exhaustive Search |
| EDP | Greedy<br>ACO |
| RCSP | Lagrangian relaxation with subgradient optimization<br>Label-setting<br>Lagrangian relaxation with near-shortest-path enumeration |
| RWA | ILP<br>Tabu search<br>Genetic algorithms |
| RNC | (new problem, no existing works) |

Table 2.1: Problem examples and existing approaches for solving them

Theorem 1 shows that the RNC problem is NP-hard in general. In Section 5.7, we propose a simple model in `LS(Graph)` for solving the RNC problem. No existing work for solving this problem is available, no comparison can thus be made. This simple model can find optimal solutions in small problem instances. Experimental results on large instances show the feasibility of our approach.

### 2.5.7 Summary

Table 2.1 summarizes the problems considered in this thesis and existing techniques for solving them. The RWA problem has attracted huge research efforts with dedicated techniques for solving it. We do not consider the pure RWA problem[5]. Rather, we consider the RWA problem with delay side constraints as described above in order to show the modeling facility and flexibility and feasibility of `LS(Graph)` on small instances.

[5]Solving the pure RWA problem with `LS(Graph)` seems not to be efficient, especially, on large instances (large number of connection requests as considered in the literature) due to the complexity of the model.

# 3

# THE LS(GRAPH) FRAMEWORK

In this chapter, we describe the framework containing abstractions that allow to model and solve COT/COP problems by local search. The framework architecture is based on that of CBLS. At the core, graph variables are specified. Over these variables, graph invariants, graph functions and graph constraints are defined. These objects are sufficient to model problems, specifying their properties. Different search components are then performed on the model. The `LS(Graph)` framework is implemented on top of the COMET programming language, enabling the combination with built-in variables (i.e., `var{int}`, `var{float}`), invariants, functions and constraints of COMET. Sections 3.2, 3.3 focus on introducing two specific variables: tree variables and path variables. These sections also define COT and COP neighborhoods for COT/COP applications. To conclude the chapter, we give an overview of the modeling abstractions of the framework.

This chapter is based on our publications [PDV09, PDH10, PDDH10].

## 3.1 Architecture

As other structures of programs for solving combinatorial applications, the structure of a local search program in `LS(Graph)` consists of declaring graph variables, stating and posting graph invariants, graph functions and graph constraints and performing a search procedure.

**Graph variables** Graph variables have been introduced in the `CP(Graph)` computation domain [DDD05]. In this framework, the domain of a graph variable is a set of graphs modeled by a graph interval. Pruning techniques are applied to narrow this interval. Branching will split the interval, and the system explores each of resulting restricted intervals until an interval becomes singleton. In the `LS(Graph)` framework, a graph variable represents what we call dynamic graphs, trees, paths which can be modified over time (e.g., by edges insertion, removal, replacement actions). These

variables are the core of the problem modeling. At each step of computation, the value of a graph variable is a graph, a tree, or a path which is a candidate solution (or a component of a candidate solution in cases that a solution to the given problem is a set of trees, a set of paths or a set of trees and paths) to the problem.

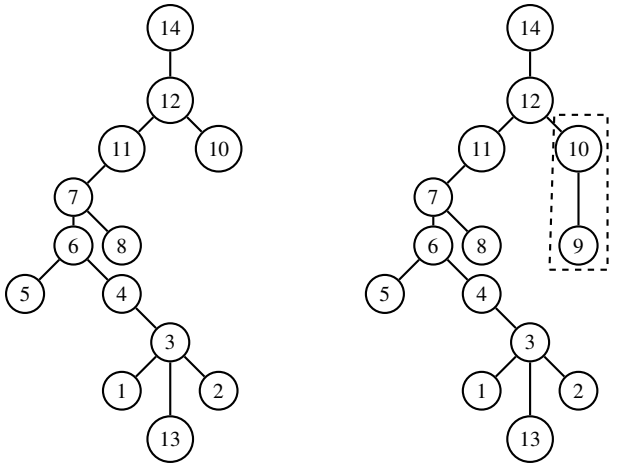
By convention, we use capital letters $A, B, ..., X, Y, Z$ to denote graph variables and lowercase $a, b, ..g, ..t$ to denote values of these variables.

**Graph invariant** Graph invariant is a concept describing objects which maintain some properties of dynamic graphs, trees, paths, such as the distances between vertices in a dynamic tree or the set of edges that can be inserted to a dynamic tree. Programming complexity can be reduced substantially by using graph invariants. Local search programs become clean, short and is close to high-level description of the algorithms. It is thus easy to understand. Users do not have to write code for updating invariants when a local move (i.e., a modification over graph variables) is taken. Instead, the stated graph invariants will be updated automatically by a built-in propagation engine.

**Graph function and graph constraint** Like graph invariant, graph function and graph constraint are concepts describing objects which maintain some properties of the considered application, such as the length of a path or the total weight of a tree or the number of violations of a given constraint. They specify the properties of the applications and are used to control the search components. Contrary to graph invariants, graph function and graph constraint are differentiable objects that allow to query the impact of various local moves over these properties (this feature is called differentiation [VM05]). Graph functions can be an objective function of the given application or they are used to state different constraints appearing in the given application. Graph functions and graph constraints can be combined flexibly by traditional arithmetic operators like $+, -, *$ or relation operators like $==, <=, >=$ for expressing more complex ones.

In the `LS(Graph)` framework, graph functions and graph constraints implement common interfaces (i.e., the differentiation interface which will be described later) enabling the genericity of search components. The search components do not need to know specific properties of the applications. Rather, they use provided interfaces of graph functions and graph constraints (e.g., differentiation methods) to guide the local search.

**The Search** Search is the procedure of continually moving from a candidate solution to another candidate solution on the neighborhood graph until some criteria are reached. The search is performed over a model, makes use of graph functions, graph constraints to direct it. At each step, the local search queries the quality of neighbors by using available interface of graph functions and graph constraints and decide to choose a desired neighbor and perform the local move (i.e., perform modification actions over graph variables of the model) to that neighbor.




a. current tree $tr$ b. $tr' = addEdge(tr, (9, 10))$

Figure 3.1: Illustrating an edge insertion

In the `LS(Graph)` framework, the search and the model are independent in the sense that we can state and post new constraints to the model without changing the search and we can also apply various (meta)heuristic strategies without modifying the model.

## 3.2 Tree variables and COT neighborhoods

### 3.2.1 Tree variable

Tree variable or *VarTree* is a concept describing dynamic trees which can be modified (e.g., by edges insertion, removal, replacement actions conserving the tree structure). A value of a tree variable is a tree. The domain of a tree variable $T$ is a set of trees modeled by a graph interval $[g_1, g_2]$, each value $tr$ of $T$ must satisfy the condition: $g_1 \subseteq tr \subseteq g_2$. *VarTree*s are used to model COT applications in which their domains are $[\perp, g]$, $g$ is the given graph under consideration.

### 3.2.2 Neighborhoods

A neighborhood of a tree is a set of trees generated by applying modification actions over the current tree.

**Related work** Some local search techniques for COT problems exploit the problem structures and propose dedicated neighborhood structures or apply specific moves without explicitly describing neighborhood structures. For instance, in the Steiner literature, specific moves have been proposed like steiner-vertex insertion and removal, key-path exchange and key-vertex elimination. The objective is to generate greedily

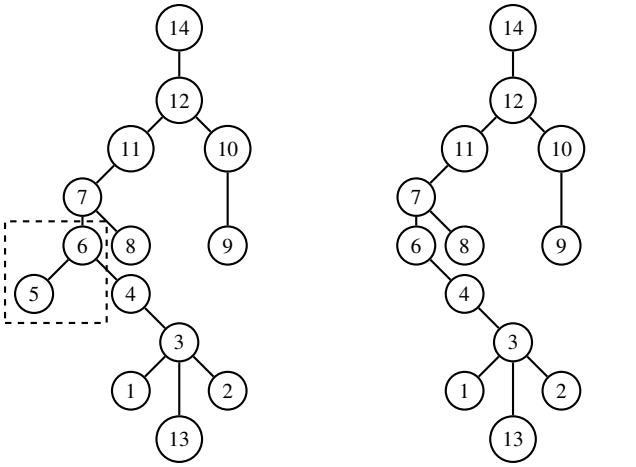



Figure 3.2: Illustrating an edge removal

a neighboring tree from the current tree without violating the Steiner constraint (i.e., the tree must contain all specified vertices). The basic idea of steiner-vertex insertion, removal and key-vertex elimination is to insert or remove some vertex (steiner vertex[1] or key vertex[2]) to the vertices set of the current tree and find a minimum spanning tree of the induced subgraph of the given graph associated with the new vertices set. The basic idea of key-path exchange is to replace a key path[3] of the current tree by another one (for more detail of these specific moves, see [DV97], [dAaRUW01], [BR01], [RS00], [RUW02] and references therein). In the Capacitated Minimum Spanning Tree problem, a very large-scale neighborhood structure has been proposed which applies the on node-based and tree-based multi-exchange. The basic idea is similar to that of above local search algorithms for Steiner tree problem in graphs which find for each subtree of the current rooted tree a minimum spanning tree of the induced subgraph of the given graph associated with the modified vertices set (by vertices exchange) of this subtree (for more detail, see [AOS03] and references therein). Obviously, these neighborhood structures are problem-dependent and cannot be applied when additional constraints need to be satisfied.

In other COT applications like edge-weighted $k$-cardinality tree (KCT) problem, degree-constrained minimum spanning tree (DCMST) problem, the proposed local search algorithms apply basic and generic moves. The neighborhood of a tree $tr$ for KCT problem proposed in [BB05] is the set of trees generated by removing a leaf edge $e$, which results in a tree $tr_1$, and adding an edge of $E(g) \setminus \{e\}$ that has exactly one end-point in $tr_1$ where $E(g)$ is the set of edges of the given graph $g$. For the DCMST problem, local search algorithms consider neighborhood generated by removing an

[1] steiner vertex is a vertex of the current tree that does not belong to the given terminals set.

[2] A key vertex is a nonterminal vertex of the current tree with degree at least 3.

[3] A key path of the current tree is the path that connects two crucial vertices (i.e., key vertex or terminal vertex) and has no internal crucial vertex.

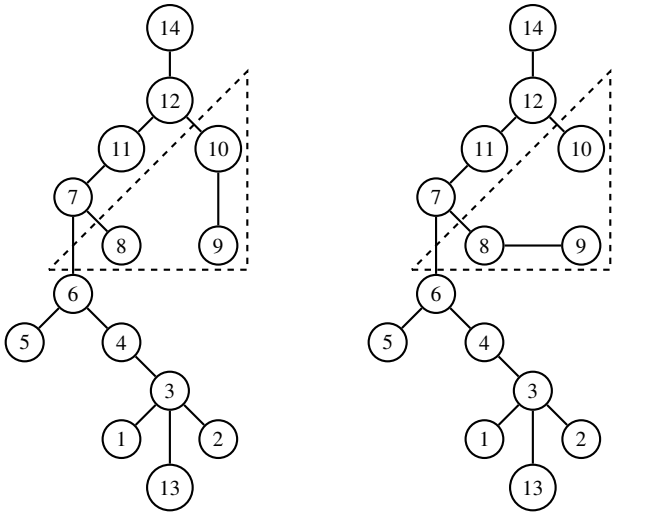



Figure 3.3: Illustrating an edge replacement

edge of the current tree, which results in two disconnected subtrees, and inserting another edge that reconnects these two disconnected components [RS02], [ALM06]. These neighborhood structures seem to be widely applicable and they can be explored in a generic way.

**COT neighborhoods** We now define some of neighborhood structures for COT applications considered in the framework. Given an undirected graph $g$ and a dynamic tree $tr$ of $g$ ($tr \subseteq g$), 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 *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:

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

Figure 3.1 illustrates the edge insertion. Figure 3.1a is the current tree $tr$ and Figure 3.1b is the neighboring tree $tr'$ generated by inserting the edge $(9, 10)$ to $tr$.

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 *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:

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

Figure 3.2 illustrates the edge removal. Figure 3.2a is the current tree $tr$ and Figure 3.2b is the neighboring tree $tr'$ generated by removing the edge $(5, 6)$ from $tr$.

3. **replace cycle edge action** [AMO93] 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 set of nodes of $tr$ is unchanged by this replacement. We denote *Repl*$(tr)$ the set of *replacing* edges of $tr$ and *Repl*$(tr, e)$ the set of *replacable* 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:

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

Figure 3.3 illustrates the edge replacement. Figure 3.3a is the current tree $tr$ and Figure 3.3b is the neighboring tree $tr'$ generated by replacing the edge $(9, 10)$ of $tr$ by the edge $(8, 9)$.

In practice, we can combine 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 $tr$ [4]. The set of such pairs of $\langle e_1, e_2 \rangle$ is denote by *RemvInst*$(tr)$. This kind of neighborhood has been considered in the tabu search algorithm of [BB05]. The formal definition of this neighborhood is the following:

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

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

## 3.3 Path variables and COP neighborhoods

### 3.3.1 Path variable

Path variable or *VarPath* is a concept describing dynamic paths which can be modified on a given graph. The domain of a path variable $P$ is a set of paths modeled by a

[4] This condition ensures tree property under the modification action.

graph interval $[g_1, g_2]$, each value $p$ of $P$ must satisfy the condition: $g_1 \subseteq G(p) \subseteq g_2$. *VarPath*s are used to model COP applications in which their domain are $[\perp, g]$, $g$ is the given graph under consideration.

### 3.3.2 Neighborhoods

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 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:

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

The 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)[5].

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

#### Related work

To our best knowledge, there exist few local search approaches for COP applications on general graphs. Moreover, these local search algorithms do not describe explicitly 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 local search programs.

On complete graphs, some local search algorithms have been applied for solving the traveling salesman problem [KP80], the vehicle routing problem [FGI05], [BGG+97]. In these approaches, a path is 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, exchange or changing position of vertices). These neighborhood structures cannot be applied for general graphs because a sequence of vertices can not be sured 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

[5] In some references, walks with no repeated vertices are referred to elementary paths.

and provides the data structure to evaluate their desirability. The rest of this section describes the neighborhood in detail. Our COP framework considers both directed and undirected graphs but, for simplifying the presentation, only undirected graphs are considered.

### 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 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 and simple updates to the RST (e.g., an edge replacement) will induce a new path from $s$ to the root. In the framework, we consider COP applications in which the sources and the destinations of paths are not fixed. Hence, the source $s$ and the destination (or root) of RST$(g, s, t)$ can also be changed.

Given an RST $tr$ over $(g, s, t)$, for a short, we denote $path(tr)$ 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 *RSTInduce(g,p)* the set of RSTs of $g$, rooted at $t(p)$ which induce $p$.

We define in the following section different neighborhood structures. 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.

### The EdgeReplacement-based Neighborhood

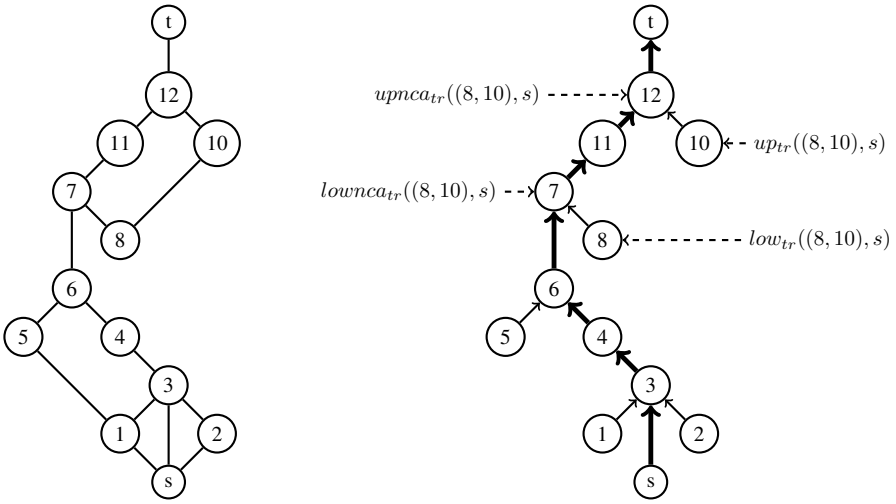
We first show in this section how to update an RST $tr$ over $(g, s, t)$ based on edge replacements to generate a new rooted spanning tree $tr'$ over $(g, s, t)$ which induces a new path from $s$ to $t$ in $g$: $path_{tr'}(s) \neq path_{tr}(s)$.

Let $tr$ be an RST over $(g, s, t)$, we consider the third basic neighborhood of $tr$ (see Section 3.2):

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

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 give 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 ances-




a. The undirected graph $g$ b. A spanning tree $tr$ rooted at $t$ of $g$

Figure 3.4: An Example of Rooted Spanning Tree

tors 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 3.4 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 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 replacable edge of $e$, and let $tr^1 = rep(tr,e',e)$. Let $su = upnca_{tr}(e,s)$ and $sv = lownca_{tr}(e,s)$. We have that $path_{tr^1}(s) \neq path_{tr}(s)$ if and only if (1) $su \neq sv$ and (2) $e' \in path_{tr}(sv,su)$.*

**Proof**

- We show that if the conditions 1 and 2 are satisfied, then $path_{tr^1}(s) \neq path_{tr}(s)$.

  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 $path_{tr^1}(s)$. Hence, $path_{tr^1}(s) \neq path_{tr}(s)$.

- We now show that if $path_{tr^1}(s) \neq path_{tr}(s)$, then the conditions 1 and 2 are satisfied.

We prove this by refutation. Suppose that $su = sv$. We denote $r = su = sv$ and $r_1 = nca_{tr}(u, v)$. We have $r\ Dom_{tr}\ r_1$ (because $r\ Dom_{tr}\ u, v$, so $r\ Dom_{tr}\ nca_{tr}(u, v) = r_1$). 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, $x\ Dom_{tr}\ nca_{tr}(s, u) = r$ (1). 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 $r\ Dom_{tr}\ fa_{tr}(x)$ (2). We see that (1) conflicts with (2). 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 $path_{tr^1}(s) = path_{tr}(s)$ (this conflicts with the hypothesis that $path_{tr^1}(s) \neq path_{tr}(s)$). 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, $path_{tr^1}(s) = path_{tr}(s)$ (this conflicts with the hypothesis). So we have $e' \in path_{tr}(su, sv)$.

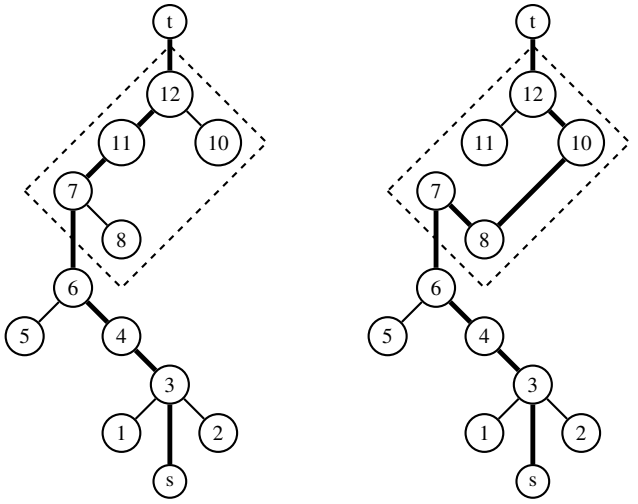
■

A *replacing* edge $e$ of $tr$ satisfying the property 1 is called a *preferred replacing* edge and a *replacable* edge $e'$ of $e$ in $tr$ satisfying condition 2 is called *preferred replacable* 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 replacable* edges of the *preferred replacing* edge $e$ on $tr$. We also denote *rep*$(tr, e', e)$ the action and the resulting RST of replacing a *preferred replacable* edge $e'$ by a *preferred replacing* edge $e$ on the RST $tr$. The EdgeReplacement-based neighborhood (called ER-neighborhood) of an RST $tr$ is defined as:

$$\textit{ERNP}_1(tr) = \{tr' = rep(tr, e', e) \mid e \in \textit{prefRepl}(tr), e' \in \textit{prefRepl}(tr, e)\}.$$

The action $rep(tr, e', e)$ is called a ER-move and is illustrated in Figure 3.5. In the current tree $tr$ (see Figure 3.5a), 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 replacable* 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 Figure 3.5b).

ER-moves ensure that the neighborhood is connected which is detailed in Proposition 2.




a. current tree $tr$ b. $tr' = rep(tr, (7,11), (8,10))$

Figure 3.5: Illustrating a Basic Move

**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 $\mathcal{P}$.*

**Proof** The proposition is proved by showing how to generate that instance $tr^k$. This can be done by Algorithm 14. 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 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 14 always terminates and returns a rooted spanning tree $tr$ inducing $\mathcal{P}$. Variable $S$ (line 1) stores the set of traversed edges.

■

### 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 $rep(tr, e'_1, e_1), \ldots, rep(tr, e'_k, e_k)$, denoted by $rep(tr, e'_1, e_1, e'_2, e_2, ..., e'_k, e_k)$, is defined as the application of the actions $rep(tr_j, e'_j, e_j)$ $j = 1, 2, ..., k$, 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***$(tr_j, e_j)$.

**Algorithm 14**: Moves($tr^0$)

**Input**: An instance $tr^0$ of RST*(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 performing $k \leq l$ 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 y ≠ fa_tr(x) then
7  |  |  ei ← (x, y);
8  |  |  eo ← replacable edge of ei that is not in S;
9  |  |  tr ← rep(tr, eo, ei);
10 |  S ← S ∪ {(x, y)};
11 |  x ← y ;
12 return tr;
```

**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 $e'_1, e_1, e'_2, e_2$,-..., $e'_k, e_k$ are all different, then these $k$ ER-moves are independent.*

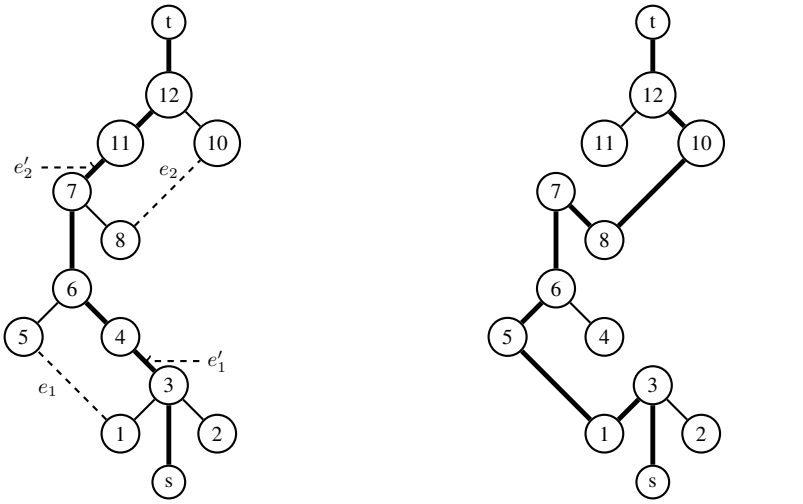
**Proof** All sequences of these basic moves are executable and the final results have the same set of edges $E(tr) \setminus \{eo_1, eo_2, ..., eo_k\} \cup \{ei_1, ei_2, ..., ei_k\}$. Thus the result trees of all execution sequences are the same. ∎

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 ER-$k$-move.

It remains to find some criterion to determine 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 $tr$, denoted by $e_1\ Dom_{tr}\ e_2$, 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 + 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 $u_2 = up_{tr}(e_2, s)$.*

**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 $su_1\ Dom_{tr}\ sv_1$, $sv_2\ Dom_{tr}\ su_1$, $su_2$




a. The Current Tree $tr$ (dashed edges are not included) b. $tr' = rep(tr, (7, 11), (8, 10), (3, 4), (1, 5))$

Figure 3.6: Illustrating a Complex Move

$Dom_{tr}$ $sv_2$, $e'_1$ 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 $tr_1 = rep(tr, e'_1, e_1)$. It is straightfoward to find that $\overline{T_{tr}}(v'_1)$ does not 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 they cannot be $su_1, sv_2, su_2$). Thus $nca_{tr_1}(s, u_2) = nca_{tr}(s, u_2) = su2$ and $nca_{tr_1}(u_1, v_2) = nca_{tr}(u_1, v_2) = x$. Moreover, from the property 1, we have $nca_{tr_1}(s, v_2) = nca_{tr_1}(u_1, v_2) = x$. Due to the fact that $sv_2$ $Dom_{tr_1}$ $su_1$ and $su_1$ $Dom_{tr_1}$ $u_1$, we have $sv_2$ $Dom_{tr_1}$ $u_1$. From the fact that $sv_2$ $Dom_{tr_1}$ $v_2$ and $sv_2$ $Dom_{tr_1}$ $u_1$, we have $sv_2$ $Dom_{tr_1}$ $nca_{tr_1}(u_1, v_2) = x$. Because $e'_2$ belongs to $path_{tr}(sv_2, su_2)$, we have $e'_2$ belongs to $path_{tr_1}(x, su_2)$. 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. ■

Figure 3.6 illustrates a complex move. In $tr$, two *preferred replacing* edges $e_1 = (1, 5)$ and $e_2 = (8, 10)$ are independent because $lownca_{tr}((8, 10), s) = 7$ which 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: $path_{tr}(s, 1)$ + $path_{tr}(5, 8)$ + $path_{tr}(10, t)$.

We specify now two other neighborhoods based on changes over sources and destinations of the paths. These neighborhoods will be applied when using sequences of

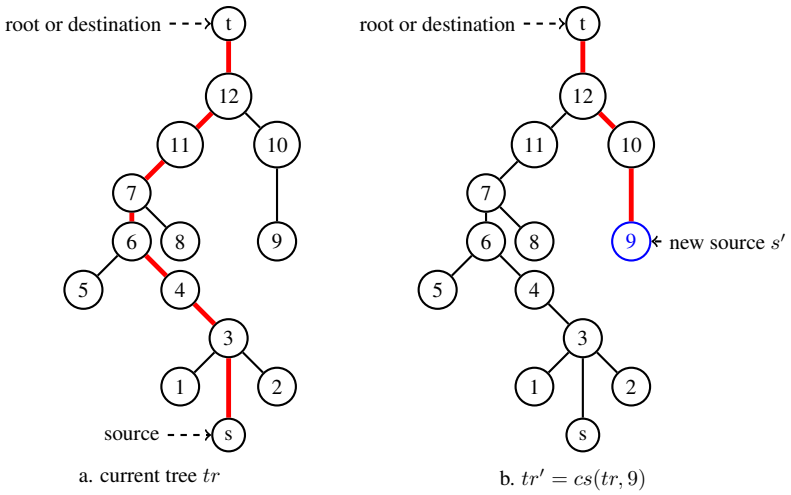



Figure 3.7: Illustrating a CS-move

RSTs for modeling general walks (vertices and edges can be repeated) as described later.

### The ChangeSource-based Basic Neighborhood

Given an instance $tr$ of RST$(g, s, t)$ and a vertex $s' \in V(g)$, we define the action $cs(tr, s')$ which change the source $s$ of $tr$ with the new one $s'$ which results a new RST $tr'$ inducing path from $s'$ to $t$ and we denote $tr' = cs(tr, s')$. This action is call ChangeSource move or CS-move. Figure 3.7 illustrates a CS-move. Figure 3.7a is the current RST which induces the path $\langle s, 3, 4, 6, 7, 11, 12, t\rangle$ and Figure 3.7b is the resulting RST $tr'$ by changing the source of $tr$ with the new source (vertex 9). $tr'$ induces the new path $\langle 9, 10, 12, t\rangle$.

We define now the neighborhood which is based on the change over the source. Given a RST $tr$ over $(g, s, t)$, the ChangeSource-based neighborhhood (or CS-neighborhood) of $tr$ is defined as

$$\mathit{CSNP}(tr) = \{tr' \mid tr' = cs(tr, s'), s' \in V(g) \setminus \{s\}\}$$

### The ChangeDestination-based Neighborhood

Given an instance $tr$ of RST$(g, s, t)$ and a vertex $t' \in V(g)$, we define the action $cr(tr, s')$ which change the root $t$ of $tr$ with the new one $t'$ which results a new RST $tr'$ inducing path from $s$ to $t'$ and we denote $tr' = cr(tr, t')$. This action is called ChangeDestination[6] move or CD-move. Figure 3.8 illustrates a CD-move. Figure 3.8a is the current RST which induces the path $\langle s, 3, 4, 6, 7, 11, 12, t\rangle$ and Figure3.8b is the resulting RST $tr'$ by changing the root of $tr$ with the new root (vertex 5). $tr'$ induces the new path $\langle s, 3, 4, 6, 5\rangle$.

[6]We also call ChangeRoot move because the root of the RST is the destination of the induced path.

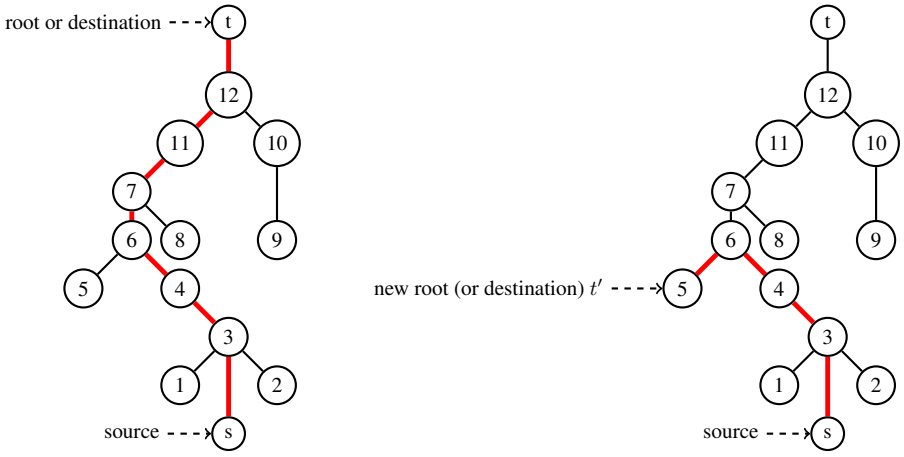



Figure 3.8: Illustrating a CD-move

We define now the neighborhood which is based on the change over the root. Given a RST $tr$ over $(g, s, t)$, the ChangeDestination-based neighborhhood (or CD-neighborhood) of $tr$ is defined as

$$\mathit{CDNP}(tr) = \{tr' \mid tr' = cr(tr, t'), t' \in V(g) \setminus \{t\}\}$$

ER-moves, CS-moves and CD-moves are called basic moves. ER-moves, CS-moves and CD-moves ensure that the neighborhood is connected which is detailed in Proposition 5.

**Proposition 5** *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 $k \leq l + 2$ basic moves, where $l$ is the length of $\mathcal{P}$.*

**Proof** The proposition is proved by showing how to generate that instance $tr^k$. This can be done by first applying two moves: $cs(tr^0, s')$ which results in an RST $tr'$ and $cr(tr', t')$ which results in an RST $tr''$ inducing path from $s'$ to $t'$ and then applying the method Moves($tr''$) in Algorithm 14 from the starting point $tr''$ instead of $tr^0$. ■

### General neighborhood

We introduce in this section a general neighborhood analyze its complexity and show how it can be exploited efficiently.

Given a RST $tr$ over $(g, s, t)$, the most general neighborhood of $tr$ is defined as:

$$\mathcal{N}_k(tr) = \{tr' \in \mathit{RSTInduce}(g, repl(p, path(tr)) \mid p \in \mathit{Paths(g,k)} \wedge s(p), t(p) \in path(tr)\}$$

Intuitively, $\mathcal{N}_k(tr)$ is the set of RSTs which induce the path $q$ where $q$ is a path constituted by replacing a subpath of the path induced by $tr$ ($path(tr)$) by a new path

of length $k$ (having $k$ edges). Generally, the size of $\mathcal{N}_k(tr)$ is too large because the number of paths having $k$ edges may be $\mathcal{O}(n^k)$ where $n$ size of the given graph $g$. Moreover, given a path $p$ in the above formula, the neighboring path $x$ is specified but there are many RSTs which induce $x$. In order to avoid overhead, in practice, we can apply this neighborhood with $k = 2, 3$ and we take randomly an RST among the set of RSTs inducing the same path. Normally, the neighboring solutions on above COP *ERNP*$_k$ are diverse and too different from the current solution. By exploring $\mathcal{N}_2(tr)$ and $\mathcal{N}_3(tr)$, we can avoid missing local minimum (if they exist) which are not too different from the current solution. Notice that the neighboring paths in $\mathcal{N}_1(tr)$ are covered by *ERNP*$_1$. Experiments have shown the benefit of using these neighborhoods, for instance, in Section 5.4.

### 3.3.3 Discussion

#### Diversification of the COP neighborhood

We discuss in this section how fundamental and robust the diversification of the proposed COP neighborhood based on rooted spanning tree is. Clearly, the above neighborhood based on rooted spanning tree features diversification in the sense that a neighboring path is usually very different from the current path.

Traditionally, a solution to a given standard CSP can be modeled by an array of variables and a neighbor of the current solution is generated by changing one variable or swapping two variables. Such a neighbor is close to the current solution: the difference between the current solution and its neighbor appears in only one or two positions. For COP problem on general graphs where a solution to the given problem is a path between two specified vertices, a neighboring path of the current path which differs from the current path of one or two edges may not exist or very few of such neighbors exists. It depends strongly on the topology of the given graphs. Figure 3.9 shows an illustrating example. The current path from 1 to 8 which is $\langle 1, 2, 3, 4, 5, 6, 7, 8\rangle$ has only two neighbors:$\langle 1, 2, 3, 13, 12, 11, 10, 9, 7, 8\rangle$ and $\langle 1, 2, 18, 17, 16, 15, 14, 6, 7, 8\rangle$ which are very different from it.

Moreover, for constraints like *PathContains*$(vp, S)$ specifying that a given set $S$ of vertices, edges must belong to the desired path $vp$, the diversification of COP neighborhood allows to explore neighboring paths which visit vertices, edges which are located far from the current path on the graph and which are not visited by the current path. In this case, such diverse neighboring paths reduce the violations of the *PathContains*$(vp, S)$ constraints while neighboring paths which are close to the current path (i.e., neighboring paths generated by replacing a subpath of the current path by a new path with a small length) might not reduce the violations of that constraint as illustrated in Figure 3.10. In this example, the current path is $\langle 1, 2, 3, 4, 5, 6, 7, 8\rangle$ and we want to find path which visit the vertices 10 and 14. Clearly, the vertices 10 and 14 are located far from the current path and they cannot be visited by non-diverse paths which are induced by RSTs in the neighborhood $\mathcal{N}_2$.

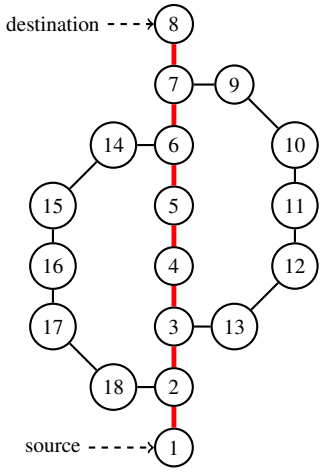



Figure 3.9: Illustrating the diversification of COP neighborhood

**Limitations**

Even though the proposed COP neighborhood can be applied on general graphs, we can see its limitations in some cases. First, if we consider a COP problem on a very large graph, for example, a graph with 10000 vertices, and we have to find a path of very small length, for example, a path of 2 or 3 edges, then maintaining a rooted spanning tree of this in this case is heavy while the COP neighborhood contains many non-promising solutions due to the fact that the length of neighboring paths are too divers. In this case, it is necessary to combine with some preprocessing techniques based on the structure of the considered problem for reducing the size of the graph. Second, if we consider a COP problem on a complete graph, and we need to find a path of fixed length (i.e., the number of edges of the path is fixed by $k$), then using the proposed COP neighborhood seems not to be efficient. In this case, using an array of variables $x[1..k+1]$ for modeling the path ($x[i]$ represents the $i^{th}$ vertex of the path) is simple and more efficient, for instance in the TSP problem [KP80], because any change over one or some variables induces a neighboring path and evaluating the quality of a neighbor as well as performing a move are more efficient.

## 3.4 Modeling constrained walk finding problems with a sequence of path variables

In many real-life routing applications, the desired routes have not necessary to be elementary (walk). For example, a vehicle can depart from a depot, traverse some roads and return to the depot and is allowed to pass visited roads. So it is required to model a route where vertices, edges can be repeated. Henceforth, we use the word "itinerary" to express routes where vertices, edges can be repeated which differs from

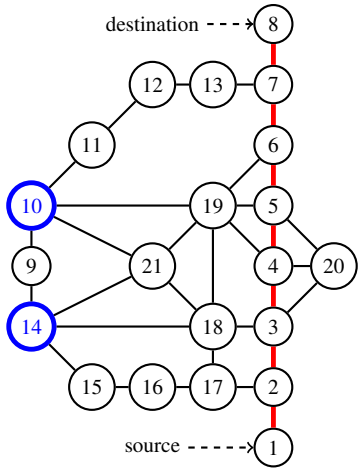



Figure 3.10: Illustrating the robustness of the diversification of COP neighborhood

"path" where vertices and edges can not be repeated. In this framework, we propose to model constrained itinerary finding problems by using a sequence of RSTs restricted by the continuity condition: the destination (or root) of a RST must be the source of the next RST in the sequence.

### 3.4.1 Sequence of rooted spanning trees

The basic idea here is to use a sequence of spanning trees: $\langle$RST($g, s = x_0, x_1$), RST($g, x_1, x_2$), ..., RST($g, x_{k-1}, x_k = t$)$\rangle$ to model a dynamic itinerary *VarItinerary*($g, s, t$) from $s$ to $t$ on the graph $g$. Each instance $it$ of *VarItinerary*($g, s, t$) is a sequence $\langle tr_0, tr_2, ..., tr_{k-1}\rangle$ where $tr_i$ is an instance of RST($g, x_i, x_{i+1}$), $\forall i = 0, 2, ..., k-1$. A constraint over the sequence which must implicitly hold at any moment is that the destination of $tr_i$ and the source of $tr_{i+1}$ is the same $\forall i = 0, 1, ..., k-2$.

Figure 3.11 shows an example where *VarItinerary*($g, s, t$)= $\langle$RST($g, s, 5$), RST($g$,-$5, t$)$\rangle$. Figure 3.11a is an instance $tr_1$ of RST($g, s, 5$) which induces the path $p_1 = \langle s, 3, 4, 8, 7, 6, 5\rangle$ and Figure 3.11b is an instance $tr_2$ of RST($g, 5, t$) which induces the path $p_2 = \langle 5, 4, 8, 10, 12, t\rangle$. Hence, the itinerary induced by *VarItinerary*($g, s, t$) is $p_1 + p_2 = \langle s, 3, 4, 8, 7, 6, 5, 4, 8, 10, 12, t\rangle$.

**Property 2** *Each instance of* $it = \langle tr_0, tr_1, ..., tr_{k-1}\rangle$ *of VarItinerary*($g, s, t$) *is an itinerary where vertices and edges are repeated at most* $k$ *times.*

### 3.4.2 Neighborhood

Given an instance $it = \langle tr_0, tr_1, ..., tr_{k-1}\rangle$ of *VarItinerary*($g, s, t$) = $\langle$RST($g, s = x_0, x_1$), RST($g, x_1, x_2$),..., RST($g, x_{k-1}, x_k = t$)$\rangle$ where $tr_i$ is an instance of RST($g$,-$x_i, x_{i+1}$), $\forall i = 0, 1, ..., k-1$, the neighborhood of $it$ is the set of itineraries gener-

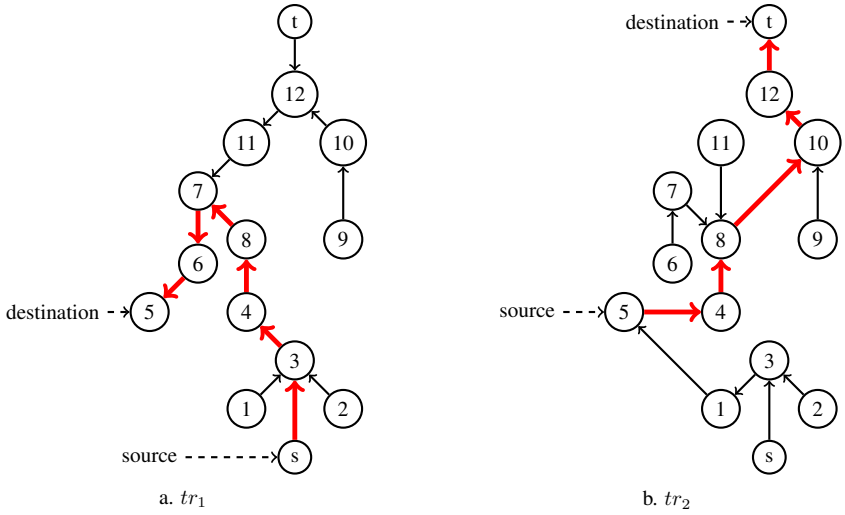



Figure 3.11: Example of itinerary

ated by taking a modification (local move) over $it$. In Section 3.3, each instance of RST$(g, s, t)$ has different neighborhoods.

We thus consider the first basic neighborhood of $it$ based on edges replacements (called EdgeReplacement-based neighborhood):

$$\mathit{ERNI}_1(it = \langle tr_0, tr_1, ...tr_i, ..., tr_{k-1}\rangle) =$$

$$\{\langle tr_0, tr_1, .., tr'_i, .., tr_{k-1}\rangle \mid tr'_i \in \mathit{ERNP}_1(tr_i), 0 \leq i \leq k-1\}$$

In the first neighborhood, we do not change the root (the destination) of each spanning tree when taking local moves. This leads to the fact that some vertices (roots of spanning trees) will always be in the itinerary and these vertices might not be in the desired solution. The search space is thus limited.

We define a second basic neighborhood of $it$ based on the changes over sources and destinations (called SourseDestinationChange-based neighborhood or SDC-neighborhood):

$$\mathit{SDCNI}_1(it = \langle tr_0, ..., tr_i, tr_{i+1}, ..., tr_{k-1}\rangle) =$$

$\{\langle tr_0, ..., tr'_i, tr'_{i+1}, ..., tr_{k-1}\rangle \mid tr'_i = cr(tr_i, y), tr'_{i+1} = cs(tr_{i+1}, y), 0 \leq i \leq k-2, y \in V(g)\}$. The action of taking a neighbor in SDC-neighborhood is called SDC-move.

Intuitively, a neighbor is generated by taking two successive spanning tree $tr_i$ and $tr_{i+1}$ and changing the root (the destination) of $tr_i$ and the source of $tr_{i+1}$ by a new vertex $y$. Figure 3.12 illustrates a SDC-move. Figures 3.12a and 3.12b is the current solution with the sequence $\langle tr_1, tr_2\rangle$ which induces the itinerary $\langle s, 3, 4, 8, 7, 6, 5, 5, 8, 10, 12, t\rangle$ and Figures 3.12c and 3.12d is a neighbor of the current solution generated

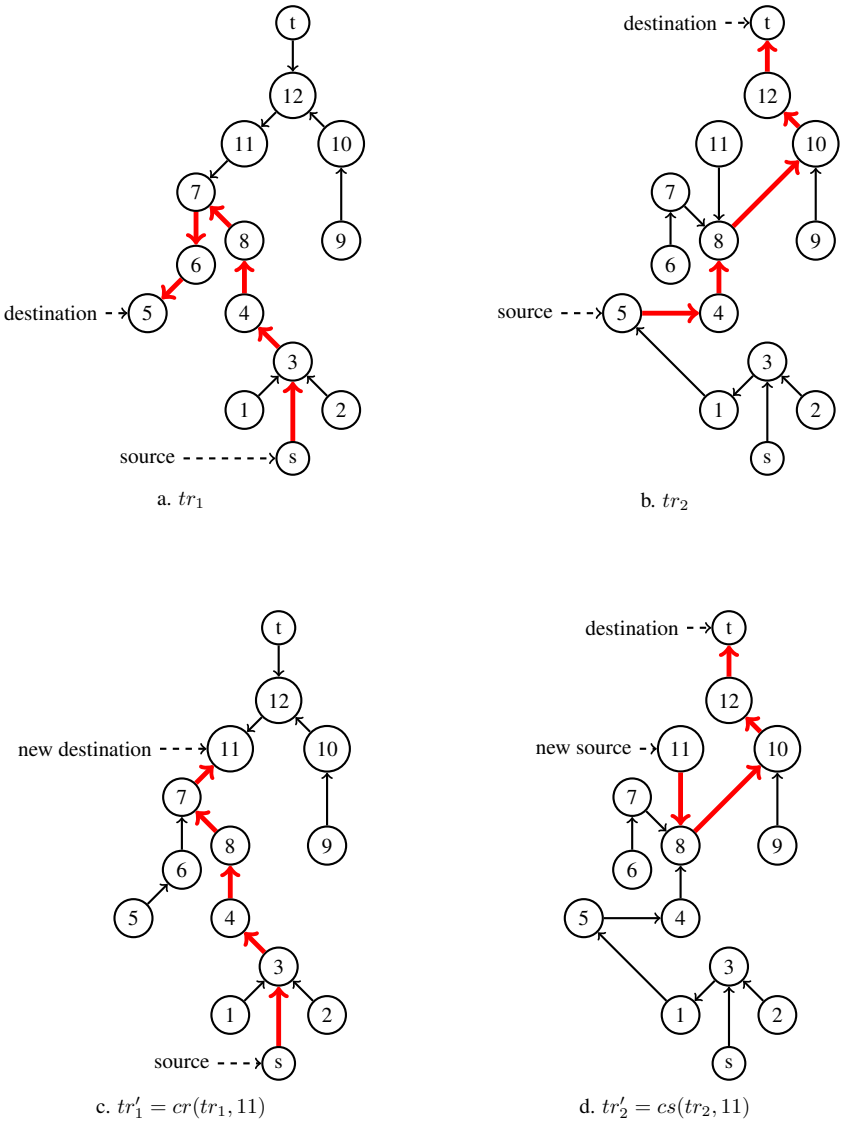



Figure 3.12: Illustrating a SDC-move

by changing the root of $tr_1$ with the new root 11 and changing the destination of $tr_2$ with the new destination 11. This neighboring solution is the sequence $\langle tr'_1, tr'_2 \rangle$ which induces the itinerary $\langle s, 3, 4, 8, 7, 11, 8, 10, 12, t \rangle$.

Even though the proposed modeling approach seems to be complex, in practice, we can exploit partially the neighborhood with dedicated heuristics. For instance, the first-improvement heuristic[7] allows to avoid the computation overhead.

## 3.5 Modeling abstractions

This section introduces abstractions available in the `LS(Graph)` framework for modeling COT/COP problems including common interfaces, classes representing graph variables, graph invariants, graph constraints, graph functions and search components. The core of the framework are graph variables (e.g., `VarTree`, `VarPath` objects representing dynamic trees, paths which can be changed) over which, graph invariants, graph constraints and graph functions are defined.

Graph invariants maintain properties of dynamic trees, paths. Graph constraints and graph functions are differentiable objects which not only maintain properties of dynamic trees, paths (for instance, the number of violations of a constraint or the value of an objective function) but also allow to determine the impact of local moves on these properties, a feature known as differentiation.

### 3.5.1 The Solver<LSGraph>

The `Solver<LSGraph>` is an abstraction representing objects which manage all variables (including COMET variables like var{int}, var{float} and graph variables like `VarTree`, `VarPath`), graph invariants, graph constraints and graph functions of the model and maintain a data structure representing the relations between these elements. The role of the `Solver<LSGraph>` is to perform a propagation method for updating invariants, functions and constraints whenever the variables over which these objects are defined are updated. The implementation of `Solver<LSGraph>` extends the `Solver<LS>` of COMET enabling the combination of COMET variables and graph variables.

### 3.5.2 Graph invariants

Invariant [VM05] is a concept describing objects maintaining some properties of the problem. In the `LS(Graph)` framework, we extend the invariant concept by introducing the graph invariant concept. Graph invariants are objects maintaining some properties of dynamic graphs, trees and paths, for example, the set of *replacing* edges of a dynamic tree, nearest common ancestors of all pairs of two vertices on a dynamic tree, etc. Users specify invariants in the model statement and these invariants are used for stating functions and constraints of the problem and for the search procedures. An

[7] The neighborhood is explored until a solution that is better than the current solution is found.

```
1.interface Invariant<LSGraph> extends Invariant<LS>{
2.  Solver<LSGraph> getLSGraphSolver();
3.  VarGraph[] getVarGraphs();
4.  void initPropagation();
5.  void propagateAddEdge(VarTree vt, Edge e);
6.  void propagateRemoveEdge(VarTree vt, Edge e);
7.  void propagateReplaceEdge(VarTree vt, Edge e1, Edge e2);
8.  void propagateReplaceEdge(VarPath vp, Edge e1, Edge e2);
9.}
```

Figure 3.13: interface of graph invariants (partial description)

```
1.interface Differentiation<LSGraph>{
2.  float getAddEdgeDelta(VarTree vt, Edge e);
3.  float getRemoveEdgeDelta(VarTree vt, Edge e);
4.  float getReplaceEdgeDelta(VarTree vt, Edge e1, Edge e2);
5.  float getReplaceEdgeDelta(VarPath vp, Edge e1, Edge e2);
6.  float getDeltaWhenUseReplacingEdge(VarPath vp, Edge[] e);
7.  float getReplaceSubPath(VarPath vp, LSGraphPath p);
8.}
```

Figure 3.14: differentiation interface (partial description)

update over variables[8] i.e., a move is performed induces a propagation method that updates all graph invariants associated with these variables thanks to the precedence graph maintained in the `Solver(LSGraph)`. Users do not have to call the procedures for updating these graph invariants.

All graph invariants must implement the `Invariant<LSGraph>` interface which is partially depicted in Figure 3.13. The interface features all the presented moves on COT and COP neighborhoods. Line 2 returns a `Solver(LSGraph)` object which manages all graph variables, graph invariants of the model. Line 3 returns the list of graph variables[9] over which the graph invariant is defined. Method in line 4 is called when the model is closed. Lines 5-8 are some propagation methods corresponding to different local moves.

### 3.5.3 Graph functions and graph constraints

Graph constraints and graph functions are differentiable objects which not only maintain properties of dynamic trees, paths (for instance, the number of violations of a constraint or the value of an objective function) but also allow to determine the impact of local moves on these properties, a feature known as differentiation. Graph functions and graph constraints are used for stating constraints and objective function of the given problem which are then used for controlling the local search procedure. The differentiation is used for evaluating the quality of a neighbor e.g., by querying the

[8] Variables here include var{int}, var{float}, `VarTree, VarPath`.

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

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

Figure 3.15: interface of graph constraints (partial description)

variation of the cost of the current path when an edge replacement is taken, we know how good a neighboring path is before making a choice.

The differentiation interface is depicted in Figure 3.14. The differentiation methods evaluate the impact of various local moves, for instance, `getAddEdgeDelta(VarTree vt, Edge e)` computes the variation of the value of the function when the edge `e` is added to the tree `vt`; the method in line 6 returns the variation of the value of the function when the *replacing* edge `e` is applied[10]. Method in line 7 evaluates the impact of moves where the subpath of `vp` between two endpoints of `p` is replaced by `p` (see the definition of most general COP neighborhood $\mathcal{N}$ at the beginning of Section 3.3.2). It enables exploration of other neighborhoods than the $NP_1$.

Figure 3.15 depicts the interface of graph constraints in which the method in line 1 returns the violations of the constraint. Line 2 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 to any violations. This information may be useful when applying multistage heuristics.

Users can also extend the `LS(Graph)` system by constructing their own invariants, graph functions, and graph constraints. These new classes must implement above interfaces.

### 3.5.4 Search components

After stating the problem model, one can perform a local search procedure for finding high-quality solutions. By using graph invariants and differentiable objects (graph functions and graph constraints) one can apply various (meta)-heuristic strategies for the local search.

In order to illustrate the modeling and the search component, we give an example in Figure 3.16 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 1-14 in which line 1 creates a `Solver<LSGraph> ls` and lines 2-3 initialize randomly a tree variable `vt` with `k` edges of a given undirected graph `g` associated with `ls`. `rpl` (line 4) is a graph invariant representing the set of *replacing* edges of `vt`. Lines 6-12 state and post constraints on degree and diameter of the tree `vt` to a graph constraint system `gcs` which is declared in line 9. Whenever the model is closed (line 14), the

[10]When a local move $replaceEdge(tr, e', e)$ is applied with the neighborhood $NP_1$, the resulting path depends only on the *replacing* edge $e$ used, not on the *replacable* edge $e'$.

```
1   Solver<LSGraph> ls();
2   int k = g.numberOfVertices()-1;
3   VarTree vt(ls,g,k);
4   ReplacingEdgesVarTree rpl(ls,vt);

6   DegreeAtMost degreeC(vt,maxDe);
7   DiameterAtMost diameterC(vt,0,maxDia);

9   ConstraintSystem<LSGraph> gcs(ls);
10  gcs.post(diameterC);
11  gcs.post(degreeC);
12  gcs.close();

14  ls.close();

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

Figure 3.16: Model for bounded diameter and degree constrained spanning tree

`initPropagation` methods of all graph invariants are called to initialize the values and internal data structures of these objects. The search is given in lines 16-24 which is a simple greedy search. At each iteration, we explore the $NT_3$ neighborhood and choose the best one w.r.t. the graph constraint system `gcs`: we choose a *replacing* edge `ei` and a *replacable* edge `eo` of `ei` such that the number of violations of `gcs` reduces most when `eo` is relaced by `ei` (see method `getReplaceEdgeDelta(vt,eo,ei)`). Line 21 is the local move which induces automatically a propagation to update all graph invariants, constraints defined over it (e.g., `rpl`, `degreeC`, `diameterC`) thanks to a 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 heuristics local search in the search component without changing the model.

In the framework, we provide a set of built-in generic neighborhood exploration procedures (see Appendix C) and search components (e.g., adaptive tabu search described in Appendix B). This enables users to use the framework as a black box and solve the problem in a declarative way. For illustrating these generic neighborhood exploration procedures, we describe one of them, i.e., the exploration of the $ERNP_1$ neighborhood in a tabu search with aspiration criterion (see Appendix for other generic neighborhood exploration procedures). The procedure (see Figure 3.17) consists of exploring neighborhood, choosing a best move or a first improvement (depends on

the input parameter `firstImprovement`) in term of a graph constraint `c` (see parameters in line 2). `tbIn, tbOut` are array tabu, each pair `tbIn[i], tbOut[i]` represent tabu list for *preferred replacing* edge and *preferred replacable* edge for the `VarPath _vps[i]` of the consider array of `VarPath _vps` of the given model. `it` is the current iteration of the local search and `fgb` is the best violations of the constraint `c` found so far. `Neighborhood N` is a COMET abstraction which stores different moves (as closure) and their evaluations. We describe now the content of this generic neighborhood exploration procedure. The array `_vps` of `VarPath`s of the model are scanned (line 8). For each `VarPath vp`, we explore its $ERNP_1$ neighborhood. `repl` in line 10 is a graph invariant representing the set of *preferred replacing* edges of `vp`. For each edge `e1` of `repl`, the variation `d` of the number of violations of the constraint `c` when the *preferred replacing* edge `e1` is applied (for the edge replacement) is computed (lines 11-12). If acceptance criterion is satisfied: the edge `e1` is not tabu at the current iteration `it` or the consider neighbor improves the best violations found so far `fgb` (line 14), we update the quality of this neighbor (lines 15-19). The chosen move is submitted to the `Neighborhood N` in lines 28-43: we choose randomly a *preferred replacable* edge `sel_eo1` of the selected *preferred replacing* edge `sel_ei1` (lines 32-34) and then submit the move and its evaluation `eval` in lines 37-42. Lines 38-41 perform the move (whenever it is called): making tabu two selected edges `sel_eo1, sel_ei1` (lines 38-39) and replacing `se_eo1` by `sel_ei1` on the selected `VarPath sel_vp`.

Table 3.1 gives a part of abstractions of the `LS(Graph)` framework including graph variables, fundamental graph invariants, graph functions, graph constraints, and some built-in generic tabu search components (see Appendix A for the manual of these classes). The `LS(Graph)` framework is open that allows users to design and implement their own graph invariants, graph functions, graph constraints respecting the specified interfaces as well as different local search components in order to integrate them to the system.

```
1   void exploreTabuMinReplace1Move1VarPath(Neighborhood N,
        Constraint<LSGraph> c,
2     GTabuEdge[] tbIn, GTabuEdge[] tbOut, int it, float fgb, bool
          firstImprovement){

4     Edge sel_ei1 = null;
5     int ind = -1;
6     float eval = System.getMAXINT();

8     forall(j in _vps.rng()){
9       VarPath vp = _vps[j];
10      ReplacingEdgesMaintainPath repl = _mapReVarPath{vp};
11      forall(e1 in repl.getSet()){
12        float d = c.getDeltaWhenUseReplacingEdge(vp,e1);

14        if(!tbIn[j].isTabu(e1,it) || d + c.violations() < fgb){
15          if(d < eval){
16            eval = d;
17            ind = j;
18            sel_ei1 = e1;
19          }
20          if(firstImprovement && eval < 0)
21            break;
22        }
23      }
24      if(firstImprovement && eval < 0)
25        break;
26    }

28    if(ind > -1){
29      VarPath vp = _vps[ind];
30      Edge sel_eo1 = null;

32      select(eo1 in getPreferredReplacableEdges(vp,sel_ei1)){
33        sel_eo1 = eo1;
34      }

36      if(sel_eo1 != null)
37        neighbor(eval,N){
38          tbIn[ind].makeTabu(sel_eo1,it);
39          tbOut[ind].makeTabu(sel_ei1,it);

41          vp.replaceEdge(sel_eo1,sel_ei1);
42        }
43    }
44  }
```

Figure 3.17: Exploring the $ERNP_1$ neighborhood

| 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 |

Table 3.1: Some graph invariants, functions and constraints of the framework (partial description)

## 3.6 Complexity

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

### 3.6.1 VarTree and Nearest Common Ancestors

For facilitating the manipulation of 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 [ST83, ST85], topology trees [Fre97], ET-trees [HK99], top trees [AHLT05, TW05], RC-trees [ABH$^+$04] (and 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. These data structures have been experimentally studied in [TW09]. These data structures are dedicated to solving specific network algorithms, for instance 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 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 nodes. At each step of the local search process, the system explores a neighborhood, queries the quality of all neighbors and decides to choose one neighbor to move. Usually, the neighborhood is large and the neighborhood exploration should be as fast as possible. This exploration requires to frequently perform the above queries over dynamic rooted trees. The queries over dynamic trees like the nearest common ancestors of all pairs of two nodes, the father of a node, the set of nodes, edges should thus be as fast as possible. For this purpose, we use direct data structure for tree by maintaining the father of each node, sets for storing nodes, edges and 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 nearest common ancestors take $\mathcal{O}(1)$ instead of $\mathcal{O}(\log n)$.

Concerning the nearest common ancestors problem, Bender et al. [BFCP$^+$05] presented a simple optimal algorithm for trees which is a sequentialized version of the more complicated PRAM algorithm of Berkman and Vishkin [BV93]. 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 [BFCP$^+$05] by an incremental implementation. That means, we update partially the data structure whenever the tree is modified (i.e., by add, remove, replace edges) instead of recomputing it from scratch. The incremental implementation does not improve the time complexity in the worst case ($\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 10000 update actions (add, remove, replace edges) conserving the size of the tree. Experimental results showed that the incremental implementation

is about 1.6 times faster than recomputing from scratch.

### 3.6.2 VarPath

The implementation of `VarPath` is based on the data structure for `VarTree`. It maintains incrementally a set of *preferred replacing* edges under edge replacement actions. Consider a $rep(tr, (u_2, v_2), (u_1, v_1))$ action over an RST $tr$ over $(g, s, t)$ (we denote $tr' = rep(tr, (u_2, v_2), (u_1, v_1))$). We show how to update incrementally *prefRepl*$(tr)$. We denote $r = nca_{tr}(s, u_2)$ and $y = nca_{tr}(s, v_2)$. Without loss of generality, suppose that $r\ Dom_{tr}\ y$ and $u_1\ Dom_{tr}\ v_1$. We denote $r_1, r_2$ respectively the children of $r$ that dominate $u_1$ and $u_2$ (see an example in Figure 3.18). The set of vertices of $tr$ is partitioned into following subsets:

- $S_1 = \{x \in V(tr) \mid r\ \overline{Dom}_{tr}\ x\}$
- $S_2 = \{x \in V(tr) \mid r_2\ Dom_{tr}\ \mathrm{x}\}$
- $S_3 = \{x \in V(tr) \mid r\ Dom_{tr}\ x \wedge r_1\ \overline{Dom}_{tr}\ x \wedge r_2\ \overline{Dom}_{tr}\ x\}$
- $S_4 = \{x \in V(tr) \mid r_1\ Dom_{tr}\ x \wedge v_1\ \overline{Dom}_{tr}\ x\}$
- $S_5 = \{x \in V(tr) \mid v_1\ Dom_{tr}\ x \wedge y\ \overline{Dom}_{tr}\ x\}$
- $S_6 = \{x \in V(tr) \mid y\ Dom_{tr}\ x \wedge z\ \overline{Dom}_{tr}\ x\}$
- $S_7 = \{x \in V(tr) \mid z\ Dom_{tr}\ x\}$

As pointed out in Section 3.3.2, an edge $e = (u, v)$ is a *preferred replacing* edge of an RST $tr$ iff $nca_{tr}(s, u) \neq nca_{tr}(s, v)$. So for updating the set of *preferred replacing* edges of an RST $tr$ over $(g, s, t)$, we first characterize the change over nearest common ancestor of $s$ and each node $x$ from $tr$ to $tr'$ in the following properties.

**Property 3** *With above notations, we have:*

- $nca_{tr}(s, x) = nca_{tr'}(s, x), \forall x \in S_1 \cup S_3 \cup S_6$
- $nca_{tr}(s, x) \neq nca_{tr'}(s, x), \forall x \in S_2 \cup S_4 \cup S_5 \cup S_7$
- $nca_{tr'}(s, x) = nca_{tr}(u_1, x), \forall x \in S_2$
- $nca_{tr'}(s, x) = r, \forall x \in S_4$
- $nca_{tr'}(s, x) = y, \forall x \in S_5$
- $nca_{tr'}(s, x) = nca_{tr}(v_1, x), \forall x \in S_7$

**Property 4** *Let $e = (u, v) \in$ Repl$(tr)$. We distinct the following cases:*

- $u \in S_1 \wedge v \in S_1$*: we have $nca_{tr}(s, u) = nca_{tr'}(s, u)$ and $nca_{tr}(s, v) = nca_{tr'}(s, v)$. This means that $e$ is preferred replacing edge of $tr$ if and only if it is preferred replacing edge of $tr'$.*

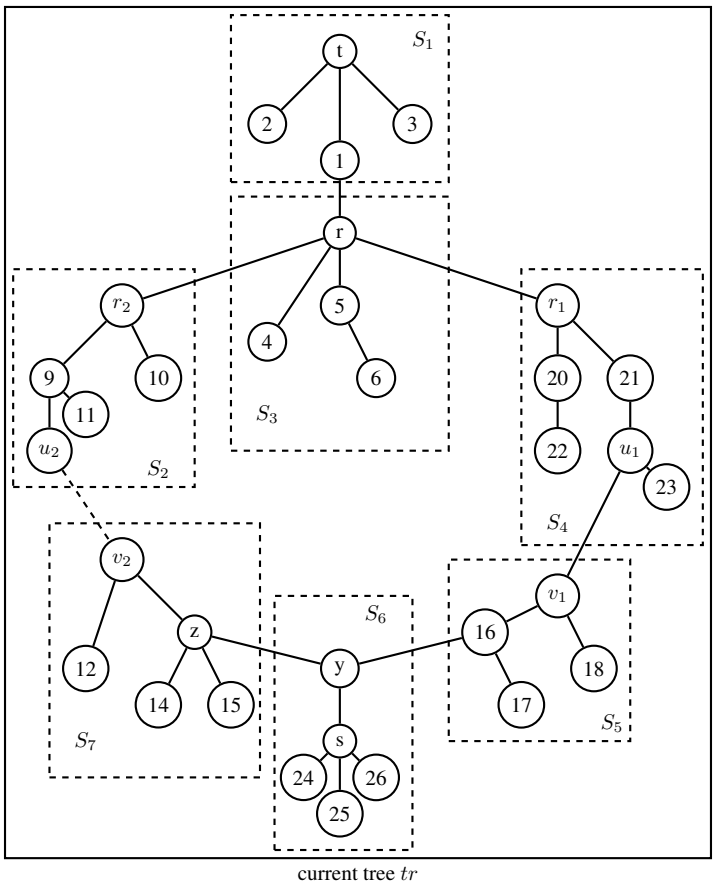



Figure 3.18: Ilustrating the update of *preferred replacing* edges under the $replaceEdge(tr, (u1, v1), (u2, , v2))$ action

- $u \in S_1 \wedge v \in S_2 \cup S_3 \cup S_4 \cup S_5 \cup S_6 \cup S_7$*: we have* $nca_{tr}(s, u) \neq nca_{tr}(s, v) \wedge nca_{tr'}(s, u) \neq nca_{tr'}(s, v)$*. This means* $e$ *is preferred replacing edge of both* $tr$ *and* $tr'$*.*

- $u \in S_2 \wedge v \in S_2$*: we have* $nca_{tr}(s, u) = nca_{tr}(s, v)$*. That means* $e$ *is not preferred replacing edge of* $tr$*. We can also see that* $nca_{tr'}(s, x) = nca_{tr}(u_2, x), \forall x \in S_2$*. So if* $nca_{tr}(u_2, u) \neq nca_{tr}(u_2, v)$ *then the edge* $e$ *will be added to prefRepl*$(tr)$ *in the update phase.*

- $u \in S_2 \wedge v \in S_3$*: we have* $nca_{tr}(s, u) = nca_{tr}(s, v) \wedge nca_{tr'}(s, u) \neq nca_{tr'}(s, v)$*. That means* $e$ *is not a preferred replacing edge of* $tr$ *but of* $tr'$*. The edge* $e$ *will thus be added to the prefRepl*$(tr)$ *in the update phase.*

- $u \in S_2 \wedge v \in S_4 \cup S_5 \cup S_6 \cup S_7$*: we have* $nca_{tr}(s, u) \neq nca_{tr}(s, v) \wedge nca_{tr'}(s, u) \neq nca_{tr'}(s, v)$*. That means* $e$ *is preferred replacing edge of both*

$tr$ *and* $tr'$.

- $u \in S_3 \wedge v \in S_3$*: we have* $nca_{tr}(s,u) = nca_{tr}(s,v) \wedge nca_{tr'}(s,u) = nca_{tr'}(s,v)$*. That means* $e$ *is neither preferred replacing edge of* $tr$ *nor of* $tr'$*.*

- $u \in S_3 \wedge v \in S_4$*: we have* $nca_{tr}(s,u) \neq nca_{tr}(s,v) \wedge nca_{tr'}(s,u) = nca_{tr'}(s,v)$*. That means* $e$ *is preferred replacing edge of* $tr$ *but* $tr'$*. The edge* $e$ *will thus be removed from prefRepl*$(tr)$ *in the update phase.*

- $u \in S_3 \wedge v \in S_5 \cup S_6 \cup S_7$*: we have* $nca_{tr}(s,u) \neq nca_{tr}(s,v) \wedge nca_{tr'}(s,u) \neq nca_{tr'}(s,v)$*. That means* $e$ *is preferred replacing edge of both* $tr$ *and* $tr'$*.*

- $u \in S_4 \wedge v \in S_4$*: we have* $nca_{tr'}(s,u) = nca_{tr'}(s,v)$*. That means* $e$ *is not preferred replacing edge of* $tr'$*. So if* $e$ *is in prefRepl*$(tr)$*, then it will be removed from prefRepl*$(tr)$ *in the update phase.*

- $u \in S_4, v \in S_5 \cup S_6 \cup S_7$*: we have* $nca_{tr}(s,u) \neq nca_{tr}(s,v) \wedge nca_{tr'}(s,u) \neq nca_{tr'}(s,v)$*. That means* $e$ *is preferred replacing edge of both* $tr$ *and* $tr'$*.*

- $u \in S_5 \wedge v \in S_5$*: we have* $nca_{tr'}(s,u) = nca_{tr'}(s,v)$*. That means* $e$ *is not preferred replacing edge of* $tr'$*. So if* $e$ *is in prefRepl*$(tr)$*, then it will be removed from prefRepl*$(tr)$ *in the update phase.*

- $u \in S_5 \wedge v \in S_6$*: we have* $nca_{tr}(s,u) \neq nca_{tr}(s,v)$ *and* $nca_{tr'}(s,u) = y$*. So if* $nca_{tr}(s,v) = y$ *then* $e$ *is preferred replacing edge of* $tr$ *but not of tr'. In this case, the edge* $e$ *will be removed from prefRepl*$(tr)$ *in the update phase. If* $nca_{tr}(s,v) \neq y$ *then* $e$ *is preferred replacing edge of both* $tr$ *and* $tr'$*.*

- $u \in S_5 \wedge v \in S_7$*: we have* $nca_{tr}(s,u) \neq nca_{tr}(s,v)$ *and* $nca_{tr'}(s,u) \neq nca_{tr'}(s,v)$*. That means* $e$ *is preferred replacing edge of both* $tr$ *and* $tr'$*.*

- $u \in S_6 \wedge v \in S_6$*: we have* $nca_{tr}(s,u) = nca_{tr'}(s,u)$ *and* $nca_{tr}(s,v) = nca_{tr'}(s,v)$*. This means that* $e$ *is preferred replacing edge of* $tr$ *if and only if it is preferred replacing edge of* $tr'$*.*

- $u \in S_6 \wedge v \in S_7$*: we have* $nca_{tr'}(s,u) \neq nca_{tr'}(s,v)$ *and* $nca_{tr}(s,v) = y \wedge nca_{tr}(s,u) = nca_{tr'}(s,u)$*. So if* $nca_{tr}(s,u) = y$ *then* $e$ *is preferred replacing edge of* $tr'$ *but not of* $tr$*. In this case, the edge* $e$ *will be added to prefRepl*$(tr)$ *in the update phase. If* $nca_{tr}(s,u) \neq y$ *then* $e$ *is preferred replacing edge of both* $tr$ *and* $tr'$*.*

- $u \in S_7 \wedge v \in S_7$*: we have* $nca_{tr}(s,u) = nca_{tr}(s,v)$*. That means* $e$ *is not preferred replacing edge of* $tr$*. We can also see that* $nca_{tr'}(s,x) = nca_{tr}(v_2,x)$*,* $\forall x \in S_7$*. So, if* $nca_{tr}(v_2,u) \neq nca_{tr}(v_2,v)$ *then the edge* $e$ *will be added to prefRepl*$(tr)$ *in the update phase.*

From above properties, we update *prefRepl*$(tr)$ in order to compute *prefRepl*$(tr')$ as follows:

- Remove all edges $e = (u, v)$ such that $u \in S_3$ and $v \in S_4$ (or vice versa).
- Remove all edges $e = (u, v)$ such that $u, v \in S_4$ or $u, v \in S_5$ (or vice versa).
- Remove all edges $e = (u, v)$ such that $u \in S_5 \wedge v \in S_6 \wedge nca_{tr}(s, v) = y$ (or vice versa).
- Add all edges $e = (u, v)$ such that $u \in S_6 \wedge v \in S_7 \wedge nca_{tr}(s, u) = y$ (or vice versa).
- Add all edges $e = (u, v)$ such that $u \in S_2$ and $v \in S_3$ (or vice versa).
- Add all edges $e = (u, v)$ such that $u, v \in S_2 \wedge nca_{tr}(u_2, u) \neq nca_{tr}(u_2, v) \vee u, v \in S_7 \wedge nca_{tr}(v_2, u) \neq nca_{tr}(v_2, v)$.

The implementation of `ReplacingEdgesMaintainPath(VarPath vp)` makes use of the `set{...}` data structure of COMET for storing the set of *preferred replacing* edges of `vp`. The worst case time complexity for maintaining this set is $\mathcal{O}(n^2 \log n)$ in which $\log n$ is the time complexity of each set operation. We evaluate the efficiency of the incremental implementation in comparison with recomputation from scratch in practice. The recomputation from scratch enumerates all edges $e = (u, v) \in E(g) \setminus E(tr)$ ($g$ is the given graph and $tr$ is the current RST over $(g, s, t)$) and check if the nearest common ancestors of $s$ and two node $u$ and $v$ are different. If it is the case, the edge $e$ will be inserted to the *preferred replacing* edges set.

Experiments setting: We take 10 graphs of 100 nodes and different number of edges: 200, 728, 1256, 1784, 2312, 2839, 3367, 3895, 4423, 4950 (complete graph). We define a `VarPath` for each of these graphs. For each graph, we generate randomly 20 runs of 10000 moves. Figure 3.19 show that average execution time of 20 runs of the incremental version and the recomputation version on given graphs. Figure 3.20 shows the speedup measured by $\frac{t}{t^*}$ where $t$ and $t^*$ are respectively the average of execution time of 20 runs of recomputation version and incremental version. The results show that the speedup increases when the density of graphs increases. It also shows the efficiency of incremental implementation in practice.

### 3.6.3 Maintaining distances between all pairs of two vertices on dynamic trees

`NodeDistances(VarTree vt, int[] ind)` is a graph invariant which maintains the distances w.r.t. weights indexed `ind` on edges between all pairs of two nodes on `vt`. This invariant allows to query the cost of path between any pair of two nodes in $\mathcal{O}(1)$, thus allowing to query the differentiation in $\mathcal{O}(1)$ in some cases, for instance, query the change of the cost of a path under edge replacement actions. For implementing this graph invariant, we use a direct data structure $dis(u, v)$ representing 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 partially updated: only $dis(u, v)$ such that $v$ dominates $u$ on the current tree $tr$ is considered.

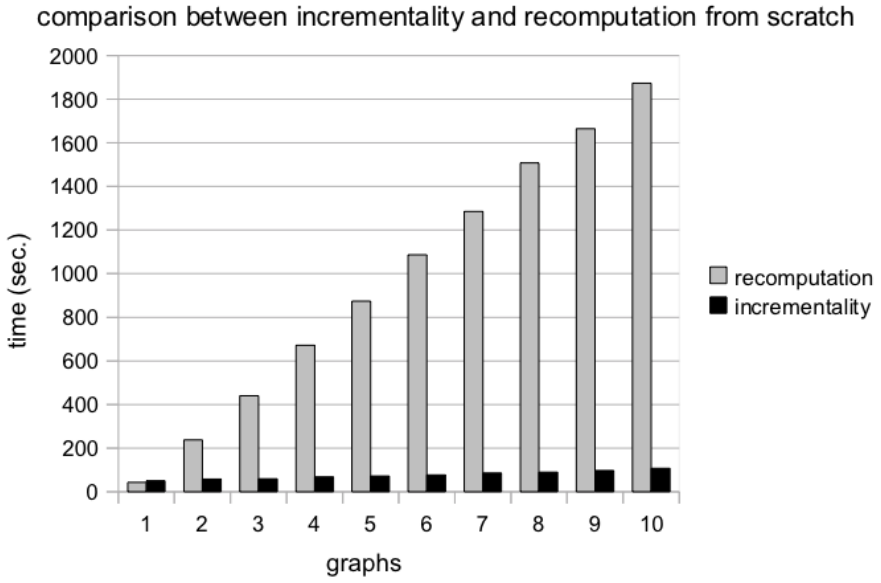



Figure 3.19: execution time of incremental version and recomputation from scratch for 10000 moves on graphs of 100 nodes and 200, 728, 1256, 1784, 2312, 2839, 3367, 3895, 4423, 4950 edges.

**Algorithm 15**: distance(x,y)

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

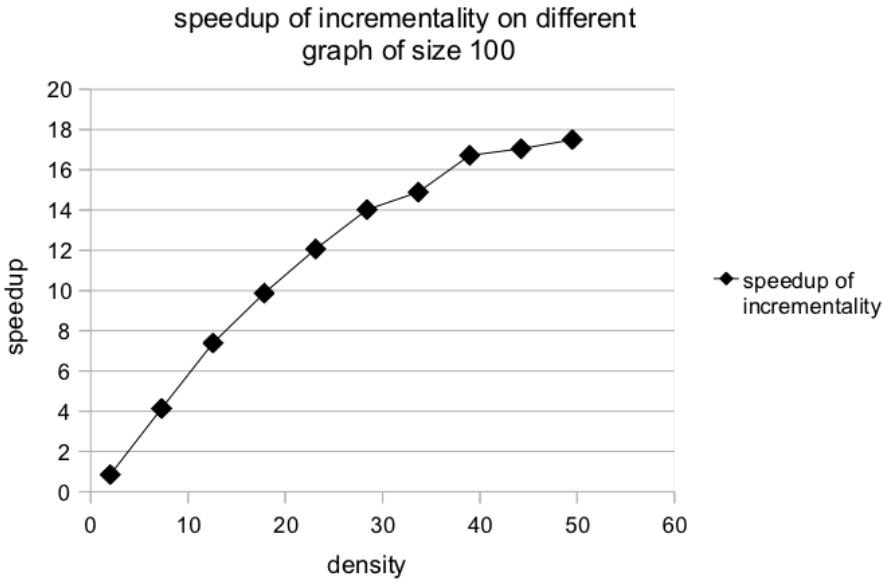



Figure 3.20: speedup of incremental version in comparison with recomputation from scratch for 10000 moves on graphs of 100 nodes and 200, 728, 1256, 1784, 2312, 2839, 3367, 3895, 4423, 4950

The cost of any two nodes $x$ and $y$ on $tr$ can be queried by Algorithm 15 in $\mathcal{O}(1)$ where line 1 can be queried in $\mathcal{O}(1)$.

We now show how to update the $dis(u, v)$ data structure under a local move on $tr$: $rep(tr, (u_1, v_1), (u_2, v_2))$. Without loss of generality, suppose that $v_1\ Dom_{tr}\ v_2$ and $u_1\ Dom_{tr}\ v_1$ (see an example in Figure 3.21). We denote $S = \{x \in V(tr) \mid v_1\ Dom_{tr}\ x\}$. 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 16 in which $c(u_2, v_2)$ is the weighted distance between $u_2$ and $v_2$ in the given graph (see line 6).

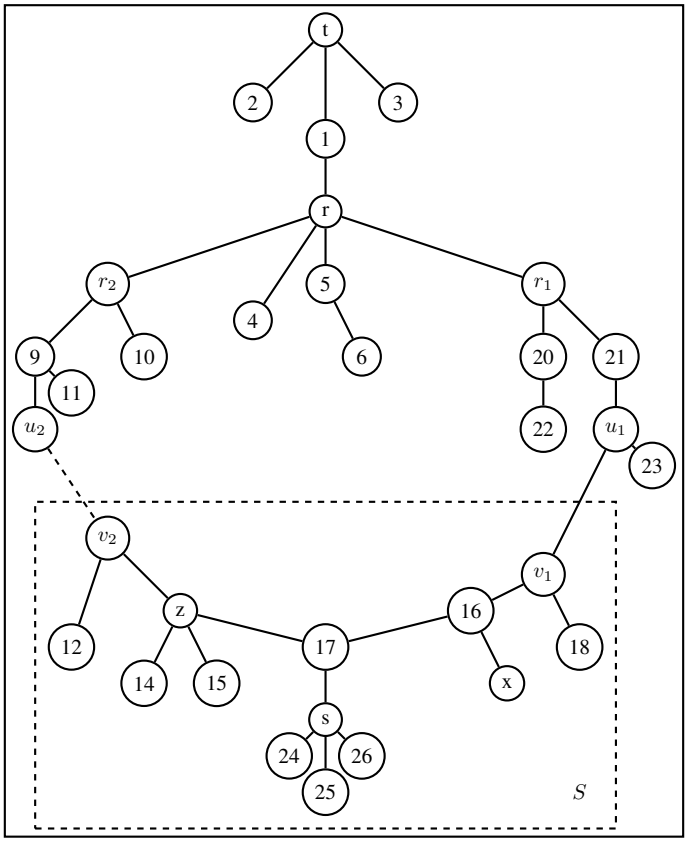
**Algorithm 16**: updateDistances

**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 efficient 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 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 to be changed at step $i$ by the incremental version. We the dynamic trees of size 98, 198, 498, 998 on complete graphs of size 100, 200, 500, 1000. For each graph, we generate randomly 20 sequences of 10000 moves. Experimental results show that the average value of $r_i$ is about $\frac{1}{10}$. Figures 3.22, 3.23 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 a replace edge action or a sequence of 2 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.




current tree $tr$

Figure 3.21: Ilustrating the update of $dis(u, v)$ under the $replaceEdge(tr, (u1, v1), (u2, v2))$ action

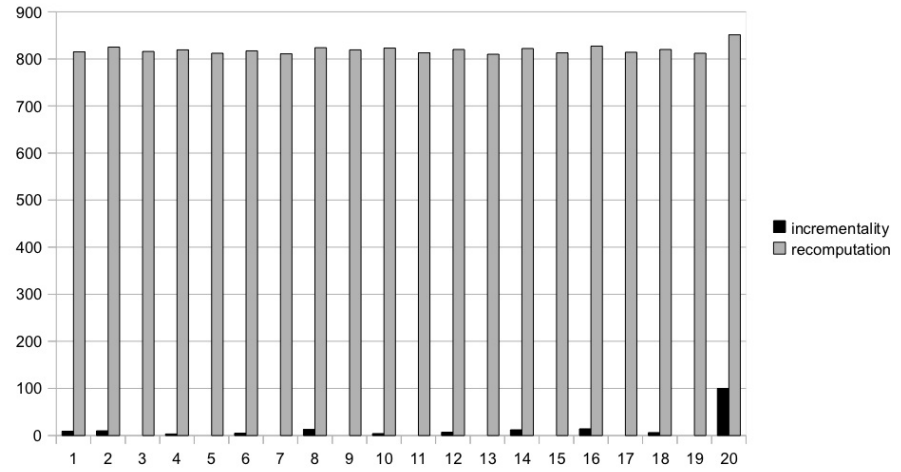



Figure 3.22: 20 last iterations for a complete graph of size 100

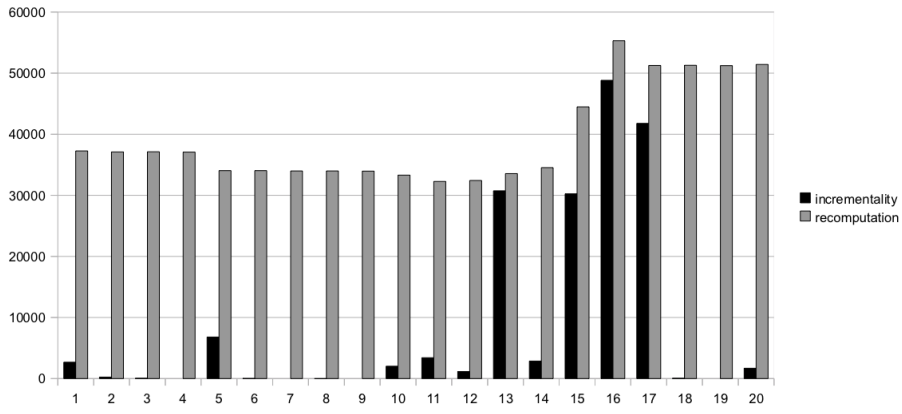



Figure 3.23: 20 last iterations for a complete graph of size 1000

# 4

# IMPLEMENTATION

This chapter describes the implementation of some essential and non-trivial classes including the `Solver<LSGraph>`, `ConstraintSystem<LSGraph>`, `PathCostOnEdges`, `IndexedPathsVisitEdges`, `AllDistinctLightPaths`. For elegance, these implementation descriptions will be depicted partially including main data structures and methods. All the consistent check statements will not be presented. The `LS(Graph)` framework is implemented on top of the COMET programming language enabling the combination of graph variables and `var{int}` and `var{float}` of COMET. The current version of `LS(Graph)` is about 25,000 lines of COMET code.

## 4.1 Class `Solver<LSGraph>`

The role of the `Solver<LSGraph>` is to manage all graph variables, graph invariants, to relate these objects and do propagation in order to update graph invariants under change of variables. The `Solver<LSGraph>` extends the `Solver<LS>` of COMET enabling the combination between graph variables and `var{int}, var{float}` of COMET.

The class is listed partially as follows:

```
1  class Solver<LSGraph> extends Solver<LS>{
2    VarGraph[]   _vg;
3    Invariant<LSGraph>[] _gi;
4    bool[,]    _depends;
5    bool     _closed;

7    void post(Invariant<LSGraph> gi){
8      super.post(gi);

10     addGraphInvariant(gi);

12     VarGraph[] vgs = gi.getVarGraphs();
13     if(vgs != null){
```

```
14          forall(i in vgs.rng()){
15             int j = getId(vgs[i]);
16             int k = getId(gi);
17             _depends[k,j] = true;
18          }
19       }
20    }

22    void close(){
23       initPropagation();
24       super.close();
25       _closed = true;
26    }

28    void initPropagation(){
29       forall(i in _gi.rng())
30          _gi[i].initPropagation();
31    }

33    void replaceEdge(VarTree vt, Edge eo, Edge ei){
34       if(_closed){
35          int id = getId(vt);
36          forall(i in _gi.rng(): _depends[i,id])
37             _gi[i].propagateReplaceEdge(vt, eo,ei);
38       }
39       vt.replaceEdge(eo,ei);
40    }
41 }
```

The data structure is simple. The array `_vg` (line 2) maintains the list of graph variables posted and the array `_gi` (line 3) stores the list of graph invariants posted in the model. Each graph variable, graph invariant has an identification value which is its index in the corresponding array. The 2-dimension array `_depends` (line 4) maintains the dependence of graph invariants on graph variables: `_depends[i,j]=true` means that the graph invariant indexed `i` depends on the graph variable indexed `j`. The `_closed` variable (line 5) specifies whether or not the solver is closed.

The `post(Invariant<LSGraph> gi)` method (lines 7-20) post the graph invariant `gi` which consists of adding it to the `_gi` array, setting its identification (method `addGraphInvariant` in line 10) and creating the dependence relation between `gi` and graph variables posted (lines 12-19): line 12 returns the list of graph variables over which `gi` is defined, methods `getId(...)` (lines 15-16) return the identification of graph variable `vgs[i]` and `gi`.

The method `close` (lines 22-26) closes the solver which mainly initializes all graph invariants (the `initPropagation` method in line 23 which is detailed in lines 28-31).

The `initPropagation` method iteratively performs the `initPropagation` method of all graph invariants posted to the solver (lines 29-31).

One of the main role of the `Solver<LSGraph>` is to do a propagation in order to update invariants under moves (the change of variables). There are several moves defined in Sections 3.2, 3.3, 3.4, each move corresponds to a propagation method specified in the `Invariant<LSGraph>` interface of the framework. We describe here one of its move method: the `replaceEdge(VarTree vt, Edge eo, Edge ei)` method.

Other move methods are essentially similar. The `replaceEdge(VarTree vt, Edge eo, Edge ei)` method (lines 33-40) updates graph invariants when the edge `eo` is replaced by the edge `ei` on the `VarTree vt`. More precisely, lines 36 scans all graph invariants `_gi[i]` which depends on `vt` (line 35 gets the identification of `vt`). For each of these `_gi[i]`, we call its corresponding propagation method (line 37). Finally, line 39 replaces the edge `eo` by `ei`.

## 4.2 Class ConstraintSystem<LSGraph>

The class `ConstraintSystem<LSGraph>` combines all constraints stated in the model and can be used to control the search procedure.

The class is depicted partially as follows:

```
1  class ConstraintSystem<LSGraph> extends GraphConstraint
2       implements Constraint<LSGraph>{

4     Constraint<LSGraph>[]  _gc;
5     float[]      _w;
6     ConstraintSystem<LS>  _CS;
7     bool       _initConstraintSystem;
8     bool       _closed;

10    void post(Constraint<LS> c){
11       if(!_initConstraintSystem){
12          _CS = new ConstraintSystem<LS>(_ls);
13          _initConstraintSystem = true;
14       }

16       _CS.post(c);
17    }

19    void post(Constraint<LSGraph> gc, int w){
20       addGraphConstraint(gc,w);
21    }

23    void close(){
24       range R = _gc.rng();

26       var{float} f[i in R] = _gc[i].violations();

28       if(_initConstraintSystem){
29          var{int} vcs = _CS.violations();
30          _violations = new var{float}(_ls) <- sum(i in R)(f[i] * _w[i])
                 + vcs;
31       }else{
32          _violations = new var{float}(_ls) <- sum(i in R)(f[i] * _w[i]);
33       }

35       _closed = true;

37    }

39    float getReplaceEdgeDelta(VarTree vt, Edge eo, Edge ei){
```

```
40        return sum(i in _gc.rng())(_w[i] *
              _gc[i].getReplaceEdgeDelta(vt,eo,ei));
41      }

43      float getAssignDelta(var{int} x, int v){
44        float delta = sum(i in _gc.rng())(_w[i] *
              _gc[i].getAssignDelta(x,v));

46        if(hasConstraintSystem())
47          delta += _CS.getAssignDelta(x,v);

49        return delta;
50      }

52      var{float} violations(){
53        return _violations;
54      }
55    }
```

The data structure consists of an array `_gc` (line 4) storing all graph constraints posted and its weights `_w` (line 5). The data structure also maintains a `ConstraintSystem<LS> _CS` (line 6) allowing users to post standard `Constraint<LS>` of COMET. The variable `_closed` (line 8) specifies wether or not the `ConstraintSystem<LSGraph>` is closed.

The method `post(Constraint<LS> c)` (lines 10-17) posts a standard `Constraint<LS> c` of COMET. This constraint is posted to `_CS` (line 16) which is initialized in lines 11-14.

The method `post(Constraint<LSGraph> gc, int w)` (lines 19-21) posts a graph constraint `gc` and its weight `w`. It is done by simply adding `gc` and `w` to the corresponding array `_gc` and `_w` (see method `addGraphConstraint` in line 20).

Whenever the `ConstraintSystem<LSGraph>` is closed (method `close` in lines 23-37), the number of violations `ConstraintSystem<LSGraph>` is set by the weighted sum of number of violations of all graph constraints (see invariant in line 32 where `f[i]` represents the number of violations of the graph constraint `_gc[i]` which is set in line 26) plus the number of violations of the `ConstraintSystem<LS> _CS` if some `Constraint<LS>` have been posted (see invariant in line 30).

One of the main features of `ConstraintSystem<LSGraph>` is the differentiation which provides a number of methods allowing to query the variation of the number of violations of the constraints under various local moves. The implementation of these methods are essentially similar. We demonstrate two of them: `getReplaceEdgeDelta(VarTree vt, Edge eo, Edge ei)` (lines 39-41) and `getAssignDelta(var{int} x, int v)` (lines 43-50).

- The `getReplaceEdgeDelta(VarTree vt, Edge eo, Edge ei)` method computes the variation of the number of violations of the `ConstraintSystem<LSGraph>` when the edge `eo` is replaced by the edge `ei` on the tree `vt`. This value is simply the weighted sum of the variation of the number of violations of all the graph constraints `_gc[i]` when this modification action is taken (line 40). In this case, all the `Constraint<LS>` posted are not taken into account because the modifica-

tion action does not affect these constraints.

- The getAssignDelta(var{int} x, int v) computes the variation of the number of violations of the ConstraintSystem<LSGraph> when the variable x is reassigned by the value v. In this case, all the Constraint<LS> posted must be taken into account because the modification action affects these constraints. The variation value delta (which is initially set in line 44) is thus added with the variation value of _cs (see lines 46-47).

## 4.3 Class PathCostOnEdges

The PathCostOnEdges(VarPath vp, int indWeight) is a Function<LSGraph> representing the cost of the path vp with respect to the weights indexed indWeight.

The implementation is listed partially as follows:

```
1  class PathCostOnEdges extends GraphFunction implements
       Function<LSGraph>{
2    NodeDistancesInvr _dis;
3    int       _indWeight;
4    VarPath     _vp;

6    PathCostOnEdges(VarPath vp, int
       indWeight):GraphFunction(vp.getLSGraphSolver(),vp){
7      _indWeight = indWeight;
8      _vp = vp;

10     VarRootedSpanningTree tr = _vp.getVarRootedSpanningTree();
11     _dis = new NodeDistancesInvr(tr,indWeight);

13     _value = new var{float}(_ls);

15     post();
16   }

18   void post(){
19     _ls.post((Invariant<LSGraph>)this);
20   }

22   var{float} getValue(){
23     return _value;
24   }

26   void initPropagation(){
27     _value :=
         _dis.getDistance(_vp.getSource(),_vp.getDestination(),
         _indWeight);
28   }


31   float getDeltaWhenUseReplacingEdge(VarPath vp, Edge ei){
32     if(_vp != vp)
33       return 0;
```

```
35        Vertex u = up(ei,vp);
36        Vertex v = low(ei,vp);
37        Vertex su = upnca(ei,vp);
38        Vertex sv = lownca(ei,vp);

40        float du = _dis.getDistance(u,vp.getDestination(),_indWeight);
41        float dv = _dis.getDistance(vp.getSource(),v,_indWeight);
42        newd = du + dv + ei.weight(_indWeight);

44        return newd - _value;
45      }


48      bool propagateReplaceEdge(VarPath vp, Edge eo, Edge ei){
49        _value := _dis.getDistance(vp.getSource(),vp.getDestination(),
              _indWeight);
50        return true;
51      }
52    }
```

The data structure mainly consists of a graph invariant `_dis` (line 2) which maintains the weighted distances between all pairs of two vertices on a dynamic RST nested under the consider path variable `_vp` (line 4). Variable `_indWeight` (line 3) stores the considered weight index.

In the constructor (lines 6-16), the considered `VarPath` and weight index are retained (lines 7-8). Line 10 gets the RST nested under the considered `VarPath` and line 11 initializes the graph invariant `_dis`.

The `initPropagation` (lines 26-28) simply initializes the variable `_value` which retains the value of the `Function<LSGraph>` (the cost of the `VarPath` in this case). This statement is consistent because the graph invariant `_dis` is posted to the `Solver<LSGraph>` before the current instance of the `PathCostOnEdges` class. The `initPropagation` of the graph invariant `_dis` is thus launched and completed before the call to the `initPropagation` method of the `PathCostOnEdges` class. Notice that the complexity of the method in line 27 is $\mathcal{O}(1)$.

We describe now one of the differentiation methods: the `getDeltaWhenUseReplacingEdge(VarPath vp, Edge ei)` method which computes the variation of the cost when the *replacing* edge `ei` is applied[1]. Lines 35-38 get the endpoints `u`, `v` of the edge `ei` and the nearest common ancestors of the source of `vp` and these endpoints such that `su` dominates `sv` on the RST nested under `vp` (recall the notations in Section 3.3). The cost of the new path is computed in lines 40-42 with time complexity $\mathcal{O}(1)$ and the variation of the cost is returned in line 42. Figure 4.1 illustrates the query. The *replacing* edge is $ei = (30, 31)$, the values of `u, v, su, sv` (lines 35-38) are respectively 31, 30, 7, 17. The value `du` (line 40) is the cost of path from the vertex 31 to the vertex $t$ and the value `dv` (line 41) is the cost of the path from the vertex $s$ to the vertex 30 on the tree in Figure 4.1.

The `propagateReplaceEdge(VarPath vp, Edge eo, Edge ei)` method updates the cost `_value` of the path `_vp` when the edge `eo` is replaced by the edge `ei`. This is

[1] In the edge replacement for `VarPath`, the new path depends only on the *replacing* edge, not on the *replacable* edge.

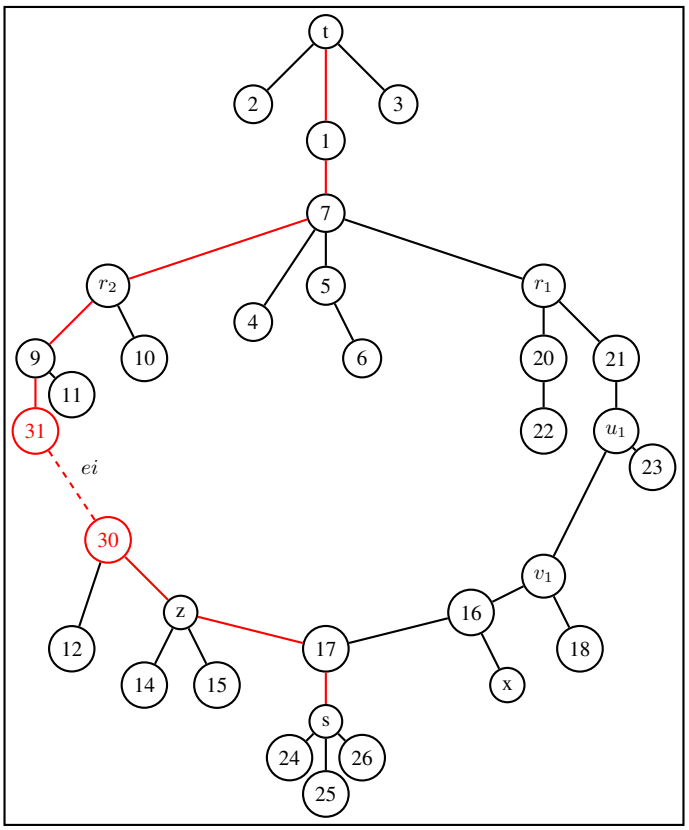



Figure 4.1: Ilustrating the variation of the cost of the path from $s$ to $t$ when the *replacing* edge $ei$ is applied

done by simply taking the distance value from the graph invariant `_dis` and assigning it to `_value` (line 49) because the update of `_dis` is completed before the call to this method.

## 4.4 Class `IndexedPathVisitEdges`

We describe now the implementation of a graph invariant combining graph variables and `var{int}` of COMETwhich will then be used for the implementation of a `Constraint<LSGraph>` appearing in the routing and wavelength assignment problem. The graph invariant `IndexedPathVisitEdges(VarPath[] vps, varint[] xw, varint[,] v)` admits an array `vps` of `VarPath` and an array `xw` of `var{int}` as sources (each path `vps[i]` has a value stored by `xw[i]`) and maintains as target the 2-dimensions array `v`: `v[i,j]` is the number of times the paths having value `i` visit the edge of identification `j`.

The implementation is partially listed as follows:

```
1  class IndexedPathVisitEdges extends GraphInvariant implements
       Invariant<LSGraph>{
2    var{int}[,]   _v;
3    VarPath[]     _vps;
4    var{int}[]    _xw;
5    range         _Paths;
6    dict{var{int}->int} _map;
7    dict{VarPath->int} _mapVP;

9    int[]         _oldxw;

11   IndexedPathVisitEdges(VarPath[] vps, var{int}[] xw, var{int}[,]
       v):GraphInvariant(vps[vps.rng().getLow()].getLSGraphSolver(),
       all(i in vps.rng())(VarGraph)vps[i]){

13     _vps = vps;
14     _xw = xw;
15     _v = v;
16     _Paths = _vps.rng();

18     _map = new dict{var{int}->int}();
19     _mapVP = new dict{VarPath->int}();

21     forall(i in _Paths){
22       _map{_xw[i]} = i;
23       _mapVP{_vps[i]} = i;
24     }

26     _oldxw = new int[i in _Paths] = _xw[i];
27   }

29   void post(InvariantPlanner<LS> ip){
30     forall(i in _Paths)
31       ip.addSource(_xw[i]);
32     forall(i in _v.rng(0), j in _v.rng(1))
33       ip.addTarget(_v[i,j]);
34   }

36   void initPropagation(){
37     forall(i in _v.rng(0), j in _v.rng(1))
38       _v[i,j] := 0;

40     forall(i in _Paths){
41       int w = _xw[i];
42       VarPath vp = _vps[i];
43       forall(e in vp.getEdges()){
44         _v[w,e.id()]++;
45       }

47       _oldxw[i] = _xw[i];
48     }
49   }

51   void propagateInt(boolean b, var{int} x){
52     int ind = _map{x};
53     int v = x;
```

```
54        int ov = _oldxw[ind];
55        if(v == ov){
56           return;
57        }
58        VarPath vp = _vps[ind];

60        forall(e in vp.getEdges()){
61           _v[ov,e.id()]--;
62        }

64        forall(e in vp.getEdges()){
65           _v[v,e.id()]++;
66        }

68        _oldxw[ind] = v;
69     }


72     bool propagateReplaceEdge(VarPath vp, Edge eo, Edge ei){
73        int ind = _mapVP{vp};
74        int w = _xw[ind];

76        VarRootedSpanningTree tr = vp.getVarRootedSpanningTree();
77        Vertex u = up(ei,vp);
78        Vertex v = low(ei,vp);
79        Vertex su = upnca(ei,vp);
80        Vertex sv = lownca(ei,vp);

82        Vertex x = sv;
83        while(x != su){
84           Edge e = tr.getFatherEdge(x);
85           _v[w,e.id()]--;
86           x = tr.getFatherVertex(x);
87        }

89        x = u;
90        while(x != su){
91           Edge e = tr.getFatherEdge(x);
92           _v[w,e.id()]++;
93           x = tr.getFatherVertex(x);
94        }
95        x = v;
96        while(x != sv){
97           Edge e = tr.getFatherEdge(x);
98           _v[w,e.id()]++;
99           x = tr.getFatherVertex(x);
100       }

102       _v[w,ei.id()]++;

104       return true;
105    }

107    dict{var{int}->int} getMap(){ return _map;}
108    dict{VarPath->int} getMapVarPath(){ return _mapVP;}
109 }
```

The data structure (lines 2-9) contains _vps, _xw, _v which retain the source and target arguments of the graph invariant, _oldxw which retains the value of _xw[2]. The data structure is initialized in the constructor (lines 11-27).

The method post(InvariantPlanner<LS> ip) specifies the sources _xw (lines 30-31) and the targets _v (lines 32-33) of the graph invariant. Other sources _vps are implicitly specified in the constructor.

The initPropagation method (lines 36-49) computes initial value of _v. The method propagateInt(boolean b, var{int} x) update _v after the variable x is updated. Lines 52 and 58 retain the VarPath vp corresponding with the variable x to be changed. Obviously, for each edge e on vp, the value _v[ov,e.id()] is reduced and the value _v[v,e.id()] is increased where v is the current value of x and ov is its previous value.

The method propagateInt(boolean b, var{int} x) (lines 51-69) performs the update over _v under the change of x. Line 52 gets the index of x in the array _xw. Lines 53 and 54 get the current v and the previous values ov of x. Line 58 gets the corresponding VarPath vp of x. When the value of vp changes from ov to v, it means clearly that the number of paths of value ov reduces by 1 and the number of paths of value v increases by 1. The value of _v[ov,e.id()] is thus decreased by 1 for all edges e of vp (lines 60-62). The value of _v[v,e.id()] is increased by 1 for all edges e of vp (lines 64-66).

The method propagateReplaceEdge(VarPath vp, Edge eo, Edge ei) (lines 72-104) performs the update over _v when a move is taken: the edge eo is replaced by the edge ei on vp. Line 73 gets the index of vp in the retaining array _vps and line 74 gets its value. Line 76 gets the rooted spanning tree tr nested under vp. Lines 77-80 get endpoints u, v of ei and the nearest common ancestors su, sv of u, v and the source of vp such that su dominates sv on tr (recall the notation in Section 3.3). The value of _v[w,e.id()] reduces for all edges e that belong to the path from sv to su on the current path[3] (see lines 82-87 where the method tr.getFatherEdge(x) returns the edge (x,y) of tr such that y is the father of x and the method tr.getFatherVertex(x) returns the father vertex of x on tr). The value of _v[w,e.id()] increases for all edges e that belong to the path from u to su and the path from v to sv[4] (lines 89-100). The value of _v[w,e.id()] is also increased (line 102) because this edge will be added to the current path under the move.

## 4.5 Class `AllDistinctLightPaths`

AllDistinctLightPaths(VarPath[] vps, var{int}[] xw) is a Constraint<LSGraph> defined over an array vps of VarPath and an array xw of var{int} which states that if vps[i] and vps[j] shares a link then the value of xw[i] and xw[j] must be different.

[2] At the moment that the propagateInt(boolean b, var{int} x) (line 51) is launched, the variable of x stores the new value, maintaining its previous value is thus necessary for performing the update.

[3] This path will be removed from the current path under the move.

[4] These paths will be added to the current path under the move.

The implementation is partially listed as follows:

```
1  class AllDistinctLightPaths extends GraphConstraint implements
       Constraint<LSGraph>{
2    VarPath[]     _vps;
3    var{int}[]    _xw;
4    var{int}[,]       _v;
5    range        _Paths;
6    range        _Waves;
7    range        _Edges;

9    dict{var{int}->int}  _map;
10   dict{VarPath->int}  _mapVP;


13   AllDistinctLightPaths(VarPath[] vps, var{int}[]
       xw):GraphConstraint(vps[vps.rng().getLow()].getLSGraphSolver()){
14     _vps = vps;
15     _xw = xw;
16     _Paths = _vps.rng();

18     int minW = min(i in xw.rng())(min(v in xw[i].getDomain())(v));
19     int maxW = max(i in xw.rng())(max(v in xw[i].getDomain())(v));
20     _Waves = minW..maxW;

22     UndirectedGraph g = _vps[_Paths.getLow()].getLUB();

24     _Edges = g.getEdgesRange();
25     _v = new var{int}[_Waves, _Edges](_ls);

27     IndexedPathVisitEdges ipve(_vps,_xw,_v);

29     _map = ipve.getMap();
30     _mapVP = ipve.getMapVarPath();

32     _violations = new var{float}(_ls) <- sum(i in _Waves, j in
         _Edges)(max(0,_v[i,j]-1));

34     _ls.post((Invariant<LSGraph>)this);
35   }


38   float getAssignDelta(var{int} x, int v){
39     if(x == v)
40       return 0;

42     int ind = _map{x};
43     int oldw = _xw[ind];
44     VarPath vp = _vps[ind];

46     int delta = 0;
47     forall(e in vp.getEdges()){
48       if(_v[oldw,e.id()] > 1)
49         delta--;
50     }

52     forall(e in vp.getEdges()){
```

```
53          if(_v[v,e.id()] >=1)
54            delta++;
55        }

57        return delta;
58      }

60      float getDeltaWhenUseReplacingEdge(VarPath vp, Edge e){
61        int ind = _mapVP{vp};
62        float delta = 0;
63        int w = _xw[ind];

65        VarRootedSpanningTree tr = vp.getVarRootedSpanningTree();
66        Vertex u = up(e,vp);
67        Vertex v = low(e,vp);
68        Vertex su = upnca(e,vp);
69        Vertex sv = lownca(e,vp);

71        // process edges to be removed
72        Vertex x = sv;
73        while(x != su){
74          Edge ei = tr.getFatherEdge(x);
75          if(_v[w,ei.id()] > 1)
76            delta -= 1;
77          x = tr.getFatherVertex(x);
78        }

80        //process edges to be added from v to sv on the current tree
81        x = v;
82        while(x != sv){
83          Edge ei = tr.getFatherEdge(x);
84          if(_v[w,ei.id()] >= 1)
85            delta += 1;
86          x = tr.getFatherVertex(x);
87        }

89        //process edges to be added from u to su on the current tree
90        x = u;
91        while(x != su){
92          Edge ei = tr.getFatherEdge(x);
93          if(_v[w,ei.id()] >= 1)
94            delta += 1;
95          x = tr.getFatherVertex(x);
96        }

98        //process the edge e which is also the edge to be added
99        if(_v[w,e.id()] >= 1)
100         delta += 1;

102       return delta;
103     }
104   }
```

The data structure consists of `_vps` (line 2) and `_xw` (line 3) which are used to refer the variables of the constraint. We maintain a 2-dimension array `_v` (line 4): `_v[i,j]` is the number of times the paths having value `i` visit the edge of identification `j` by

making use of the graph invariant `IndexedPathVisitEdges` described above (see the initialization of this invariant in line 27).

In the constructor (lines 13-35), the number of violations of the constraint is stated as invariant defined over `_v` (line 32) where `_Waves` is the range of values of all `_xw[i]` which is computed in lines 18-20 and `_Edges` is the range of identifications of all edges of the graph over which all `VarPath _vps[i]` are specified (line 22).

The `getAssignDelta(var{int} x, int v)` (lines 38-58) computes the variation of the number of violations of the constraint when the variable `x` is assigned by the value `v`. Line 42 gets the index of `x` from the retaining array `_xw` and line 44 gets the corresponding `VarPath vp`. The variation is computed in lines 46-55. Obviously, the number of violations of the constraint over all edges of `vp` corresponding with its current value `oldw` (line 43) reduces (see lines 47-50) and the number of violations of the constraint over all edges of `vp` corresponding with its new value `v` increases (see lines 52-55).

The `getDeltaWhenUseReplacingEdge(VarPath vp, Edge e)` method (lines 60-103) computes the variation of the number of violations of the constraint when the *replacing* edge `e` is applied in the edge replacement over `vp`. Line 61 gets the index of `vp` in the retaining array `_vps` and line 63 gets the value of `vp`. Line 65 gets the rooted spanning tree `tr` nested under `vp`. Lines 66-69 get endpoints `u`, `v` of `ei` and the nearest common ancestors `su`, `sv` of `u`, `v` and the source of `vp` such that `su` dominates `sv` on `tr`. The number of violations of the constraint reduces along the subpath of `vp` to be removed under the move (lines 72-78) and the number of violations of the constraint increases along subpaths to be added under the move (lines 81-100).

# 5

# APPLICATIONS

This chapter presents the applications of the `LS(Graph)` framework to the resolution of some COT/COP problems including the edge-weighted $k$-cardinality tree (KCT) problem, the quorumcast routing (QR) problem, the resource constrained shortest path (RCSP) problem, the edge-disjoint paths (EDP) problem, the routing and wavelength assignment (RWA) problem, the routing for network covering (RNC) problem. All these problems have been specified in Section 2 together with related works.

`LS(Graph)` has also been successfully applied on a Traffic Engineering in Switched Ethernet Networks problem which consists of finding a spanning tree on a given network minimizing the traffic congestion which is reported in [HFD+10]. This application has been specified and solved by HO in his doctoral research and we do not describe this application in this thesis.

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

## 5.1 Specifying tabu search parameters

Most of the applications in this chapter apply the tabu search algorithm. There are many parameters for the tabu search algorithm. Trying all possible values of the parameters for selecting the best ones is not realizable. In this thesis, we sample some values of the parameters, analyze their influence on the solutions, and select the most promising ones for the QR and the RWA-D problems.

In order to compare two results produced by two algorithms, we apply the technique based on the concept of confidence interval for comparing two alternatives of [Jai91] (the paired observations case described in the page 209) which is described as follows. We denote $\mathcal{A}(P_1)$ and $\mathcal{A}(P_2)$ respectively the tabu search algorithm $\mathcal{A}$ instantiated by the two parameters $P_1$ and $P_2$ each of which consists of a set of elements (e.g., $\{tbMin, tbMax, tinc, maxStable\}$, see Appendix B for more description detail about the tabu search schema and its parameters). Given a set of problem instances

$\{1, 2, ..., n\}$, we denote $R_1 = \{x_1^1, x_2^1, ..., x_n^1\}$ and $R_2 = \{x_1^2, x_2^2, ..., x_n^2\}$ the results produced by $\mathcal{A}(P_1)$ and $\mathcal{A}(P_2)$ in which $x_j^i$ is the average objective value found[1] of the problem instance $j, \forall i = 1..2, j = 1..n$. We denote $z_i = x_i^1 - x_i^2, \forall i = 1, 2, ..., n$.

- The sample mean
$$\overline{z} = \frac{1}{n}\sum_{i=1}^{n} z_i$$
- The sample standard deviation
$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(z_i - \overline{z})^2}$$
- The 90% confidence interval for mean $[l, u]$ = $[\overline{z} - t\frac{s}{\sqrt{n}}, \overline{z} + t\frac{s}{\sqrt{n}}]$, with $t$ is the 0.95-quantile of a $t$-variate with $n-1$ degrees of freedom (see Table A.4 of Appendix of [Jai91] for the value of $t$ which depends on $n$).

Without loss of generality, we consider the minimization problem. If $[l, u]$ includes zero, then the two results are not different. If $u < 0$, then the result $R_1$ is better than $R_2$ (and vice versa).

## 5.2 The edge-weighted $k$-cardinality tree (KCT) problem

### 5.2.1 Problem formulation

Given an undirected weighted graph $G = (V, E)$ and an integral value $k$, the KCT problem consists of finding a connected and acyclic subgraph (i.e., a tree) of $G$ having exactly $k$ edges such that the sum of weights of edges is minimal.

### 5.2.2 The model

We implement a tabu search (denoted by KCT_MTABU) applying the same local search schema as [BB05] but also exploiting the $NT_3$ neighborhood. The model is given in Figure 5.1 in which line 1 declares a `Solver<LSGraph> ls`. Line 2 creates a `VarTree tr` associated with `ls` of the given graph `g` which is initialized randomly with `k` edges. The size of the tree `tr` is maintained by the initialization and by the considered neighborhoods). A `Model<LSGraph> mod` (line 4) is an object which encapsulates all variables, constraints, and objective function to be optimized. In this model, we have only one variable `tr` and an objective function to be minimized `weight`. The constraint specifying that the number of edges of `tr` must be equal to $k$ is always satisfied

[1] For each problem instance, each algorithm executes 20 times and gives 20 objective values which are normally different.

```
1  Solver<LSGraph> ls();
2  VarTree tr(ls,g,k);
3  Weight<Tree> weight(tr);
4  Model<LSGraph> mod(tr,weight,NonSpanningTree,MINIMIZATION);

6  KCTSearch se(mod);
7  se.setCard(k);
8  se.setMaxIter(100000);
9  se.setMaxTime(1800);

11 se.search();
```

Figure 5.1: Model in `LS(Graph)` for the KCT problem

```
1  class KCTSearch extends TabuSearch<LSGraph>{
2    int _card;

4    KCTSearch(Model<LSGraph> mod):TabuSearch<LSGraph>(mod){
5    }
6    void setCard(int ca){
7      _card = ca;
8    }
9    void restartSolution(){
10     initSolution();
11   }
12   void initSolution(){
13     VarTree tr = getFirstVarTree();
14     InsertableEdgesVarTree inst = getInsertableEdges(tr);

16     tr.clear();
17     select(e in inst.getSet()){
18       tr.addEdge(e);
19     }

21     forall(i in 1.._card-1)
22       selectMin(e in inst.getSet())(e.weight()){
23         tr.addEdge(e);
24       }
25   }
26   void exploreNeighborhood(Neighborhood N){
27     exploreTabuMinAddRemove1VarTree(N,true);
28     exploreTabuMinReplace1VarTree(N,true);
29   }
30 }
```

Figure 5.2: Search component for the KCT problem

in the initial solution and during the search procedure. Line 6 creates a search component `KCTSearch` (depicted in Figure 5.2). Lines 7-9 set its parameters including the cardinality `k`, the maximum number of iterations and the time limit. Line 11 performs the search procedure. The search component is given in Figure 5.2 that extends the generic tabu search `TabuSearch<LSGraph>`[2] of the framework by re-implementing the `exploreNeighborhood` method (lines 26-29): we consider only $NT_{1+2}$ and the $NT_3$ neighborhood which conserves the number of edges of the tree during the search. The `initSolution` is overridden in lines 12-25 which constructs the tree in a greedy random way. Line 13 gets the only variable `tr` of the model and line 14 gets the graph invariant representing the set of *insertable* edges of `tr`. Line 16 clears the tree. Lines 17-25 is the tree construction by selecting randomly the first edge (lines 17-19) and iteratively selecting an edge of minimal weight for adding to the tree until the number of edges of the constructed tree `tr` is equal to `_card` (lines 21-25). Notice that if we remove line 28 in Figure 5.2, we obtain the tabu search algorithm of [BB05][3]. This shows the compositionality of our framework.

### 5.2.3 Experiments

The model has been experimented in two benchmarks. The first benchmark consists of 35 4-regular graphs of different size (from 25 to 1000) and the value of cardinality $k$ is set by 20. The second benchmark is a diverse set of problem instances including grid graphs, graphs from the steiner tree benchmark and one graph from the graphs coloring benchmark. These two benchmarks are available on the KCT library [BB].

The KCT_MTABU is executed 20 times for each instance with 30 minutes of time limit. The tabu search parameters are set as follows: `tbMin` = 5, `tbMax` = 33, `tinc` = 8, `maxStable` = 200 (see Appendix B for more description detail about the tabu search schema and its parameters).

The first benchmark (denoted by blxh instances) is easy (from literature study) and does not allow to differentiate algorithms. Experimental results of our KCT_MTABU are shown in Table 5.1. The Table shows that the KCT_MTABU finds optimal solutions (presented in column 2 of Table 5.1), but is slightly slower than dedicated C++ implementation of the tabu search algorithm (TS) and two other meta-heuristic algorithms (genetic and ACO) of [BB05]. The last three columns present the speedup of the three meta-heuristic methods of [BB05] measured by $\frac{\overline{t_0}}{\overline{t}}$ where $\overline{t_0}$ is the average time of finding best solutions of the meta-heuristic of [BB05] runned on AMD Athlon 1100 MHz CPU and $\overline{t}$ is the average time of finding best solution of our KCT_MTABU runned on XEN virtual machines with 1 core of a CPU Intel Core2 Quad Q6600 @2.40GHz and 1GB of RAM. The reason why our KCT_MTABU finds optimal solutions slower than TS is that the KCT_MTABU explores the $NT_3$ neighborhood which is too large. Moreover, for the neighborhood $NT_{1+2}$, the dedicated algorithm (implemented in C++) maintained two ordered list of *insertable* and *remov-*

[2]Note that the tabu search schema of `TabuSearch<LSGraph>` is the same as that of [BB05].

[3]However, the neighborhood exploration is different: the algorithm of [BB05] maintained two sorted list of *insertable* edges and *removable* edges and scan them in a systematical way while our tabu search scan them in a generic way.

```
1  Solver<LSGraph> ls();
2  VarTree tr(ls,g,k);
3  Weight<Tree> weight(tr);
4  Model<LSGraph>
       mod(tr,DegreeAtMost(tr,D),weight,NonSpanningTree,MINIMIZATION);

6  KCTSearch se(mod);
7  se.setCard(k);
8  se.setMaxIter(100000);
9  se.setMaxTime(1800);

11 se.search();
```

Figure 5.3: Model in `LS(Graph)` for the KCT with the degree side constraint problem

*able* edges and explores it systematically. In our model, these two list are stored by general `set{...}` data structure of COMET and are explored in a generic way.

The second benchmark is much more difficult and consists of 230 instances. The start-of-the-art algorithm for the KCT problem is the exact ILP-based algorithm using directed cuts [CKIL09]. The optimal solutions are reported on [CKIL]. Our KCT_MTABU model finds optimal solutions in 80 out of 230 instances (generally, it finds optimal solution for all graphs with cardinality $k = 2$ and $n - 2$) with average gap for all instances[4] is about 0.045 while the tabu search of [BB05] finds optimal solutions in 72 out of 230 instances. Especially, the KCT_MTABU implemented in high-level programming language COMET reaches optimal solutions (with cardinality $k = 2$ and $n - 2$ where $n$ is the number of vertices of the input graphs) very fast as shown in Table 5.2. Notice however that the ILP-based algorithm is an exact method that thus proves the optimality while our KCT_MTABU is not able to prove optimality.

### 5.2.4 The KCT problem with the degree side constraint

In order to show the compositionality and facility of the modeling, we consider the KCT problem with the degree side constraint (denoted by KCT-D) which specifies that the degree of each vertex of the tree cannot exceed a given value $D$. For solving this problem, we simply add the degree side constraint into the model as shown in Figure 5.3 (see line 4). Note that we do not have to change the search component: it is depicted in Figure 5.2.

Experimental results for the KCT-D problem with blxh instances is presented in Table 5.3 (the maximum value $D$ imposed on the degree of vertices of the tree is set to 3). Columns 2, 3, 4, 5 respectively present the average, the minimal, the maximal and the standard deviation of the objective values found of 20 executions. Column 6 presents the optimal objective value for the KCT problem (without the degree side

[4]The average gap for each instance $i$ is measured by $\frac{\overline{f}_i - opt_i}{opt_i}$ where $\overline{f}_i$ is the average objective value obtained in 20 executions for instance $i$ and $opt_i$ is the optimal objective value of that instance.

| instances | $f^*$ | $min$ | $max$ | $\overline{f}$ | $\sigma$ | $\bar{t}$ | speedup ACO | speedup EC | speedup TS |
|---|---|---|---|---|---|---|---|---|---|
| g25-4-01 | 219 | 219 | 219 | 219 | 0 | 0.03 | 0.15 | 3.13 | 0.15 |
| g25-4-02 | 607 | 607 | 607 | 607 | 0 | 0.01 | 0.45 | 5.76 | 1.21 |
| g25-4-03 | 464 | 464 | 464 | 464 | 0 | 0.01 | 0.6 | 8.6 | 0.8 |
| g25-4-04 | 620 | 620 | 620 | 620 | 0 | 0.01 | 0.7 | 5.14 | 1.69 |
| g25-4-05 | 573 | 573 | 573 | 573 | 0 | 0.01 | 1.37 | 7.02 | 3.95 |
| g50-4-01 | 460 | 460 | 460 | 460 | 0 | 0.3 | 1.3 | 2.56 | 1.82 |
| g50-4-02 | 421 | 421 | 421 | 421 | 0 | 0.06 | 0.44 | 2.06 | 0.44 |
| g50-4-03 | 565 | 565 | 565 | 565 | 0 | 0.21 | 0.75 | 2.32 | 1.26 |
| g50-4-04 | 434 | 434 | 434 | 434 | 0 | 0.03 | 0.48 | 3.53 | 0.06 |
| g50-4-05 | 387 | 387 | 387 | 387 | 0 | 0.37 | 0.16 | 0.58 | 0.16 |
| g75-4-01 | 366 | 366 | 366 | 366 | 0 | 0.13 | 1.23 | 1.43 | 0.16 |
| g75-4-02 | 295 | 295 | 295 | 295 | 0 | 0.11 | 3.72 | 1.98 | 0.15 |
| g75-4-03 | 412 | 412 | 412 | 412 | 0 | 0.26 | 2.48 | 6.81 | 0.05 |
| g75-4-04 | 430 | 430 | 430 | 430 | 0 | 0.31 | 1.83 | 4.48 | 0.46 |
| g75-4-05 | 284 | 284 | 284 | 284 | 0 | 0.09 | 0.17 | 1.36 | 0.11 |
| g100-4-01 | 363 | 363 | 363 | 363 | 0 | 0.12 | 8.75 | 2.22 | 1.12 |
| g100-4-02 | 335 | 335 | 335 | 335 | 0 | 0.35 | 3.11 | 3.13 | 0.37 |
| g100-4-03 | 412 | 412 | 412 | 412 | 0 | 0.19 | 0.07 | 0.74 | 0.1 |
| g100-4-04 | 442 | 442 | 442 | 442 | 0 | 0.57 | 0.03 | 0.3 | 0.11 |
| g100-4-05 | 388 | 388 | 388 | 388 | 0 | 1.03 | 0.21 | 1.45 | 0.16 |
| g200-4-01 | 308 | 308 | 308 | 308 | 0 | 0.45 | 0.16 | 0.81 | 0.13 |
| g200-4-02 | 299 | 299 | 299 | 299 | 0 | 0.24 | 0.28 | 1.73 | 0.64 |
| g200-4-03 | 300 | 300 | 300 | 300 | 0 | 0.23 | 0.17 | 1.08 | 0.13 |
| g200-4-04 | 304 | 304 | 304 | 304 | 0 | 2.16 | 0.33 | 0.49 | 0.14 |
| g200-4-05 | 357 | 357 | 357 | 357 | 0 | 0.32 | 0.15 | 0.82 | 0.08 |
| g400-4-01 | 253 | 253 | 253 | 253 | 0 | 0.32 | 0.23 | 0.83 | 0.06 |
| g400-4-02 | 328 | 328 | 328 | 328 | 0 | 1.1 | 3.63 | 2.27 | 0.17 |
| g400-4-03 | 302 | 302 | 302 | 302 | 0 | 0.79 | 2.11 | 5.48 | 0.29 |
| g400-4-04 | 306 | 306 | 306 | 306 | 0 | 0.87 | 0.72 | 1.21 | 0.1 |
| g400-4-05 | 320 | 320 | 320 | 320 | 0 | 0.96 | 1.92 | 2.48 | 0.32 |
| g1000-4-01 | 263 | 263 | 263 | 263 | 0 | 11.57 | 0.55 | 0.46 | 0.73 |
| g1000-4-02 | 281 | 281 | 281 | 281 | 0 | 3.31 | 0.86 | 2.03 | 0.31 |
| g1000-4-03 | 289 | 289 | 289 | 289 | 0 | 2.41 | 0.6 | 2.64 | 0.36 |
| g1000-4-04 | 298 | 298 | 306 | 298.8 | 2.4 | 28.08 | 0.12 | 0.12 | 0.22 |
| g1000-4-05 | 268 | 268 | 268 | 268 | 0 | 0.74 | 0.53 | 1.74 | 0.1 |

Table 5.1: Experimental results of the KCT_MTABU on the first benchmark

| ♯ vertices | $< 1000$ | 1000-1089 | 2500 |
|---|---|---|---|
| KCT_MTABU | 0.67 | 23.41 | 155.27 |
| ILP-based | 11.7 | 124.8 | 3704.1 |

Table 5.2: Average times (in seconds) for reaching optimal solutions of KCT_MTABU with $k = 2, n - 2$ and of the algorithm of [CKIL09] with all values of cardinality $k$ on grid graphs 15x15, 45x5, 33x33, 100x10 and 50x50.

constraint). Column 7 presents the average times (in seconds) for finding the best solutions of 20 executions. The results show that the model finds quickly optimal solutions for all instances except the instances g1000-4-03.dat and g1000-4-04.dat.

We conclude that by using the `LS(Graph)`, a local search model can be implemented easily, shortly and flexibly which gives good results.

## 5.3 The quorumcast routing (QR) problem

### 5.3.1 Problem formulation

Given a weighted undirected graph $G = (V, E)$, a source node $s \in V$, an integral value $q$ and a set $S \subseteq V$, the quorumcast routing problem consists of finding a minimum tree of $G$ spanning $s$ and $q$ nodes of $S$.

### 5.3.2 The model

We propose a tabu search model in `LS(Graph)` exploring different neighborhoods for solving this problem. The model is given in Figure 5.4 in which line 1 creates a `Solver<LSGraph> ls` and line 2 declares a `VarTree tr` associated with `ls`. Lines 4-7 state 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 the `TabuSearch<LSGraph>` which is depicted in Figure 5.5. Lines 12-14 set parameters for the search and line 16 call the search procedure.

We now describe the search component in Figure 5.5. 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 15-29) constructs the tree in a greedy random way. It clears the tree `tr` (line 20) and selects randomly a first edge containing `_root` (lines 21-23). It then iteratively selects with minimal weight for adding to the constructed tree `tr` (lines 25-28). The `exploreNeighborhood` of the `TabuSearch<LSGraph>` is also overriden (lines 32-37) with different neighborhoods: $NT_1$ (line 35), $NT_2$ (line 34), $NT_{1+2}$ (line 35) and $NT_3$ (line 36).

| Instances | $\overline{f}$ | $f\_min$ | $f\_max$ | $\sigma$ | $f^*$ | $\bar{t}$ |
|---|---|---|---|---|---|---|
| g25-4-01.dat | 219 | 219 | 219 | 0 | 219 | 0.07 |
| g25-4-02.dat | 607 | 607 | 607 | 0 | 607 | 0.04 |
| g25-4-03.dat | 464 | 464 | 464 | 0 | 464 | 0.04 |
| g25-4-04.dat | 620 | 620 | 620 | 0 | 620 | 0.04 |
| g25-4-05.dat | 573 | 573 | 573 | 0 | 573 | 0.04 |
| g50-4-01.dat | 460 | 460 | 460 | 0 | 460 | 0.78 |
| g50-4-02.dat | 421 | 421 | 421 | 0 | 421 | 0.26 |
| g50-4-03.dat | 565 | 565 | 565 | 0 | 565 | 0.5 |
| g50-4-04.dat | 434 | 434 | 434 | 0 | 434 | 0.18 |
| g50-4-05.dat | 387 | 387 | 387 | 0 | 387 | 1.45 |
| g75-4-01.dat | 366 | 366 | 366 | 0 | 366 | 0.4 |
| g75-4-02.dat | 295 | 295 | 295 | 0 | 295 | 0.42 |
| g75-4-03.dat | 412 | 412 | 412 | 0 | 412 | 0.93 |
| g75-4-04.dat | 430 | 430 | 430 | 0 | 430 | 1.55 |
| g75-4-05.dat | 284 | 284 | 284 | 0 | 284 | 0.38 |
| g100-4-01.dat | 363 | 363 | 363 | 0 | 363 | 0.53 |
| g100-4-02.dat | 335 | 335 | 335 | 0 | 335 | 1.64 |
| g100-4-03.dat | 412.05 | 412 | 413 | 0.22 | 412 | 2.88 |
| g100-4-04.dat | 442 | 442 | 442 | 0 | 442 | 0.92 |
| g100-4-05.dat | 388 | 388 | 388 | 0 | 388 | 3.81 |
| g200-4-01.dat | 308 | 308 | 308 | 0 | 308 | 2.17 |
| g200-4-02.dat | 299 | 299 | 299 | 0 | 299 | 1.01 |
| g200-4-03.dat | 300 | 300 | 300 | 0 | 300 | 0.8 |
| g200-4-04.dat | 304 | 304 | 304 | 0 | 304 | 7.16 |
| g200-4-05.dat | 357 | 357 | 357 | 0 | 357 | 1.49 |
| g400-4-01.dat | 253 | 253 | 253 | 0 | 253 | 0.93 |
| g400-4-02.dat | 328 | 328 | 328 | 0 | 328 | 5.3 |
| g400-4-03.dat | 302 | 302 | 302 | 0 | 302 | 2.01 |
| g400-4-04.dat | 306 | 306 | 306 | 0 | 306 | 2.6 |
| g400-4-05.dat | 320 | 320 | 320 | 0 | 320 | 2.96 |
| g1000-4-01.dat | 263.25 | 263 | 268 | 1.09 | 263 | 52.95 |
| g1000-4-02.dat | 281 | 281 | 281 | 0 | 281 | 15.64 |
| g1000-4-03.dat | 294 | 294 | 294 | 0 | 289 | 6.37 |
| g1000-4-04.dat | 302.9 | 302 | 306 | 1.61 | 298 | 47.09 |
| g1000-4-05.dat | 268 | 268 | 268 | 0 | 268 | 2.08 |

Table 5.3: Experimental results of the KCT_MTABU on the blxh benchmark with the degree side constraint

```
1  Solver<LSGraph> ls();
2  VarTree tr(ls,g);

4  ConstraintSystem<LSGraph> gcs(ls);
5  gcs.post(q <= NBVisitedVerticesTree(tr,S));
6  gcs.post(NBVisitedVerticesTree(tr,s) == 1);
7  gcs.close();

9  Model<LSGraph>
       mod(tr,gcs,Weight<Tree>(tr,1),NonSpanningTree,MINIMIZATION);

11 QRSearch se(mod);
12 se.setMaxIter(100000);
13 se.setMaxTime(1800);
14 se.setCard(q);
15 se.setRoot(s);

17 se.search();
```

Figure 5.4: A model in `LS(Graph)` for the QR problem

### 5.3.3 Experiments

We compare the model in `LS(Graph)` with the IMP heuristic which is the best heuristic among three heuristic algorithms in [CA94]. The IMP algorithm has been re-implemented in `COMET`.

We have conducted a preliminary experiments and decided to choose the tabu search parameters as follows: `tbMin` = $n/p\_tbMin$, `tbMax` = $n/p\_tbMax$, `tinc` = `(tbMax-tbMin)`/$p\_tinc+1$, and `maxStable` = $p\_maxStable$ in which $n$ is the number of vertices of the given graph with (see also Table 5.4):

- $p\_tbMin \in \{3, 5\}$,
- $p\_tbMax \in \{2\}$,
- $p\_tinc \in \{1, 2, 4\}$,
- $p\_maxStable \in \{50, 200\}$.

The preliminary results show that on small instances (i.e., graphs up to 100 vertices), the parameters do not influence on the quality of the solutions. Figures 5.6, 5.7, 5.8 depict the influence of parameters on the quality of the solutions on some large instances. The X-axis presents the indices of the parameters (see Table 5.4) and the Y-axis presents the average objective values of 20 executions for each instance. We can see that the parameters indexed by 12 of Table 5.4 are most promising. Finally, we select $p\_tbMin = 5$, $p\_tbMax = 2$, $p\_tinc = 4$, $p\_maxStable = 200$.

**Problem instances** We take 35 graphs from the benchmark of the KCT problem [BB05] which are 4-regular graphs of size from 25 to 1000 nodes. For each graph of

```
1  class QRSearch extends TabuSearch<LSGraph>{
2    Vertex  root;
3    int    _card;
4    QRSearch(Model<LSGraph> mod): TabuSearch<LSGraph>(mod){
5    }
6    void setCard(int ca){
7      _card = ca;
8    }
9    void setRoot(Vertex r){
10     root = r;
11   }
12   void restartSolution(){
13     initSolution();
14   }
15   void initSolution(){
16     Solver<LSGraph> ls = getLSGraphSolver();
17     VarTree tr = getFirstVarTree();
18     InsertableEdgesVarTree inst = getInsertableEdges(tr);

20     tr.clear();
21     select(e in inst.getSet():e.contains(root)){
22       tr.addEdge(e);
23     }

25     forall(i in 1..._card-1)
26       selectMin(e in inst.getSet())(e.weight()){
27         tr.addEdge(e);
28       }
29   }


32   void exploreNeighborhood(Neighborhood N){
33     exploreTabuMinAdd1VarTree(N,true);
34     exploreTabuMinRemove1VarTree(N,true);
35     exploreTabuMinAddRemove1VarTree(N,true);
36     exploreTabuMinReplace1VarTree(N,true);
37   }
38 }
```

Figure 5.5: The search component for the QR problem

| index | $p\_tbMin$ | $p\_tbMax$ | $p\_tinc$ | $p\_maxStable$ |
|---|---|---|---|---|
| 1 | 3 | 2 | 1 | 50 |
| 2 | 3 | 2 | 1 | 200 |
| 3 | 3 | 2 | 2 | 50 |
| 4 | 3 | 2 | 2 | 200 |
| 5 | 3 | 2 | 4 | 50 |
| 6 | 3 | 2 | 4 | 200 |
| 7 | 5 | 2 | 1 | 50 |
| 8 | 5 | 2 | 1 | 200 |
| 9 | 5 | 2 | 2 | 50 |
| 10 | 5 | 2 | 2 | 200 |
| 11 | 5 | 2 | 4 | 50 |
| 12 | 5 | 2 | 4 | 200 |

Table 5.4: Parameters tried for the QR problem

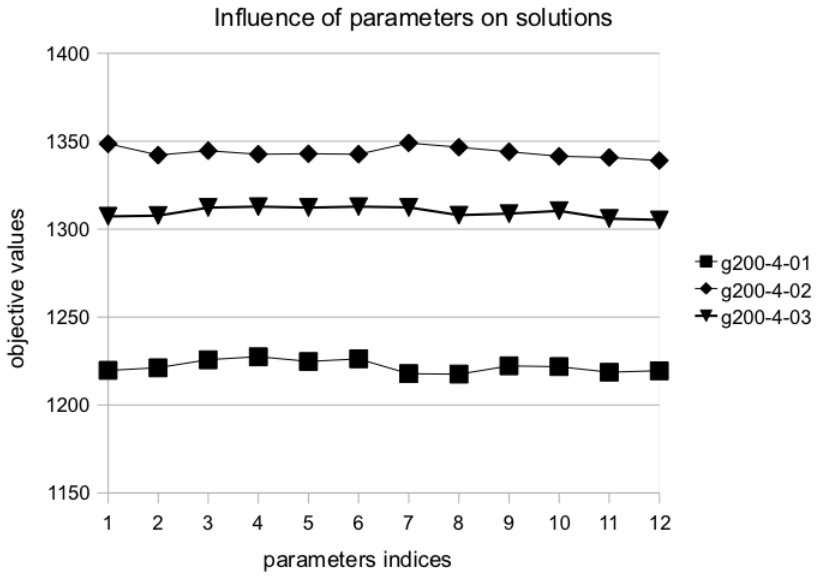



Figure 5.6: Influence of parameters on solutions (instances of 200 vertices)

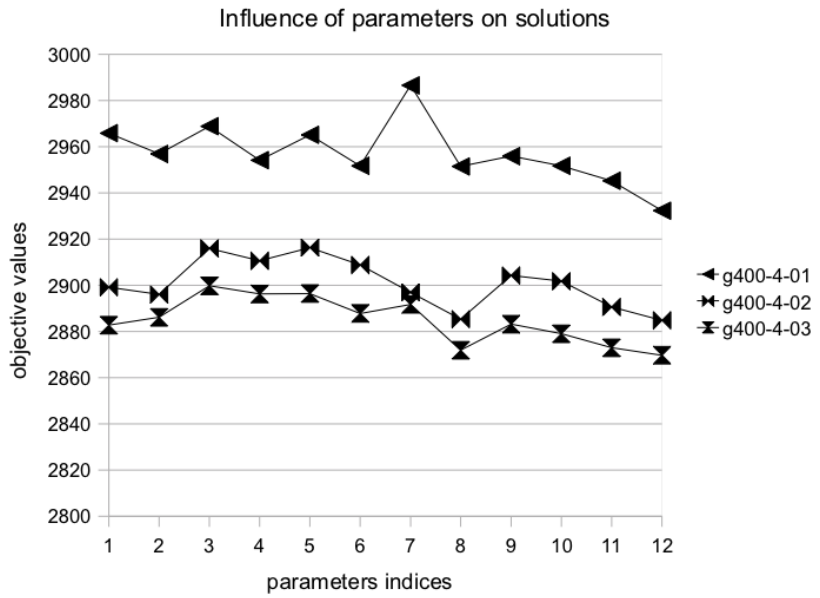



Figure 5.7: Influence of parameters on solutions (instances of 400 vertices)

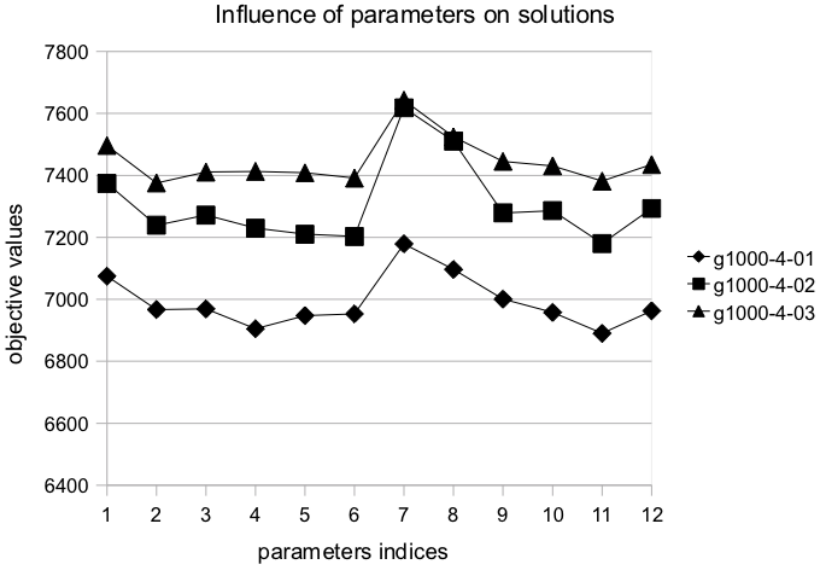



Figure 5.8: Influence of parameters on solutions (instances of 1000 vertices)

size $n$, we generate randomly $n/2$ nodes for the set $S$, the value for $q$ is set to $n/4$ and the root is set to the 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 minutes. Experimental results are shown in Table 5.5. Columns 2-5 present the average, the minimal, the maximal, the standard deviation of the best objective value found in 20 executions. The same information for our tabu search model is presented in columns 7-10. Column 6 is the average of execution time (in seconds) of the IMP algorithm while column 11 presents the average of time (in seconds) for finding the best solutions in 20 executions. Figure 5.9 presents the comparison between the average objective values (among 20 executions) of best solutions found by the IMP heuristic and our tabu search model. Figure 5.10 presents the comparison between the objective value of the best solution (among 20 executions) found by the IMP heuristic and the objective value of the worst solution (among 20 executions) found by the tabu search model. We can see that our tabu search model finds better solutions than the IMP in average. Moreover, the worst solutions found by our model are even better than the best solution found by the IMP (among 20 executions) in most cases.

The results also show that for small instances (from 25 nodes to 100 nodes), the results among 20 runs are stable while for larger instances (from 200 nodes to 1000 nodes), the standard deviation slightly increases. Once again, we find that the local search model is easily and shortly implemented using the `LS(Graph)` framework. Experimental results on large instances show the benefit of the framework from both modeling and computational standpoints.

## 5.4 The resource constrained shortest path (RCSP) problem

### 5.4.1 Problem formulation

Given a directed graph $G = (V, E)$, each arc $e$ is associated with a length $c(e) \geq 0$ and a vector $r(e) \geq 0$ of resources consumed in traversing the arc $e$. Given a source node $s$, a destination node $t$ and two vectors $L$, $U$ of resources corresponding to the minimum and maximum amount that can be used on the chosen path (i.e., a lower and an upper limit on the resources consumed on the path). The length of a path $\mathcal{P}$ is defined as $f(\mathcal{P}) = \sum_{e \in \mathcal{P}} c(e)$. The resources consumed in traversing $\mathcal{P}$ is defined as $r(\mathcal{P}) = \sum_{e \in \mathcal{P}} r(e)$ The formulation of RCSP is then given by:

$$
\begin{array}{lll}
\min & f(\mathcal{P}) & (1) \\
\text{s.t.} & L \leq r(\mathcal{P}) & (2) \\
 & r(\mathcal{P}) \leq U & (3) \\
 & \mathcal{P} \text{ is a path from } s \text{ to } t \text{ on } G & (4)
\end{array}
$$

| Instances | IMP | | | | | LS(Graph) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | avg | min | max | std_dev | avg_t | avg | min | max | std_dev | avg_t |
| g25-4-01.dat | 110 | 110 | 110 | 0 | 1 | 110 | 110 | 110 | 0 | 0.14 |
| g25-4-02.dat | 245 | 245 | 245 | 0 | 1 | 231 | 231 | 231 | 0 | 0.17 |
| g25-4-03.dat | 149 | 149 | 149 | 0 | 1 | 149 | 149 | 149 | 0 | 0.02 |
| g25-4-04.dat | 210 | 210 | 210 | 0 | 0.99 | 204 | 204 | 204 | 0 | 0.01 |
| g25-4-05.dat | 217 | 217 | 217 | 0 | 0.99 | 217 | 217 | 217 | 0 | 0.45 |
| g50-4-01.dat | 325 | 325 | 325 | 0 | 1.12 | 325 | 325 | 325 | 0 | 0.1 |
| g50-4-02.dat | 471 | 471 | 471 | 0 | 1.09 | 455 | 455 | 455 | 0 | 0.77 |
| g50-4-03.dat | 473 | 473 | 473 | 0 | 1.08 | 464 | 464 | 464 | 0 | 0.57 |
| g50-4-04.dat | 480 | 480 | 480 | 0 | 1.1 | 441 | 441 | 441 | 0 | 2.56 |
| g50-4-05.dat | 368 | 368 | 368 | 0 | 1.1 | 334 | 334 | 334 | 0 | 0.13 |
| g75-4-01.dat | 546 | 546 | 546 | 0 | 1.32 | 480 | 480 | 480 | 0 | 2.15 |
| g75-4-02.dat | 501.6 | 498 | 506 | 3.98 | 1.31 | 465 | 465 | 465 | 0 | 11.89 |
| g75-4-03.dat | 545 | 545 | 545 | 0 | 1.33 | 526 | 526 | 526 | 0 | 3.49 |
| g75-4-04.dat | 750 | 750 | 750 | 0 | 1.33 | 665 | 665 | 665 | 0 | 13.58 |
| g75-4-05.dat | 431 | 431 | 431 | 0 | 1.33 | 427 | 427 | 427 | 0 | 0.51 |
| g100-4-01.dat | 774 | 774 | 774 | 0 | 1.82 | 756 | 756 | 756 | 0 | 20.85 |
| g100-4-02.dat | 765 | 765 | 765 | 0 | 1.84 | 701 | 701 | 701 | 0 | 19.12 |
| g100-4-03.dat | 952.5 | 947 | 957 | 4.97 | 1.84 | 859 | 859 | 859 | 0 | 59.81 |
| g100-4-04.dat | 872 | 872 | 872 | 0 | 1.81 | 835 | 835 | 835 | 0 | 124.9 |
| g100-4-05.dat | 681 | 681 | 681 | 0 | 1.83 | 641 | 641 | 641 | 0 | 43.68 |
| g200-4-01.dat | 1362 | 1362 | 1362 | 0 | 7.52 | 1219.4 | 1217 | 1225 | 2.6 | 532.68 |
| g200-4-02.dat | 1469.7 | 1437 | 1480 | 14.62 | 7.6 | 1339 | 1334 | 1344 | 3.08 | 405.93 |
| g200-4-03.dat | 1445 | 1445 | 1445 | 0 | 7.63 | 1305.25 | 1297 | 1318 | 6.67 | 531.59 |
| g200-4-04.dat | 1665.65 | 1646 | 1677 | 9.71 | 7.52 | 1530.85 | 1527 | 1537 | 2.67 | 731.35 |
| g200-4-05.dat | 1523 | 1523 | 1523 | 0 | 7.49 | 1428.65 | 1425 | 1439 | 4.35 | 689.38 |
| g400-4-01.dat | 3003.95 | 3000 | 3017 | 6.51 | 51.2 | 2932.3 | 2910 | 2951 | 10.19 | 728.36 |
| g400-4-02.dat | 3245.1 | 3168 | 3291 | 33.95 | 51.38 | 2884.8 | 2862 | 2915 | 13.58 | 763.07 |
| g400-4-03.dat | 3112.3 | 3094 | 3134 | 12.21 | 51.23 | 2869.7 | 2853 | 2889 | 8.3 | 772.34 |
| g400-4-04.dat | 3137.5 | 3115 | 3187 | 17.18 | 51.13 | 2923.6 | 2880 | 2954 | 22.71 | 820.53 |
| g400-4-05.dat | 3152.4 | 3140 | 3160 | 7.14 | 51.09 | 2853.1 | 2831 | 2879 | 14.19 | 787 |
| g1000-4-01.dat | 7103.4 | 7066 | 7173 | 30.56 | 788.11 | 6963.1 | 6790 | 7290 | 124.45 | 1430.34 |
| g1000-4-02.dat | 7606.35 | 7509 | 7702 | 55.62 | 786.58 | 7293.3 | 7118 | 7483 | 89.48 | 1461.48 |
| g1000-4-03.dat | 7785.1 | 7745 | 7853 | 31.56 | 788.36 | 7435.4 | 7263 | 7567 | 78.15 | 1332.6 |
| g1000-4-04.dat | 8030.2 | 7996 | 8069 | 24.24 | 787.59 | 7764.6 | 7651 | 7933 | 70.39 | 1234.6 |
| g1000-4-05.dat | 7642.05 | 7559 | 7730 | 59.71 | 787.05 | 7363.55 | 7220 | 7590 | 99.09 | 1335.74 |

Table 5.5: Comparison between the IMP heuristic [CA94] and a local search algorithm implemented in `LS(Graph)`

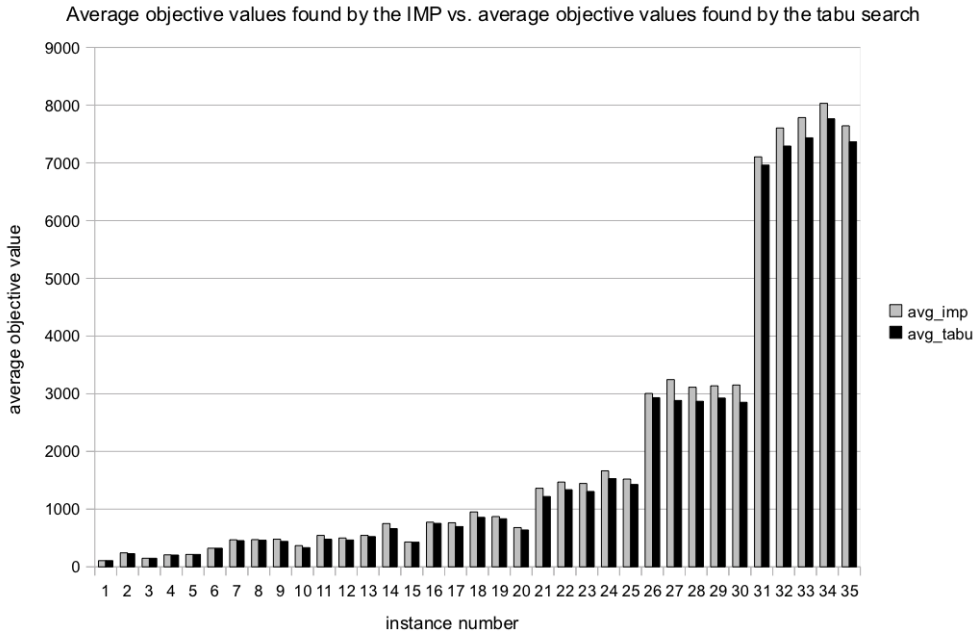



Figure 5.9: Comparison between average objective value found by the IMP heuristic (avg_imp) and those found by the tabu search algorithm in LS(Graph) (avg_tabu) among 20 executions for each instance

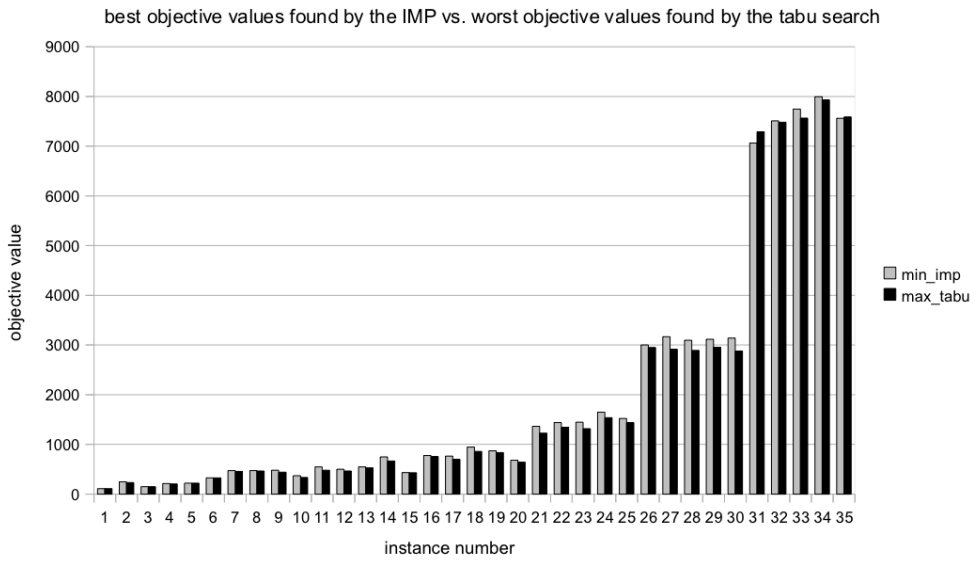



Figure 5.10: Comparison between min objective values found by the IMP heuristic (min_imp) and max objective values found by the tabu search algorithm in LS(Graph) (max_tabu) among 20 executions for each instance

### 5.4.2 The Model

The model is given in Figure 5.11. Line 1 creates a `Solver<LSGraph> ls` and line 3 initializes a `VarPath vp`. Lines 5-7 initialize graph functions `r[i]` representing the $i^{th}$ resource consumed along `vp` where `PathCostOnEdgePrune` extends `PathCostOnEdges` adding pruning methods. For example, when the cost of a given path is required to be less than or equal to a given value, we can apply a simple pruning method [BC89] for removing vertices, edges from the given graph and this pruning allows to reduce the size of the problem. Line 9 create the objective function `len` to be minimized[5]. Line 11 initializes a `ConstraintSystem<LSGraph> gcs` and lines 12-16 state and post all constraints over resource consumed along the path `vp`. Line 18 encapsulates variable `vp`, constraint to be satisfied `gcs` and objective function to be minimized `len` into a `Model<LSGraph>` object `mod`. Line 20 creates a search component `se` which extends the built-in `TabuSearch<LSGraph>` (see Figure 5.12). Line 23 creates a `LSGraphPrune` object `prune` which apply pruning methods of all constraints posted (line 24). Whenever the search `se` finds a new best solution (line 26), the length of optimal solution should be less than or equal to the length of this new best solution (see method `se.getBestObjValue()`, we can thus prune the solution space (remove vertices, edges that do not belong to any optimal solutions) by applying the `pruneLessThan` method in line 27. After the pruning (some edges, vertices may be removed from the given graph), the `VarPath vp` is reinitialized randomly and line 28 updates the best solution if the new initial solution is better than the best solution found so far. After the pruning, the given graph may be null (all edges, vertices are removed). In this case, we stop the search and we obtain optimal solution (line 30-31). Line 34 performs the search.

The search is depicted in Figure 5.12 which extends the built-in `TabuSearch<LSGraph>`. The solution is initialized randomly (lines 10-16) and the `restartSolution` simply calls the `initSolution`. The `exploreNeighborhood` is overridden (lines 18-21) which explores the neighborhood $ERNP_1$ and $\mathcal{N}_2(tr)$ (see Appendix for the detail about these neighborhood explorations). We choose the tabu search parameters as follows: `tbMin` = 5, `tbMax` = 33, `tinc` = 8, `maxStable` = 200 (see Appendix B for more description detail about the tabu search schema and its parameters).

### 5.4.3 Experiments

#### Problem instances

We experiment the model over two kinds of benchmark. For the first benchmark, we take the 24 graphs experimented in the paper [BC89] from OR-library (called OR graphs) and for the second benchmark, we use the generator in [CRW08] for generating graphs of structures A-H described in that paper for the aircraft routing (called aircraft graphs). For each of these graphs, we generate randomly the lower bound and upper bound of the resources consumed as follows. We first generate randomly

[5]Each edge `e` has a vector of properties `w[0..nbrRCs]` where `w[0]` represents the length of the edge and `w[1..nbrRCs]` is the vector of resources consumed in traversing `e`.

```
1  Solver<LSGraph> ls();

3  VarPath vp(ls,g,s,t);

5  GraphFunctionPrune<LSGraph> r[resources];
6  forall(i in resources)
7    r[i] = new PathCostOnEdgesPrune(vp,i);

9  PathCostOnEdgesPrune len(vp,0);

11 ConstraintSystem<LSGraph> gcs(ls);
12 forall(i in resources){
13   gcs.post(L[i] <= r[i]);
14   gcs.post(r[i] <= U[i]);
15 }
16 gcs.close();

18 Model<LSGraph> mod(vp,gcs,len,MINIMIZATION);

20 RCSPSearch se(mod);
21 se.setMaxIter(10000);

23 LSGraphPrune prune(ls);
24 prune.post(gcs);

26 whenever se@evtNewBest(){
27   prune.pruneLessThan(len,se.getBestObjValue());
28   se.update();

30   if(vp.isNull())
31     se.stopSearch();
32 }

34 se.search();
```

Figure 5.11: Model for the RCSP problem

```
1  class RCSPSearch extends TabuSearch<LSGraph>{
2    RCSPSearch(Model<LSGraph> mod): TabuSearch<LSGraph>(mod){

4    }

6    void restartSolution(){
7      initSolution();
8    }

10   void initSolution(){
11     VarPath[] vps = getVarPaths();
12     forall(i in vps.rng()){
13       vps[i].init();
14     }
15     return;
16   }

18   void exploreNeighborhood(Neighborhood N){
19     exploreTabuMinReplace1Move1VarPath(N,true);
20     exploreTabuMin1ReplaceXVY1VarPath(N,true);
21   }
22 }
```

Figure 5.12: Search component for the RCSP problem

1000 paths from the source to the destination (i.e., by running 1000 random moves over a `VarPath`). We then choose randomly 5 paths, each path $i$ has a resource consumed $R_i = (r_1^i, r_2^i, ..., r_k^i)$. The lower bound and upper bound of resource consumed is $L = (l_1, l_2, ..., l_k)$ and $U = (u_1, u_2, ..., u_k)$ where $l_j = min\{r_j^1, r_j^2, r_j^3, r_j^4, r_j^5\}$ and $u_j = max\{r_j^1, r_j^2, r_j^3, r_j^4, r_j^5\}$. This ensures that feasible solutions to the RCSP problem always exist.

Each problem instance consists of a graph and a pair of lower bound and upper bound vector of resources consumed. For each of the OR graphs, we generate only one pair of lower bound and upper bound of vector resources consumed and for each of the aircraft graphs, we generate three such pairs. Note that the number of resources consumed in the aircraft graphs is one and in the OR graphs is one or ten. For each problem instance, we execute the model 20 times with the time limit of 30 minutes.

### Results

Table 5.6 presents the experimental result of our tabu search model in `LS(Graph)` on benchmark from OR-Library where the constraint over lower bound of resource consumed is not considered (to do it, we simply remove the line 13 in Figure 5.11). Column 6 presents the execution time (sec.) of the LRE algorithm of [CRW08] implemented in C++ and is executed on 3.8 GHz Intel Pentium IV processor, 3 giga- bytes of RAM, and the Microsoft Windows XP Professional operating system. Columns 7-10 present the average, the minimal, the maximal and the standard deviation of time (in sec.) for finding optimal solutions in 20 executions of our tabu search model. The

| Instances | $\|V\|$ | $\|E\|$ | $R$ | $opt$ | LRE | LS(Graph) | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | $t^*$ | $\bar{t}$ | $t\_min$ | $t\_max$ | $\sigma$ |
| rcsp1.txt | 100 | 955 | 1 | 131 | 0.00 | 2.07 | 1.5 | 2.89 | 0.37 |
| rcsp2.txt | 100 | 955 | 1 | 131 | 0.00 | 2.02 | 1.42 | 2.57 | 0.29 |
| rcsp3.txt | 100 | 959 | 1 | 2 | 0.00 | 2.56 | 1.81 | 3.82 | 0.55 |
| rcsp4.txt | 100 | 959 | 1 | 2 | 0.00 | 2.88 | 1.75 | 3.57 | 0.47 |
| rcsp5.txt | 100 | 990 | 10 | 100 | 0.02 | 2.51 | 1.6 | 3.36 | 0.68 |
| rcsp6.txt | 100 | 990 | 10 | 100 | 0.02 | 2.66 | 1.5 | 4.18 | 0.76 |
| rcsp7.txt | 100 | 999 | 10 | 6 | 0.02 | 5.65 | 2.37 | 12.17 | 2.73 |
| rcsp8.txt | 100 | 999 | 10 | 14 | 0.02 | 8.26 | 2.7 | 21.38 | 4.57 |
| rcsp9.txt | 200 | 2040 | 1 | 420 | 0.00 | 6.46 | 2.49 | 11.77 | 2.93 |
| rcsp10.txt | 200 | 2040 | 1 | 420 | 0.00 | 7.53 | 1.94 | 16.75 | 4.62 |
| rcsp11.txt | 200 | 1971 | 1 | 6 | 0.00 | 5.08 | 2.68 | 8.47 | 1.33 |
| rcsp12.txt | 200 | 1971 | 1 | 6 | 0.00 | 5.03 | 2.73 | 8.45 | 1.42 |
| rcsp13.txt | 200 | 2080 | 10 | 448 | 0.05 | 15.49 | 3.82 | 46.98 | 9.8 |
| rcsp15.txt | 200 | 1960 | 10 | 9 | 0.05 | 14.78 | 2.99 | 25.31 | 5.61 |
| rcsp16.txt | 200 | 1960 | 10 | 17 | 0.05 | 11.28 | 2.98 | 26.45 | 6.56 |
| rcsp17.txt | 500 | 4858 | 1 | 652 | 0.00 | 63.47 | 9.26 | 130.3 | 30.54 |
| rcsp18.txt | 500 | 4858 | 1 | 652 | 0.00 | 47.26 | 5.67 | 130.15 | 34.5 |
| rcsp19.txt | 500 | 4978 | 1 | 6 | 0.00 | 16.23 | 4.06 | 32.21 | 6.85 |
| rcsp20.txt | 500 | 4978 | 1 | 6 | 0.02 | 20.02 | 8.89 | 31.31 | 6.04 |
| rcsp21.txt | 500 | 4847 | 10 | 858 | 0.09 | 52.45 | 15.28 | 82.99 | 22.07 |
| rcsp22.txt | 500 | 4847 | 10 | 858 | 0.09 | 60.66 | 20.85 | 148 | 32.38 |
| rcsp23.txt | 500 | 4868 | 10 | 4 | 0.08 | 43.83 | 7.54 | 95.88 | 22.92 |
| rcsp24.txt | 500 | 4868 | 10 | 5 | 0.08 | 64.17 | 14.52 | 152.28 | 44.89 |

Table 5.6: Experimental results of our local search model implemented in `LS(Graph)` on OR benchmark

table shows that the tabu search can find optimal solutions for all instances in all executions but slower than the exact LRE algorithm of [CRW08] implemented in C++.

Experimental results for the RCSP problem considering both constraints on lower bound (2) and upper bound (3) of resource consumed are presented in Tables 5.7 and 5.8. For this problem, no comparison is made. We report the average, the minimal and the maximal value of the best objective value found, the average of time (in sec.) for finding these values and the standard deviation of the best objective value found in 20 executions (see columns 5-9 in Table 5.7 and columns 6-10 in Table 5.8). Experimental results show that for the first benchmark with smaller graphs, the best objective values are convergent, the standard deviation is 0 for all instances. For the second benchmark with larger graphs, the best objective values found in 20 executions are quite divers and the average time for reaching these value is higher than in the first benchmark.

| Instances | $\|V\|$ | $\|E\|$ | $R$ | $\overline{f}$ | $f\_min$ | $f\_max$ | $\bar{t}(sec.)$ | $\sigma$ |
|---|---|---|---|---|---|---|---|---|
| rcsp1.txt.ext | 100 | 955 | 1 | 80 | 80 | 80 | 1.48 | 0 |
| rcsp2.txt.ext | 100 | 955 | 1 | 80 | 80 | 80 | 1.55 | 0 |
| rcsp3.txt.ext | 100 | 959 | 1 | 2 | 2 | 2 | 2.88 | 0 |
| rcsp4.txt.ext | 100 | 959 | 1 | 1 | 1 | 1 | 2.6 | 0 |
| rcsp5.txt.ext | 100 | 990 | 10 | 89 | 89 | 89 | 1.88 | 0 |
| rcsp6.txt.ext | 100 | 990 | 10 | 89 | 89 | 89 | 4.93 | 0 |
| rcsp7.txt.ext | 100 | 999 | 10 | 4 | 4 | 4 | 17.24 | 0 |
| rcsp8.txt.ext | 100 | 999 | 10 | 5 | 5 | 5 | 29.83 | 0 |
| rcsp9.txt.ext | 200 | 2040 | 1 | 248 | 248 | 248 | 6.73 | 0 |
| rcsp10.txt.ext | 200 | 2040 | 1 | 248 | 248 | 248 | 5.63 | 0 |
| rcsp11.txt.ext | 200 | 1971 | 1 | 6 | 6 | 6 | 12.34 | 0 |
| rcsp12.txt.ext | 200 | 1971 | 1 | 6 | 6 | 6 | 15.58 | 0 |
| rcsp13.txt.ext | 200 | 2080 | 10 | 260 | 260 | 260 | 10.52 | 0 |
| rcsp14.txt.ext | 200 | 2080 | 10 | 289 | 289 | 289 | 26.24 | 0 |
| rcsp15.txt.ext | 200 | 1960 | 10 | 8 | 8 | 8 | 241.17 | 0 |
| rcsp16.txt.ext | 200 | 1960 | 10 | 8 | 8 | 8 | 546.97 | 0 |
| rcsp17.txt.ext | 500 | 4858 | 1 | 690 | 690 | 690 | 37.1 | 0 |
| rcsp18.txt.ext | 500 | 4858 | 1 | 455 | 455 | 455 | 51.77 | 0 |
| rcsp19.txt.ext | 500 | 4978 | 1 | 6 | 6 | 6 | 48.66 | 0 |
| rcsp20.txt.ext | 500 | 4978 | 1 | 6 | 6 | 6 | 117.24 | 0 |
| rcsp21.txt.ext | 500 | 4847 | 10 | 735 | 735 | 735 | 89.32 | 0 |
| rcsp22.txt.ext | 500 | 4847 | 10 | 611 | 611 | 611 | 74.99 | 0 |
| rcsp23.txt.ext | 500 | 4868 | 10 | 4 | 4 | 4 | 181.55 | 0 |
| rcsp24.txt.ext | 500 | 4868 | 10 | 5 | 5 | 5 | 900.5 | 0 |

Table 5.7: Experimental results of our local search model implemented in `LS(Graph)` on OR benchmark. Both constraints over lower bound and upper bound of resource consumed are considered.

| Instances | $\|V\|$ | $\|E\|$ | $r\_min$ | $r\_max$ | $\overline{f}$ | $f\_min$ | $f\_max$ | $\bar{t}(sec.)$ | $\sigma$ |
|---|---|---|---|---|---|---|---|---|---|
| aircraftA.txt | 988 | 4712 | 540.22 | 576.1 | 0.5 | 0.16 | 1.1 | 1521.27 | 0.26 |
| | | | 487.53 | 569.47 | 0.39 | 0.17 | 0.65 | 1192.73 | 0.18 |
| | | | 519.53 | 606.16 | 0.36 | 0.08 | 0.86 | 1530.81 | 0.24 |
| aircraftB.txt | 988 | 11048 | 452.88 | 632.78 | 0.17 | 0.07 | 0.51 | 1397.21 | 0.13 |
| | | | 529.12 | 571.11 | 0.23 | 0.07 | 0.53 | 1635.04 | 0.15 |
| | | | 449.12 | 630.84 | 0.14 | 0.07 | 0.52 | 1410.64 | 0.13 |
| aircraftC.txt | 988 | 22222 | 475.91 | 598.8 | 0.23 | 0.07 | 0.49 | 1609.81 | 0.16 |
| | | | 472.55 | 604.05 | 0.24 | 0.07 | 0.6 | 1324.19 | 0.19 |
| | | | 489.28 | 578.9 | 0.2 | 0.07 | 0.52 | 1743.36 | 0.12 |
| aircraftD.txt | 988 | 123166 | 422.68 | 692.99 | 0.19 | 0.07 | 0.58 | 1619.51 | 0.16 |
| | | | 459.27 | 712.5 | 0.61 | 0.07 | 3.07 | 1290.25 | 0.89 |
| | | | 416.07 | 656.61 | 0.17 | 0.07 | 0.68 | 1478.74 | 0.18 |
| aircraftE.txt | 988 | 228042 | 435.81 | 684.11 | 0.34 | 0.11 | 1.5 | 1480.43 | 0.42 |
| | | | 432.07 | 571.01 | 0.54 | 0.1 | 1.64 | 1414.83 | 0.59 |
| | | | 352.71 | 641.83 | 0.1 | 0.07 | 0.17 | 1374.37 | 0.04 |
| aircraftF.txt | 988 | 223330 | 383.19 | 660.94 | 0.65 | 0.09 | 2.62 | 1380.09 | 0.75 |
| | | | 403.08 | 903.99 | 0.22 | 0.07 | 0.75 | 1527.77 | 0.19 |
| | | | 431.5 | 602.64 | 0.52 | 0.07 | 1.33 | 1416.74 | 0.49 |
| aircraftG.txt | 988 | 195110 | 456.02 | 633.4 | 0.43 | 0.11 | 1.19 | 1556.23 | 0.36 |
| | | | 422.6 | 838.68 | 0.36 | 0.08 | 1.19 | 1334.84 | 0.42 |
| | | | 311.83 | 811.45 | 0.2 | 0.08 | 1.04 | 1259.76 | 0.29 |
| aircraftH.txt | 988 | 118454 | 458.18 | 829.56 | 0.16 | 0.07 | 0.27 | 1584.35 | 0.07 |
| | | | 421.12 | 704.79 | 0.73 | 0.07 | 5.56 | 1478.78 | 1.62 |
| | | | 355.02 | 550.19 | 0.37 | 0.07 | 1.98 | 1448.65 | 0.56 |

Table 5.8: Experimental results of our local search model implemented in `LS(Graph)` on aircraft benchmark. Both constraints over lower bound and upper bound of resource consumed are considered.

## 5.5 The Edge-Disjoint Paths problem

### 5.5.1 Problem formulation

We are given an undirected graph $G = (V, E)$ and a set $T = \{\langle s_i, t_i\rangle \mid i = 1, 2, ..., |T|; s_i \neq t_i \in V\}$ 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 exists multually 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, the formulation of the EDP problem is given by:

| | | |
|---|---|---|
| *min* | $\sharp T'$ | (1) |
| *s.t.* | $T' \subseteq T$ | (2) |
| | $T'$ is *edp*-feasible | (3) |

#### The Simple Greedy Algorithm

We first describe the simple greedy algorithm SGA [Kle96] because our proposed algorithm (detailed later) will apply this as sub-procedure (see Algorithm 17). The algorithm starts with an empty solution $S$ (line 1). At each iteration $j$ (line 3), it tries to find a shortest path $P_j$ from $s_j$ to $t_j$ of the commodity $j$ in the graph $G_1 = (V, E_1)$ where the set of edges $E_1$ is initialized by $E$ (line 2). If such path exits, it is inserted to $S$ and the set $E_1$ is updated by removing all edges of the path $P_j$ for the next step.

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

**Algorithm 17**: 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 ← ∅;
2 E_1 ← E;
3 foreach T_j = ⟨s_j, t_j⟩ ∈ T do
4     if s_j and t_j can be connected by a path in G_1 = (V, E_1) then
5         P_j ← shortest path from s_j to t_j in G_1 = (V, E_1);
6         S ← S ∪ {P_j};
7         E_1 ← E_1 \ {e | e ∈ P_j};
8 return S;
```

In the ACO algorithm of [BB07], 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 $t_j$):

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

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 most edges in common with other paths, until all remaining paths are mutually edge-disjoint (see Algorithm 18).

**Algorithm 18**: Extract(S)

**Input**: set $S$ of paths
**Output**: subset of edges-disjoint paths of $S$

```
1 S_0 ← S;
2 while C(S_0) > 0 do
3     foreach p ∈ S_0 do
4         c(p) ← number of edges of the path p in common with other paths of S_0;
5     p* ← argMax_{p∈S_0} c(p);
6     S_0 ← S_0 \ {p*};
7 return S_0;
```

In this section, we propose two heuristic algorithms based on local search for solving this problem: the LS-SGA and LS-R algorithms. These heuristic algorithms are sophisticated and cannot be viewed as standard CBLS programs with two independent components: the model and the search. Rather, they perform a local search applying the `LS(Graph)` framework combining with additional processing: the extraction method (Algorithm 18) and the Simple Greedy Algorithm. These algorithms make use 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 proposed local search algorithm tries to minimize this criterion.

### The LS-SGA algorithm

The LS-SGA algorithm is first proposed in our paper [PDH10]. The main idea of the LS-SGA algorithm (detailed in Algorithm 19) is to perform a local search algorithm aiming at minimizing the number of the violation of the $PathsEdgeDisjoint(P_1, P_2, ..., P_k)$ constraint. 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 take 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) combining with a 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 10-11). The best solution is updated in line 17 and lines 18-20 update some paths of $S$ by new found paths of $S_2$.

**Algorithm 19**: 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 ← ∅;
2  S ← ∅;
3  foreach ⟨s_i, t_i⟩ ∈ T do
4      p_i ← random path from s_i to t_i on G;
5      S ← S ∪ {p_i};
6  while termination criterion is not reached do
7      hasMove ← LocalMove(S);
8      if hasMove then
9          S_1 ← Extract(S);
10         T' ← set of commodities that are connected by paths in S_1;
11         T'' ← T \ T';
12         E' ← set of edges of paths of S_1;
13         E'' ← E \ E';
14         G'' ← (V, E'');
15         S_2 ← SGA(G'', T'');
16         if ♯S_1 + ♯S_2 > ♯S_best then
17             S_best ← S_1 ∪ S_2;
18             foreach p_i ∈ S_2 do
19                 p is a path of S \ S_1 such that starting point of p ≡ starting
                   point of p_i and terminating point of p ≡ terminating point of p_i;
20                 p ← p_i;
21     else
22         RandomMoves(S);
23 return S_best;
```

### The LS-R algorithm

The idea is to connect recursively as much as possible commodities of $T$ (see Algorithm 20). The core is the recursive method LS-Recursive in Algorithm 21 receives a graph $G$ and a list of commodities $T$ as input and computes a set of edge-disjoint paths connecting commodities of $T$. This paths set is then accumulated in the solution $Sol$ ($Sol$ is a global variable) and all of 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 local search method GreedyLocalSearch. Lines 2-3 update the solution by adding new edge-disjoint paths of $S_i$ found. Lines 3-4 computes the set of connected components 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 a set of commodities $T_i$ that belong to $G_i$, we perform recursively the LS-Recursive method (see lines 6-8).

**Algorithm 20**: 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 $T$

```
1 S_best ← ∅;
2 while termination criterion is not reached do
3 |   Sol ← ∅;
4 |   LS-Recursive(G, T);
5 |   if ♯Sol > ♯S_best then
6 |   |   S_best ← Sol;
```

**Algorithm 21**: 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 ← GreedyLocalSearch(G, T);
2 foreach p ∈ S_i do
3 |   Sol ← Sol ∪ {p};
4 E' ← set of edges of paths of S_i;
5 CC ← set of connected components of the graph (V, E \ E');
6 foreach G_i ∈ CC do
7 |   T_i ← 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);
```

```
1  class EDP_LS_SGA{
2    UndirectedGraph    g;
3    int          K;
4    Vertex[]       si;
5    Vertex[]       ti;

7    PathsEdgeDisjoint     disjoint;
8    NeighborhoodExplorer<LSGraph> ne;
9    Model<LSGraph>    mod;
10   set{LSGraphPath}    b_sol;
11   int        maxCard;
12   float        t_best;
13   ...
14 }
```

Figure 5.13: The data structure of the LS-SGA algorithm

## 5.5.2 The Model

We describe now the implementation of the two above algorithms in `LS(Graph)` framework.

### The LS-SGA Model

The global data structure is given in Figure 5.13. Lines 2-5 represent the input data including the graph `g` and a list of commodities $\langle$`si[i],ti[i]`$\rangle, \forall$ `i=1..K`. Line 7 declares a `Constraint<LSGraph> disjoint` which specifies the edge-disjointness of the set of paths. A `NeighborhoodExplorer<LSGraph> ne` (line 8) will be used for exploring the neighborhood. Lines 10-11 store the best solution found.

The main method is the `ls_sga` method given in Figure 5.14. Line 2 creates a `Solver<LSGraph> ls`, and lines 4-6 initialize randomly a set of paths `vps` connecting commodities. The `Constraint<LSgraph> disjoint` is imposed on the set of paths `vps` (line 8). Line 10 creates a model `mod` including variables `vps` and the constraint `disjoint`. At each step of the loop, we perform a local move (line 17) exploring different neighborhoods (see Figure 5.17) and perform a move which decrease the number of violations of the `disjoint` constraint. If no such move is found, a random move is taken (lines 18-19). Line 21 processes the current paths set `vps` (method `extract_and_improve`) which extracts (see line 2 in Figure 5.15)a set of edge-disjoint paths (based on Algorithm 18 and the implementation is given the `extract` method in Figure 5.16) and applies a simple greedy algorithm SGA (see Algorithm 17) in hope of improving the solution found by the `extract` method (see line 4 in Figure 5.15). Method `updateBest` which is detailed in Figure 5.18 checks and updates the best solution by the set of edge-disjoint paths found `sol`.

The `localmove` method (see Figure 5.17) explores different neighborhoods (lines 4, 8, 12, 16) in order to find a move that decreases the number of violations of the `disjoint` constraint. If such a move is found, we take it and return `true`. Otherwise, the method returns `false`.

```
1   void ls_sga(float maxT){
2     Solver<LSGraph> ls();

4     set{VarPath} vps();
5     forall(i in 1..K)
6       vps.insert(new VarPath(ls,g,si[i],ti[i]));

8     disjoint = new PathsEdgeDisjoint(ls,vps);

10    mod = new Model<LSGraph>(vps,disjoint);
11    ne = new NeighborhoodExplorer<LSGraph>(mod);

13    maxCard = 0;
14    b_sol = new set{LSGraphPath}();

16    while(System.getCPUTime()*0.001 < maxT && maxCard < K){
17      if(!localmove()){
18        forall(i in 1..5)
19          randomMove(vps);
20      }
21      extract_and_improve(vps);
22    }
23  }
```

Figure 5.14: The main method of the LS-SGA algorithm

### The LS-R Model

The main method is the `ls_r` method given in Figure 5.20. Lines 2-5 initialize the set of paths `vps` connecting given commodities. Current solution `sol` and best solution `b_sol` are initialized in lines 7-8. The main method iteratively calls the core method `ls_recursive` (detailed in Figure 5.21) method in line 21 after the initialization of the global variables: graph `g`, current solution `sol`, and the set of commodities `coms` (lines 11-19). These global variables are updated during the call to recursive method `ls_recursive`. Lines 25-29 initialize randomly paths of the set `vps` that do not belong to the current solution `sol`.

The `ls_recursive` is given in Figure 5.21. Line 2 computes a set of edge-disjoint paths `S` by a greedy local search combined with an extraction method (see detail in Figure 5.22). These paths are then added to the current solution `sol` in lines 5-6. All edges of these paths are removed from the graph `g` (lines 8-12) and all commodities connected by these paths are also removed from the set `coms` in lines 13-14. A set of connected components `cc` of the remaining graph is computed in lines 17-18. For each of these connected component `gi` (line 20), we call recursively the `ls_recursive` method (line 26) where `tvps` is the set of paths connecting commodities on the graph `gi` which are initialized randomly (see lines 21-23).

The `greed_ls_search` is given in Figure 5.22 which finds as much as possible edge-disjoint paths of `vps` on the graph `g`. It applies a greedy local search aiming at minimizing the number of violations of the constraint `disjoint` over the `vps` and

```
1   void extract_and_improve(set{VarPath} vps){
2       set{VarPath} S = extract(vps,g);

4       set{VarPath} Si = tryImprove(S,vps);
5       set{VarPath} sol();
6       forall(vp in S){
7         sol.insert(vp);
8       }

10      if(Si != null){
11        forall(vp in Si){
12          sol.insert(vp);
13        }
14      }

16      updateBest(sol);
17  }

19  set{VarPath} tryImprove(set{VarPath} S,set{VarPath} vps){
20    set{LSGraphPath} Si = sga(S,vps);
21    set{VarPath} retS();

23    if(S.getSize() + Si.getSize() > maxCard){
24      forall(pi in Si){
25        forall(vp in vps: !member(vp,S) && vp.getSource() ==
              pi.getSource() && vp.getDestination() ==
              pi.getDestination()){
26          vp.assign(pi);
27          retS.insert(vp);
28        }
29      }
30      return retS;
31    }
32    return null;
33  }
```

Figure 5.15: The `process` method in the LS-SGA algorithm

```
1   set{VarPath} extract(set{VarPath} vps, UndirectedGraph g){
2     int cc[g.getEdgesRange()] = 0;

4     set{VarPath} S();
5     forall(vp in vps)
6       S.insert(vp);

8     forall(vp in S){
9       Edge e = vp.getFirstEdge();
10      while(e != null){
11        cc[e.id()]++;
12        e = vp.getNextEdge();
13      }
14    }

16    bool finished = false;
17    while(!finished){
18      int maxVio = 0;
19      VarPath maxVioPath = null;
20      forall(pp in S){
21        int co = 0;
22        Edge e = pp.getFirstEdge();
23        while(e != null){
24          if(cc[e.id()] > 1)
25            co++;
26          e = pp.getNextEdge();
27        }

29        if(co > maxVio){
30          maxVio = co;
31          maxVioPath = pp;
32        }
33      }
34      if(maxVio == 0){
35        finished = true;
36      }else{
37        S.delete(maxVioPath);

39        Edge e = maxVioPath.getFirstEdge();
40        while(e != null){
41          cc[e.id()]--;
42          e = maxVioPath.getNextEdge();
43        }
44      }
45    }
46    return S;
47  }
```

Figure 5.16: The `extract` method of the LS-SGA algorithm

```
1   bool localmove(){
2     MinNeighborSelector N();

4     VarPath[] vps = mod.getVarPaths();

6     ne.exploreDegradeMultiStageReplace1Move1VarPath(N,vps,disjoint,true);
7     if(N.hasMove()){
8       call(N.getMove());
9     }else{
10      ne.exploreDegradeMultiStageReplace2Moves1VarPath(N,vps,disjoint,true);
11      if(N.hasMove()){
12        call(N.getMove());
13      }else{
14        ne.exploreDegradeMultiStageReplace1Move2VarPaths(N,vps,disjoint,false);
15        if(N.hasMove()){
16          call(N.getMove());
17        }else{
18          ne.exploreDegradeReplace2Moves1VarPath(N,vps,disjoint,true);
19          if(N.hasMove()){
20            call(N.getMove());
21          }else{
22            return false;
23          }
24        }
25      }
26    }
27    return true;
28  }

30  void randomMove(set{VarPath} vps){
31    select(vp in vps){
32      ReplacingEdgesMaintainPath rpl = ne.getReplacingEdges(vp);
33      select(ei in rpl.getSet()){
34        select(eo in getPreferredReplacableEdges(vp,ei)){
35          vp.replaceEdge(eo,ei);
36        }
37      }
38    }
39  }
```

Figure 5.17: The `localmove` and `randomMove` methods of the LS-SGA algorithm

```
1     bool updateBest(set{VarPath} sol){
2       if(maxCard < sol.getSize()){
3         maxCard = sol.getSize();
4         t_best = System.getCPUTime()*0.001;
5         cout << "UPDATE BEST = " << maxCard << " time = " << t_best <<
              endl;

7         b_sol.reset();
8         forall(vp in sol){
9           LSGraphPath p = vp.getLSGraphPath();
10          b_sol.insert(p);
11        }
12        return true;
13      }

15      return false;
16    }
```

Figure 5.18: The updateBest method of the LS-SGA algorithm

```
1  class EDP_LS_R{
2    UndirectedGraph       g0;
3    UndirectedGraph       g;

5    int              K;
6    Vertex[]           si;
7    Vertex[]           ti;
8    set{SourceSink}       coms;

10   set{VarPath}        sol;
11   set{LSGraphPath}        b_sol;
12   ...
13 }
```

Figure 5.19: The data structure of the LS-R algorithm

extract (line 15) the set of edge-disjoint paths follow the Algorithm 18. In this method, the search component `se` in line 11 extends the built-in `GreedyLocalSearch<LSGraph>` overriding the `localmove` method (see Figure 5.23). The `localmove` method explores two different neighborhoods (line 12 and 17) for finding a move which decreases the number of violations of the `disjoint` constraint. It takes that move if it is found and returns `true`. If no such move is discovered, the `localmove` method returns `false`.

### 5.5.3 Experiments

#### Problem instances

We experiment the two proposed algorithms on three kinds of benchmark. The first benchmark is instance on 4 graphs provided by Blesa [BB07]. The second benchmark

```
1   void ls_r(float maxT){
2     set{VarPath} vps();
3     forall(ss in coms){
4       vps.insert(new VarPath(g0,ss.getSource(),ss.getSink()));
5     }

7     sol = new set{VarPath}();
8     b_sol = new set{LSGraphPath}();

10    while(System.getCPUTime()*0.001 < maxT && b_sol.getSize() < K){
11      sol.reset();

13      coms.reset();
14      forall(i in 1..K){
15        SourceSink ss(si[i],ti[i]);
16        coms.insert(ss);
17      }

19      g = g0.copy();

21      ls_recursive(g,vps);
22      updateBest();

24      // generate initial paths for next iteration
25      if(System.getCPUTime()*0.001 < maxT && b_sol.getSize() < K){
26        forall(vp in vps: !member(vp,sol)){
27          vp = new VarPath(g0,vp.getSource(),vp.getDestination());
28        }
29      }
30    }
31  }
```

Figure 5.20: The main method of the LS-R algorithm

```
1   void ls_recursive(UndirectedGraph g, set{VarPath} vps){
2        set{VarPath} S = greedy_ls_search(g,vps);

4        // remove edges covered by S from g and all SourceSinks of S
             from coms
5        forall(vp in S){
6          sol.insert(vp);

8          set{Edge} S1 = vp.getEdges();
9          forall(e in S1)
10           if(g.contains(e)){
11             g.removeEdge(e);
12           }
13         select(ss in coms: vp.getSource() == ss.getSource() &&
               vp.getDestination() == ss.getSink())
14           coms.delete(ss);
15       }

17       BasicGraphAlgorithm algo();
18       set{UndirectedGraph} cc = algo.computeConnectedComponents(g);

20       forall(gi in cc){
21         set{VarPath} tvps();
22         forall(ss in coms: gi.contains(ss.getSource()) &&
               gi.contains(ss.getSink()))
23           tvps.insert( new
                 VarPath(gi,ss.getSource(),ss.getSink()));

25         if(tvps.getSize() > 0)
26           ls_recursive(gi,tvps);
27       }
28  }
```

Figure 5.21: The ls_recursive method of the LS-R algorithm

```
1    set{VarPath} greedy_ls_search(UndirectedGraph g, set{VarPath}
         vps){
2      Solver<LSGraph> ls();
3      forall(vp in vps){
4        ls.post(vp);
5      }

7      PathsEdgeDisjoint disjoint(ls,vps);

9      Model<LSGraph> mod(vps,disjoint);

11     EDPSearch se(mod);

13     se.search();

15     set{VarPath} S = extract(vps,g);

17     return S;
18   }
```

Figure 5.22: The greedy_ls_search method of the LS-R algorithm

```
1  class EDPSearch extends GreedyLocalSearch<LSGraph>{
2    EDPSearch(Model<LSGraph> mod): GreedyLocalSearch<LSGraph>(mod){
3    }


6    void initSolution(){
7      // do nothing, just to use random initiation
8    }

10   bool localmove(){
11     MinNeighborSelector N();

13     exploreDegradeMultiStageReplace1Move1VarPath(N,true);
14     if(N.hasMove()){
15       call(N.getMove());
16       return true;
17     }
18     exploreDegradeMultiStageReplace2Moves1VarPath(N,true);
19     if(N.hasMove()){
20       call(N.getMove());
21       return true;
22     }

24     return false;
25   }
26 }
```

Figure 5.23: The search component of the LS-R algorithm

| Name | $\|V\|$ | $\|E\|$ | Degree avg. |
|---|---|---|---|
| bl-wr2-wht2.10-50.rand | 500 | 1020 | 4.08 |
| bl-wr2-wht2.10-50.sdeg | 500 | 1020 | 4.08 |
| mesh15x15 | 225 | 420 | 3.73 |
| mesh25x25 | 625 | 1200 | 3.84 |
| steinb4.txt | 50 | 100 | 4.00 |
| steinb10.txt | 75 | 150 | 4.00 |
| steinb16.txt | 100 | 200 | 4.00 |
| steinc6.txt | 500 | 1000 | 4.00 |
| steinc11.txt | 500 | 2500 | 10.00 |
| steinc16.txt | 500 | 12500 | 50.00 |
| planar-n50 | 50 | 135 | 5.4 |
| planar-n100 | 100 | 285 | 5.7 |
| planar-n200 | 200 | 583 | 5.83 |
| planar-n500 | 500 | 1477 | 5.91 |

Table 5.9: Description of graphs of the benchmarks

is instances on some graphs of the Steiner benchmark from Or-Library [Bea]. The third benchmark consists of instances on random planar graphs. Table 5.9 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 [BB07] are not available. As a result, we use the instance generator described in [BB07] 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 2 instances[6] with 0.10*$n$, 0.25*$n$ and 0.40*$n$ commodities. We do the same for each graph of the second set but we generate only 1 instance for each rate of commodities instead of 2.

For comparison, we have reimplemented the ACO algorithm described in [BB07] in the COMET programming language. For each problem instance, we execute the three algorithms 20 times. Due to the high complexity of the problem, we set the time limit to 30 minutes for each execution. In total, we have 54 problem instances with 1080 executions.

### Results

Experimental results are shown in Tables 5.10 and 5.11. The structure of these tables are the same and are described as follows. First column presents the instance name. Columns 2-5 present the result of the ACO [BB07] algorithm including the average, the minimal and the maximal of the best objective values found in 20 executions, and the average of time for finding these best objective values. The same information of

[6]This is different from what we did in [PDH10] where we generate randomly 20 instances with each rate of commodities and for each instance, the algorithm is executed only once.

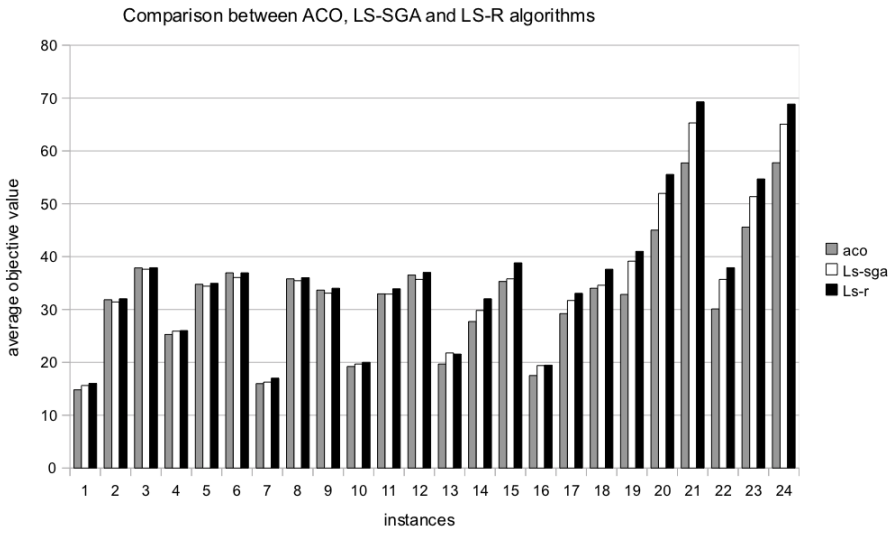



Figure 5.24: Comparison between the ACO, LS-SGA and LS-R algorithms on mess instances

the LS-SGA and LS-R are presented in columns 7-10 and columns 12-15. Column 6 compares the ACO and LS-SGA algorithms under format $a/b$ where $a$ is the number of times the ACO algorithm find better solution than the LS-SGA algorithm and $b$ is the number of time the LS-SGA finds better solution than the ACO algorithm in 20 executions. Column 16 presents the same information as the column 6 but for the comparison between the ACO and the LS-R algorithms. Figures 5.24, 5.25, and 5.26 provide more information about the comparison between the three algorithms in term of average objective values found among 20 executions for each instance.

Experiments results show that in average the LS-R is better than two other algorithms: ACO and LS-SGA algorithms and the LS-SGA is better than the ACO algorithm. The LS-SGA finds better solutions than the ACO algorithm in 534 out of 1080 executions while the ACO algorithm finds better solutions in 96 out of 1080 executions. The LS-R finds better solution than the ACO in 614 out of 1080 executions while the ACO algorithm finds better solution than the LS-R in 7 out of 1080 executions.

## 5.6 The Routing and Wavelength Assignment problem

### 5.6.1 Problem formulation

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 $c : E \rightarrow \mathbb{R}$, $c(e)$

| Instances | ACO | | | | LS-SGA | | | | | LS-R | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $\overline{f}$ | $m$ | $M$ | $\overline{t}$ | $\overline{f}$ | $m$ | $M$ | $\overline{t}$ | $\tau_1$ | $\overline{f}$ | $m$ | $M$ | $\overline{t}$ | $\tau_2$ |
| bl-wr2-wht2.10-50.rand.bb_com10_ins1 | 14.8 | 14 | 16 | 131.6 | 15.6 | 14 | 16 | 410.76 | 2/13 | 16 | 16 | 16 | 194.71 | 0/19 |
| bl-wr2-wht2.10-50.rand.bb_com25_ins1 | 31.85 | 31 | 32 | 165.22 | 31.4 | 30 | 32 | 564.99 | 10/3 | 32 | 32 | 32 | 263.71 | 0/3 |
| bl-wr2-wht2.10-50.rand.bb_com40_ins1 | 37.85 | 37 | 38 | 219.56 | 37.6 | 36 | 38 | 322.96 | 6/2 | 37.9 | 37 | 38 | 230.29 | 1/2 |
| bl-wr2-wht2.10-50.rand.bb_com10_ins2 | 25.25 | 25 | 26 | 95.41 | 25.9 | 25 | 26 | 434.43 | 0/13 | 26 | 26 | 26 | 151.09 | 0/15 |
| bl-wr2-wht2.10-50.rand.bb_com25_ins2 | 34.75 | 34 | 35 | 97.33 | 34.4 | 32 | 35 | 544.02 | 8/4 | 34.95 | 34 | 35 | 303.26 | 0/4 |
| bl-wr2-wht2.10-50.rand.bb_com40_ins2 | 36.95 | 36 | 37 | 185.14 | 36.05 | 34 | 37 | 422.18 | 13/0 | 36.95 | 36 | 37 | 293.27 | 1/1 |
| bl-wr2-wht2.10-50.sdeg.bb_com10_ins1 | 15.95 | 15 | 16 | 89.24 | 16.25 | 16 | 17 | 529.03 | 0/6 | 17 | 17 | 17 | 430.99 | 0/20 |
| bl-wr2-wht2.10-50.sdeg.bb_com25_ins1 | 35.8 | 35 | 36 | 67.08 | 35.45 | 34 | 36 | 536.4 | 8/3 | 36 | 36 | 36 | 423.68 | 0/4 |
| bl-wr2-wht2.10-50.sdeg.bb_com40_ins1 | 33.65 | 33 | 34 | 169.9 | 33.1 | 31 | 34 | 472.56 | 11/4 | 34 | 34 | 34 | 557.57 | 0/7 |
| bl-wr2-wht2.10-50.sdeg.bb_com10_ins2 | 19.2 | 19 | 20 | 401.19 | 19.65 | 19 | 20 | 522.22 | 2/11 | 20 | 20 | 20 | 448.59 | 0/16 |
| bl-wr2-wht2.10-50.sdeg.bb_com25_ins2 | 32.95 | 32 | 34 | 365.09 | 32.9 | 31 | 34 | 880.04 | 9/7 | 33.9 | 33 | 34 | 516.21 | 1/14 |
| bl-wr2-wht2.10-50.sdeg.bb_com40_ins2 | 36.5 | 35 | 37 | 133 | 35.7 | 35 | 37 | 936.57 | 11/2 | 37 | 37 | 37 | 583.99 | 0/9 |
| mesh15x15.bb_com10_ins1 | 19.65 | 19 | 21 | 457.46 | 21.75 | 21 | 22 | 644.11 | 0/20 | 21.55 | 21 | 22 | 360.53 | 0/19 |
| mesh15x15.bb_com25_ins1 | 27.7 | 26 | 29 | 470.98 | 29.8 | 29 | 31 | 335.51 | 0/19 | 32 | 31 | 33 | 887.93 | 0/20 |
| mesh15x15.bb_com40_ins1 | 35.3 | 32 | 38 | 871.22 | 35.8 | 33 | 39 | 763.25 | 6/10 | 38.8 | 37 | 40 | 960.97 | 0/20 |
| mesh15x15.bb_com10_ins2 | 17.5 | 17 | 19 | 479.89 | 19.4 | 19 | 20 | 515.65 | 0/19 | 19.45 | 19 | 20 | 568.74 | 0/18 |
| mesh15x15.bb_com25_ins2 | 29.2 | 28 | 31 | 1010.52 | 31.7 | 30 | 33 | 480.98 | 0/20 | 33.05 | 32 | 34 | 592.96 | 0/20 |
| mesh15x15.bb_com40_ins2 | 34 | 33 | 36 | 750.55 | 34.6 | 33 | 37 | 649.26 | 4/13 | 37.6 | 36 | 39 | 910.63 | 0/20 |
| mesh25x25.bb_com10_ins1 | 32.85 | 29 | 36 | 996.96 | 39.15 | 36 | 41 | 864.6 | 0/20 | 41 | 39 | 43 | 946.47 | 0/20 |
| mesh25x25.bb_com25_ins1 | 45 | 42 | 49 | 1104.82 | 51.95 | 49 | 56 | 1053.57 | 0/20 | 55.55 | 54 | 59 | 1111.8 | 0/20 |
| mesh25x25.bb_com40_ins1 | 57.7 | 53 | 61 | 797.14 | 65.3 | 60 | 69 | 950.87 | 0/20 | 69.3 | 67 | 72 | 1520.61 | 0/20 |
| mesh25x25.bb_com10_ins2 | 30.1 | 28 | 33 | 944.36 | 35.7 | 34 | 37 | 875.12 | 0/20 | 37.9 | 36 | 40 | 945.08 | 0/20 |
| mesh25x25.bb_com25_ins2 | 45.6 | 44 | 48 | 1015.84 | 51.35 | 47 | 54 | 673.59 | 0/20 | 54.7 | 52 | 59 | 1042.11 | 0/20 |
| mesh25x25.bb_com40_ins2 | 57.75 | 54 | 61 | 939.82 | 65.05 | 62 | 68 | 1409.13 | 0/20 | 68.85 | 66 | 71 | 1040.24 | 0/20 |

Table 5.10: Experimental results of the first graphs set

| Instances | ACO | | | | LS-SGA | | | | | LS-R | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $\overline{f}$ | $m$ | $M$ | $\overline{t}$ | $\overline{f}$ | $m$ | $M$ | $\overline{t}$ | $\tau_1$ | $\overline{f}$ | $m$ | $M$ | $\overline{t}$ | $\tau_2$ |
| steinb4.txt_com10_ins1 | 5 | 5 | 5 | 0.01 | 5 | 5 | 5 | 1.12 | 0/0 | 5 | 5 | 5 | 1.09 | 0/0 |
| steinb4.txt_com25_ins1 | 12 | 12 | 12 | 0.44 | 12 | 12 | 12 | 1.22 | 0/0 | 12 | 12 | 12 | 1.4 | 0/0 |
| steinb4.txt_com40_ins1 | 20 | 20 | 20 | 51.11 | 20 | 20 | 20 | 5.45 | 0/0 | 19.9 | 19 | 20 | 2.8 | 2/0 |
| steinb10.txt_com10_ins1 | 7 | 7 | 7 | 0.02 | 7 | 7 | 7 | 1.35 | 0/0 | 7 | 7 | 7 | 1.16 | 0/0 |
| steinb10.txt_com25_ins1 | 17.85 | 17 | 18 | 96.48 | 18 | 18 | 18 | 13.42 | 0/3 | 18 | 18 | 18 | 5.2 | 0/3 |
| steinb10.txt_com40_ins1 | 24.35 | 23 | 26 | 242.04 | 26.25 | 26 | 27 | 682.84 | 0/20 | 27.3 | 27 | 28 | 505.22 | 0/20 |
| steinb16.txt_com10_ins1 | 10 | 10 | 10 | 0.25 | 10 | 10 | 10 | 1.46 | 0/0 | 10 | 10 | 10 | 1.52 | 0/0 |
| steinb16.txt_com25_ins1 | 24.35 | 24 | 25 | 364.99 | 25 | 25 | 25 | 93.53 | 0/13 | 25 | 25 | 25 | 8.86 | 0/13 |
| steinb16.txt_com40_ins1 | 32.45 | 32 | 34 | 658.25 | 34.1 | 33 | 36 | 747.34 | 0/19 | 35.95 | 35 | 37 | 646.19 | 0/20 |
| steinc6.txt_com10_ins1 | 49.1 | 47 | 50 | 572.75 | 50 | 50 | 50 | 184.44 | 0/12 | 50 | 50 | 50 | 240.75 | 0/12 |
| steinc6.txt_com25_ins1 | 89.9 | 85 | 94 | 728.76 | 92.2 | 87 | 100 | 734.88 | 4/12 | 104.95 | 102 | 108 | 1370.88 | 0/20 |
| steinc6.txt_com40_ins1 | 109.8 | 106 | 117 | 924.1 | 112.05 | 106 | 118 | 971.4 | 2/14 | 121.4 | 119 | 125 | 1372.37 | 0/20 |
| steinc11.txt_com10_ins1 | 50 | 50 | 50 | 23.59 | 50 | 50 | 50 | 42.19 | 0/0 | 50 | 50 | 50 | 37.64 | 0/0 |
| steinc11.txt_com25_ins1 | 123.3 | 122 | 125 | 521.8 | 125 | 125 | 125 | 128.49 | 0/17 | 125 | 125 | 125 | 262.38 | 0/17 |
| steinc11.txt_com40_ins1 | 194.25 | 190 | 198 | 494.64 | 200 | 200 | 200 | 395.54 | 0/20 | 200 | 200 | 200 | 473.81 | 0/20 |
| steinc16.txt_com10_ins1 | 50 | 50 | 50 | 6.89 | 50 | 50 | 50 | 55.12 | 0/0 | 50 | 50 | 50 | 46.01 | 0/0 |
| steinc16.txt_com25_ins1 | 125 | 125 | 125 | 17.13 | 125 | 125 | 125 | 194.36 | 0/0 | 125 | 125 | 125 | 113.83 | 0/0 |
| steinc16.txt_com40_ins1 | 200 | 200 | 200 | 45.32 | 200 | 200 | 200 | 366.69 | 0/0 | 200 | 200 | 200 | 183.32 | 0/0 |

Table 5.11: Experimental results of the steiner graphs set

| Instances | ACO | | | | LS-SGA | | | | LS-R | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $\overline{f}$ | $m$ | $M$ | $\overline{t}$ | $\overline{f}$ | $m$ | $M$ | $\overline{t}$ | $\tau_1$ | $\overline{f}$ | $m$ | $M$ | $\overline{t}$ | $\tau_2$ |
| planar-n50.ins1.txt_com10_ins1 | 5 | 5 | 5 | 0.03 | 5 | 5 | 5 | 0.86 | 0/0 | 5 | 5 | 5 | 0.8 | 0/0 |
| planar-n50.ins1.txt_com25_ins1 | 12 | 12 | 12 | 0.16 | 12 | 12 | 12 | 0.96 | 0/0 | 12 | 12 | 12 | 0.97 | 0/0 |
| planar-n50.ins1.txt_com40_ins1 | 20 | 20 | 20 | 36.38 | 20 | 20 | 20 | 25.12 | 0/0 | 19.9 | 19 | 20 | 31.18 | 2/0 |
| planar-n100.ins1.txt_com10_ins1 | 10 | 10 | 10 | 0.12 | 10 | 10 | 10 | 1.14 | 0/0 | 10 | 10 | 10 | 1.07 | 0/0 |
| planar-n100.ins1.txt_com25_ins1 | 25 | 25 | 25 | 20.22 | 25 | 25 | 25 | 7.05 | 0/0 | 25 | 25 | 25 | 5.33 | 0/0 |
| planar-n100.ins1.txt_com40_ins1 | 34 | 33 | 36 | 680.72 | 35.3 | 34 | 37 | 813.56 | 0/16 | 36 | 35 | 37 | 698.88 | 0/18 |
| planar-n200.ins1.txt_com10_ins1 | 20 | 20 | 20 | 13.46 | 20 | 20 | 20 | 4.06 | 0/0 | 20 | 20 | 20 | 5.23 | 0/0 |
| planar-n200.ins1.txt_com25_ins1 | 41.8 | 39 | 43 | 889.07 | 44.85 | 43 | 47 | 988.81 | 0/20 | 45.95 | 45 | 48 | 853.18 | 0/20 |
| planar-n200.ins1.txt_com40_ins1 | 49.35 | 47 | 51 | 790.65 | 53.35 | 51 | 56 | 1033.97 | 0/19 | 55.7 | 54 | 58 | 901.74 | 0/20 |
| planar-n500.ins1.txt_com10_ins1 | 44.95 | 42 | 47 | 1100.41 | 49.95 | 49 | 50 | 484.84 | 0/20 | 50 | 50 | 50 | 309.24 | 0/20 |
| planar-n500.ins1.txt_com25_ins1 | 60.95 | 57 | 65 | 954.35 | 73.85 | 70 | 77 | 1345.74 | 0/20 | 78.2 | 77 | 80 | 1044.03 | 0/20 |
| planar-n500.ins1.txt_com40_ins1 | 82.85 | 78 | 86 | 1235.13 | 93.95 | 91 | 99 | 1366.27 | 0/20 | 100.15 | 97 | 102 | 1455.43 | 0/20 |

Table 5.12: Experimental results of the planar graphs set

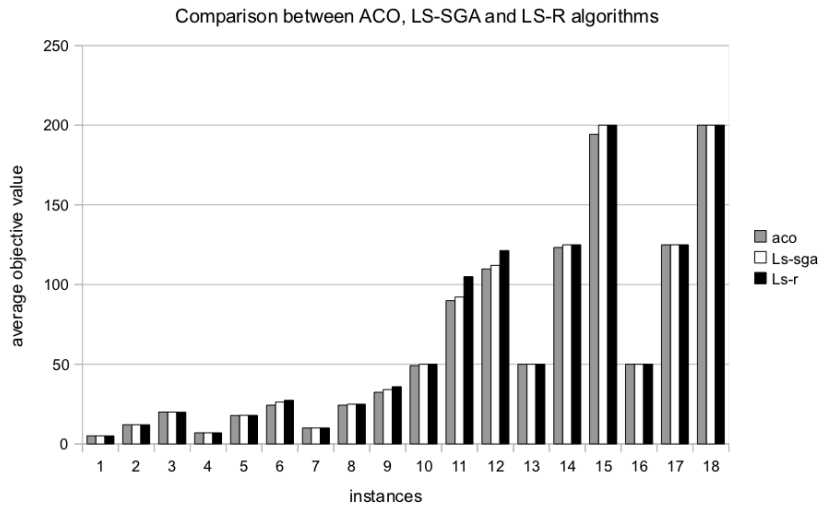



Figure 5.25: Comparison between the ACO, LS-SGA and LS-R algorithms on steiner instances

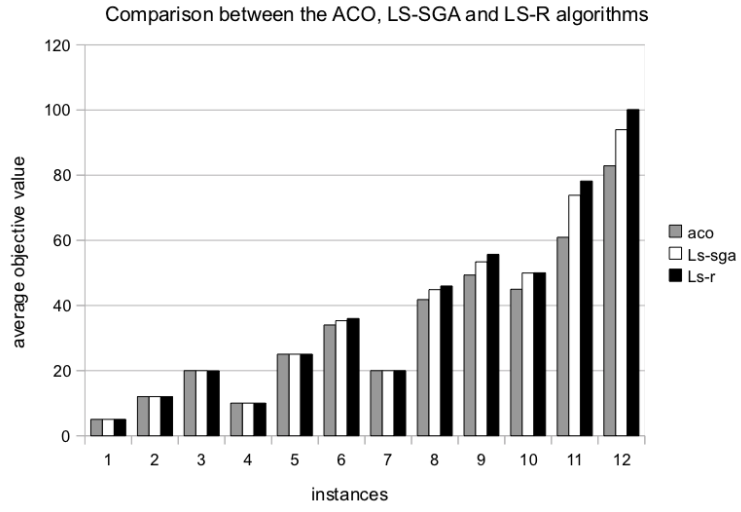



Figure 5.26: Comparison between the ACO, LS-SGA and LS-R algorithms on planar instances

represents the delay in traversing $e$. Given a set of connection requests $R \subseteq V \times V$, $R = \{\langle s_1, t_1\rangle, \langle s_2, t_2\rangle, ..., \langle s_k, t_k\rangle\}$ and a value $D$. The RWA-D problem consists in finding routes $p_i$ from $s_i$ to $t_i$ and its wavelength 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 wavelength is minimized (objective function).

### 5.6.2 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.

The model is given in Figure 5.27. 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 Figure 5.28 receives an array of `VarPath vps`, a value `W` of the number of wavelengths, and local search parameters `maxIt`, `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 14-15 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` and line 20 creates a search component which is detailed in Figure 5.29. Lines 22-24 set parameters for the search and line 26 performs the search. The value of `xw` is stored in `vw` for the next iteration (see lines 28-29): all paths `vps[i]` and its wavelength `xw[i]` are conserved for the next `localsearch` . The `localsearch` returns `true` if a feasible solution to the RWA-D problem is discovered (lines 31-33).

The search component is given in Figure 5.29 which 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 cost is greater than `Lmax`. This aims at quickly satisfying the delay constraint. The `initSolution` is also overriden which do nothing in order not to change the value of variables initialized in the previous step of the search. The search explores two neighborhoods (lines 7-10) (see appendix for the detail about these neighborhood explorations).

```
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;

13    bool finished = false;
14    int W = 1;
15    while(!finished){
16      if(localsearch(vps,W,maxIt,maxT)){
17        finished = true;
18      }else{
19        W++;
20      }
21    }
22    cout << "best objective value = " << W << endl;
23  }
```

Figure 5.27: Model for the RWA-D problem

### 5.6.3 Experiments

The model has been experimented on different instances (graphs from 16 nodes - 33 edges to 100 nodes - 261 edges and 10, 20, 50 connections requests for each graph) with different number of iterations (20, 50, 100, 200, 500) for the tabu search (the value of `maxIt` in line 16 of Figure 5.27). For each problem instance and a given number of iterations, the model is executed 20 times. We have conducted a preliminary experiments in order to choose the tabu search parameters. For each specified number of iterations, we try the parameters as follows: `tbMin` = $n/p\_tbMin$, `tbMax` = $n/p\_tbMax$, `tinc` = `(tbMax-tbMin)/`$p\_tinc$`+1`, and `maxStable` = $p\_maxStable$ in which $n$ is the number of vertices of the given graph with (see also Table 5.13):

- $p\_tbMin \in \{3, 5\}$,
- $p\_tbMax \in \{2\}$,
- $p\_tinc \in \{1, 2, 4\}$,
- $p\_maxStable \in \{50, 200\}$.

The preliminary results show that the influence of the parameters on solutions is quite divers (for some instances, the parameters do not influence on the solutions found) and no parameters are good for all instances. Figures 5.30 and 5.31 depict the influence of the parameters on the solutions for some instances in case of 200

```
1   bool localsearch(VarPath[] vps, int W, int maxIt, float maxT){
2     Solver<LSGraph>  ls();

4     forall(i in vps.rng()){
5       ls.post(vps[i]);
6     }

8     xw = new var{int}[i in vps.rng()](ls,1..W) := vw[i];

10    ConstraintSystem<LSGraph> CS(ls);

12    CS.post(AllDistinctLightPaths(vps,xw));

14    forall(i in vps.rng())
15      CS.post(PathCostOnEdges(vps[i]) <= Lmax,1000);

17    CS.close();

19    Model<LSGraph> mod(vps,xw,CS);
20    RWASearch se(mod,Lmax);

22    se.setMaxIter(maxIt);
23    se.setMaxTime(maxT);
24    se.setParameters(tbMin, tbMax, tinc, maxStable);

26    se.search();

28    forall(i in xw.rng())
29      vw[i] = xw[i];

31    if(CS.violations() == 0){
32      return true;
33    }
34    return false;
35  }
```

Figure 5.28: The local search procedure for the RWA-D problem

```
1  class RWASearch extends TabuSearch<LSGraph>{
2    float  _Lmax;
3    RWASearch(Model<LSGraph> mod, float Lmax):
         TabuSearch<LSGraph>(mod){
4      _Lmax = Lmax;
5    }

7    void exploreNeighborhood(Neighborhood N){
8      exploreTabuMinMultiStageAssign(N,true);
9      exploreTabuMinMultiStageReplace1Move1VarPath(N,true);
10   }


13   void restartSolution(){
14     // init paths with shortest versions for paths whose current
           cost greater than Lmax
15     forall(k in _vps.rng()){
16       VarPath vp = _vps[k];
17       float d = sum(e in vp.getEdges())(e.weight());
18       if(d > _Lmax){
19         LSGraphPath pa(vp.getLUB());
20         pa.dijkstra(vp.getSource(),vp.getDestination());
21         vp.assign(pa);
22       }
23     }
24   }
25   void initSolution(){
26   }
27 }
```

Figure 5.29: The search component

| index | $p\_tbMin$ | $p\_tbMax$ | $p\_tinc$ | $p\_maxStable$ |
|---|---|---|---|---|
| 1 | 3 | 2 | 1 | 50 |
| 2 | 3 | 2 | 1 | 200 |
| 3 | 3 | 2 | 2 | 50 |
| 4 | 3 | 2 | 2 | 200 |
| 5 | 3 | 2 | 4 | 50 |
| 6 | 3 | 2 | 4 | 200 |
| 7 | 5 | 2 | 1 | 50 |
| 8 | 5 | 2 | 1 | 200 |
| 9 | 5 | 2 | 2 | 50 |
| 10 | 5 | 2 | 2 | 200 |
| 11 | 5 | 2 | 4 | 50 |
| 12 | 5 | 2 | 4 | 200 |

Table 5.13: Parameters tried for the RWA-D problem

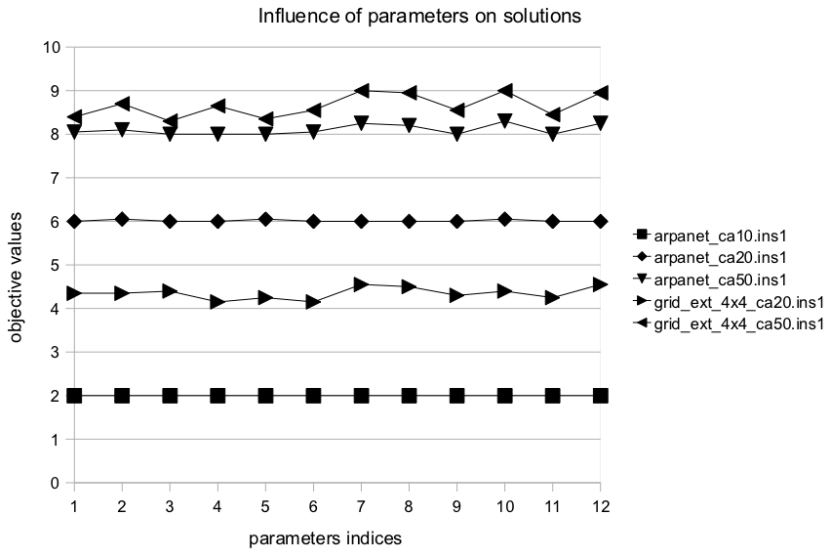



Figure 5.30: Influence of parameters on solutions

| ♯iterations | $p\_tbMin$ | $p\_tbMax$ | $p\_tinc$ | $p\_maxStable$ |
|---|---|---|---|---|
| 20 | 3 | 2 | 1 | 200 |
| 50 | 5 | 2 | 1 | 200 |
| 100 | 3 | 2 | 1 | 50 |
| 200 | 3 | 2 | 1 | 50 |
| 500 | 3 | 2 | 2 | 50 |

Table 5.14: Tabu search parameters selection

iterations. The X-axis presents the indices of the parameters (see Table 5.13) and the Y-axis presents the average objective values of 20 executions for each instance.

Selected values for $p\_tbMin$, $p\_tbMax$, $p\_tinc$, $p\_maxStable$ depend on the number of iterations of the local search procedure (see Figure 5.14) and are presented in Table 5.13.

Table 5.15 show the experimental results. For each number of iterations, the table shows the average of the best objective value and the execution time (in seconds) of 20 runs. Experimental results show that in general, when the number of iterations increases, the solutions founds are slightly better except some instances (see lines 2, 4, 10) but the execution times are much higher.

One 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 show the flexibility and compositionality of the framework.

| instance | 20 iters | | 50 iters | | 100 iters | | 200 iters | | 500 iters | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $\overline{f}$ | $\overline{t}$ | $\overline{f}$ | $\overline{t}$ | $\overline{f}$ | $\overline{t}$ | $\overline{f}$ | $\overline{t}$ | $\overline{f}$ | $\overline{t}$ |
| arpanet_ca10.ins1 | 2 | 1.12 | 2 | 1.31 | 2 | 1.62 | 2 | 2.3 | 2 | 4.12 |
| arpanet_ca20.ins1 | 6.2 | 3.18 | 6.05 | 4.85 | 6 | 7.86 | 6 | 13.4 | 6 | 29.53 |
| arpanet_ca50.ins1 | 9.25 | 12.37 | 8.75 | 23.25 | 8.25 | 39.44 | 8.05 | 73.01 | 8 | 165.51 |
| grid_ext_4x4_ca10.ins1 | 2 | 1.05 | 2 | 1.19 | 2 | 1.4 | 2 | 1.84 | 2 | 2.92 |
| grid_ext_4x4_ca20.ins1 | 4.6 | 2.16 | 4.5 | 3.16 | 4.4 | 4.62 | 4.35 | 7.82 | 4.1 | 16.26 |
| grid_ext_4x4_ca50.ins1 | 9.95 | 9.96 | 9.25 | 18.21 | 8.8 | 31.53 | 8.4 | 58.7 | 7.75 | 125.78 |
| grid_ext_5x5_ca10.ins1 | 2.5 | 1.27 | 2.3 | 1.46 | 2.2 | 1.88 | 2.05 | 2.61 | 2 | 4.49 |
| grid_ext_5x5_ca20.ins1 | 3.1 | 2 | 2.9 | 2.81 | 2.9 | 4.32 | 2.9 | 7.32 | 2.8 | 15.01 |
| grid_ext_5x5_ca50.ins1 | 8.65 | 11.55 | 7.35 | 19.95 | 6.7 | 32.82 | 6.45 | 58.72 | 5.85 | 127.69 |
| grid_ext_6x6_ca10.ins1 | 2.05 | 1.27 | 2 | 1.47 | 2.05 | 1.98 | 2 | 2.82 | 2 | 4.74 |
| grid_ext_6x6_ca20.ins1 | 3.05 | 2.38 | 2.7 | 3.3 | 2.45 | 4.71 | 2.2 | 7.58 | 2.3 | 15.79 |
| grid_ext_6x6_ca50.ins1 | 7.4 | 13.14 | 6.3 | 21.98 | 5.75 | 35.49 | 5.5 | 64.81 | 5.4 | 151.36 |
| grid_ext_8x8_ca10.ins1 | 2.4 | 1.82 | 2.25 | 2.24 | 2.1 | 2.95 | 2.1 | 4.47 | 2 | 8.72 |
| grid_ext_8x8_ca20.ins1 | 3.85 | 4.33 | 3.75 | 6.49 | 3.4 | 9.28 | 3.55 | 17 | 3.35 | 36.2 |
| grid_ext_8x8_ca50.ins1 | 6.45 | 17.52 | 5.7 | 27.97 | 5.5 | 44.11 | 5.1 | 76.96 | 4.8 | 168.78 |
| grid_ext_10x10_ca10.ins1 | 3.45 | 3.34 | 3.45 | 4.78 | 3.25 | 6.97 | 2.8 | 10.22 | 2.6 | 18.8 |
| grid_ext_10x10_ca20.ins1 | 4.95 | 7.77 | 4.75 | 11.15 | 4.5 | 16.38 | 4.3 | 27.24 | 4 | 53.36 |
| grid_ext_10x10_ca50.ins1 | 7.35 | 31.13 | 6.1 | 44.68 | 5.45 | 66.78 | 5 | 109.63 | 4.6 | 238.44 |

Table 5.15: Experimental results of the RWA-D problem over 20, 50, 100, 200, 500 iterations

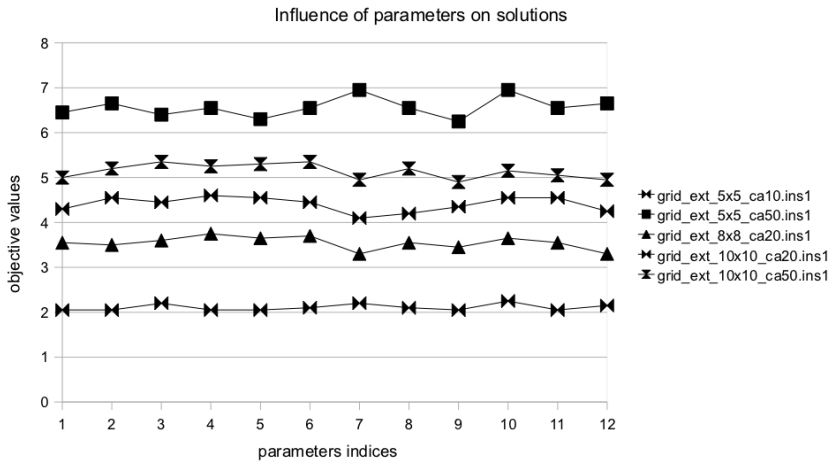



Figure 5.31: Influence of parameters on solutions

## 5.7 The Routing for Network Covering problem

### 5.7.1 Problem formulation

Given an undirected weighted graph $G = (V, E)$, a vertex $d \in V$ representing the depot, a set $S$ of edges of $G$ ($S \subseteq E$) and a value $L$, the RNC problems consists of finding a minimal cardinality set of paths starting from and terminating at $d$ whose cost are less than or equal to $L$ that covers all edges of $S$.

### 5.7.2 The Model

To our best knowledge, this problem has not been considered earlier. We propose in this section a model based on local search for solving the RNC problem. In the presentation of the model, `g` is the given graph, `S` is the set of edges that need to be covered and `L` is the maximum value allowed of the cost of each itinerary in the solution.

The model is a greedy constructive search which is depicted in Figure 5.32. At each step, we find greedily a feasible itinerary `vi` that covers as many edges of `S` (`S` is reduced after each iteration (see lines 8-9)) as possible (see method `greedy` at line 5 of Figure 5.32) until `S` is empty. `sol` (line 2) stores the solution which is updated when a new itinerary is discovered (line 6).

The method `greedy` is detailed in Figure 5.33. Line 2 creates a `Solver<LSGraph> ls` which manages all graph variables, graph invariants, graph constraints and graph functions of the model. Line 3 declares an object `vi` representing *VarItinerary* which is composed by a sequence of `k` *VarPath*s starting and terminating at depot `d`, rooted at

```
1   void rncModel(set{Edge} S, float L, int nbTrials, float maxT){
2     set{VarItinerary} sol();

4     while(S.getSize() > 0){
5       VarItinerary vi = greedy(S,L,nbTrials,maxT);
6       sol.insert(vi);

8       forall(e in vi.getEdges()){
9         S.delete(e);
10      }
11    }
12    cout << "Best objective value = " << sol.getSize() << endl;
13    cout << "time = " << System.getCPUTime()*0.001 << endl;
14  }
```

Figure 5.32: Model for the RNC problem

random vertices except the last one. Line 5 initializes a graph function `cost` representing the cost of `vi` and line 7 initializes a graph function `visitedEdges` representing the number of edges of `S` visited by `vi`. Line 9 creates a model with a decision variable `vi`, a constraint `cost <= L` to be satisfied and an objective function `visitedEdges` to be maximized. A search object which applies to the model `mod` and maximizes `visitedEdges` is initialized in line 11 (see Figure 5.34 for detail of the implementation). Tabu search parameters are set in lines 13-15 where the maximum number of iterations `nbTrials` and time window `maxT` are set in lines 14 and 15. Line 17 performs the tabu search.

The search (Figure 5.34) extends the built-in `TabuSearch<LSGraph>` overriding the `initSolution` (lines 12-45) and the `exploreNeighborhood` (lines 47-50) methods. The initial solution is an itinerary that visits at least a random selected edge `e=(u,v)` of the set of edges to be covered `_S` (lines 16-18) formed by 3 paths in a greedy way: the first path `p1` is the shortest path from the depot `_depot` to an endpoint `u` of `e` (lines 19-20); the second path contains only the edge `e` which is the path `u->v` (line 22); the third path is the shortest path from another endpoint `v` of `e` to the depot `_depot` (lines 24-25). `low..up` (lines 27-28) is the range of `VarPath` of `vi`. The first three `VarPaths vp1`, `vp2`, `vp3` are respectively assigned to `p1`, `p2` and `p3` (lines 30-36). The remaining `VarPaths` of `vi` contains only one vertex `_depot` (lines 38-43). In the `exploreNeighborhood` (line 47-50), we exploit two neighborhoods: $ERNP_1$ (line 48) and $SDCNI_1$ (line 49).

### 5.7.3 Experiments

We experimented the above model on some grid graphs of size 4x4, 5x5, 10x10, 15x15, 20x20, 25x25 and 30x30 vertices. For each graph $g$, the depot is chosen randomly. The weights of edges are generated randomly with respect to a uniform distribution between 1 and 10. The set of edges to be covered is generated randomly and its cardinality is $r * \sharp E(g)$ where $r \in \{0.5, 1\}$. The value of $L$ is $1.5 * L_0$ where

```
1   VarItinerary greedy(set{Edge} S, float L, int nbTrials, float maxT){
2     Solver<LSGraph> ls();
3     VarItinerary vi(ls,g,d,d,k);

5     Function<LSGraph> cost = new ItineraryCost(ls,vi,g);

7     Function<LSGraph> visitedEdges = new
          VisitedEdgesItinerary(ls,vi,S,g);

9     Model<LSGraph> mod(vi,cost <= L, visitedEdges, MAXIMIZATION);

11    RNCSearch se(mod,S,d);

13    se.setParameters(tbMin, tbMax, tinc, maxStable);
14    se.setMaxIter(nbTrials);
15    se.setMaxTime(maxT);

17    se.search();

19    return vi;
20  }
```

Figure 5.33: the `greedy` method of the model for the RNC problem

$L_0$ is the cost of the shortest path from the depot that visits the farthest edge and returns the depot. This ensures that feasible solutions to this problem always exist. We generate 2 instances for each graph and each value of $r$. In total, there are 28 problem instances.

An exact branch-and-bound method has been implemented. The objective here is to test whether or not the local search model can find optimal solutions in small instances.

**Parameters** The parameters for the model are {`k, alpha, beta, tbMin, tbMax, tinc, maxStable`} where `k` is the number of RST of each `VarItinerary`; `alpha` and `beta` are used as coefficients for combining constraints and the objective function in the model; `tbMin, tbMax, tinc, maxStable` are parameters for the generic search procedure (see Figure 5.33). Normally, the parameters depends on the size of the input. If `k` is high, the model is heavy which influence the performance. From our experiments, we choose the values for parameters as follows. The value of `k` is set to 3. The value of `alpha` should be higher than `beta` in order to prioritize the search towards feasible solutions. We thus set `beta` = 1, `alpha` = 1000. For the remaining parameters, we set the values of `tbMin = n/10, tbMax = n/3, tinc = n/4, maxStable = n/5` where `n` is the size of the given graph.

Each model is executed 20 times for each instance with a specified number of iterations `nbTrials` in line 14 of Figure 5.33). Due to the huge complexity of the problem, each local search procedure is performed with 20, 50, 100, 200, 500, 1000 iterations in order to analyze the evolution of the best objective value over iterations.

```
1  class RNCSearch extends TabuSearch<LSGraph>{
2    set{Edge} _S;
3    Vertex  _depot;

5    RNCSearch(Model<LSGraph> mod, set{Edge} S, Vertex
         depot):TabuSearch<LSGraph>(mod){
6      _S = S;
7      _depot = depot;
8    }

10   void restartSolution(){
11   }
12   void initSolution(){
13        VarItinerary vi = getFirstItinerary();
14        UndirectedGraph g = vi.getLUB();

16        select(e in _S){
17          Vertex u = e.fromVertex();
18          Vertex v = e.toVertex();
19          LSGraphPath p1(g);
20          p1.dijkstra(_depot,u);

22          LSGraphPath p2(g,e);

24          LSGraphPath p3(g);
25          p3.dijkstra(v,_depot);

27          int low = vi.rng().getLow();
28          int up = vi.rng().getUp();

30          VarPath vp1 = vi.get(low);
31          VarPath vp2 = vi.get(low+1);
32          VarPath vp3 = vi.get(low+2);

34          vp1.assign(p1);
35          vp2.assign(p2);
36          vp3.assign(p3);

38          forall(j in low+3..up){
39            VarPath vp = vi.get(j);
40            LSGraphPath pi(g);
41            pi = pi + _depot;
42            vp.assign(pi);
43          }
44        }
45   }

47   void exploreNeighborhood(Neighborhood N){
48     exploreTabuMinReplace1Move1VarPath(N,true);
49     exploreTabuMinChangeDestinationVarItinerary(N,true);
50   }
```

Figure 5.34: Search component for the RNC problem

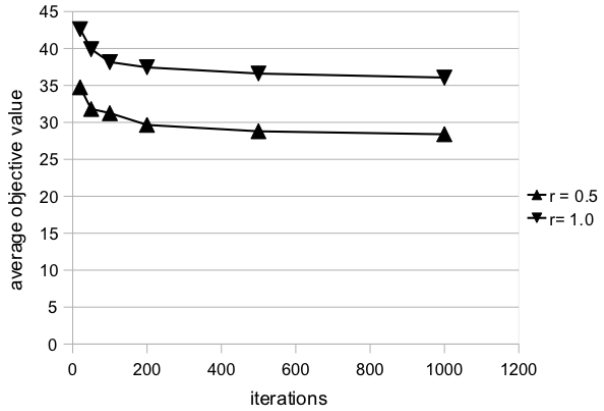



Figure 5.35: Grid 25x25: average objective value over iterations

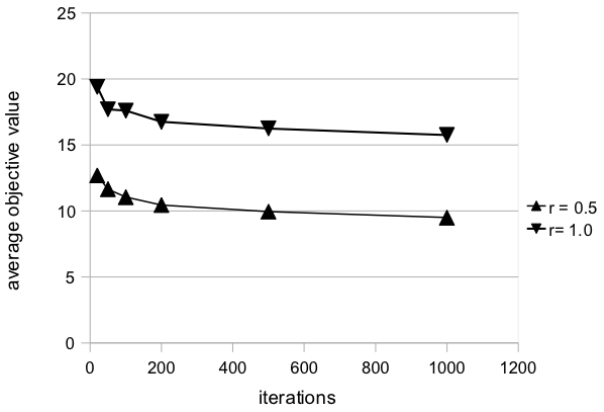



Figure 5.36: Grid 15x15: average objective value over iterations

In total, we have 33600 executions.

**Results** The results are shown in Table 5.16 with different number of iterations. Columns 2 and 3 present the depot vertex and the factor $r$ used for instances generation described above. Columns 4-15 present the min, max, average value of the best objective value found and the average execution time in the 20 runs of each instance with 20, 100, 200 iterations. Column 16 presents the optimal objective value found by the Branch-and-Bound method within 30 minutes for instances on grid_4x4 and grid_5x5. The Tables show that the diversity of the best objective values found in the 20 runs for each problem instance increases when the size of the input graph and the number of edges to be covered increase. The execution time increases when the number of edges to be covered increases. Figures 5.36, 5.35 show the evolution of the average best objective value over iterations. They show that the best objective values found reduce substantially in first 200 iterations and are quite stable from 200 to 1000 iterations.

Due to the huge complexity of the problem, because edges and vertices are allowed to be repeated on paths, the exact method can not find solution on instances with grids of size larger than 6x6 within 20 hours. The Table shows that the local search model can find optimal solutions on some small instances (for the grid_4x4 and grid_5x5).

| Graph | $d$ | $r$ | 20 iters | | | | 100 iters | | | | 200 iters | | | | B & B |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | $m$ | $M$ | $\overline{f}$ | $\overline{t}$(sec.) | $m$ | $M$ | $\overline{f}$ | $\overline{t}$(sec.) | $m$ | $M$ | $\overline{f}$ | $\overline{t}$(sec.) | $f^*$ |
| grid_4x4 | 4 | 0.5 | 2 | 3 | 2.05 | 1.51 | 2 | 2 | 2 | 1.86 | 2 | 2 | 2 | 2.2 | 2 |
| | 2 | 0.5 | 2 | 3 | 2.05 | 1.54 | 2 | 2 | 2 | 1.84 | 2 | 2 | 2 | 2.25 | 2 |
| | 1 | 1 | 3 | 3 | 3 | 1.78 | 3 | 3 | 3 | 2.51 | 3 | 3 | 3 | 3.32 | 3 |
| | 10 | 1 | 3 | 4 | 3.4 | 1.91 | 3 | 4 | 3.05 | 2.47 | 3 | 3 | 3 | 3.23 | 3 |
| grid_5x5 | 24 | 0.5 | 2 | 3 | 2.95 | 1.93 | 2 | 3 | 2.7 | 2.69 | 2 | 3 | 2.55 | 3.71 | 2 |
| | 18 | 0.5 | 3 | 5 | 3.7 | 2.14 | 3 | 4 | 3.2 | 3.05 | 3 | 4 | 3.1 | 4.36 | 3 |
| grid_10x10 | 14 | 0.5 | 7 | 10 | 8.6 | 8.93 | 7 | 9 | 8 | 26.19 | 7 | 9 | 7.7 | 47.49 | - |
| | 16 | 0.5 | 7 | 9 | 7.75 | 8.15 | 6 | 7 | 6.6 | 22.5 | 6 | 7 | 6.7 | 42.55 | - |
| | 54 | 1 | 14 | 17 | 15.4 | 15.13 | 13 | 16 | 14.25 | 48.26 | 13 | 15 | 13.65 | 84.93 | - |
| | 69 | 1 | 11 | 14 | 12.35 | 12.66 | 11 | 12 | 11.35 | 41.61 | 10 | 12 | 10.9 | 73.7 | - |
| grid_15x15 | 61 | 0.5 | 11 | 14 | 12.7 | 31.15 | 10 | 12 | 11.05 | 100.22 | 9 | 11 | 10.45 | 182.77 | - |
| | 218 | 0.5 | 12 | 16 | 14.25 | 35.5 | 11 | 14 | 12.3 | 106.89 | 11 | 13 | 12.15 | 201.59 | - |
| | 169 | 1 | 18 | 21 | 19.4 | 49.18 | 16 | 19 | 17.6 | 168.06 | 16 | 18 | 16.75 | 310.11 | - |
| | 52 | 1 | 20 | 23 | 21.4 | 53.42 | 18 | 20 | 18.85 | 175.64 | 18 | 20 | 18.85 | 332.02 | - |
| grid_20x20 | 207 | 0.5 | 23 | 26 | 23.65 | 127.51 | 19 | 22 | 20.5 | 365.96 | 19 | 20 | 19.65 | 649.45 | - |
| | 309 | 0.5 | 20 | 23 | 21.35 | 116.33 | 18 | 20 | 18.55 | 341.52 | 17 | 19 | 18.1 | 623.27 | - |
| | 19 | 1 | 22 | 25 | 23.9 | 141.15 | 19 | 22 | 20.9 | 474.56 | 18 | 21 | 20.15 | 871.45 | - |
| | 241 | 1 | 26 | 29 | 27.55 | 155.56 | 23 | 25 | 24.45 | 508.23 | 23 | 26 | 24.1 | 955.71 | - |
| grid_25x25 | 381 | 0.5 | 33 | 37 | 34.75 | 365.83 | 30 | 33 | 31.25 | 997.67 | 28 | 31 | 29.65 | 1727.41 | - |
| | 420 | 0.5 | 27 | 31 | 29.25 | 314.45 | 25 | 27 | 26.25 | 892.06 | 24 | 26 | 25 | 1545.11 | - |
| | 418 | 1 | 40 | 44 | 42.6 | 473.79 | 36 | 40 | 38.15 | 1383.21 | 36 | 39 | 37.45 | 2528.95 | - |
| | 440 | 1 | 41 | 46 | 43 | 472.14 | 38 | 41 | 38.65 | 1392.56 | 37 | 39 | 37.55 | 2522.91 | - |
| grid_30x30 | 66 | 0.5 | 32 | 38 | 33.55 | 657.68 | 27 | 30 | 28.85 | 1713.57 | 27 | 31 | 28.1 | 3081.24 | - |
| | 130 | 0.5 | 36 | 40 | 38.35 | 744.43 | 31 | 36 | 33 | 1881.9 | 30 | 33 | 31.85 | 3307.93 | - |
| | 305 | 1 | 51 | 55 | 53.1 | 1063.87 | 46 | 50 | 47.75 | 2929.8 | 45 | 48 | 46 | 5235.79 | - |
| | 247 | 1 | 50 | 52 | 50.8 | 1011.71 | 43 | 48 | 45.2 | 2854.93 | 42 | 47 | 43.85 | 5075.64 | - |

Table 5.16: Experimental results of the RNC problem with 20, 100, 200 iterations

# 6

# CONCLUSION

The objective of this thesis comes from the observation that various constrained optimum trees and paths applications on graphs have been approached by specific, and sophisticated techniques which are difficult to extend. Hence, when a new problem needs to be solved, it requires huge efforts of research and programming. Section 6.1 summarizes the results of this thesis and Section 6.2 gives some directions for future research.

## 6.1 Results

In this thesis, we designed and implemented the `LS(Graph)` framework which allows to model in a high-level language and to solve constrained optimum trees and paths problems on graphs by local search. As an extension of COMET , `LS(Graph)` (the current version is about 25,000 lines of COMET code) strengthens the modeling benefits of CBLS: compositionality, modularity, reuse. Users do not have to pay attention to the manipulation of sophisticated data structures and algorithms on graphs. Rather, they can concentrate on stating the model and exploring different local search strategies. The `LS(Graph)` framework is open; it allows users to design and implement their own invariants, constraints and objective functions and integrate them into the system. The framework supports graphs with several weights on edges, vertices. It does not depend on whether these weights are negative or positive, although positive weights constraints are required for most of the specific algorithms. The `LS(Graph)` library as well as some data sets experimented in this thesis are available at *http://becool.info.ucl.ac.be/lsgraph*.

As presented in Section 3.3, one of the main technical contributions of this thesis is the use of rooted spanning tree for representing an elementary path and its neighborhood in COP applications: a rooted spanning tree $tr$ induces a unique path $p$ from a specified vertex $s$ (called the source of $tr$) to its root. An update over this tree (i.e., by an edge replacement) creates a neighboring rooted spanning tree of $tr$ which induces

a neighboring path of $p$. To the best of our knowledge, there exist few applications of local search approaches for solving constrained paths problems on general graphs except those on complete graphs (on complete graphs, the path can explicitly be represented by a sequence of vertices and any change (e.g., remove, insert or replace vertices) over these vertices induces a new path). In some applications, sophisticated and specific which are problem-dependent moves are applied. The benefit of this representation is twofold. On the one hand, the size of neighborhood is polynomial (e.g., in comparison with the most general neighborhood $\mathcal{N}$), it can thus be explored efficiently. On the other hand, it features diversity (i.e., neighboring paths of a path are usually very different from the current path) which is fundamental in local search.

As a result, we also proposed a simple way to represent a walk (i.e., paths where vertices and edges can be repeated) by a sequence of rooted spanning tree: the root of a tree is the source of the subsequent tree in the sequence. A limitation of this representation is the need to predefine the number of rooted spanning trees in the sequence. This parameter depends on particular applications and can only be evaluated experimentally.

In term of implementation, dedicated data structures and incremental algorithms have been exploited. The query of the nearest common ancestor of two vertices on a tree is fundamental for implementing various abstractions of the framework. State-of-the-art data structures and algorithms for maintaining nearest common ancestors of all pairs of two vertices on dynamic tree have been used. In the framework, we have implemented this data structure and algorithm in an incremental way. By maintaining the distance between all pairs of two vertices on dynamic trees, the cost of a path and its variation under different moves can be queried in $\mathcal{O}(1)$. As shown in Section 3.6, these algorithms cannot improve the worst case complexity but are efficient in practice.

Finally, the constructed framework has been applied to three constrained optimum trees problems (the edge-weighted $k$-cardinality tree problem, the quorumcast routing problem and the design of spanning tree protocol problem-application on traffic engineering in switched ethernet networks) and four constrained optimum paths problems (the resource constrained shortest path problem with both constraints over minimum and maximum resources consumed along the path, the edge-disjoint paths problem, the routing and wavelength assignment with delay side constraints and the routing for network covering problem). The application on the traffic engineering in switched ethernet networks is tackled in collaboration with Ho in [HFD+10], and is not presented in this thesis. For many applications, it is difficult to obtain competitive results in comparison with dedicated algorithms. Rather, we show how to solve them flexibly and shortly especially when side constraints need to be added. This is one of the main objective of this thesis. However, for the quorumcast routing problem, our proposed tabu search gives better results than the IMP heuristic algorithms which is in our best knowledge, the state-of-the-art heuristic for this problem. For the edge-disjoint paths problem, we proposed two algorithms based on local search which give competitive results in comparison with the state-of-the-art ACO algorithm.

## 6.2 Future work

Our future work can be specified in four main directions:

1. **Improving `LS(Graph)`**. Some abstractions still need to be improved. We give three examples. First, the diameter-bound constraint on tree stating that the diameter of the tree $tr$ cannot exceed $D$ is now modeled by the constraint: $Diamter(tr) \leq D$ or by the global constraint *DiameterAtMost(tr,D)*. In the first choice, the function $Diameter(tr)$ has been efficiently implemented but this expression does not allow to differentiate trees because several trees may have the same value of diameter. In the second choice, we define the number of violations of the *DiameterAtMost(tr,D)* constraint by $\sum_{u,v \in V(tr)} max(0, d_{tr}(u,v) - D)$ where $d_{tr}(u,v)$ is the distance between $u$ and $v$ on $tr$. This definition is better than the previous one but its complexity makes it difficult to have an efficient implementation. As a second example, in the current implementation of the constraint *PathContains*$(vp, S)$ specifying that the path $vp$ must visit a set of specified vertices or edges $S$, the time complexity for querying the variation of the number of its violations is proportional to the length of the subpath to be removed and the new path to be added[1]. By maintaining an auxiliary data structure, this query could be performed in $\mathcal{O}(1)$. Finally, another important feature of the framework that needs to be developed is the visualization which animates the local search behavior. This can help to adapt local search strategy in order to improve it.

2. **Applications**. In term of applications, many interesting applications can be tackled by applying the `LS(Graph)` framework.

   - We will first explore other local search models for solving the routing for network covering problem and experiment over extensive data set including real urban networks.
   - Second, we are interested in two other constrained walks finding problems like Capacitated Arc Routing problem. Clearly, these problems can be modeled and solved with `LS(Graph)`. Existing techniques for these problems were experimented on graphs up to 255 vertices which are not too large. We would like to implement local search algorithms in `LS(Graph)` for solving them, and compare with existing techniques on standard and larger benchmarks.
   - Third, we are interested in the Finding Two Disjoint Paths in a Network with Normalized $\alpha^+$-MIN-SUM Objective Function problem [YZL05] which arises in the reliable telecommunication networks. To our best knowledge, there are few works for this problem. In [YZL05], the complexity has been analyzed and an approximation algorithm has been proposed for solving this problem but no implementation and experimental results have been given.

[1] The move for a *VarPath* is the replacement of a subpath of the current path by a new path.

- Finally, we are interested in the edge-disjoint spanning trees problem on graphs [BJGG07]. This problem can be easily modeled and solved by using the `LS(Graph)`. In the paper [BJGG07], a local search has been proposed making use of the neighborhood $NT_3$ of `LS(Graph)`. We intend to, by using the `LS(Graph)` framework, re-implement this local search algorithm, try a tabu search algorithm and compare them.

3. **Hybrid platform**. In many cases, solving optimally and efficiently constrained trees and paths problems on small instances is very important. Dooms in his doctoral research [DDD05] introduces a `CP(Graph)` computation domain in Constraint Programming. We could extend this work by constructing a generic solver which allows to model and solve these two specific classes of problem (COT and COP) by Constraint Programming. Global constraints on trees and paths will be designed and implemented aiming at pruning vertices, edges of the given graph that do not belong to optimal solutions. Another objective is an hybrid platform which combines `LS(Graph)` with this exact solver. We could concentrate on two hybrid directions. The first one is for constraint optimization problem which uses local search as main routine. Whenever the local search finds a new best solution, the constraint stating that the objective function must be better than the best new objective value will be posted, pruning techniques will then be applied for reducing problem size. This approach has been applied in the RCSP application presented in Section 5.4. The second approach is the Large Neighborhood Search (LNS) [Sha98]. For traditional CSP in which the problem is modeled by a set of scalar decision variables, the LNS at each step applies the Constraint Programming technique for reassigning optimally a subset of decision variables. For COT/COP problems, the LNS idea is to use local search as main routine and apply Constraint Programming technique for finding appropriate moves: at each step, we remove a subtree or subpath of the current tree, path and find (by Constraint Programming) another appropriate subtree, subpath for replacing the removed subtree, subpath.

4. **Specific modeling languages**. Monette in his doctoral research [MDVH09] proposes a generic solver that allows to model in a flexible way and solve scheduling problems with different side constraints. The problem is modeled from high-level way. A synthesizer will analyze the problem structure and generate appropriate search algorithms for finding solutions. We have seen two important problems and their variants with various side constraints in industries: Steiner tree problem and routing problem. One research direction is inspired from the approach of [MDVH09] for constructing a generic solver for these problems and their variants. The power of such a solver is, on the one hand, the ability of solving classical well-known problems with state-of-the-art search algorithms and, on the other hand, the flexibility of modeling and solving the problem with different side constraints. The challenge from this approach is how to design a rich modeling language, implement abstractions (constraints, functions) supporting this language and how to recognize well-known problems from the modeling.

# A

# MODELING API OF LS(GRAPH)

This appendix gives main modeling API of the `LS(Graph)` framework. It presents only the public methods of each class (methods which are available for users). Sections A.4 and A.5 depict the graph functions and graph constraints of `LS(Graph)`. These classes implement the common interfaces `Function<LSGraph>` and `Constraint<LSGraph>`. All methods appearing in these interfaces are thus not presented. Instead, we present only the constructor with different parameters for these classes.

## A.1 Solver<LS(Graph)>

- `Solver<LSGraph>()`
- `void post(VarGraph vg)`
- `void post(Invariant<LSGraph> inv)`
- `close()`

## A.2 Variables

### A.2.1 VarTree

- `VarTree(Solver<LSGraph> ls, UndirectedGraph g)`
- `VarTree(Solver<LSGraph> ls, UndirectedGraph g, int k)`: the tree is randomly initialized with `k` edges
- `VarTree(UndirectedGraph g)`
- `Solver<LSGraph> getLSGraphSolver()`

- `bool contains(Vertex v)`: return true if the tree contains the vertex `v`
- `bool contains(Edge e)`: return true if the tree contains the edge `e`
- `set{Vertex} getVertices()`: return the set of vertices of the tree
- `set{Edge} getEdges()`: return the set of edges of the tree
- `void clear()`: clear the tree, all vertices and edges are removed
- `void addEdge(Edge e)`: add the edge `e` to the tree
- `void removeEdge(Edge e)`: remove the edge `e` from the tree
- `void replaceEdge(Edge eo, Edge ei)`: replace the *replacable* edge `eo` by the *replacing* edge `ei`
- `UndirectedGraph getLUB()`: return the undirected graph over which the tree is specified.

### A.2.2 VarRootedTree extends VarTree

- `VarRootedTree(Solver<LSGraph> ls, UndirectedGraph g, Vertex root)`
- `VarRootedTree(UndirectedGraph g, Vertex root)`
- `Vertex root()`: return the root of the tree
- `nca(Vertex u, Vertex v)`: return the nearest common ancestor of two vertices `u` and `v`
- `getFatherVertex(Vertex v)`: return the father vertex of `v` on the tree
- `getFatherEdge(Vertex v)`: return the father edge of `v` on the tree (the edge connecting `v` and its father vertex)

### A.2.3 VarRootedSpanningTree extends VarRootedTree

- `VarRootedSpanningTree(Solver<LSGraph> ls, UndirectedGraph g, Vertex root)`
- `VarRootedSpanningTree(Solver<LSGraph> ls, GenericGraph g, LSGraphPath p)`: the rooted spanning tree is randomly generated inducing the path `p`

### A.2.4 VarPath

- `VarPath(Solver<LSGraph> ls, GenericGraph g, Vertex s, Vertex t)`: initialize randomly a path from `s` to `t` on the graph (directed or undirected) `g`
- `VarPath(Solver<LSGraph> ls, GenericGraph g, LSGraphPath p)`: initialize the `VarPath` with `p`
- `VarPath(GenericGraph g, Vertex s, Vertex t)`
- `VarPath(GenericGraph g, LSGraphPath p)`
- `Solver<LSGraph> getLSGraphSolver()`
- `set{Edge} getEdges()`: return the set of edges of the path
- `set{Vertex} getVertices()`: return the set of vertices of the path
- `Vertex getSource()`
- `Vertex getDestination()`
- `PATH_TYPE getType()`: return the type (DIRECTED or UNDIRECTED) of the path. This depends on the type of the graph over which the path is specified.
- `UndirectedGraph getLUB()`: if the graph over which the path is specified is directed, then the method returns its corresponding undirected graph, otherwise, the method returns this graph
- `DirectedGraph getDLUB()`: if the graph over which the path is specified is directed, then the method returns this graph, otherwise, the method returns null
- `bool isNull()`: return true if the path has no vertices
- `Edge getEdgeFrom(Vertex v)`: return the edge of the path starting from `v`
- `Vertex getNextVertex(Vertex v)`: return the next vertex of `v` on the path
- `replaceEdge(Edge eo, Edge ei`: replace the *preferred replacable* edge `eo` by the *preferred replacing* edge `ei` on the corresponding rooted spanning tree
- `changeSource(Vertex newSrc)`: change the source of the path with new the source `newSrc`
- `changeDestination(Vertex newDes)`: change the destination of the the path with the new destination `newDes`
- `assign(LSGraphPath p)`: assign the path `p` to the current `VarPath`. This is done by generating randomly a rooted spanning tree inducing `p`

### A.2.5 VarItinerary

- `VarItinerary(Solver<LSGraph> ls, GenericGraph g, Vertex s, Vertex t, int k)`: initialize randomly the itinerary from `s` to `t` with `k` rooted spanning trees
- `VarItinerary(GenericGraph g, Vertex s, Vertex t, int k)`
- `VarItinerary(Solver<LSGraph> ls, GenericGraph g, LSGraphPath p, int k)`: initialize the itinerary with the path `p` having `k` rooted spanning trees
- `VarItinerary(GenericGraph g, LSGraphPath p, int k)`
- `Solver<LSGraph> getLSGraphSolver()`
- `set{Edge} getEdges()`: return the set of edges
- `set{Vertex} getVertices()`: return the set of vertices
- `Vertex getSource()`: return the source
- `Vertex getDestination()`: return the destination
- `PATH_TYPE getType()`: return the type of the itinerary
- `UndirectedGraph getLUB()`: if the graph over which the path is specified is directed, then the method returns its corresponding undirected graph, otherwise, the method returns this graph
- `DirectedGraph getDLUB()`: if the graph over which the path is specified is directed, then the method returns this graph, otherwise, the method returns null
- `bool isNull()`: return true if the path has no vertices
- `Edge getEdgeFrom(Vertex v)`: return the edge of the path starting from `v`
- `Vertex getNextVertex(Vertex v)`: return the next vertex of `v` on the path
- `VarPath[] getVarPaths()`: return the array of `VarPath` constituting the itinerary
- `range rng()`: return the range of the array of `VarPath`
- `VarPath get(int i)`: return the `VarPath` at the `i` position in the array
- `replaceEdge(VarPath vp, Edge eo, Edge ei`: replace the *preferred replacable* edge `eo` by the *preferred replacing* edge `ei` on the corresponding rooted spanning tree of `vp`
- `changeSource(Vertex newSrc)`: change the source of the path with new the source `newSrc`
- `changeDestination(Vertex newDes)`: change the destination of the the path with the new destination `newDes`

- `changeDestination(VarPath vp, Vertex newDes)`: change the destination of `VarPath vp` of the itinerary and change the source of the successive `VarPath` of `vp` on the itinerary.
- `assign(LSGraphPath p)`: assign the path `p` to the itinerary.

## A.3 Invariants

### A.3.1 InsertableEdgesVarTree

- `InsertableEdgesVarTree(VarTree vt)`
- `set{Edge} getSet()`

### A.3.2 RemovableEdgesVarTree

- `RemovableEdgesVarTree(VarTree vt)`
- `set{Edge} getSet()`

### A.3.3 ReplacingEdgesVarTree

- `ReplacingEdgesVarTree(VarTree vt)`
- `set{Edge} getSet()`

### A.3.4 NodeDistancesInvr

- `NodeDistancesInvr(VarTree vt, GenericGraph g, int[] inds)`
- `NodeDistancesInvr(VarTree vt, GenericGraph g)`
- `NodeDistancesInvr(VarTree vt, GenericGraph g, int ind)`
- `float getDistance(Vertex u, Vertex v)`
- `float getDistance(Vertex u, Vertex v, int ind)`

### A.3.5 ReplacingEdgesMaintainPath

- `ReplacingEdgesMaintainPath(VarPath vp)`
- `set{Edge} getSet()`

### A.3.6 IndexedPathVisitEdges

- `IndexedPathVisitEdges(VarPath[] vps, var{int}[] xw, var{int}[,] v)`

## A.4 Functions

### A.4.1 WeightTree

- `WeightTree(VarTree vt, int ind)`
- `WeightTree(VarTree vt)`

### A.4.2 LongestPath

- `LongestPath(VarTree vt, int ind)`
- `LongestPath(VarTree vt)`

### A.4.3 PathCostOnEdges

- `PathCostOnEdges(VarPath vp)`
- `PathCostOnEdges(VarPath vp, int ind)`

### A.4.4 NBVisitedEdgesPath

- `NBVisitedEdgesPath(VarPath vp, set{Edge} S)`
- `NBVisitedEdgesPath(VarPath[] vps, set{Edge} S)`

### A.4.5 NBRVisitsEdgePath

- `NBRVisitsEdgePath(VarPath vp, set{Edge} S)`
- `NBRVisitsEdgePath(VarPath[] vps, set{Edge} S)`

### A.4.6 NBRVisitsNodePath

- `NBRVisitsNodePath(VarPath vp, set{Vertex} S)`
- `NBRVisitsNodePath(VarPath[] vps, set{Vertex} S)`

### A.4.7 NBVisitedVerticesTree

- `NBVisitedVerticesTree(VarTree vt, set{Vertex} S)`
- `NBVisitedVerticesTree(VarTree[] vt, set{Vertex} S)`

### A.4.8 FunctionCombinator<LSGraph>

- `FunctionCombinator<LSGraph>(Solver<LSGraph> ls)`
- `void add(Function<LSGraph> f)`
- `void add(Function<LSGraph> f, float w)`
- `void add(Constraint<LSGraph> c)`
- `void add(Constraint<LSGraph> c, float w)`
- `void close()`

## A.5 Constraints

### A.5.1 DiameterAtMost

- `DiameterAtMost(VarTree vt, int D)`
- `DiameterAtMost(VarTree vt)`

### A.5.2 DegreeAtMost

- `DegreeAtMost(VarTree vt, int D)`

### A.5.3 PathsEdgeDisjoint

- `PathsEdgeDisjoint(VarPath[] vps)`

### A.5.4 PathsContainEdges

- `PathsContainEdges(VarPath[] vps, set{Edge} S)`
- `PathsContainEdges(VarPath vp, set{Edge} S)`

### A.5.5 PathsContainVertices

- `PathsContainVertices(VarPath[] vps, set{Vertex} S)`
- `PathsContainVertices(VarPath vp, set{Vertex} S)`

### A.5.6 AllDistinctLightPaths

- `AllDistinctLightPaths(VarPath[] vps, var{int}[] xw)`

### A.5.7 ConstraintSystem<LSGraph>

- `ConstraintSystem<LSGraph>(Solver<LSGraph> ls)`
- `void post(Constraint<LSGraph> c)`
- `void post(Constraint<LSGraph> c, float w)`
- `void close()`

## A.6 Model<LSGraph>

This is an abstraction representing a model which encapsulates variables, constraints, functions

- `Model<LSGraph>(VarTree vt, TREE_TYPE treetype, Constraint<LSGraph> c)`
- `Model<LSGraph>(VarTree vt, TREE_TYPE treetype, Constraint<LSGraph> c, Function<LSGraph> f, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarTree vt, TREE_TYPE treetype, Constraint<LSGraph> c, Function<LSGraph> f, float alpha, float beta, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarTree vt, TREE_TYPE treetype, Function<LSGraph> f, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarTree vt, TREE_TYPE treetype, Function<LSGraph> f, float alpha, float beta, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarTree[] vts, TREE_TYPE treetype, Constraint<LSGraph> c)`
- `Model<LSGraph>(VarTree[] vts, TREE_TYPE treetype, Constraint<LSGraph> c, Function<LSGraph> f, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarTree[] vts, TREE_TYPE treetype, Constraint<LSGraph> c, Function<LSGraph> f, float alpha, float beta, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarTree[] vts, TREE_TYPE treetype, Function<LSGraph> f, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarTree[] vts, TREE_TYPE treetype, Function<LSGraph> f, float alpha, float beta, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarPath vp, Constraint<LSGraph> c)`
- `Model<LSGraph>(VarPath vp, Constraint<LSGraph> c, Function<LSGraph> f, MODEL_TYPE modtype)`

- `Model<LSGraph>(VarPath vp, Constraint<LSGraph> c, Function<LSGraph> f, float alpha, float beta, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarPath vp, Function<LSGraph> f, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarPath vp, Function<LSGraph> f, float alpha, float beta, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarPath[] vps, Constraint<LSGraph> c)`
- `Model<LSGraph>(VarPath[] vps, Constraint<LSGraph> c, Function<LSGraph> f, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarPath[] vps, Constraint<LSGraph> c, Function<LSGraph> f, float alpha, float beta, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarPath[] vps, Function<LSGraph> f, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarPath[] vps, Function<LSGraph> f, float alpha, float beta, MODEL_TYPE modtype)`
- `Model<LSGraph>(var{int} x, VarPath vp, Constraint<LSGraph> c)`
- `Model<LSGraph>(var{int} x, VarPath vp, Constraint<LSGraph> c, Function<LSGraph> f, MODEL_TYPE modtype)`
- `Model<LSGraph>(var{int} x, VarPath vp, Constraint<LSGraph> c, Function<LSGraph> f, float alpha, float beta, MODEL_TYPE modtype)`
- `Model<LSGraph>(var{int} x, VarPath vp, Function<LSGraph> f, MODEL_TYPE modtype)`
- `Model<LSGraph>(var{int} x, VarPath vp, Function<LSGraph> f, float alpha, float beta, MODEL_TYPE modtype)`
- `Model<LSGraph>(var{int}[] x, VarPath[] vps, Constraint<LSGraph> c)`
- `Model<LSGraph>(var{int}[] x, VarPath[] vps, Constraint<LSGraph> c, Function<LSGraph> f, MODEL_TYPE modtype)`
- `Model<LSGraph>(var{int}[] x, VarPath[] vps, Constraint<LSGraph> c, Function<LSGraph> f, float alpha, float beta, MODEL_TYPE modtype)`
- `Model<LSGraph>(var{int}[] x, VarPath[] vps, Function<LSGraph> f, MODEL_TYPE modtype)`
- `Model<LSGraph>(var{int}[] x, VarPath[] vps, Function<LSGraph> f, float alpha, float beta, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarItinerary vi, Constraint<LSGraph> c)`

- `Model<LSGraph>(VarItinerary vi, Constraint<LSGraph> c, Function<LSGraph> f, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarItinerary vi, Constraint<LSGraph> c, Function<LSGraph> f, float alpha, float beta, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarItinerary vi, Function<LSGraph> f, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarItinerary vi, Function<LSGraph> f, float alpha, float beta, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarItinerary[] vis, Constraint<LSGraph> c)`
- `Model<LSGraph>(VarItinerary[] vis, Constraint<LSGraph> c, Function<LSGraph> f, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarItinerary[] vis, Constraint<LSGraph> c, Function<LSGraph> f, float alpha, float beta, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarItinerary[] vis, Function<LSGraph> f, MODEL_TYPE modtype)`
- `Model<LSGraph>(VarItinerary[] vis, Function<LSGraph> f, float alpha, float beta, MODEL_TYPE modtype)`
- `var{int}[] getVarInts()`: return all `var{int}` of the model
- `VarTree[] getVarTrees()`: return all `VarTree` of the model
- `VarPath[] getVarPaths()`: return all `VarPath` of the model
- `VarItinerary[] getVarItineraries()`: return all `VarItinerary` of the model
- `Constraint<LSGraph> getConstraint()`: return the constraint to be satisfied of the model
- `Function<LSGraph> getFunction()`: return the function to be optimized of the model
- `Function<LSGraph> getGlobalFunction()`: return the function which combines the constraint to be satisfied and the function to be optimized with the specified parameters `alpha, beta`
- `MODEL_TYPE getType()`
- `Solver<LSGraph> getLSGraphSolver()`

## A.7 NeighborhoodExplorer<LSGraph>

- `NeighborhoodExplorer<LSGraph>(Model<LSGraph> mod)`
- `void exploreTabuMinMultiStageAssign(Neighborhood N, varint[] x, GTabuInt tbi, Constraint<LSGraph> c, int it, float fgb, bool firstImprovement)`
- `void exploreTabuMinAdd1VarTree(Neighborhood N, VarTree[] vts, Function<LSGraph> f, GTabuEdge[] tbIn, GTabuEdge[] tbOut, int it, float fgb, bool firstImprovement)`
- `void exploreTabuMinRemove1VarTree(Neighborhood N, VarTree[] vts, Function<LSGraph> f, GTabuEdge[] tbIn, GTabuEdge[] tbOut, int it, float fgb, bool firstImprovement)`
- `void exploreTabuMinAddRemove1VarTree(Neighborhood N, VarTree[] vts, Function<LSGraph> f, GTabuEdge[] tbIn, GTabuEdge[] tbOut, int it, float fgb, bool firstImprovement)`
- `void exploreTabuMinReplace1VarTree(Neighborhood N, VarTree[] vts, Function<LSGraph> f, GTabuEdge[] tbIn, GTabuEdge[] tbOut, int it, float fgb, bool firstImprovement)`
- `void exploreTabuMinMultiStageReplace1Move1VarPath(Neighborhood N, VarPath[] vps, Constraint<LSGraph> c, GTabuEdge[] tbIn, GTabuEdge[] tbOut, int it, float fgb, bool firstImprovement)`
- `void exploreTabuMin1ReplaceXVY1VarPath(Neighborhood N, VarPath[] vps, Function<LSGraph> f, GTabuVertex[] tbVIn, int it, float fgb, bool firstImprovement)`
- `void exploreTabuMin1ReplaceXVY1VarPath(Neighborhood N, VarPath[] vps, Constraint<LSGraph> f, GTabuVertex[] tbVIn, int it, float fgb, bool firstImprovement)`
- `void exploreTabuMinReplace2MovesVarPath(Neighborhood N, VarPath[] vps, Function<LSGraph> f, GTabu[] tbIn, GTabu[] tbOut, int it, float fgb, bool firstImprovement)`
- `void exploreTabuMinReplace1Move1VarPath(Neighborhood N, VarPath[] vps, Function<LSGraph> f, GTabuEdge[] tbIn, GTabuEdge[] tbOut, int it, float fgb, bool firstImprovement)`
- `void exploreTabuMinChangeDestinationVarItinerary(Neighborhood N, VarItinerary[] vis, Function<LSGraph> f, GTabuVertex[] tbVIn, int it, float fgb, bool firstImprovement)`
- `void exploreDegradeMultiStageReplace1Move1VarPath(Neighborhood N, VarPath[] vps, Constraint<LSGraph> c, bool firstImprovement)`
- `void exploreDegradeMultiStageReplace2Moves1VarPath(Neighborhood N, VarPath[] vps, Constraint<LSGraph> c, bool firstImprovement)`

- `void exploreDegradeMultiStageReplace1Move2VarPaths(Neighborhood N, VarPath[] vps, Constraint<LSGraph> c, bool firstImprovement)`
- `void exploreDegradeReplace2Moves1VarPath(Neighborhood N, VarPath[] vps, Constraint<LSGraph> c, bool firstImprovement)`

## A.8 TabuSearch<LSGraph>

- `TabuSearch<LSGraph>(Model<LSGraph> mod)`: the tabu search receive a model (`Model<LSGraph>`) as parameter.
- `void setMaxIter(int maxIter)`: set the maximum number of iterations for the search
- `void setMaxTime(float maxTime)`: set the time window for the search
- `void setParameters(int tbMin, int tbMax, int tinc, int maxStable)`: set the parameters of the tabu search.
- `stopSearch()`: stop the search
- `void search()`: perform the search
- `int getCurrentIter()`: return the current iteration of the search
- `float getCurrentTime()`: return the current time of the search
- `float getFGB()`: return the best value found so far of the global function (returned by the method `getGlobalFunction` of the `Model<LSGraph>` class)
- `float getFCur()`: return the current value of the global function (returned by the method `getGlobalFunction` of the `Model<LSGraph>` class)
- `Model<LSGraph> getModel()`: return the model
- `Solver<LSGraph> getLSGraphSolver()`: return the solver
- `NeighborhoodExplorer<LSGraph> getNeighborhoodExplorer()`: return the neighborhood explorer

## A.9 Arithmetic Operators

- `operator GenericGraph + (GenericGraph g1, GenericGraph g2)`
- `operator GenericGraph + (GenericGraph g1, Vertex v)`
- `operator GenericGraph + (GenericGraph g1, Edge e)`
- `operator LSGraphPath + (LSGraphPath p1, LSGraphPath p2)`

- operator LSGraphPath + (LSGraphPath p1, Vertex v)
- operator LSGraphPath + (LSGraphPath p1, Edge e)
- operator Function<LSGraph> + (Function<LSGraph> obj1, Function<LSGraph> obj2)
- operator Function<LSGraph> + (Function<LSGraph> obj1, Constraint<LSGraph> obj2)
- operator Function<LSGraph> + (Constraint<LSGraph> obj1, Function<LSGraph> obj2)
- operator Function<LSGraph> + (Function<LSGraph> obj1, float c)
- operator Function<LSGraph> + (float c, Function<LSGraph> obj1)
- operator Function<LSGraph> * (Function<LSGraph> obj1, Function<LSGraph> obj2)
- operator Function<LSGraph> * (Function<LSGraph> obj1, float c)
- operator Function<LSGraph> * (float c, Function<LSGraph> obj1)
- operator Function<LSGraph> - (Function<LSGraph> obj1, Function<LSGraph> obj2)
- operator Function<LSGraph> - (Function<LSGraph> obj1, float c)
- operator Function<LSGraph> - (float c, Function<LSGraph> obj2)
- operator Function<LSGraph> * (Constraint<LSGraph> f, float c)
- operator Function<LSGraph> * (float c, Constraint<LSGraph> f)
- operator Function<LSGraph> + (Constraint<LSGraph> f, float c)
- operator Function<LSGraph> + (float c, Constraint<LSGraph> f)
- operator Function<LSGraph> - (float c, Constraint<LSGraph> f)

## A.10 Logical Operators

- operator Constraint<LSGraph> == (Function<LSGraph> go1, Function<LSGraph> go2)
- operator Constraint<LSGraph> == (Function<LSGraph> go1, float c)
- operator Constraint<LSGraph> == (float c, Function<LSGraph> go1)
- operator Constraint<LSGraph> <= (Function<LSGraph> go1, Function<LSGraph> go2)

- `operator Constraint<LSGraph> <= (Function<LSGraph> go1, float c)`
- `operator Constraint<LSGraph> <= (float c, Function<LSGraph> go1)`
- `operator GraphConstraintPrune<LSGraph> <= (GraphFunctionPrune<LSGraph> go1, float c)`
- `operator GraphConstraintPrune<LSGraph> <= (float c, GraphFunctionPrune<LSGraph> go1)`

# B

# GENERIC TABU SEARCH

This appendix describes the generic adaptive tabu search with restart schema of the `LS(Graph)` framework (see Figure B.3). The length of tabu lists `tbl` varies from `tbMin` and `tbMax` with an increasing step `tinc`.

The search preparation consists of the `initSearch` method (line 2) which set initial values for variables (e.g., the best objective value), the `initSolution` method (line 5) which computes the initial solution to the given problem and the `resetTabu` which initializes the tabu lists, set the `tbl` to `tbMin`.

Variable `nic` counts the number of successive local moves which do not find better solutions than the best-restart solution (i.e., the best solution found from restart). If the best restart solution cannot be improved in `maxStable` successive local moves (condition in line 13) then the tabu length `tbl` is increased by `tinc` (lines 18-19) or the tabu search is restarted (line 15). The core of the procedure is the `localmove` method (see line 22 which is detailed in Figure B.1) which explores specified neighborhoods and perform a local move. If a local move is taken: the `localmove` method return true then we check and update the best solution (line 26). Otherwise, the tabu search is restarted (line 28).

Figure B.2 describes the generic restart procedure of this tabu search which consists of setting `tbl` to `tbMin` (line 2), resetting tabu lists (line 3), generating new initial solutions (line 4). Variables `fCur` and `frb` (line 5) represent the current and best-restart value of the function which controls the search.

In this generic tabu search schema, `tbMin, tbMax, maxStable, tinc` are search parameters. Normally, users can override the `initSolution` and `restartSolution` methods depending on the problem under consideration. The built-in `restartSolution` does nothing and the `initSolution` method generates solutions (i.e., trees, paths) randomly.

```
1   bool localmove(){
2     MinNeighborSelector N();

4     exploreNeighborhood(N);
5     if(N.hasMove()){
6       call(N.getMove());
7       return true;
8     }

10    return false;
11  }
```

Figure B.1: Generic local move

```
1   void performRestart(){
2     tbl = tbMin;
3     resetTabu();
4     restartSolution();
5     frb = fCur;
6     update();
7     nic = 1;
8   }
```

Figure B.2: Generic restart procedure

```
1   void search(){
2     initSearch();
3     update();

5     initSolution();
6     update();

8     resetTabu();

10    it = 1;
11    _finished = false;
12    while(it < maxIt && System.getCPUTime()*0.001 < maxT &&
          !_finished && !checkOptimum()){
13      if(nic % maxStable == 0){
14        if(tbl + tinc > tbMax){
15          performRestart();
16        }else{
17          tbl = tbl + tinc;
18          updateTabuLists();
19        }
20      }

22      bool ok = localmove();

24      if(ok){
25        update();
26      }else{
27        performRestart();
28      }

30      notify evtLocalMove();

32      it++;
33    }
34    restoreBest();
35  }
```

Figure B.3: Generic adaptive tabu search schema

# C

# NEIGHBORHOOD EXPLORATION

This Appendix describes major generic neighborhood explorations in `LS(Graph)` which are applied to above applications in Chapter 5. Users can implement their own neighborhood exploration methods. Different neighborhoods are constituted by updating over `var{int}`, `VarPath`, `VarTree` and `VarItinerary`. A neighbor is considered as an improvement if it reduces the value of a `Function<LSGraph> f` or the number of violations of a `Constraint<LSGraph> c`. The parameter `firstImprovement` determines whether we desire to find the first neighbor who improves the given `Function<LSGraph>` or `Constraint<LSGraph>`. If `firstImprovement` is `true`, the exploration stops when it finds an improving neighbor. Otherwise, the exploration find the best neighbor. The parameter `Neighborhood N` is a COMET abstraction which stores different moves (as closure) and their evaluations. Sections C.1, C.2, C.3, C.4, C.5, C.6, C.8, C.8 , C.10, C.9, C.11 are neighborhood explorations for a tabu search with aspiration criterion. Only neighbors which are not tabu or they are better than the best solution found so far in term of the `Function<LSGraph> f` or the `Constraint<LSGraph> c`. In the parameters list of each method, apart from variables (`var{int}[]`, `VarPath[]`, `VarTree[]` and `VarItinerary[]`) to be scanned and tabu lists, it is required to provide the current iteration of local search `it` and the best value `fgb` of the `Function<LSGraph> f` or the number of violations of the `Constraint<LSGraph> c`. Sections C.12, C.13, C.14, C.15 explore neighborhoods and accept only improving neighbors. In these procedures the parameter `it` and `fgb` are not required.

## C.1 exploreTabuMinMultiStageAssign

The following code is a method that explore neighborhood constituted by reassigning a `var{int}` applying multistage heuristic.

```
1    void exploreTabuMinMultiStageAssign(Neighborhood N, var{int}[] x,
         GTabuInt tbi, Constraint<LSGraph> c, int it, float fgb, bool
         firstImprovement){
```

```
3       c.computeVariableViolations();

5       float eval = System.getMAXINT();
6       var{int} sel_var = null;
7       int sel_val = -1;


10      selectMax(i in x.rng())(c.violations(x[i])){
11        forall(v in x[i].getDomain()){
12          float d = c.getAssignDelta(x[i],v);
13          if(!tbi.isTabu(x[i],v,it) || d + c.violations() < fgb){
14            if(eval > d){
15              eval = d;
16              sel_var = x[i];
17              sel_val = v;

19              if(firstImprovement) if(eval < 0) break;
20            }
21          }
22        }
23      }

25      if(sel_var != null){
26        neighbor(eval,N){
27          sel_var := sel_val;
28          tbi.makeTabu(sel_var,sel_val,it);
29        }
30      }
31    }
```

Line 3 computes the number of violations of each variable. Variables `eval`, `sel_var` and `sel_val` in lines 5, 6, 7 represent the best evaluation, the best variable, value for the assignment found during the exploration. The neighborhood is explored in lines 10-23. Line 10 select the most violating variable `x[i]`, line 11 scans all its values `v`. Line 12 computes the variation `d` of the number of violations of `c` when `x[i]` is assigned by `v`. Line 13 checks tabu condition or the aspiration criterion is reached. Lines 14-20 check and store the best assignment found.

If the desired neighbor is found, it will be submitted to the `Neighborhood N` (lines 25-30). The move (lines 27-28) consists of assigning `sel_val` to `sel_var` and making the move tabu.

## C.2 exploreTabuMinAdd1VarTree

This method explores the neighborhood generated by replacing a tree $tr$ by one of its neighbors in $NT_1(tr)$.

```
1     void exploreTabuMinAdd1VarTree(Neighborhood N, VarTree[] vts,
          Function<LSGraph> f, GTabuEdge[] tbIn, GTabuEdge[] tbOut, int
          it, float fgb, bool firstImprovement){
2       float eval = System.getMAXINT();
3       Edge sel_ei = null;
```

```
4       int ind = -1;

6       forall(j in vts.rng()){
7         VarTree tree = vts[j];
8         InsertableEdgesVarTree inst = _mapI{tree};

10        forall(ei in inst.getSet()){
11          float d = f.getAddEdgeDelta(tree,ei);
12          if(!tbIn[j].isTabu(ei,it) || d + f.getValue() < fgb){
13            if(eval > d){
14              eval = d;
15              sel_ei = ei;
16              ind = j;
17            }
18            if(firstImprovement)if(eval < 0){
19              break;
20            }
21          }
22        }
23        if(firstImprovement)if(eval < 0)
24          break;
25      }

27      if(sel_ei != null){
28        neighbor(eval,N){
29          tbOut[ind].makeTabu(sel_ei,it);
30          vts[ind].addEdge(sel_ei);
31        }
32      }
33    }
```

Lines 6-7 scan all `VarTree tree` of the input variables `vts`. Line 8 retains a graph invariant `inst` which represents the set of *insertable* edges of `tree`. Line 10 scans all *insertable* edges `ei` and line 11 evaluates the quality of neighboring tree created by inserting `ei` in term of the variation of the value of `f`. Lines 12-21 check tabu condition and update the best move representing by the index of chose tree `ind` and the selected *insertable* edge `sel_ei`. The selected move (lines 29-30) which consists of making `sel_ei` tabu and performing the edge insertion will be submitted to `Neighborhood N` in line 28.

## C.3 exploreTabuMinRemove1VarTree

This method explores the neighborhood generated by replacing a tree $tr$ by one of its neighbors in $NT_2(tr)$.

```
1     void exploreTabuMinRemove1VarTree(Neighborhood N, VarTree[] vts,
          Function<LSGraph> f, GTabuEdge[] tbIn, GTabuEdge[] tbOut, int
          it, float fgb, bool firstImprovement){
2       float eval = System.getMAXINT();
3       Edge sel_eo = null;
4       int ind = -1;

6       forall(j in vts.rng()){
```

```
7         VarTree tree = vts[j];
8         RemovableEdgesVarTree remv = _mapR{tree};

10        forall(eo in remv.getSet()){
11             float d = f.getRemoveEdgeDelta(tree,eo);
12             if(!tbOut[j].isTabu(eo,it) || d + f.getValue() < fgb){
13               if(eval > d){
14                  eval = d;
15                  sel_eo = eo;
16                  ind = j;
17               }
18               if(firstImprovement)if(eval < 0){
19                  break;
20               }
21             }
22        }
23        if(firstImprovement)if(eval < 0)
24          break;
25      }

27      if(sel_eo != null ){
28        neighbor(eval,N){
29          tbIn[ind].makeTabu(sel_eo,it);
30          vts[ind].removeEdge(sel_eo);
31        }
32      }
33    }
```

The schema is similar to that of **exploreTabuMinAdd1VarTree** except the fact that we scan all *removable* edges instead of *insertable* edges (line 8) and do the edge removal (line 30).

## C.4 exploreTabuMinAddRemove1VarTree

This method explores the neighborhood generated by replacing a tree $tr$ by one of its neighbors in $NT_{1+2}(tr)$.

```
1     void exploreTabuMinAddRemove1VarTree(Neighborhood N, VarTree[]
          vts, Function<LSGraph> f, GTabuEdge[] tbIn, GTabuEdge[]
          tbOut, int it, float fgb, bool firstImprovement){
2       float eval = System.getMAXINT();
3       Edge sel_eo = null;
4       Edge sel_ei = null;
5       int ind = -1;

7       forall(j in vts.rng()){
8         VarTree tree = vts[j];
9         InsertableEdgesVarTree inst = _mapI{tree};
10        RemovableEdgesVarTree remv = _mapR{tree};

12        forall(eo in remv.getSet()){
13          Vertex u = eo.fromVertex();
14          if(tree.getAdjEdges()[u.id()].getSize() > 1)
15            u = eo.toVertex();
```

```
16            forall(ei in inst.getSet():!ei.contains(u)){
17               float d = f.getAddRemoveEdgesDelta(tree,eo,ei);
18               if(!tbOut[j].isTabu(eo,it) || !tbIn[j].isTabu(ei,it) ||
                      d + f.getValue() < fgb){
19                  if(eval > d){
20                     eval = d;
21                     sel_eo = eo;
22                     sel_ei = ei;
23                     ind = j;
24                     if(firstImprovement)if(eval < 0) break;
25                  }
26               }
27            }
28            if(firstImprovement)if(eval < 0) break;
29         }
30      }

32      if(sel_eo != null && sel_ei != null){
33         neighbor(eval,N){
34            tbIn[ind].makeTabu(sel_eo,it);
35            tbOut[ind].makeTabu(sel_ei,it);

37            vts[ind].removeEdge(sel_eo);
38            vts[ind].addEdge(sel_ei);
39         }
40      }
41   }
```

The best move selected is represented (lines 2-5) by its evaluation `eval`, the edge to be removed `eo`, the edge to be inserted `ei` and the index `ind` of the tree over which we do the change. Lines 7-8 scan all `VarTree tree`. Lines 9-10 retains two graph invariants representing the set of *insertable* and *removable* edges of `tree`. Lines 12-16 scan all pairs $\langle$`ei,eo`$\rangle$ of *RemvInst*(`tree`) (see Section 3.2 for the formal definition). Line 17 evaluates the quality of the neighboring tree created by removing `eo` and inserting `ei`. Lines 18-26 check the tabu condition and updates the best move (if any). Finally, the selected move is submitted to the `Neighborhood N` in line 32-40.

## C.5 exploreTabuMinReplace1VarTree

This method explores the neighborhood generated by replacing a tree $tr$ by one of its neighbors in $NT_3(tr)$.

```
1    void exploreTabuMinReplace1VarTree(Neighborhood N, VarTree[] vts,
        Function<LSGraph> f, GTabuEdge[] tbIn, GTabuEdge[] tbOut, int
        it, float fgb, bool firstImprovement){
2       Edge sel_eo = null;
3       Edge sel_ei = null;
4       int ind = -1;
5       float eval = System.getMAXINT();
6       forall(j in vts.rng()){
7          VarTree tree = vts[j];
8          ReplacingEdgesVarTree repl = _mapRe{tree};
9          forall(ei in repl.getSet(), eo in getReplacableEdges(tree,ei)){
```

```
10            float d = f.getReplaceEdgeDelta(tree,eo,ei);
11            if(!tbOut[j].isTabu(eo,it) || !tbIn[j].isTabu(ei,it) || d
                  + f.getValue() < fgb){
12              if(d < eval){
13                eval = d;
14                ind = j;
15                sel_ei = ei;
16                sel_eo = eo;
17                if(firstImprovement)if(eval < 0) break;
18              }
19            }
20          }
21          if(firstImprovement)if(eval < 0) break;
22        }

24        if(ind > -1){
25          neighbor(eval,N){
26            tbIn[ind].makeTabu(sel_eo,it);
27            tbOut[ind].makeTabu(sel_ei,it);

29            vts[ind].replaceEdge(sel_eo,sel_ei);
30          }
31        }

33      }
```

Lines 6-7 scan all `VarTree tree`. Line 8 retains a graph invariant `repl` representing the set of *replacing* edges of `tree`. Line 9 scans all pair $\langle$`ei,eo`$\rangle$ of *replacing* and *replacable* edges for the replacement. Line 10 evaluates the quality of the replacement. Lines 11-19 check tabu condition and update the best move in which `ind` stores the index of the best `VarTree` for the update and `sel_ei` and `sel_eo` store the best *replacing* and *replacable* edges. The selected move (if any) is submitted in lines 25-30.

## C.6 exploreTabuMinMultiStageReplace1Move1VarPath

The following method presents the exploration of the neighborhood $ERNP_1$ applying the multistage strategy.

```
1     void exploreTabuMinMultiStageReplace1Move1VarPath(Neighborhood N,
          VarPath[] vps, Constraint<LSGraph> c, GTabuEdge[] tbIn,
          GTabuEdge[] tbOut, int it, float fgb, bool firstImprovement){
2       c.computeVariableViolations();

4       float eval = System.getMAXINT();
5       Edge sel_e = null;
6       int sel_id = -1;

8       selectMax(i in vps.rng())(c.violations(vps[i])){
9         VarPath vp = vps[i];
10        sel_id = i;
11        ReplacingEdgesMaintainPath rpl = _mapReVarPath{vp};
12        forall(e in rpl.getSet()){
```

```
13            float d = c.getDeltaWhenUseReplacingEdge(vp,e);
14            if(!tbIn[i].isTabu(e,it) || d + c.violations() < fgb){
15               if(eval > d){
16                  eval = d;
17                  sel_e = e;
18                  if(firstImprovement)if(eval < 0) break;
19               }
20            }
21         }
22      }

24      if(sel_e != null){
25         neighbor(eval,N){
26            select(eo in
                  getPreferredReplacableEdges(vps[sel_id],sel_e)){
27               vps[sel_id].replaceEdge(eo,sel_e);

29               tbIn[sel_id].makeTabu(eo,it);
30               tbOut[sel_id].makeTabu(sel_e,it);
31            }
32         }
33      }
34   }
```

Line 2 computes the number of violations of all graph variables and lines 8-9 select the most violating `VarPath vp` of the constraint `c`. Line 11 retains the graph invariant `rpl` representing the set of *preferred replacing* edges of `vp`. For each *preferred replacing* edge `e`, line 13 evaluates the quality of move applying this edge. Lines 14-20 check the tabu condition and update the best move. Lines 25-32 submit the selected move (if any) which consists of selecting randomly a *preferred replacable* edge `eo` of sel_ei, performing the move (replace `eo` by `sel_ei` on the selected `VarPath vps[sel_id]` and making two edges `eo`, `sel_ei` tabu.

## C.7 exploreTabuMin1ReplaceXVY1VarPath

The following method presents the exploration of neighborhood generated by replacing a RST $tr$ by one of its neighbors in $\mathcal{N}_2(tr)$.

```
1    void exploreTabuMin1ReplaceXVY1VarPath(Neighborhood N, VarPath[]
        vps, Function<LSGraph> f, GTabuVertex[] tbVIn, int it, float
        fgb, bool firstImprovement){
2       float eval = System.getMAXINT();
3       VarPath sel_vp = null;
4       Vertex sel_v = null;
5       Vertex sel_x = null;
6       Vertex sel_y = null;
7       int ind = -1;

9       forall(k in vps.rng()){
10         VarPath vp = vps[k];
11         GenericGraph g = null;
12         if(vp.getType() == UNDIRECTED){
13            g = vp.getLUB();
```

```
14          }else{
15            g = vp.getDLUB();
16          }

18          set{Vertex} S = g.getVertices();
19          forall(v in S: !vp.contains(v)){
20            Vertex x = vp.getSource();
21            while(x != vp.getDestination()){
22              Vertex y = vp.getNextVertex(x);
23              while(y != vp.getDestination()){
24                if(g.edge(x,v) != null && g.edge(v,y) != null){
25                  float d = f.getDeltaWhenUseReplacingPath(vp,v,x,y);
26                  if(!tbVIn[k].isTabu(v,it) || d + f.getValue() < fgb){
27                    if(d < eval){
28                      eval = d;
29                      sel_vp = vp;
30                      sel_v = v;
31                      sel_x = x;
32                      sel_y = y;
33                      ind = k;
34                      if(eval < 0 && firstImprovement){
35                        break;
36                      }
37                    }
38                  }
39                }
40                y = vp.getNextVertex(y);
41              }
42              if(eval < 0 && firstImprovement) break;
43              else
44                x = vp.getNextVertex(x);
45            }
46            if(eval < 0 && firstImprovement) break;
47          }
48          if(eval < 0 && firstImprovement) break;
49        }

51        if(sel_vp != null){
52          neighbor(eval,N){
53            sel_vp.replaceSubPath(sel_v,sel_x,sel_y);

55            tbVIn[ind].makeTabu(sel_v,it);
56          }
57        }
58      }
```

Lines 9-10 scan all `VarPath vp`. Lines 19-47 examine all triple $\langle$`x,v,y`$\rangle$ such that `x` is located before `y` on the current path `vp` and `v` is not in `vp`. The move here is the replacement of subpath from `x` to `y` of `vp` by the new path `x` $\longrightarrow$ `v` $\longrightarrow$ `y`. Line 24 checks the condition that `(x,v)` and `(v,y)` are edges of the given graph `g` retrieved in lines 11-16. Line 25 evaluates the quality of this move. Lines 26-38 check the tabu condition and update the best move. The selected move is submitted in lines 51-57.

## C.8 exploreTabuMin1ReplaceXVY1VarPath

The following method is similar to the above method except that it explores the neighborhood taking into account the graph constraint `c` instead of the graph function `f`.

```
1   void exploreTabuMin1ReplaceXVY1VarPath(Neighborhood N, VarPath[]
        vps, Constraint<LSGraph> c, GTabuVertex[] tbVIn, int it, float
        fgb, bool firstImprovement){
2     float eval = System.getMAXINT();
3     VarPath sel_vp = null;
4     Vertex sel_v = null;
5     Vertex sel_x = null;
6     Vertex sel_y = null;
7     int ind = -1;

9     forall(k in vps.rng()){
10      VarPath vp = vps[k];
11      GenericGraph g = null;
12      if(vp.getType() == UNDIRECTED){
13        g = vp.getLUB();
14      }else{
15        g = vp.getDLUB();
16      }

18      set{Vertex} S = g.getVertices();
19      forall(v in S: !vp.contains(v)){
20        Vertex x = vp.getSource();
21        while(x != vp.getDestination()){
22          Vertex y = vp.getNextVertex(x);
23          while(y != vp.getDestination()){
24            if(g.edge(x,v) != null && g.edge(v,y) != null){
25              float d = c.getDeltaWhenUseReplacingPath(vp,v,x,y);
26              if(!tbVIn[k].isTabu(v,it) || d + c.violations() <
                    fgb){
27                if(d < eval){
28                  eval = d;
29                  sel_vp = vp;
30                  sel_v = v;
31                  sel_x = x;
32                  sel_y = y;
33                  ind = k;
34                  if(eval < 0 && firstImprovement){
35                    break;
36                  }
37                }
38              }
39            }
40            y = vp.getNextVertex(y);
41          }
42          if(eval < 0 && firstImprovement) break;
43          else
44            x = vp.getNextVertex(x);
45        }
46        if(eval < 0 && firstImprovement) break;
47      }
48      if(eval < 0 && firstImprovement) break;
```

```
49      }

51      if(sel_vp != null){
52        neighbor(eval,N){
53          sel_vp.replaceSubPath(sel_v,sel_x,sel_y);

55          tbVIn[ind].makeTabu(sel_v,it);
56        }
57      }
58    }
```

## C.9 exploreTabuMinReplace1Move1VarPath

The following method explores the neighborhood generated by replacing a RST $tr$ by one of its neighbors in $ERNP_1(tr)$.

```
1     void exploreTabuMinReplace1Move1VarPath(Neighborhood N, VarPath[]
          vps, Function<LSGraph> f, GTabuEdge[] tbIn, GTabuEdge[]
          tbOut, int it, float fgb, bool firstImprovement){
2       Edge sel_ei = null;
3       int ind = -1;
4       float eval = System.getMAXINT();

6       forall(j in vps.rng()){
7         VarPath vp = vps[j];
8         ReplacingEdgesMaintainPath repl = _mapReVarPath{vp};
9         forall(e in repl.getSet()){
10          float d = f.getDeltaWhenUseReplacingEdge(vp,e);
11          if(!tbIn[j].isTabu(e,it) || d + f.getValue() < fgb){
12            if(d < eval){
13              eval = d;
14              ind = j;
15              sel_ei = e;

17            }
18            if(firstImprovement)if(eval < 0)
19              break;
20          }
21        }
22        if(firstImprovement)if(eval < 0)
23          break;

25      }

27      if(ind > -1){
28        Edge sel_eo = null;
29        select(eo in getPreferredReplacableEdges(vps[ind],sel_ei)){
30          sel_eo = eo;
31        }

33        if(sel_eo != null)neighbor(eval,N){
34            tbIn[ind].makeTabu(sel_eo,it);
35            tbOut[ind].makeTabu(sel_ei,it);
```

```
37              vps[ind].replaceEdge(sel_eo,sel_ei);
38          }
39      }

41  }
```

Lines 6-7 scan all `VarPath vp`. Line 8 retains the graph invariant `repl` representing the set of *preferred replacing* edges of `vp`. For each *preferred replacing* edge `e`, line 10 evaluates the quality of move applying this edge. Lines 11-20 check the tabu condition and update the best move. Lines 28-39 submit the selected move (if any) which consists of selecting randomly a *preferred replacable* edge `sel_eo` of sel_ei, performing the move (replace `sel_eo` by `sel_ei` on the selected `VarPath vps[ind]` and making two edges `sel_eo`, `sel_ei` tabu.

## C.10 exploreTabuMinReplace2MovesVarPath

The following method explore the neighborhood generated by replacing a RST $tr$ by one of its neighbors in $ERNP_2(tr)$.

```
1   void exploreTabuMinReplace2MovesVarPath(Neighborhood N, VarPath[]
        vps, Function<LSGraph> f, GTabu[] tbIn, GTabu[] tbOut, int
        it, float fgb, bool firstImprovement){
2     Edge sel_ei1 = null;
3     Edge sel_ei2 = null;
4     int ind = -1;
5     float eval = System.getMAXINT();

7     forall(j in vps.rng()){
8       VarPath vp = vps[j];
9       ReplacingEdgesMaintainPath repl = _mapReVarPath{vp};
10      forall(e1 in repl.getSet(), e2 in repl.getSet():
            dominate(e2,e1,vp)){
11        float d = f.getDeltaWhenUseReplacingEdge(vp,e1,e2);
12        if(!tbIn[j].isTabu(e1,it) || !tbIn[j].isTabu(e2,it) || d +
              f.getValue() < fgb){
13          if(d < eval){
14            eval = d;
15            ind = j;
16            sel_ei1 = e1;
17            sel_ei2 = e2;
18          }
19          if(firstImprovement)if(eval < 0)
20            break;
21        }
22      }
23      if(firstImprovement)if(eval < 0)
24        break;
25    }

27    if(ind > -1){
28      Edge sel_eo1 = null;
29      Edge sel_eo2 = null;
30      select(eo1 in getPreferredReplacableEdges(_vps[ind],sel_ei1),
```

```
31              eo2 in getPreferredReplacableEdges(_vps[ind],sel_ei2):
                    !tbOut[ind].isTabu(eo1,it) ||
                    !tbOut[ind].isTabu(eo2,it)){
32            sel_eo1 = eo1;
33            sel_eo2 = eo2;
34          }

36          if(sel_eo1 != null && sel_eo2 != null)neighbor(eval,N){
37              tbIn[ind].makeTabu(sel_eo1,it);
38              tbIn[ind].makeTabu(sel_eo2,it);

40              tbOut[ind].makeTabu(sel_ei1,it);
41              tbOut[ind].makeTabu(sel_ei2,it);

43              vps[ind].replaceEdge(sel_eo1,sel_ei1);
44              vps[ind].replaceEdge(sel_eo2,sel_ei2);

46          }
47        }

49    }
```

The method is similar to C.9 except that we scan all pairs of two independent *preferred replacing* edges `e1,e2` (line 10) instead of only one.

## C.11 exploreTabuMinChangeDestinationVarItinerary

The following method explores the neighborhood generated by replacing a *VarItinerary vi* by one of its neighbors in $SDCNI_1(vi)$.

```
1     void exploreTabuMinChangeDestinationVarItinerary(Neighborhood N,
          VarItinerary[] vis, Function<LSGraph> f, GTabuVertex[]
          tbVIn, int it, float fgb, bool firstImprovement){
2       float eval = System.getMAXINT();
3       VarPath sel_vp = null;
4       Vertex sel_des = null;
5       VarItinerary sel_vi = null;
6       int ind = -1;
7       forall(i in vis.rng()){
8         VarItinerary vi = vis[i];
9         forall(j in vi.rng(): j < vi.getSize()){
10          VarPath vp = vi.get(j);
11          int k = _mapVP{vp};

13          VarRootedSpanningTree t = vp.getVarRootedSpanningTree();
14          Vertex des = vp.getDestination();
15          forall(newDes in t.getVertices(): newDes != des){

17            float d = f.getChangeDestinationDelta(vi,vp,newDes);
18            if(!tbVIn[k].isTabu(newDes,it) || d + f.getValue() <
                  fgb){
19              if(d < eval){
20                eval = d;
21                sel_vp = vp;
22                sel_des = newDes;
```

```
23                  sel_vi = vi;
24                  ind = k;
25                  if(eval < 0 && firstImprovement) break;
26                }
27              }
28            }
29            if(eval < 0 && firstImprovement) break;
30          }
31          if(eval < 0 && firstImprovement) break;
32        }

34        if(sel_vi != null){
35          neighbor(eval,N){
36            sel_vi.changeDestination(sel_vp,sel_des);

38            tbVIn[ind].makeTabu(sel_des,it);
39          }
40        }

42      }
```

Lines 7-8 scan all *VarItinerary* `vi`. Lines 9-10 scan all *VarPath* `vp` which is not the last element of `vi`. Lines 13-15 scan all vertices `newDes` of the given graph which will be the new destination for `vp`. Line 17 evaluates the quality of the move by making `newDes` the new root (or destination) of `vp` and making `newDes` the new source of the successive *VarPath* of `vp` in `vi`. Lines 18-26 check the tabu condition and update the best move. Finally, the selected move (if any) is submitted in lines 35-39.

## C.12 exploreDegradeMultiStageReplace1Move1VarPath

The following method explores the neighborhood generated by replacing a *VarPath* $vp$ by one of its neighbors in $ERNP_1(vp)$, selecting only moves which are better than the current solution in term of the constraint `c`, applying the multistage strategy.

```
1     void exploreDegradeMultiStageReplace1Move1VarPath(Neighborhood N,
          VarPath[] vps, Constraint<LSGraph> c, bool firstImprovement){
2       c.computeVariableViolations();

4       Edge sel_ei = null;
5       float eval = System.getMAXINT();
6       VarPath sel_vp = null;
7       selectMax(k in vps.rng())(c.violations(vps[k])){
8         ReplacingEdgesMaintainPath rpl = _mapReVarPath{vps[k]};
9         forall(ei in rpl.getSet()){
10          float d = c.getDeltaWhenUseReplacingEdge(vps[k],ei);
11          if(d < eval){
12            eval = d;
13            sel_vp = vps[k];
14            sel_ei = ei;
15            if(firstImprovement)if(eval < 0)break;
16          }
17        }
18        if(sel_vp != null && eval < 0){
```

```
19            neighbor(eval,N){
20              select(sel_eo in
                    getPreferredReplacableEdges(sel_vp,sel_ei)){
21                sel_vp.replaceEdge(sel_eo,sel_ei);
22              }
23            }
24          }
25        }
26    }
```

Line 2 computes the violations of all graph variables of the constraint `c`. Line 7 selects the most violating `VarPath vps[k]`. Line 8 retains the set of *preferred replacing* edges of `vps[k]` and line 9 scans all its elements `ei`. Line 10 evaluates the quality of move by taking into account `ei`. Lines 11-16 update the best move. Line 18 check is the best neighbor associating with the best move is better than the current solution. If it is the case, line 19-23 submit the selected move.

## C.13 exploreDegradeMultiStageReplace2Moves1VarPath

The objective of the following method is the same with that of C.12 except that it explores the neighborhood generated by replacing a *VarPath* $vp$ by one of its neighbors in $ERNP_2(vp)$ instead of $ERNP_1(vp)$.

```
1     void exploreDegradeMultiStageReplace2Moves1VarPath(Neighborhood
          N, VarPath[] vps, Constraint<LSGraph> c, bool
          firstImprovement){
2       c.computeVariableViolations();

4       Edge sel_ei1 = null;
5       Edge sel_ei2 = null;
6       float eval = System.getMAXINT();
7       VarPath sel_vp = null;
8       selectMax(k in vps.rng())(c.violations(vps[k])){
9         ReplacingEdgesMaintainPath rpl = _mapReVarPath{vps[k]};

11        forall(ei1 in rpl.getSet(), ei2 in rpl.getSet():dominate(ei2,
              ei1, vps[k])){
12          float d = c.getDeltaWhenUseReplacingEdge(vps[k],ei1,ei2);
13          if(d < eval){
14            eval = d;
15            sel_vp = _vps[k];
16            sel_ei1 = ei1;
17            sel_ei2 = ei2;
18            if(firstImprovement)if(eval < 0) break;
19          }
20        }
21        if(sel_vp != null && eval < 0){
22          neighbor(eval,N){
23            select(sel_eo1 in getPreferredReplacableEdges(sel_vp,
                  sel_ei1),
24                 sel_eo2 in getPreferredReplacableEdges(sel_vp,
                       sel_ei2)){
```

```
26                sel_vp.replaceEdge(sel_eo1,sel_ei1);
27                sel_vp.replaceEdge(sel_eo2,sel_ei2);
28              }
29            }
30          }
31        }
32      }
```

Line 11 scans all pair of independent *preferred replacing* edges ⟨ei1,ei2⟩ and line 12 evaluates the move by applying these edges. Lines 13-19 update the best move. If the best neighbor associating with the best move is better than the current solution (line 21), then the best move is submitted to `Neighborhood N` in lines 22-29.

## C.14 exploreDegradeMultiStageReplace1Move2VarPaths

The objective of the following method is the same with that of C.12 except that it explores the neighborhood generated by replacing two *VarPath*s $vp_1$ and $vp_2$ by one of its neighbors in $ERNP_1(vp_1)$ and $ERNP_1(vp_2)$.

```
1     void exploreDegradeMultiStageReplace1Move2VarPaths(Neighborhood
          N, VarPath[] vps, Constraint<LSGraph> c, bool
          firstImprovement){
2       if(_vps.rng().getUp() - _vps.rng().getLow() + 1 < 2)
3         return;

5       c.computeVariableViolations();

7       Edge sel_ei1 = null;
8       Edge sel_ei2 = null;
9       int k1;
10      int k2;
11      float eval = System.getMAXINT();
12      VarPath sel_vp1 = null;
13      VarPath sel_vp2 = null;

15      selectMax(k in vps.rng())(c.violations(vps[k]))
16        k1 = k;
17      selectMax(k in vps.rng():k != k1)(c.violations(vps[k]))
18        k2 = k;

20        ReplacingEdgesMaintainPath rpl1 = _mapReVarPath{vps[k1]};
21        ReplacingEdgesMaintainPath rpl2 = _mapReVarPath{vps[k2]};

23        forall(ei1 in rpl1.getSet(), ei2 in rpl2.getSet()){
24          float d =
                c.getDeltaWhenUseReplacingEdge(vps[k1],ei1,vps[k2],ei2);
25          if(d < eval){
26            eval = d;
27            sel_vp1 = vps[k1];
28            sel_vp2 = vps[k2];
29            sel_ei1 = ei1;
30            sel_ei2 = ei2;
31            if(firstImprovement)if(eval < 0) break;
32          }
```

```
33          }
34          if(sel_vp1 != null && eval < 0){
35            neighbor(eval,N){
36              select(sel_eo1 in getPreferredReplacableEdges(sel_vp1,
                    sel_ei1),
37                sel_eo2 in getPreferredReplacableEdges(sel_vp2,
                      sel_ei2)){

39                sel_vp1.replaceEdge(sel_eo1,sel_ei1);
40                sel_vp2.replaceEdge(sel_eo2,sel_ei2);
41              }
42            }
43          }
44      }
```

Lines 15-18 select two most violating `VarPath vps[k1], vps[k2]`. Lines 20-21 retain two graph invariants representing the set of *preferred replacing* edges of `vps[k1]` and `vps[k2]`. Line 23 scans all `ei1,ei2` of these sets. Line 24 evaluates the quality of move by applying `ei1` on `vps[k1` and applying `ei2` on `vps[k2]`. Lines 25-32 update the best move. If the best neighbor associating with the best move is better than the current solution (line 34), then the best move is submitted in lines 35-42.

## C.15 exploreDegradeReplace2Moves1VarPath

The objective of the following method is the same with that of C.13 except that it does not apply the multistage strategy.

```
1     void exploreDegradeReplace2Moves1VarPath(Neighborhood N,
          VarPath[] vps, Constraint<LSGraph> c, bool firstImprovement){

3               float delta = System.getMAXINT();
4               int sel_k = -1;
5               Edge sel_ei1;
6               Edge sel_ei2;

8               forall(k in vps.rng()){
9                 ReplacingEdgesMaintainPath rpl =
                      _mapReVarPath{vps[k]};
10                forall(e1 in rpl.getSet()){
11                  forall(e2 in rpl.getSet(): dominate(e2,e1,vps[k])){
12                    float d =
                          c.getDeltaWhenUseReplacingEdge(vps[k],e1,e2);
13                    if(d < delta){
14                      sel_k = k;
15                      sel_ei1 = e1;
16                      sel_ei2 = e2;
17                      delta = d;
18                      if(firstImprovement)if(delta < 0) break;
19                    }
20                  }
21                  if(firstImprovement) if(delta < 0) break;
22                }
23                if(firstImprovement) if(delta < 0)
```

```
24                  break;
25              }

27              if(delta < 0)neighbor(delta,N){
28                  select(eo1 in
                        getPreferredReplacableEdges(vps[sel_k],sel_ei1),
29                    eo2 in
                        getPreferredReplacableEdges(vps[sel_k],sel_ei2)){
30                    vps[sel_k].replaceEdge(eo1,sel_ei1);
31                    vps[sel_k].replaceEdge(eo2,sel_ei2);
32                  }

34              }
35      }
```

Line 8 scans all `VarPath vps[k]` instead of considering only the most violating `VarPath` in C.13. The remaining code is the same.

# BIBLIOGRAPHY


[ABH+04] U.A. Acar, G.E. Blelloch, R. Harper, J.L. Vittes, and S.L.M. Woo. An experimental analysis of change propagation in dynamic trees. *In proc. 15th SODA*, pages 524–533, 2004. 61

[AGLR94] Baruch Awerbuch, Rainer Gawlick, Tom Leighton, and Yuval Rabani. On-line admission control and circuit routing for high performance computing and communication. In *35th IEEE Symposium on Foundations of Computer Science (FOCS1994)*, pages 412–423, 1994. 25

[AHLT05] Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, 2005. 61

[ALM06] Rafael Andrade, Abilio Lucena, and Nelson Maculan. Using lagrangian dual information to generate degree constrained spanning trees. *Discrete Applied Mathematics*, pages 703–717, 2006. 1, 3, 37

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, New Jersey, United States, 1993. 38

[AOS03] Ravindra K. Ahuja, James B. Orlinb, and Dushyant Sharma. A composite very large-scale neighborhood structure for the capacitated minimum spanning tree problem. *Operations Research Letters 31*, pages 185–194, 2003. 1, 36

[ARD99] M. Ali, B. Ramamurthy, and J. Deogun. Genetic algorithm for routing in wdm optical networks with power considerations. part i: The unicast case. In *Proceedings of the 8th IEEE ICCCNÕ99, Boston-Natick, MA, USA*, pages 335–340, 1999. 27

[ARD00] M. Ali, B. Ramamurthy, and J. Deogun. Routing and wavelength assignment with power considerations in optical networks. *Computer Networks*, 32:539–555, 2000. 27

[BB] C. Blum and M. Blesa. Kctlib - a library for the edge-weighted k-cardinality tree problem. http://iridia.ulb.ac.be/ cblum/kctlib/. 23, 88

[BB05] C. Blum and M. Blesa. New metaheuristic approaches for the edge-weighted k-cardinality tree problem. *Computers and Operations Research*, pages 32(6):1355–1377, 2005. 1, 3, 23, 24, 36, 38, 86, 88, 89, 93

[BB07] M. Blesa and C. Blum. Finding edge-disjoint paths in networks: An ant colony optimization algorithm. *Journal of Mathematical Modelling and Algorithms, 6(3)*, pages 361–391, 2007. 1, 25, 26, 107, 116, 120

[BBXG00] Maria J. Blesa, M. Josep Blesa, Fatos Xhafa, and Jordi Girona. A c++ implementation of tabu search for k-cardinality tree problem based on generic programming and component reuse. In *GCSE Young Researchers Workshop 2000 (Part of the Second International Symposium on Generative and Component-based Software Engineering) October 9-12*, pages 648–652. Net.ObjectDaysForum, 2000. 23, 24

[BC89] J. E. Beasley and N. Christofides. An algorithm for the resource constrained shortest path problem. *Network, vol. 19*, pages 379–394, 1989. 1, 3, 26, 101

[BE03] Christian Blum and Matthias Ehrgott. Local search algorithms for the k-cardinality tree problem. *Discrete Appl. Math*, 128:511–540, 2003. 23

[Bea] J. E. Beasley. Or-library, url=http://people.brunel.ac.uk/ mastjjb/jeb/info.html. 26, 27, 120

[BFCP+05] Michael A. Bender, Martín Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms 57*, pages 75–94, 2005. 61

[BGG+97] P. Badeau, M. Gendreau, F. Guertin, J.-Y. Potvin, and E. Taillard. A parallel tabu search heuristic for the vehicle routing problem with time windows. *Transportation Research - C 5*, pages 109–122, 1997. 39

[BJGG07] Jørgen Bang-Jensena, Daniel Gonçalvesb, and Inge Li Gørtzc. Finding well-balanced pairs of edge-disjoint trees in edge-weighted graphs. *Discrete Optimization*, 4(1):334–348, 2007. 142

[Blu02] Christian Blum. Ant colony optimization for the edge-weighted k-cardinality tree problem. In *In GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 27–34. Morgan Kaufmann Publishers, 2002. 23

[Blu06] C. Blum. A new hybrid evolutionary algorithm for the huge k-cardinality tree problem. *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 515–522, 2006. 3, 23

[BM00] Dhritiman Banerjee and Biswanath Mukherjee. Wavelength-routed optical networks: Linear formulation, resource budgeting tradeoffs, and a reconfiguration study. *IEEE/ACM Transactions on Networking*, 8:598–607, 2000. 27

[BMP04] D. Banerjee, V. Mehta, and S. Pandey. A genetic algorithm approach for solving the routing and wavelength assignment problem in wdm networks. *3rd IEEE/IEE International Conference on Networking, ICNÕ2004, Paris*, pages 70–78, 2004. 27

[BMX01] Maria J. Blesa, Pablo Moscato, and Fatos Xhafa. A memetic algorithm for the minimum weighted k-cardinality tree subgraph problem. *In: Proceedings of the Metaheuristics International Conference MICÕ2001*, pages 85–90, 2001. 23

[BR01] M. P. Bastos and C. C. Ribeiro. Reactive tabu search with path-relinking for the steiner problem in graphs. *In C. C. Ribeiro and P. Hansen, editors, Essays and Surveys in Metaheuristics*, pages 39–58, 2001. 36

[BS00] Alok Baveja and Aravind Srinivasan. Approximation algorithms for disjoint paths and related routing and packing problems. *Math. Oper. Res.*, 25(2):255–280, 2000. 25

[BS04] T.N. Bui and G. Sundarraj. Ant system for the k-cardinality tree problem. *In K. Deb et al., editor, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2004)*, 3102:36–47, 2004. 3, 23

[BV93] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, 1993. 61

[BYC97] S. Banerjee, J. Yoo, and C. Chen. Design of wavelength routed optical networks for packet switched traffic. *IEEE Journal of Lightware Technology*, 15(9):1636–1646, 1997. 27

[CA94] Shun Yan Cheung and Kumar A. Efficient quorumcast routing algorithms. *In: Proceedings of INFOCOM'94*, pages 840–847, 1994. xv, 24, 25, 93, 98

[CB96] Chien Chen and Subrata Banerjee. A new model for optimal routing and wavelength assignment in wavelength division multiplexed optical networks. *INFOCOM 1996*, pages 164–171, 1996. 1, 2, 25, 27

[CCP03] Joao C. N. Clímaco, José M. F. Craveirinha, and Marta M. B. Pascoal. A bicriterion approach for routing problems in multimedia networks. *Networks*, 41:206–220, 2003.

[CGK92] I. Chlamtac, A. Ganz, and G. Karmi. Lightpath communications: An approach to high bandwidth optical WANS. *IEEE Transactions on Communications*, 40(7):1171–1182, 1992. 27

[CK03] Chandra Chekuri and Sanjeev Khanna. Edge disjoint paths revisited. *In Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms (SODA2003)*, pages 628–637, 2003. 1, 25

[CKIL] M. Chimani, M. Kandyba, and P. Mutzel I. Ljubic. Efficient exact algorithm for the k-cardinality tree problem. url = http://ls11-www.cs.uni-dortmund.de/people/kandyba/kcard.html. 89

[CKIL09] M. Chimani, M. Kandyba, and P. Mutzel I. Ljubic. Obtaining optimal k-cardinality trees fast. *ACM Journal of Experimental Algorithmics*, 14(2):5.1–5.23, 2009. xv, 1, 3, 23, 89, 91

[CMS02] A. Corberán, R. Martí, and J. M. Sanchis. A grasp heuristic for the mixed chinese postman problem. *European Journal of Operational Research*, 142:70–80, 2002. 2

[CRW08] W. Matthew Carlyle, Johannes O. Royset, and R. Kevin Wood. Lagrangian relaxation and enumeration for solving constrained shortest-path problems. *Networks*, 52:256–270, 2008. 26, 27, 101, 103, 104

[CW03] W. Matthew Carlyle and R. Kevin Wood. Near-shortest and k-shortest simple paths. *Networks*, 46:98–109, 2003. 27

[dAaRUW01] M. Poggi de Aragão, C. C. Ribeiro, E. Uchoa, and R. F. Werneck. Hybrid local search for the steiner problems in graphs. *In Ext. abstract of the 4th Metaheuristics International Conference*, pages 429–433, 2001. 36

[dAUW01] M.P. de Aragão, E. Uchoa, and R.F. Werneck. Dual heuristics on the exact solution of large Steiner problems. In *Proceedings of the Brazilian Symposium on Graphs, Algorithms and Combinatorics GRACO'01*, Fortaleza, 2001. 1

[DB03] I. Dumitrescu and N. Boland. Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem. *Networks*, 42:135–153, 2003. 1, 3, 26, 27

[DDD05] G. Dooms, Y. Deville, and P. Dupont. Cp(graph): Introducing a graph computation domain in constraint programming. *International Conference on Principles and Practice on Constraint Programming, LNCS 3709*, pages 211–225, 2005. 1, 21, 33, 142

[DGTW96] B. Du, J. Gu, D.H.K. Tsang, and W. Wang. Quorumcast routing by multispace search. *Proceedings of IEEE Globecom1996*, pages 1069–1073, 1996. 24, 25

[DR59] G. B. Dantzig and R.H. Ramser. The truck dispatching problem. *Management Science*, 6:80–91, 1959. 28

[DR00] Rudra Dutta and George N. Rouskas. A survey of virtual topology design algorithms for wavelength routed optical networks. *Optical Networks*, 1(1):73–89, 2000. 27

[DS04] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. The MIT Press, 2004. 19

[DV97] C. Duin and S. Voß. Efficient path and vertex exchange in steiner tree algorithms. *Networks*, 29:89–105, 1997. 36

[EFHM97] Matthias Ehrgott, Jörg Freitag, Horst W. Hamacher, and Francesco Maffioli. Heuristics for the k-cardinality tree and subgraph problem. *Asia-Pacific Journal of Operational Research*, 14(1):87–114, 1997. 23

[FGI05] Birger Funke, Tore Grünert, and Stefan Irnich. Local search for vehicle routing and scheduling problems: Review and conceptual integration. *Journal of Heuristics*, 11(4):267–306, 2005. 39

[Fis07] Thomas Fischer. Improved local search for large optimum communication spanning tree problems. *In MIC'2007 - 7th Metaheuristics International Conference*, 2007. 1

[Fre97] G. N. Frederickson. A data structure for dynamically maintaining rooted trees. *Journal of Algorithms*, 24(1):37–65, 1997. 61

[GCJ09] Teresa Gomes, Jos'e Craveirinha, and Lúsa Jorge. An effective algorithm for obtaining the minimal cost pair of disjoint paths with dual arc costs. *Computer & Operations Research 36(5)*, pages 1670–1682, 2009. 1

[GH97] Naveen Garg and D.S. Hochbaum. An o(log k) approximation algorithm for the k minimum spanning tree problem in the plane. *Algorithmica*, 18(1):111–121, 1997. 23

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A guide to the Theory of NP-Completeness*. W. H. Freeman, 1st ed., 1979. 29, 30

[GvHR06] M. Gruber, J.I. van Hemert, and G.R. Raidl. Neighborhood searches for the bounded diameter minimum spanning tree problem embedded in a vns, ea, and aco. *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1187–1194, 2006. 1

[GW81] B.L. Golden and R.T. Wong. Capacitated arc routing problems. *Networks*, 11:305–315, 1981. 2, 28

[GY03] Jonathan L. Gross and Jay Yellen. *Handbook of Graph Theory*. CRC Press, 2003. 9

[HFD+10] Trong Viet Ho, Pierre Francois, Yves Deville, Quang Dung Pham, and Olivier Bonaventure. Using local search for traffic engineering in switched ethernet networks. In *Proceedings of 22nd International Teletraffic Congress (ITC-22), Amsterdam, Netherlands*, 2010. 23, 85, 140

[HK99] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J ACM*, 46(4):502–516, 1999. 61

[hlk08] Sanne Wøhlk. *A Decade of Capacitated Arc Routing Problem*. In The Vehicle Routing Problem : Latest Advances and New Challenges. Editors: Bruce Golden ; S. Raghavan ; Edward Wasil. Springer, 2008. 28

[HM01] Alain Hertz and Michel Mittaz. A variable neighborhood descent algorithm for the undirected capacitated arc routing problem. *Transportation Science*, 35(4):425–434, 2001. 2

[Hyy04] E. Hyytia. Resource allocation and performance analysis problems in optical networks, ph.d. thesis. In *Dpt. of Electrical and Communications Engineering, Helsinki University of Technology, Helsinki, Sweden*, 2004. 27

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-InterScience, New York, 1991. 85, 86

[JMT07] B. Jaumard, C. Meyer, and B. Thiongane. Comparison of ilp formulations for the rwa problem. *Optical Switching and Networking*, 4:157–172, 2007. 1, 2, 27

[JMY06] B. Jaumard, C. Meyer, and X. Yu. How much wavelength conversion allows a reduction in the blocking rate? *Journal of Optical Networking*, 5(12):881–900, 2006. 27

[JSMR01] Alpár Jüttner, Balázs Szviatovszki, I. Mécs, and Zsolt Rajkó. Lagrange relaxation based method for the qos routing problem. *Proc IEEE INFOCOM*, 2:859–868, 2001. 26

[KES01] Mohan Krishnamoorthy, Andreas T. Ernst, and Yazid M. Sharaiha. Comparison of algorithms for the degree constrained minimum spanning tree. *Journal of Heuristics*, pages 587–611, 2001. 1

[KK01a] T. Korkmaz and M. Krunz. Multi-constrained optimal path selection. *Proc IEEE INFOCOM*, 2:834–843, 2001. 26

[KK01b] T. Korkmaz and M. Krunz. A randomized algorithm for finding a path subject to multiple qos constraints. *Comput Networks*, 36:251–268, 2001. 26

[KKKM04] F. Kuipers, T. Korkmaz, M. Krunz, and P. Van Mieghem. Performance evaluation of constraint-based path selection algorithms. *IEEE Network*, 18:16–23, 2004. 26

[KKT02] Turgay Korkmaz, Marwan Krunz, and Spyros Tragoudas. An efficient algorithm for finding a path subject to two additive constraints. *Computer Communications*, 25:225–238, 2002. 26

[Kle96] J. Kleinberg. *Approximation algorithms for disjoint-paths problems. PhD thesis*. The MIT Press, Cambridge, USA, 1996. 1, 4, 25, 107

[KP80] Paris-C. Kanellakis and Christos H. Papadimitriou. Local search for the asymmetric traveling salesman problem. *OPERATIONS RESEARCH*, 28(5):1086–1099, 1980. 39, 49

[KS01a] Petr Kolman and Christian Scheideler. Simple on-line algorithms for the maximum disjoint paths problem. *13th ACM Symposium on Parallel Algorithms and Architectures (SPAAÕ01)*, pages 38–47, 2001. 1, 25

[KS01b] R. M. Krishnaswamy and K. N. Sivarajan. Algorithms for routing and wavelength assignment based on solutions of LP-relaxations. *IEEE Commun. Lett.*, 5(10):435–437, 2001. 27

[KS04] Stavros G. Kolliopoulos and Clifford Stein. Approximating disjoint-path problems using packing integer programs. *Mathematical Programming*, 99(1):63–87, 2004. 1, 4, 25

[KY10] Ali Kansou and Adnan Yassine. New upper bounds for the multi-depot capacitated arc routing problem. *Int. J. Metaheuristics*, 1(1):81–95, 2010. 2

[LHH07] Y. Li, J. Harms, and R. Holte. Fast exact multiconstraint shortest path algorithms. *IEEE Int Conf Commun*, pages 123–130, 2007. 26

[LKLP02] K. Lee, K. Kang, T. Lee, and S. Park. An optimization approach to routing and wavelength assignment in wdm all-optical mesh networks without wavelength conversion. *ETRI Journal*, 24(2):131–141, 2002. 27

[LMSL92] Chung-Lun Li, S. Thomas McCormick, and David Simchi-Levi. Finding disjoint paths with different path-costs: Complexity and algorithms. *Networks*, pages 653–67, 1992. 1

[Low98] Chor Ping Low. A fast search algorithm for the quorumcast routing problem. *Information Processing Letters*, 66:87–92, 1998. 24

[MAK07] Wil Michiels, Emile Aarts, and Jan Korst. *Theoretical Aspects of Local Search*. Springer, 2007. 13

[MDVH09] Jean-Noel Monette, Yves Deville, and Pascal Van Hentenryck. Aeon: Synthesizing scheduling algorithms from high-level models. In *Proceedings of 2009 INFORMS Computing Society Conference (ICS09), Charleston, South Carolina*, pages 11–13, 2009. 142

[MF00] Blesa MJ and Xhafa F. A c++ implementation of tabu search for k-cardinality tree problem based on generic programming and component reuse. *In: Net.ObjectDays 2000 Tagungsband, Erfurt, Germany*, pages 648–52, 2000.

[mis]

[MJ96] Ehrgott M and Freitag J. K_tree/k_subgraph: a program package for minimal weighted k-cardinality-trees and -subgraphs. *European Journal of Operational Research*, 1(93):214–225, 1996. 23

[MK04] P. Van Mieghem and F.A. Kuipers. Concepts of exact qos routing algorithms. *IEEE/ACM Trans Network*, 12:851–864, 2004. 26

[MSI06] MSI/IFI. Around project, url = http://www.ifi.auf.org/site/content/view/48/84/. 2006. 23, 28

[MTY09] Yi Mei, Ke Tang, and Xin Yao. A global repair operator for capacitated arc routing problem. *IEEE Transaction on Systems, MAN, and Cybernetics-Part B*, 39(3):723–734, 2009. 2

[MU01] N. Mladenovic and D. Uroševic. Variable neighborhood search for the k-cardinality tree problem. *In Proceedings of the Metaheuristics International Conference MIC'2001*, 2:743–747, 2001. 23

[Muk06] B. Mukherjee. *Optical WDM Networks*. Springer, 2006. 27

[NH79] S. C. Ntafos and S. L. Hakimi. On path cover problems in digraphs and applications to program testing. *IEEE Transactions on Software Engineering*, pages 520–529, 1979. 1

[NP01] Enrico Nardelli and Guido Proietti. Finding all the best swaps of a minimum diameter spanning tree under transient edge failures. *Journal of Graph Algorithms and Applications vol. 5, no. 5*, pages 39–57, 2001. 1

[NR06] T. Noronha and C. Ribeiro. Routing and wavelength assignment by partition coloring. *European Journal of Operational Research*, 171(3):797–810, 2006. 27

[OB03] A. Ozdaglar and D. Bersekas. Routing and wavelength assignment in optical networks. *IEEE/ACM Transactions on Networking*, 11(2):259–272, 2003. 27

[PDDH10] Quang Dung Pham, Phan-Thuan Do, Yves Deville, and Tuong-Vinh Ho. Constraint-based local search for solving non-simple paths problems on graphs: Application to the routing for network covering problem problems. In *In Proceedings of Symposium on Information and Communication Technology, SoICT2010, Hanoi, Vietnam*, pages 1–8, 2010. 33

[PDH10] Q. D. Pham, Y. Deville, and P. Van Hentenryck. 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*, pages 267–281, 2010. 33, 108, 120

[PDV09] Quang Dung Pham, Yves Deville, and Pascal Van Hentenryck. LS(Graph & Tree): A local search framework for constraint optimization on graphs and trees. *Proceedings of the 24th Annual ACM Symposium on Applied Computing (SAC'09)*, pages 1402–1407, 2009. 33

[RL04] Marc Reimann and Marco Laumanns. A hybrid aco algorithm for the capacitated minimum spanning tree problem. *Proceedings of First International Workshop on Hybrid Metaheuristics*, pages 1–10, 2004. 1

[RS95] R. Ramaswami and K. Sivarajan. Routing and wavelength assignment in all-optical networks. *IEEE/ACM Trans. Network*, 3(5):489–500, 1995. 27

[RS00] C. C. Ribeiro and M. C. Souza. Tabu search for the steiner problem in graphs. *Networks*, 36:138–146, 2000. 36

[RS02] C.C. Ribeiro and M.C. Souza. Variable neighborhood search for the degree constrained minimum spanning tree problem. *Discrete Applied Mathematics*, 118:43–54, 2002. 37

[RUW02] C. C. Ribeiro, E. Uchoa, and R. F. Werneck. A hybrid grasp with perturbations for the steiner problem in graphs. *INFORMS J. Computing*, 14(3):228–246, 2002. 36

[RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh. *HANDBOOK OF CONSTRAINT PROGRAMMING*. Elsevier Science Inc., New York, USA, 2006. 13

[Sha98] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *In Michael J. Maher and Jean-Francois Puget, editors, Principles and Practice of Constraint Programming - CP98, 4th International Conference, volume 1520 of Lecture Notes in Computer Science*, pages 417–431, 1998. 142

[Smi06] K Smilowitz. Multi-resource routing with flexible tasks: an application in drayage operation. *IIE Transactions*, pages 38(7):555–568, 2006. 1

[ST83] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *JCSS*, 26(3):362–391, 1983. 61

[ST85] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the Association for Computing Machinery*, 32(3):652–686, 1985. 61

[TMP02] Massimo Tornatore, Guido Maier, and Achille Pattavina. Wdm network optimization by ilp based on source formulation. In *IN PROCEEDINGS OF IEEE INFOCOM*, pages 1813–1821, 2002. 27

[tsp] Tsp-library, url= http://www.tsp.gatech.edu/.

[TW05] Robert E. Tarjan and Renato F. Werneck. Self-adjusting top trees. *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 813–822, 2005. 61

[TW09] Robert E. Tarjan and Renato F. Werneck. Dynamic trees in practice. *Journal of Experimental Algorithmics (JEA)*, 14(Article No.: 5), 2009. 61

[VM05] Pascal Van Hentenryck and Laurent Michel. *Constraint-based local search*. The MIT Press, London, England, 2005. 4, 9, 13, 20, 34, 53

[vrp] Vrp-library, url= http://neo.lcc.uma.es/radi-aeb/webvrp/.

[WH04] Bin Wang and Jennifer C. Hou. An efficient qos routing algorithm for quorumcast communication. *Computer Networks Journal*, 44(1):43–61, 2004. 24, 25

[YC02] Kriangchai Yaoyuenyong and Peerayuth Charnsethikul. A heuristic algorithm for the mixed chinese postman problem. *Optimization and Engineering*, 3:157–187, 2002. 2

[YCCL] Y. Ye, T. Chai, T. Cheng, and C. Lu. Algorithms for the design of wdm translucent optical networks. *Optics Express*, 11(22). 27

[YLR10] Emre Yetginer, Zeyu Liu, and George N. Rouskas. 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)*, 2010. 27

[YZL05] Bing Yang, S.Q. Zheng, and Enyue Lu. Finding two disjoint paths in a network with normalized $\alpha^+$-min-sum objective function. In *Lecture Notes in Computer Science, Volume 3827/2005*, pages 954–963, 2005. 141

[Zac99] M. Zachariasen. Local Search for the Steiner Tree Problem in the Euclidean Plane. *European Journal of Operational Research*, 119:282–300, 1999. 1

[ZJM00] Hui Zang, Jason P. Jue, and Biswanath Mukherjee. A review of routing and wavelength assignment approaches for wavelength- routed optical wdm networks. *Optical Networks Magazine*, 1(1):47–60, 2000. 1, 2, 27