LS(Graph & Tree): A Local Search Framework for Constraint Optimization on Graphs and Trees

Pham Quang Dung
University of Louvain
1348 Louvain-la-Neuve, Belgium
quang.pham@uclouvain.be

Yves Deville
University of Louvain
1348 Louvain-la-Neuve, Belgium
Yves.Deville@uclouvain.be

Pascal Van Hentenryck
Brown University
Providence, RI 02912, USA
pvh@cs.brown.edu

ABSTRACT

`LS(Graph & Tree)` is a local search framework which aims at simplifying the modeling of Constraint Satisfaction Optimization Problems on graphs (CSOP on graphs or GCSOP). Optimum Constrained Trees (OCT) problems (a subclass of CSOP on graphs) in which we need to find an optimum subtree with additional constraints of a given weighted graph arise in many real-life applications. This paper introduces the `LS(Graph & Tree)` framework and local search abstractions for OCT problems. These abstractions are applied to model and solve the edge weighted k-Cardinality Tree (KCT) problem. The modeling as well as experimental results show the significance of the abstractions.

Categories and Subject Descriptors

H.4 [**Constraint Optimization**]

Keywords

Constraint Optimization, Constrained Tree Problems, KCT, Local Search, Graph Theory

1. INTRODUCTION

Constraint Satisfaction Optimization on graphs (CSOP on graphs or GCSOP) appears in various real-life applications such as telecommunication and transportation networks [11] distributed mutual exclusion [18], bit compression for information retrieval [6], etc. Optimum Constrained Tree (OCT) problems especially arise in the telecommunication network design such as: Degree Constrained Minimum Spanning Tree (DCMST) [16, 2], Bounded Diameter Minimum Spanning Tree (BDMST) [12], Capacitated Minimum Spanning Tree Problem (CMST) [19, 1], Minimum Diameter Spanning Tree (MDST) [17], Edge-Weighted k-Cardinality Tree (KCT), [5, 7], Steiner Minimal Tree (SMT) [20, 8], Optimum Communication Spanning Tree Problems (OCST) [10], etc. For solving these problems, metaheuristic approaches like local search have been shown to be competitive in comparison with other exact techniques. Unfortunately, these local search algorithms are often large, intricate, and are tedious to design, implement and maintain.

We introduce in this paper the `LS(Graph & Tree)` framework which aims at simplifying the modeling of local search algorithms for CSOP on graphs and trees. This framework supports constraint-based architecture [13] which uses constraints and objective functions to describe and control local search, and features compositionality, modularity and reuse. Programmers do not have to pay attention to complicated data structures and algorithms on graphs (for example, the computation of the number of connected components of a graph, diameter of a tree, etc.) because these are implemented within `LS(Graph & Tree)`. Rather they can concentrate on the modeling and on the exploration over various heuristic and metaheuristic strategies for the search procedure. For each constraint appearing on the considered problem, two approaches are possible. In the first approach, called soft constraint, one relaxes this constraint, allowing solutions to have a number of constraint violations and direct the search in order to reduce these constraint violations. In the second, called hard constraint, one maintains this constraint and considers only local moves that conserve it. To do this, one has to manipulate a data structure allowing to compute at each step the set of appropriate local moves. For the constrained tree problems on graphs, most of local search algorithms in the literature follow this approach and manipulate a dynamic tree (a dynamic graph in which the *Tree* constraint is always satisfied).

In this paper, we present abstractions which are dedicated to the modeling and solving of OCT problems by local search techniques. It proposes abstractions which model dynamic trees, abstractions describing *GraphInvariant*s, *GraphConstraint*s and *GraphObjective*s which are defined over these dynamic trees.

`LS(Graph & Tree)` offers openness and extensibility. One can extend it by designing and implementing new components such as *GraphInvariant*, *GraphConstraint* or *GraphObjective*. We have implemented two versions of the abstractions: one in `COMET` [13] and one in C++. We also compare these two versions. The C++ version has been applied to model and solve the KCT problem.

The paper is organized as follows. In Section 2, we briefly introduce the `LS(Graph & Tree)` framework. Section 3 presents abstractions representing dynamic trees as well as invariants, constraints and objective functions which are defined over these dynamic trees. Section 4 gives an overview

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.
Copyright 2009 ACM 978-1-60558-166-8/09/03 ...$5.00.

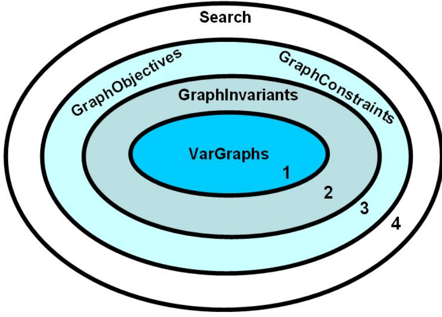

Figure 1: LS(Graph & Tree) architecture

of the framework implementation. We show in Section 5 an application of these abstractions to the modeling and solving of a particular OCT problem: the edge weighted k-Cardinality (KCT) problem. Section 6 summarizes our contributions and proposes future research directions.

2. THE LS(GRAPH & TREE) FRAMEWORK

The architecture of the `LS(Graph & Tree)` is based on the `COMET` architecture [13] and is organized into four layers depicted in Figure 1. It is composed of two main independent components: a declarative component (layers 1, 2, 3) which models the problem in terms of constraints and functions, and a search component which specifies the heuristic and meta-heuristic search algorithms. The architecture enables local search algorithms to be high-level, compositional, and modular. It is possible to add new constraints and to modify or remove existing ones, without having to worry about the global effect of these changes. It allows programmers to experiment with different search heuristics and meta-heuristics without affecting the problem modeling.

Graph variable Graph variables have already been introduced in constraint programming [9]. Graph variables are called *VarGraph*, the core of our local search framework. A *VarGraph* describes dynamic graphs (a graph G that can be modified, by adding or removing some nodes and edges, within a bound $[glb(G), lub(G)]$ [1]) over which, *GraphInvariants*, *GraphConstraints* and *GraphObjectives* are defined. *GraphInvariant* is a concept representing objects which maintain some properties of a dynamic graphs (for instance, the sum of weights of all the edges of a graph, the diameter of a tree, etc.). *GraphConstraint* and *GraphObjective* are concepts describing differentiable objects which maintain some properties (for instance, the number of violations of a constraint or the value of an objective function) of a dynamic graph. The main difference of *GraphConstraint* and *GraphObjective* from *GraphInvariant* is the available interface, allowing to query the impact of local moves (modification of the dynamic graph) on these properties. Each modification over a *VarGraph* induces a propagation which updates automatically all *GraphInvariants*, *GraphConstraints* and *GraphObjectives* defined over that *VarGraph* thanks to a dependency graph which represents the dependence between these objects. In the framework, `VarGraph` is an abstract class from which, `VarUndirectedGraph`, `VarDirectedGraph`, `VarTree` (which is detailed in Section 3), etc. are derived.

```
interface GraphConstraint{
  1.  VarGraph[]      getVarGraphs();
  2.  var{int}        violations();
  3.  var{boolean}    isTrue();
  4.  int getAddNodeDelta(VarGraph g, Node v);
  5.  int getRemoveNodeDelta(VarGraph g, Node v);
  6.  int getAddEdgeDelta(VarGraph g, Edge e);
  7.  int getRemoveEdgeDelta(VarGraph g, Edge e);
  8.  int getReplaceEdgeDelta(VarGraph g, Edge oe, Edge ne);
  ...
}
```

Figure 2: Interface `GraphConstraint` (partial description)

The Neighborhood and Local move The Neighborhood of a solution is a set of neighboring solutions of the current solution which are generated by a local change over that solution. A local move is the action of taking a local change over the current solution to generate a new solution. Naturally, a local change over a dynamic graph is an addition or a removal of one or some nodes (edges) or the replacement of some nodes (edges) by other nodes (edges). In this framework, we consider the following basic local moves on graphs: *addNode*, *removeNode*, *addEdge*, *removeEdge*, and *replaceEdge*. We can also combine some of the above actions to generate more complex local moves (i.e., in variable neighborhood search).

There exist various constraints on graphs such as: $Tree(G)$ which specifies that the *VarGraph* G is an undirected tree; $SimplePath(G, s, t)$ which specifies that the *VarGraph* G is a simple path from node s to node t; $Connected(G)$ which specifies that the *VarGraph* G is connected, etc. Such constraints have already been introduced in constraint programming [9]. In order to foster the compositionality and reuse, all soft graph constraints implement the same interface `GraphConstraint` (see Figure 2). Moreover, this makes it possible to design constraint combinators [14] (e.g. *GraphConstraintSystem*). The method `getVarGraphs` (line 1) gives access to a list of `VarGraph`s within the *GraphConstraint*. The violations and the truth value of the *GraphConstraint* are returned by the methods `violations` (line 2) and `isTrue` (line 3). The remaining methods (lines 4-8) provide the differentiable API of the *GraphConstraint*. They make it possible to query the variation of the violations under various update actions (local moves). The *Tree* constraint *Tree(G)*, for instance, can thus be violated by a solution. But the interface allows to query its violations and to select a neighbor reducing the violations of this constraint.

3. TREE VARIABLES

This section presents the *VarTree* (also called tree variable) abstraction providing a hard *Tree* constraint in `LS(Graph &Tree)`. By using this abstraction, the *Tree* constraint is thus satisfied all along the search.

3.1 Basic local moves for a *VarTree*

Given a dynamic unrooted tree T, 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 $e = (u, v) \in E(lub(T)) \setminus E(T)$ can be added to T if T is empty, or if there is exactly one node u or v in the tree T: $u \in V(T)$ `XOR`

[1] Given a dynamic graph G, we denote $glb(G)$, $lub(G)$ respectively the lower bound and upper bound of G, we also denote $V(G), E(G)$ respectively the set of nodes and the set of edges of G.

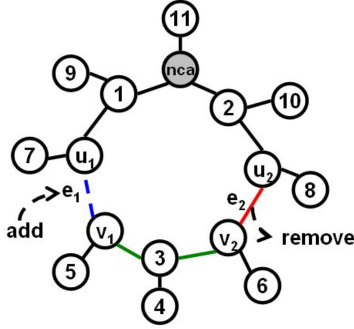

Figure 3: replace an edge by another edge

$v \in V(T)$. This edge is called *insertable* edge. The set $V(T)$ is also updated.

2. **remove edge action** An edge $e = (u, v) \in E(T)$ can be removed from T if one node u or v is a leaf of T: $deg_T(u) = 1 \vee deg_T(v) = 1$. This edge is called *removable* edge. The set $V(T)$ is also updated.

3. **replace cycle edge action** An edge e_2 of T can be replaced by another edge $e_1 = (u_1, v_1) \in E(lub(T)) \setminus E(T)$ with $u_1, v_1 \in V(T)$ conserving the tree property in the following case: e_2 is not a leaf of T, thus the removal of e_2 from T disconnects T, and then the insertion of e_1 reconnects two subtrees generated by the removal of e_2 from T (see Figure 3). The edge e_1 is called *replacing* edge, and e_2 is called *replaceable* edge of e_1. In other words, the insertion of e_1 creates a cycle containing e_2 and the removal of e_2 removes the cycle and restores the tree property. The set of nodes of T is unchanged by this replacement.

3.2 The Abstractions

The objective of the *VarTree* abstraction is to provide an easy way to navigate in the neighborhood as well as to perform local moves while maintaining the tree property. Programmers do not have to manipulate sophisticated data structures and algorithms to maintain the sets of *insertable*, *removable*, *replacing* and *replaceable* edges. Rather, they can focus on exploring various metaheuristics strategies. The key center is the class `VarTree` which is partially described in the following `COMET` snippet:

```
1:  class VarTree extends VarGraph{
2:    set{Edge} getInsertableEdges();
3:    set{Edge} getRemovableEdges();
4:    set{Edge} getReplacingEdges();
5:    set{Edge} getReplaceableEdges(Edge e);
6:  }
```

The methods give access to the lists of *insertable* edges (line 2), *removable* edges (line 3), *replacing* edges (line 4) and *replaceable* edges of a given edge `e` (line 5). Over this abstraction, we can reuse some *GraphConstraint*s, *GraphObjective*s which are defined over `VarGraph`, such as `Weight(G, ind)` [2] which represents an objective function specifying the sum of weights of all the edges of the given *VarGraph* `G`, and `DegreeAtMost(G,d)` which represents a constraint specifying that the degree of each node of the *VarGraph* `G` cannot exceed `d`.

[2]In this framework, each edge or node of a graph can have multiple properties. `ind` is the index of the considered property of each edge.

```
1:  void stateModel(){
2:    LSGraphSolver ls = new LSGraphSolver();
3:    UndirectedGraph g = new UndirectedGraph("graph.inp");
4:    VarSpanningTree tree = new VarSpanningTree(ls,g);
5:    GraphConstraint diameterCstr = new DiameterAtmost(ls,tree,D);
6:    Weight weight = new Weight(ls,tree,0);
7:    GraphObjective goc = alpha*diameterCstr + beta*weight;
8:    ls.close();
9:  }
10:  void localmove(){
11:    selectMin(ei in tree.getReplacingEdges(),
        eo in tree.getReplaceableEdges(ei))
           (goc.getReplaceEdgeDelta(tree,eo,ei)){
12:      ls.replaceEdge(tree,eo,ei);
13:    }
14:}
```

Figure 4: The BDMST problem : modeling and greedy local move

The `VarSpanningTree` abstraction is an extension of `VarTree` representing a dynamic spanning tree of a given undirected graph. This abstraction is dedicated to model and solve Optimum Constrained Spanning Tree problems, for instance, DCMST [16, 2], BDMST [12], OCST [10]. In this abstraction, only **replace cycle edge action** is allowed, *insertable* edges list and *removable* edges list are hence empty.

The `VarRootedTree` abstraction is an extension of `VarTree` representing the dynamic rooted tree (with a fixed node representing the root of the tree) of a given directed or undirected graph. This abstraction can be used to model and solve CMST problem [19, 1]. Over this abstraction, some *GraphObjective*s are defined, for instance, `Height(T,v)` representing the *height* of the node `v` in the `VarRootedTree T`; `Capacity(T,v,ind)` representing the sum of weights of all the nodes of the subtree of the `VarRootedTree T` rooted at node `v` (`ind` is the index of considered weight of each node.)

3.3 Example

In order to illustrate these abstractions, we give an example of a modeling for BDMST [12]. Given an undirected weighted graph G and an integral value D, the BDMST problem is to find a minimum spanning tree of G whose *diameter* (the maximal number of edges on any path of the tree) cannot exceed D.

For elegance, all the example codes here are presented in `COMET`. The modeling is presented in Figure 4 in which line 2 initializes an `LSGraphSolver` object `ls` which manages all the *VarGraph*, *VarTree*, *GraphInviant*s, *GraphConstraint*s and *GraphObjective*s and relations (dependency graph) between these objects. The input graph `g` and a `VarSpanningTree tree` are created and initialized in lines 3-4. The *GraphConstraint* `diameterCstr` constrains the diameter of the tree. It is initialized in line 5. BDMST is a problem with a constraint (over the diameter) to be satisfied and an objective function (the total weight of the tree) to be minimized. These are then combined into a global objective function with weights `alpha` and `beta` (line 7). Lines 10-14 show a simple greedy local move for BDMST. It selects a *replacing* edge `ei` and a *replaceable* edge `eo` of `ei` such that `goc` reduces most when replacing `eo` by `ei` (line 11). Line 12 performs the move which also updates automatically all *GraphConstraint*s and *GraphObjective*s defined over `tree` thanks to a dependency graph maintained in `ls`.

4. IMPLEMENTATION

For implementing a differentiable object likes *GraphConstraint* (representing soft constraints) or *GraphObjective*, we have designed a dedicated data structure and (incremental) algorithms allowing to efficiently maintain the considered properties and to query the variations of these properties under various local moves. To implement an abstraction representing a hard constraint likes *VarTree*, we maintain an auxiliary data structure allowing to efficiently navigate in the neighborhood while maintaining this hard constraint. For lack of space, we do not present here the implementation of all abstractions and their complexities, but we summarize the implementation of the *VarTree* abstraction which is essential in the framework.

In order to facilitate their manipulation, all `VarTree` abstractions including `VarSpanningTree, VarRootedTree` are stored as rooted tree with a special node representing the root of the tree. Each node (except the root) of the tree has exactly one father node. On this rooted tree, the **nearest common ancestor** [3] of each pair of two nodes are fundamental for the **replace cycle edge action**. Given a *replacing* edge $e = (u, v)$, we can easily compute the list of *replaceable* edges of e by iterating all edges from u and v towards $nca(u, v)$ on the current tree. Nearest common ancestors are also helpful when implementing some *GraphConstraint*s and *GraphObjective*s on *VarTree*, for instance, `LongestPath`, `Capacity`. Berkman and Vishkin [4] proposed an algorithm to compute the nearest common ancestor of all pairs of nodes of the tree. It was reused in [3]. An intermediate data structure is precomputed in $O(n\ log\ n)$; each query $nca(u, v)$ is then computed in $O(1)$ time. We extend this algorithm with an incremental implementation. Our incremental algorithm does not improve the time complexity in the worst case ($O(n\ log\ n)$ for each local move) but is efficient in practice. Experimental results, not reported here for lack of space, showed that incremental update is 2 times faster than recomputation from scratch.

We implemented two versions of the framework: one in C++ (C++ version) and one in `COMET` [13] (`COMET` version) (each version contains about 10,000 lines of code). `COMET` is a novel programming language providing a number of innovative control abstractions for local search. By using the `COMET` version, we gain the facility when carrying out various metaheuristics strategies with the nice built-in control abstractions of `COMET`. We also benefit from the incremental variables, built-in invariants, constraints and objectives that simplify the modeling of local search algorithms. The local search programs are also short and concise. In contrast, by using the C++ version, we gain the performance (the `VarTree` implementation in `COMET` is about 2.5 times slower than in C++), but from a programming standpoint, it is more sophisticated. An integration of the C++ implementation within the `COMET` solver (written in C++) would make the `COMET` version as efficient as the C++ version.

5. EXPERIMENTS

We present in this section an application of the framework to the modeling and solving of the KCT problem [5, 7].

[3]Given a rooted tree T and two nodes $u, v \in V(T)$, the **nearest common ancestor** of u and v denoted by $nca(u, v)$ is the ancestor of u and v that located farthest from the root of T.

Given an edge-weighted graph $G = (V, E)$ and a value k ($1 \leq k \leq |V| - 1$), KCT consists of finding a subtree of G with exactly k edges, such that the sum of weights of all the edges is minimal. KCT has gained considerable interest of many researchers. KCTLib [15] proposes three different metaheuristic approaches for KCT in which a Tabu Search exploits a neighborhood structure using the two first local moves described in Section 3.1. Our third update action (**replace cycle edge action**) is thus not exploited. A C++ implementation of this Tabu Search (denoted by TS_KCT) is also distributed in this library. We implement a Tabu Search using `VarTree` abstraction in the C++ version of `LS(Graph & Tree)` framework (denoted by MTS_KCT_VT) exploiting the same Tabu Search schema than TS_KCT, but exploring a larger neighborhood by including the **replace cycle edge action**.

We compare our MTS_KCT_VT with TS_KCT on two experiments. The first experiment is carried out over a subset of standard benchmarks on KCTLib which are sparse graphs. The second experiment is performed over new dense graphs which are complete euclidean graphs generated randomly as follows: we generate randomly n nodes with coordinates (x, y) where x and y are generated by a uniform distribution in the interval [1..500], the weight of each edge is the euclidean distance between their endpoints. All the experiments are carried out on an Intel Pentium R dual-core processor 1.60GHz and 512MB of memory with Ubuntu 7.10.

In the first experiment, we test over 35 4-regular graphs whose order varies from 25 to 1000 (the value of k is 20 for this benchmark). The algorithm is executed 20 times for each instance with the time limit of 5 minutes. For lack of space, the complete results are not reported here. These instances are not difficult and all the runs return the best known solution. On half of the instances, MTS_KCT_VT is faster, with an average speed-up of 12. For the other instances, TS_KCT is faster with an average speed-up of 10. The two algorithms are thus complementary.

In the second experiment, two graphs of order 500 of the new data set (complete euclidean graphs) are chosen. For each of these two graphs, we experiment with k = 200, 300, 400, 480. Each of the two tabu searches was run 20 times for each problem instance (combination of a input graph and a value of k). The time limit is 60 minutes. The experimental results are shown in Table 1. For each instance, columns 3-6 report the minimal, maximal, average values and the standard deviation of the objective function of best found solution (in 20 runs) by TS_KCT, and the columns 7-8 present the average value (in 20 runs) of the earliest iteration and the earliest time (in sec.) obtaining this solution. The same information of MTS_KCT_VT are presented in the columns 9-14. The experimental results show that MTS_KCT_VT finds better solution than TS_KCT thanks to an exploration over a larger neighborhood. Moreover, the average value of the objective function of the best found solutions and the average value of the number of iterations for reaching these solutions of MTS_KCT_VT are smaller than TS_KCT. The standard deviations of the objective function found by two tabu searches show that among 20 random runs, the best solutions found by TS_KCT vary much more than the ones found by MTS_KCT_VT. However, the size of neighborhood in MTS_KCT_VT is large, hence slowing down the neighborhood exploration, but the average time is often bet-

Graph gEcl500	k	TS_KCT[15]						MTS_KCT_VT						p_val
		min	Max	avg	stdev.	avgIt	avgT	min	Max	avg	stdev.	avgIt	avgT	
in3	200	2459	2936	2657.3	146.64	1173.0	2270.6	**2422**	**2586**	**2479.9**	**41.09**	**542.9**	2317.8	5.50e-05
	300	3857	4253	4067.3	111.83	1224.9	2500.6	**3807**	**3960**	**3899.1**	**52.25**	**224.6**	1446.6	7.79e-06
	400	5332	5773	5519.6	117.36	1500.2	2644.8	**5206**	**5326**	**5299.1**	**23.59**	**267.8**	1841.2	9.87e-08
	480	6711	6882	6786.0	58.04	3470.3	1803.2	**6693**	**6700**	**6696.9**	**1.82**	**359.1**	2661.6	1.51e-06
in5	200	2483	3102	2765.1	160.34	1135.3	1975.3	**2391**	**2599**	**2471.2**	**46.07**	**486.2**	2139.5	1.97e-07
	300	3857	4457	4135.9	126.81	1208.5	2536.9	**3827**	**3947**	**3864.8**	**35.17**	**300.3**	1946.0	1.64e-08
	400	5518	5839	5654.6	79.82	1401.9	2362.4	**5403**	**5470**	**5447.9**	**18.95**	**249.9**	1953.1	5.47e-10
	480	6856	7037	6931.5	53.40	4016.9	1823.3	**6815**	**6842**	**6827.1**	**11.36**	**223.1**	1752.2	5.43e-08

Table 1: Comparison between TS_KCT and MTS_KCT_VT on random euclidean graphs

ter. The p-values of the statistical student-hypothesis test (last column) indicate that there is enough evidence to support the conclusion that the objective values reached with MTS_KCT_VT are significantly smaller than with TS_KCT. In conclusion, MTS_KCT_VT and TS_ KCT are both good on simple problems, but complementary. However, on complex dense graphs, our MTS_KCT_VT is better than the TS_KCT.

6. CONCLUSIONS

We presented in this paper the `LS(Graph & Tree)` framework which aims at simplifying the modeling and solving of Constraint Satisfaction Optimization Problems on graphs and trees by local search. `LS(Graph & Tree)` supports both *VarGraph* (and its extensions) and standard `COMET` incremental variables (e.g. `var{int}`). However, in the current version, one cannot combine *VarGraph* and standard `COMET` variables in the same constraint or objective function. This limitation can however be lifted without computational overhead. The proposed computation model features constraint-based architecture [13] and has number of benefits. From a programming standpoint, local search algorithms are short and concise. From a computational standpoint, some built-in components are efficiently implemented with auxiliary data structures and (incremental) algorithms. One can implement efficiently existing local search algorithms without having to manipulate complex data structures as well as implementing sophisticated algorithms on graphs. From a language standpoint, the computational model features compositionality, modularity and reuse. It is easy to add new constraints or objective functions to the model without modifying the search component. We can also explore various heuristics and metaheuristics over a same model. The development time of programs is much reduced allowing to quickly implement various local search algorithms. We also compared two implementations of our framework; one in C++ and one in `COMET`. The `VarTree` abstractions have been applied to solve the KCT problem. The problem modeling and experimental results show the significance of the abstractions. Our future work will extend `LS(Graph & Tree)` to other structures such as directed acyclic graphs, paths, etc., and will focus on the design and implementation of local search abstractions dedicated for the modeling and solving Optimum Constrained Path problems on graphs.

Acknowledgments

We would like to thank Christian Blum and Maria José Blesa Aguilera who have kindly provided the C++ software for solving the KCT problem. Thanks to the reviewers for their helpful comments.

7. REFERENCES

[1] R. K. Ahujaa, J. B. Orlinb, and D. Sharma. A composite very large-scale neighborhood structure for the capacitated minimum spanning tree problem. *Operations Research Letters 31*, pages 185–194, 2003.

[2] R. Andrade, A. Lucena, and N. Maculan. Using lagrangian dual information to generate degree constrained spanning trees. *Discrete Applied Mathematics*, pages 703–717, 2006.

[3] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms 57*, pages 75–94, 2005.

[4] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, 1993.

[5] 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.

[6] A. Bookstein and S. T. Klein. Compression of correlated bit-vectors. *Information Systems*, 16(4):387–400, 1991.

[7] M. Chimani, M. Kandyba, I. Ljubic, and P. Mutzel. Obtaining optimal k-cardinality trees fast. *10th Workshop on Algorithm Engineering and Experiments 2008, San Francisco (ALENEX08), SIAM*, pages 27–36, 2008.

[8] M. de Aragão, E. Uchoa, and R. 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.

[9] 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.

[10] T. Fischer. Improved local search for large optimum communication spanning tree problems. *In MIC'2007 - 7th Metaheuristics International Conference*, 2007.

[11] M. Gomes, R. Andrade, C. Santiago, and N. Maculan. Spanning tree algorithms to some hard combinatorial problems. *in : Proceedings of Optimization Days, Montreal/Canada*, pages 83–84, 1997.

[12] M. Gruber, J. van Hemert, and G. 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.

[13] P. V. Hentenrych and L. Michel. *Constraint-based local search*. The MIT Press, London, England, 2005.

[14] P. V. Hentenryck, L. Michel, and L. Liu. Constraint-based combinators for local search. *International Conference on Constraint Programming, LNCS 3258*, pages 47–61, 2004.

[15] KCTLib. *http://iridia.ulb.ac.be/ cblum/kctlib/*, 2003.

[16] M. Krishnamoorthy, A. T. Ernst, and Y. M. Sharaiha. Comparison of algorithms for the degree constrained minimum spanning tree. *Journal of Heuristics*, pages 587–611, 2001.

[17] E. Nardelli and G. 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.

[18] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 7(1):61–77, 1989.

[19] M. Reimann and M. Laumanns. A hybrid aco algorithm for the capacitated minimum spanning tree problem. *Proceedings of First International Workshop on Hybrid Metaheuristics*, pages 1–10, 2004.

[20] M. Zachariasen. Local Search for the Steiner Tree Problem in the Euclidean Plane. *European Journal of Operational Research*, 119:282–300, 1999.