Constraint-Based Local Search for Constrained Optimum Paths Problems

Quang Dung PHAM[1], Yves DEVILLE[1], Pascal Van HENTENRYCK[2]

[1] Université catholique de Louvain B-1348 Louvain-la-Neuve, Belgium
{quang.pham,yves.deville}@uclouvain.be
[2] Brown University, Box 1910 Providence, RI 02912, USA
pvh@cs.brown.edu

Abstract. Constrained Optimum Path (COP) problems arise in many real-life applications and are ubiquitous in communication networks. They have been traditionally approached by dedicated algorithms, which are often hard to extend with side constraints and to apply widely. This paper proposes a constraint-based local search (CBLS) framework for 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 COP applications at a high level of abstraction, while cleanly separating the model and the search procedure. The main technical contribution is a connected neighborhood based on rooted spanning trees to find high-quality solutions to COP problems. The framework, implemented in `COMET`, is applied to Resource Constrained Shortest Path (RCSP) problems (with and without side constraints) and to the edge-disjoint paths problem (EDP). Computational results show the potential significance of the approach.

1 Introduction

Constrained Optimum Path (COP) problems appear in many real-life applications, especially in communication and transportation networks (e.g., [5]). They aim at finding one or more paths from some origins to some destinations satisfying some constraints and optimizing an objective function. For instance, in telecommunication networks, routing problems supporting multiple services involve the computation of paths minimizing transmission costs while satisfying bandwidth and delay constraints [3, 6]. Similarly, the problem of establishing routes for connection requests between network nodes is one of the basic operations in communication networks and it is typically required that no two routes interfere with each other due to quality-of-service and survivability requirements. This problem can be modeled as edge-disjoint paths problem [4]. Most of COP problems are NP-hard. They are often approached by dedicated algorithms, such as the Lagrangian-based branch and bound in [3] and the vertex labeling algorithm from [7]. These techniques exploit the structure of constraints and objective functions but are often difficult to extend and reuse.

This paper proposes a constraint-based local search (CBLS) [10] framework for COP applications to support the compositionality, reuse, and extensibility at the core of CBLS and CP systems. It follows the trend of defining domain-specific CBLS frameworks, capturing modeling abstractions and neighborhoods for classes of applications exhibiting significant structures. The COP framework can also be viewed as an extension of the `LS(Graph & Tree)` framework [8] for those applications where the output of the optimization model is one or more elementary paths (i.e., paths with no repeated nodes). As is traditional for CBLS, the resulting COP framework allows the model to be compositional and easy to extend, and provides a clean separation of concerns between the model and the search procedure. Moreover, the framework captures structural moves that are fundamental in obtaining high-quality solutions for COP applications. The key technical contribution underlying the COP framework 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 availability of high-level abstractions (the "what") and the underlying connected neighborhood for elementary paths (the "how") make the COP framework particularly appealing for modeling and solving complex COP applications. The COP framework, implemented in `COMET`, was evaluated experimentally on two classes of applications: Resource-Constrained Shortest Path (RCSP) problems with and without side constraints and Edge-Disjoint Path (EDP) problems. The experimental results show the potential of the approach.

The rest of this paper is organized as follows. Section 2 gives the basic definitions and notations. Section 3 specifies our novel neighborhoods for COP applications and Section 4 presents the modeling framework. Section 5 applies the framework to two various COP applications and Section 6 concludes the paper.

2 Definitions and Notations

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

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 $u\ Dom_{tr}\ v$.
- $path_{tr}(u, v)$ for the path from u to v in t ($u, v \in V(tr)$).
- $nca_{tr}(u, v)$ for the nearest common ancestor of two nodes u and v. In other words, $nca_{tr}(u, v)$ is the common dominator of u and v such that there is no other common dominator of u and v that is a descendant of $nca_{tr}(u, v)$.

3 The COP Neighborhoods

A neighborhood for COP problems defines the set of paths that can be reached from the current solution. To obtain a reasonable efficiency, a local-search algorithm must maintain incremental data structures that allow a fast exploration of this neighborhood and a fast evaluation of the impact of the moves (differentiation). The key novel contribution of our COP framework is to use a rooted spanning tree to represent the current solution and its neighborhood. It is based on the observation that, given a spanning tree tr whose root is t, the path from a given node s to t in tr is unique. Moreover, the spanning tree implicitly specifies a set of paths that can be reached from the induced path and provides 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 space reasons, 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: $path_{tr}(s) =< v_1, v_2, ..., v_k >$ in which $s = v_1, t = v_k$ and $v_{i+1} = fa_{tr}(v_i)$, $\forall i = 1, \dots, k-1$. By maintaining RSTs for COP problems, our framework avoids an explicit representation of paths and enables the definition of an 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.

The Basic Neighborhood We now consider the definition of our COP neighborhood. We first show how to update an RST tr over (g, s, t) to generate a new rooted spanning tree tr' over (g, s, t) which induces a new path from s to t in g: $path_{tr'}(s) \neq path_{tr}(s)$.

Given an RST over (g, s, t), an edge $e = (u, v)$ such that $e \in E(g) \setminus E(tr)$ is called a *replacing* edge of tr and we denote by $rpl(tr)$ the set of *replacing* edges of tr. An edge e' belonging to $path_{tr}(u,v)$ is called a *replacable* edge of e and we denote by $rpl(tr,e)$ the set of *replacable* edges of e. Intuitionally, a *replacing* edge e is an edge that is not in the tree tr but that can be added to tr. This edge insertion creates a cycle C and all the edges of this cycle except e are *replacable* edges of e. Let tr be an RST over (g,s,t), e a *replacing* edge of tr and e' a *replacable* edge of e. We consider the following traditional edge replacement action [1]:

1. Insert the edge $e=(u,v)$ to tr. This creates an undirected graph g' with a cycle C containing the edge e'.
2. Remove e' from g'.

The application of these two actions yields a new rooted spanning tree tr' of g, denoted $tr' = rep(tr,e',e)$. The neighborhood of tr could then be defined as

$$N(tr) = \{tr' = rep(tr,e',e) \mid e \in rpl(tr), e' \in rpl(tr,e)\}.$$

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 $N^k(tr) \subseteq N(tr)$ such that $path_{tr'}(s) \neq path_{tr}(s), \forall tr' \in N^k(tr)$.

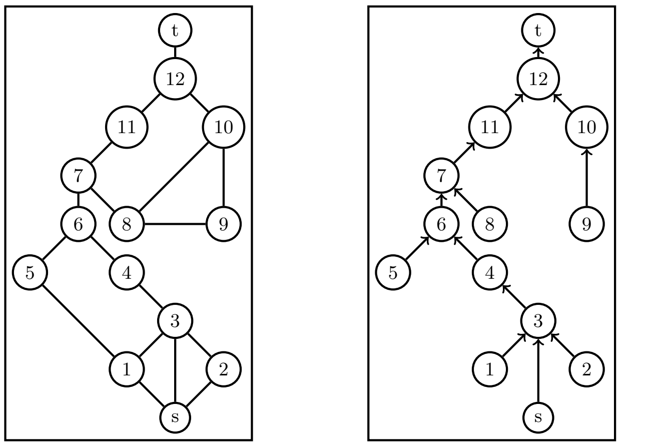
We first give some notations to be used in the following presentation. Given an RST tr over (g,s,t) and a *replacing* edge $e=(u,v)$, the nearest common ancestors of s and the two endpoints u, v of e are both located on the path from s to t. We denote by $lownca_{tr}(e,s)$ and $upnca_{tr}(e,s)$ the nearest common ancestors of s on the one hand and one of the two endpoints of e on the other hand, with the condition that $upnca_{tr}(e,s)$ dominates $lownca_{tr}(e,s)$. We denote by $low_{tr}(e,s)$, $up_{tr}(e,s)$ the endpoints of e such that $nca_{tr}(s, low_{tr}(e,s)) = lownca_{tr}(e,s)$ and $nca_{tr}(s, up_{tr}(e,s)) = upnca_{tr}(e,s)$. Figure 1 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. $lownca_{tr}((8,10)) = 7$ and $upnca_{tr}((8,10)) = 12$ because 12 Dom_{tr} 7; $low_{tr}((8,10)) = 8$; $up_{tr}((8,10)) = 10$.

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 $tr^1 = rep(tr,e',e)$. We have that $path_{tr^1}(s) \neq path_{tr}(s)$ if and only if (1) $su \neq sv$ and (2) $e' \in path_{tr}(sv,su)$, where $su = upnca_{tr}(e,s)$ and $sv = lownca_{tr}(e,s)$.*

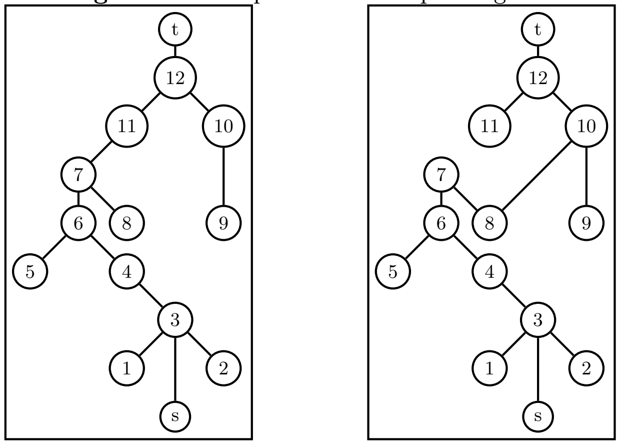
A *replacing* edge e of tr satisfying condition 1 is called a *preferred replacing* edge and a *replacable* edge e' of e in tr satisfying condition 2 is called *preferred replacable* edge of e. We denote by *prefRpl*(tr) the set of *preferred replacing* edges of tr and by *prefRpl*(tr,e) the set of *preferred replacable* edges of the *preferred replacing* edge e on tr. The basic COP neighborhood of an RST tr is defined as

$$N^1(tr) = \{tr' = rep(tr,e',e) \mid e \in prefRpl(tr), e' \in prefRpl(tr,e)\}.$$



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

Fig. 1. An Example of Rooted Spanning Tree



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

Fig. 2. Ilustrating a Basic Move

The action $rep(tr, e', e)$ is called a basic move and is illustrated in Figure 2. In the current tree tr (see Figure 2a), 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: $< s$, 3, 4, 6, 7, 11, 12, $t >$. The path induced by tr' is: $< s$, 3, 4, 6, 7, 8, 10, 12, $t >$ (see Figure 2b).

Basic moves ensure that the neighborhood is connected.

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

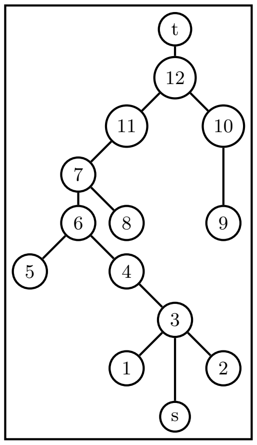
Proof. The proposition is proved by showing how to generate that instance tr^k. This can be done by Algorithm 1. The idea is to travel the sequence of nodes of $\mathcal{P}$ on the current tree tr. Whenever we get stuck (we cannot go from the current node x to the next node y of $\mathcal{P}$ by an edge (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 1 always terminates and returns a rooted spanning tree tr inducing $\mathcal{P}$. Variable S (line 1) stores the set of traversed edges.

Algorithm 1: Moves

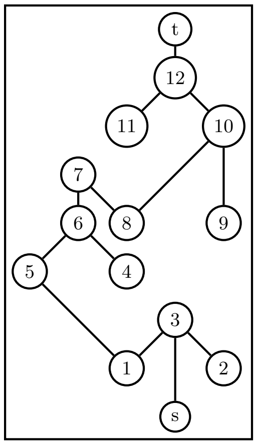
```
Input: An instance $tr^0$ of $RST$ on $(g, s, t)$ and a path $\mathcal{P}$ on $g$, $s$ =
       firstNode($\mathcal{P}$), $t$ = lastNode($\mathcal{P}$)
Output: A tree inducing $\mathcal{P}$ computed by taking $k \leq l$ basic moves ($l$ is the
        length of $\mathcal{P}$)
1  $S \leftarrow \emptyset$;
2  $tr \leftarrow tr^0$;
3  $x \leftarrow$ firstNode($\mathcal{P}$);
4  while $x \neq$ lastNode($\mathcal{P}$) do
5      $y \leftarrow$ nextNode($x, \mathcal{P}$);
6      if $(x, y) \notin E(tr)$ then
7          $ei \leftarrow (x, y)$;
8          $eo \leftarrow$ replacable edge of $ei$ that is not in $S$;
9          $tr \leftarrow rep(tr, eo, ei)$;
10     $S \leftarrow S \cup \{(x, y)\}$;
11     $x \leftarrow y$ ;
12 return $tr$;
```

Neighborhood of Independent Moves It is possible to consider more complex moves by applying a set of independent basic moves. Two basic moves are independent if the execution of the first one does not affect the second one and vice versa. The sequence of basic 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 $tr_{j+1} = rep(tr_j, e'_j, e_j)$, $j = 1, 2, ..., k - 1$. It is feasible if the basic moves are feasible, i.e., $e_j \in$ *prefRpl*(tr_j) and $e'_j \in$ *prefRpl*(tr_j, e_j), $\forall j = 1, 2, ..., k$.

Proposition 3. *Consider k basic 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 basic moves are independent.*



a. The current tree tr



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

Fig. 3. Illustrating a Complex Move

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

It remains to find some criterion to determine whether two basic 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 $u_2 = up_{tr}(e_2, s)$.*

Figure 3 illustrates a complex move. In tr, two *preferred replacing* edges (1,5) and (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: $<$ s, 3 ,1, 5, 6, 7, 8, 10, 12, $t >$ which is actually the path: $path_{tr}(s, 1) + path_{tr}(5, 8) + path_{tr}(10, t)$.

4 The COP Modeling Framework

Our COP modeling framework is implemented in `COMET` as an extension of the `LS(Graph & Tree)` framework which provides graph invariants, graph constraints, and graph objectives [8]. Graph invariants maintain properties of dynamic graphs, such as the sum of weights of all the edges and the diameter of a tree, etc. Graph constraints and graph objectives are differentiable objects which maintain some properties of a dynamic graphs (for instance, the number

```
1. LSGraphSolver ls();
2. VarPath path(ls,g,s,t);
3. PreferredReplacingEdges prefReplacing(path);
4. PreferredReplacableEdges prefReplacable(path);
...
9. int d = MAXINT;
10. forall(ei in prefReplacing.getSet())
11.     forall(eo in prefReplacable.getSet(ei))
12.         d = min(d,C.getReplaceEdgeDelta(path,eo,ei));
```

Fig. 4. Exploring the Basic Neighborhood

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.

Our COP modeling framework introduces a new type of variable `VarPath` to model elementary paths. A path variable *path(g,s,t)* encapsulates an RST over (g, s, t) and may appear in a variety of constraints and objectives. For instance, `PathCostOnEdges(path,k)`, where `k` is the index of a considered weight on the edges of a graph, maintains the total weight accumulated along the path *path* from s to t, `PathEdgeDisjoint(Paths)` is a graph constraint defined over an array of paths that specifies that these paths are mutually edge-disjoint, while `MinEdgeCost(path,k)`, `MaxEdgeCost(path,k)` maintain the minimal and maximal weight of edges on the same path. `NodesVisited(path,S)` maintains the number of nodes of `S` visited by *path*. These abstractions are examples of graph objectives which are fundamental when modeling COP problems. For example, in QoS, we consider shortest path from an origin to a destination with constraints over bandwidth which is defined to be the minimum weight of edges on the specified path. As usual in CBLS, the objectives can be combined with traditional arithmetic operators (with +,-,* operators) and used in constraints expressed with relational operators.

Figure 4 illustrates the COP framework with a simple snippet to explore the basic neighborhood. Line 1 initializes a `LSGraphSolver` object `ls` which manages all the *VarGraph*, *VarPath*, graph invariants, graph constraints and graph objectives objects. Line 2 declares and initializes randomly a `VarPath` variable. This variable encapsulates an RST over (g, s, t) which evolves during the local search. `prefReplacing` and `prefReplacable` are graph invariants which maintain the set of *preferred replacing* edges and *preferred replacable* edges (lines 3-4). Lines 9–12 explore the basic neighborhood to find the best basic moves with respect to a graph constraint `C`. The `getReplaceEdgeDelta` (line 12) method returns the variation of the number of violations of `C` when the *preferred replacable* edge `eo` is replaced by the *preferred replacing* edge `ei` on the RST representing `path`.

5 Applications

5.1 The Resource Constrained Shortest Path (RCSP) problem

The Resource constrained shortest path problem (RCSP) [3] 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 [3] over instances from the OR-Library [2]. Given a directed graph $G = (V, E)$, each arc e is associated with a length $c(e)$ and a vector $r(e)$ 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:

min $f(\mathcal{P})$
s.t. $L \leq r(\mathcal{P}) \leq U$
$\mathcal{P}$ is elementary path from s to t on G.

The RCSP problem with only constraints on the maximum resources consumed is also considered in [5, 7]. The algorithm based on Lagrangian relaxation and enumeration from [5] and the vertex-labeling algorithm combining with several preprocessing techniques in [7] are known to be state-of-the-art for this problem. We give a Tabu search model (RCSP_TABU) for solving the RCSP problem considering both constraints of minimum and maximum resources consumed. This problem is rather pure and does not highlight the benefits of our framework but it is interesting as a starting point and a basis for comparison.

The RCSP Modeling The model using the COP framework is as follows:

```
void stateModel{
    LSGraphSolver ls();
    VarPath path(ls,g,s,t);
    range Resources = 1..K;
    GraphObjective go[Resources];
    forall(k in Resources)
        go[k] = PathCostOnEdges(path,k);
    PathCostOnEdges len(path,0);
    GraphConstraintSystem gcs(ls);
    forall(k in Resources){
        gcs.post(L[k] <= go[k]);
        gcs.post(go[k] <= U[k]);
    }
    gcs.close();
    GraphObjectiveCombinator goc(ls);
    goc.add(len,1);
16.  goc.add(gcs,1000);
17.  goc.close();
18.  PreferredReplacingEdges prefReplacing(path);
19.  PreferredReplacableEdges prefReplacable(path);
20.  ls.close();
21.}
```

Line 1 declares a `LSGraphSolver ls`. A `VarPath` variable representing an *RST* over (g, s, t) is declared and initialized in line 2. Lines 3–6 create `K` graph objectives representing resources consumed in traversing the path from `s` to `t` where `PathCostOnEdges(ls,path,k)` (line 6) is a modeling abstraction representing the total weights of type k accumulated along the path from `s` to `t`[3]. Variable `len` represents the length of the path from `s` to `t` (line 7). Lines 9–12 initialize and post to the `GraphConstraintSystem gcs` (line 8) the constraints over resources consumed in traversing the path from `s` to `t`.

In this model, we combine the graph constraint `gcs` with coefficient 1000 and the graph objective `len` with coefficient 1 in a global `GraphObjectiveCombinator goc` to be minimized (lines 14–17). We introduce two graph invariants `prefReplacing` and `prefReplacable` to represent the set of *preferred replacing* edges of `path` and the sets *preferred replacable* edges of all *preferred replacing* edges of `path` (lines 18–19).

The search procedure not described here is based on tabu search on the neighborhood N^2 because the exploration on basic neighborhood N^1 gave poor results. At each local move, we consider the best neighbor which minimizes `goc`. We take this neighbor if it improves the current solution. Otherwise, a random neighbor which is not tabu will be taken. Solutions are made tabu by putting the edges appearing in the replacements into two tabu lists: one list for storing the edges to be added and another one for storing the edges to be removed. The length of these lists are set to be the number of vertices divided by 5.

Experimental Results We compare the model with the algorithm described in [3] over the benchmarks from OR-Library [2] and over a modification of these benchmarks. The original benchmarks contains 24 instances whose orders vary from 100 to 500 and the number of resources are 1 or 10. Instance 14 does not have any feasible solution. On instances from the original benchmarks, the upper limit values of the resources consumed is small such that the pruning techniques used in [3] reduce substantially the problem sizes. The Algorithm in [3] is thus particularly efficient on these instances. The second benchmark is generated by modifying some upper limit values as follows. We chose the large instances of order 500 (instances numbered from 17 to 24). For instances numbered 17–20 (number of resources is equal to 1), we slightly decrease the upper limit values. For instances numbered 21–24 (number of resources is equal to 10), we multiply some upper limit values by 10. In order to compare the model with the algorithm

[3] Each arc has multiple properties, the property indexed by 0 is the length and properties indexed from 1 to `k` are resources consumed in traversing this arc.

instances	opt	t*	min	max	%	avr_t	min_t	max_t	std_dev	min'	max'	%
rcsp1	131	0.62	131	131	100	**0.26**	**0.25**	**0.28**	0.01	131	131	100
rcsp2	131	0.05	131	131	100	0.26	0.24	0.26	0.01	131	131	100
rcsp3	2	0.60	2	2	100	2.11	**0.48**	5.82	1.29	2	2	100
rcsp4	2	0.09	2	2	100	3.82	0.82	10.19	2.96	2	7	100
rcsp5	100	0.84	100	100	100	**0.83**	**0.6**	0.97	0.1	100	100	100
rcsp6	100	0.84	100	100	100	**0.75**	**0.6**	0.95	0.1	100	119	100
rcsp7	6	1.40	6	6	100	21.44	3.48	55.08	15.17	6	9	100
rcsp8	14	1.58	14	14	100	51.28	1.22	183.94	47.03	14	∞	80
rcsp9	420	0.04	420	420	100	122.5	2.14	483.43	115.26	420	∞	60
rcsp10	420	0.03	420	420	100	71.04	4.14	416.6	92.89	420	∞	90
rcsp11	6	0.11	6	6	100	7.75	1.84	18.44	3.81	6	7	100
rcsp12	6	0.09	6	6	100	9.12	2.34	25.12	6.52	6	6	100
rcsp13	448	0.44	448	448	100	90.06	7.94	276.02	66.81	448	∞	70
rcsp14	-	-	-	-	-	-	-	-	-	-	-	-
rcsp15	9	9.28	9	9	100	93.43	31.89	284.25	60.53	9	∞	70
rcsp16	17	8.84	17	17	100	279.89	33.43	1049.57	253.27	17	∞	30
rcsp17	652	55.91	652	652	100	56.64	**19.9**	106.07	23.65	652	652	100
rcsp18	652	56.45	652	652	100	57.27	**25.61**	116.98	22.56	652	652	100
rcsp19	6	0.59	6	6	100	28.15	7.92	66.72	13.32	6	6	100
rcsp20	6	1.07	6	6	100	46.85	12.79	118.63	31.08	6	15	100
rcsp21	858	3.20	858	858	100	242.3	61.68	679.96	190.7	858	∞	50
rcsp22	858	1.86	858	858	100	294.94	108.41	827.04	186.13	858	∞	50
rcsp23	4	50.74	4	4	100	280.36	**11.92**	1053.61	279.03	4	∞	90
rcsp24	5	54.05	5	∞	80	719.92	**24.13**	1737.43	574.94	5	∞	20

Table 1. First Experimental Results of RCSP_TABU: Original Instances.

from [3], we implemented that algorithm in `COMET` (denoted by RCSP_BEA) following the description in the paper.

The RCSP_TABU model is executed 20 times with a time window of 30 minutes for each instance. The first experimental results are shown in Table 1 (columns 1–10). The structure of the table is described as follows. Column 2 is the optimal value of the objective function and column 3 is the execution time (in seconds) of the RCSP_BEA model. Columns 4 and 5 present the minimal and maximal value of the objective function in 20 runs of RCSP_TABU. Column 6 is the rate of finding the optimal solution. Columns 7–10 show the average, the minimal, maximal, and the standard deviation of the execution time necessary to find the optimal solution. The results show that the RCSP_TABU model found the optimal solutions in all 20 runs over all instances except the instance rcsp24 (only 15 runs found the optimal solution) and the instance rcsp14 (a feasible solution does not exist). The table also shows that on the original benchmark, the RCSP_BEA model found optimal solutions faster than our RCSP_TABU model except for some instances (see lines 1, 3, 5, 6, 17, 18, 23, 24).

The experimental results for the second set of benchmarks are shown on Table 2 (Columns 11–13 should be ignored for now). Column 2 presents the execution times of the RCSP_BEA model for finding optimal solutions. The remaining columns report results of the RCSP_TABU model. Columns 3–6 show

instance	t*	avr_t	min_t	max_t	std_dev	%solved	avr_it
rcsp17_01	57.58	**54.16**	**26.18**	159.76	30.63	100	12.95
rcsp17_02	58.88	**56.26**	**30.1**	138.43	26.95	100	18.25
rcsp18_01	56.57	57.4	**27.35**	120.06	23.19	100	17.55
rcsp18_02	56.64	60.32	**16.98**	266.61	52.56	100	16.95
rcsp19_01	75.56	**40.65**	**11.78**	108.21	25.02	100	41.05
rcsp19_02	59.36	**56.9**	**8.74**	134.95	35.54	100	56.95
rcsp20_01	74.98	**49.32**	**6.66**	136.49	38.17	100	57.4
rcsp20_02	58.34	**51.6**	**8.72**	177.01	36.72	100	55.9
rcsp21_01	164.5	**72.91**	**31.74**	108.89	22.07	100	11.15
rcsp21_02	154.67	**63.88**	**34.02**	128.04	21.13	100	12.15
rcsp22_01	157.6	**73.85**	**28.28**	118.14	24.48	100	11.1
rcsp22_02	150.95	**72.42**	**33.07**	180.05	30.97	100	11.7
rcsp23_01	130.08	**76.99**	**21.58**	216.21	51.99	100	38.9
rcsp23_02	129.34	**70.3**	**21.64**	250.5	54.97	100	30.65
rcsp24_01	129.09	153.04	**34.35**	418.23	113.86	100	75.7
rcsp24_02	129.44	**94.54**	**24.13**	402.08	80.31	100	49.25

Table 2. Second Experimental Results of RCSP_TABU: More Difficult Instances.

the average, minimal, maximal, and standard deviation of execution times to find optimal solutions. Column 7 present the percentage for finding optimal solutions. The final column reports the average of the number of moves. Experimental results show that our RCSP_TABU model found optimal solutions faster than the RCSP_BEA model in most cases. The reason is that, on these instances, the constraints over resources consumed are easy to satisfy but the search space is much larger. The reduction techniques in [3] do not reduce the search space much and the search procedure of the RCSP_BEA model is thus much slower.

5.2 The RCSP problem with Multiple Choice Side Constraints

In order to illustrate the flexibility of our modeling approach, we consider the RCSP problem with the following side constraint over nodes on the path: The set of nodes V is partitioned into Q subsets $S_1, S_2, ..., S_Q$ and the path is required to visit at most one node from each subset. This constraint arises when solving a subproblem in a branch-and-price method for a variation of the vehicle routing problem, known as Multi-Resource Routing Problem (MRRP) [9]. This problem cannot be solved with RCSP_BEA without a substantial programming effort.

The Modeling The MC_RCSP problem is modeled by extending the RCSP model: the Multiple Choice constraints are stated and posted into the `GraphConstraintSystem gcs`. This can be done by simply adding the following snippet to the RCSP model:

```
1. GraphObjective nv[1..Q];
2. forall(q in 1..Q){
3.   nv[q] = NodesVisited(path,S[q]);
4.   gcs.post(nv[q] <= 1);
5. }
```

where `NodesVisited(ls,path,S[q])` is an abstraction representing the number of nodes in `S[q]` visited by the path. Notice that such a side constraint cannot be handled by the algorithm of [3].

Experimental Results We experiment the model over the benchmark from the OR-library where the set of subsets $S_1, S_2, ..., S_Q$ is generated as follows. We take a random feasible solution to the RCSP problem which is an elementary path $v_1, v_2, ..., v_q$ satisfying the resource constraints. Then, we partition V into $Q = 3*q$ sets $S_1, S_2, ..., S_Q$ where $v_j \in S_j, \forall j \in \{1, 2, ..., q\}$ and the size differences are at most one. This ensures that there exists at least one feasible solution $v_1, v_2, ..., v_q$ to the MC_RCSP problem. The model is executed 10 times with 10 minutes of time window for each instance. Experimental results are shown in Table 1 (columns 11–13) where column 13 presents the rate of finding feasible solutions. Columns 11–12 present the minimal and maximal value of the best solution in different executions. In most cases, the rate for finding feasible solutions is high except instances 16 and 24. In some cases, the model finds optimal solutions.

5.3 The EDP problem

We are given an undirected graph $G = (V, E)$ and a set $T = \{< s_i, t_i >|\ s_i \neq t_i \in V\}$ representing a list of commodities ($\sharp T = k$). The EDP problem consists of finding a maximal cardinality set of mutually edge-disjoint paths from s_i to t_i on G ($< s_i, t_i > \in T$). In [4], a Multi-start Simple Greedy and an ACO algorithms are presented. The ACO is known to be state-of-the-art for this problem. We propose a local search algorithm using the following model:

```
void stateModel{
1.   LSGraphSolver ls();
2.   VarPath path[j in 1..k](ls,g,s[j],t[j]);
3.   PathEdgeDisjoint ed(path);
4.   ls.close();
5.}
```

where line 2 initializes an array of `k` `VarPath`s representing `k` paths between commodities. The edge-disjoint constraint `ed` is defined over paths from `s[i]` to `t[i]` and is stated in line 3.

In [4], 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. The number of violations of the $PathEdgeDisjoint(P_1, P_2, ..., P_k)$ constraint in the framework is defined to be $C(\{P_1, P_2, ..., P_k\})$ and the proposed local search algorithm tries to minimize this criterion.

instance	com.	MSGA		ACO		Local search	
		$\overline{q}$	$\overline{t}$	$\overline{q}$	$\overline{t}$	$\overline{q}$	$\overline{t}$
mesh25x25.bb	62	36.95	546.854	31.1	880.551	**38.85**	1165.47
	156	44.65	863.007	47.5	965.921	**55.5**	1082.78
	250	50.5	672.962	60.5	972.396	**67.95**	967.087
mesh15x15.bb	22	20.55	517.601	18.6	500.812	**21**	384.828
	56	27.15	651.27	28.35	988.782	**30.3**	485.693
	90	31	797.534	34.55	746.96	**36.05**	435.308
bl-wr2-wht2.10-50.rand.bb	50	18.7	688.651	19.6	201.235	**20.05**	228.382
	125	27.2	643.51	31.15	338.446	**31.2**	241.047
	200	36.6	625.138	41.55	164.783	**41.7**	202.186
bl-wr2-wht2.10-50.sdeg.bb	50	18.65	470.26	19.75	223.396	**20.1**	311.887
	125	28.1	662.916	31.55	163.151	**31.85**	357.25
	200	33.3	487.999	38.05	217.362	**38.25**	178.417

Table 3. Experimental results for the EDP problem

From a solution which is normally a set of k non-disjoint path, a feasible solution to the EDP problem can be extracted by iteratively removing the path which has most edges in common with other paths until all remaining paths are mutually edge-disjoint as suggested in [4]. In our local search model, we extend this idea by taking a simple greedy algorithm over the remaining paths after that extraction procedure in hope of improving the number of edge-disjoint paths.

Experimental Results For the experimentation, we re-implemented in `COMET` the Multi-start Greedy Algorithm (MSGA) and the ACO (the extended version) algorithm described in [4] and compare them with our local search model. The instances in the original paper [4] are not available (except some graphs). As a result, we use the instance generator described in [4] and generate new instances as follows. We take 4 graphs from [4]. For each graph, 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 20 instances with 0.10*n, 0.25*n and 0.40*n commodities. In total, we have 240 problems instances. Due to the high complexity of the problem, we execute each problem instance once with a time limit of 30 minutes for each execution. Experimental results are shown in Table 3. The time window for the MSGA and the ACO algorithms are also 30 minutes. The Table reports the average values of the objective function and the average execution times for obtaining the best solutions of 20 instances (a graph $G = (V, E)$ and a set of $r * |V|$ commodities, $r = 0.10, 0.25, 0.40$). Table 3 shows that our local search model gives very competitive results. It finds better solutions than MGSA in 217/240 instances, while MSGA find better solutions in 4/240 instances. On the other hand, in comparison with the ACO model, our model finds better solutions in 144/240 instances, while the ACO model find better solutions in 11/240 instances. This clearly demonstrates the potential benefits of our COP framework, from a modeling and computational standpoint.

6 Conclusion

This paper considered Constrained Optimum Path (COP) problems which arise in many real-life applications. It proposes a domain-specific constraint-based local search (CBLS) framework for COP applications, enabling models to be high level, compositional, and extensible and allowing for a clear separation between model and search. The key technical contribution to support the COP framework is a novel neighborhood based on a rooted spanning tree that implicitly defines a path between the source and the target and its neighbors, and provides an efficient data structure for differentiation. The paper proved that the neighborhood obtained by swapping edges in this tree is connected and presented a larger neighborhood involving multiple independent moves. The COP framework, implemented in `COMET`, was applied to Resource Constrained Shortest Path problems (with and without side constraints) and to the edge-disjoint paths problem. Computational results showed the potential significance of the approach, both from a modeling and computational standpoint.

Acknowledgments We would like to thank Maria José Blesa Aguilera who has kindly provided some graphs for the experimentation. This research is partially supported by the Interuniversity Attraction Poles Programme (Belgian State, Belgian Science Policy) and the FRFC project 2.4504.10 of the Belgian FNRS (National Fund for Scientific Research).

References

1. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, New Jersey, United States, 1993.
2. J. E. Beasley. Or-library, url=http://people.brunel.ac.uk/ mastjjb/jeb/info.html.
3. J. E. Beasley and N. Christofides. An algorithm for the resource constrained shortest path problem. *Network, vol. 19*, pages 379–394, 1989.
4. 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.
5. W. M. Carlyle and R. K. Wood. Lagrangian relaxation and enumeration for solving constrained shortest-path problems. *Proceedings of the 38th Annual ORSNZ Conference*, 2003.
6. J. C. N. Clímaco, J. M. F. Craveirinha, and M. M. B. Pascoal. A bicriterion approach for routing problems in multimedia networks. *Network*, 41:206–220, 2003.
7. I. Dumitrescu and N. Boland. Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem. *Networks*, 42:135–153, 2003.
8. Q. D. Pham, Y. Deville, and P. 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)*, 2009.
9. K. Smilowitz. Multi-resource routing with flexible tasks: an application in drayage operation. *IIE Transactions*, pages 38(7):555–568, 2006.
10. P. Van Hentenryck and L. Michel. *Constraint-based local search.* The MIT Press, London, England, 2005.