Constraint-based local search for solving non-simple paths problems on graphs: Application to the Routing for Network Covering Problem

Quang-Dung PHAM
Université catholique de Louvain
B-1348 Louvain-la-Neuve, Belgium
quang.pham@uclouvain.be

Phan-Thuan DO
School of Information and Communication Technology
Hanoi University of Technology
Hanoi, Vietnam
thuandp@it-hut.edu.vn

Yves DEVILLE
Université catholique de Louvain
B-1348 Louvain-la-Neuve, Belgium
yves.deville@uclouvain.be

Tuong-Vinh HO
Institut de la Francophonie pour l'Informatique (IFI)
Hanoi,Vietnam
ho.tuong.vinh@auf.org

ABSTRACT

Routing problems have been considered as central problems in the fields of transportation, distribution and logistics. `LS(Graph)` is a generic framework allowing to model and solve constrained optimum paths problems on graphs by local search where paths are known to be elementary (i.e., edges, vertices cannot be repeated on paths). In many real-world situations, the paths to be determined are not known to be neither simple nor elementary. In this paper, we extend the `LS(Graph)` framework by designing and implementing abstractions that allow to model and solve constrained paths problem where edges, vertices can be repeated on paths (call non-simple paths). We also propose an instance of such problem class: the routing for network covering (RNC) problem which arises in the context of rescue after a natural disaster in which we have to route a fleet of identical vehicles with limited capacity on a transportation network in order to collect the informations of the disaster. Given an undirected weighted graph $G = (V, E)$ representing a transportation network and a vertex $v_0 \in V$ representing the depot, the RNC problem consists of routing a fleet of unlimited number of identical vehicles with limited capacity that cannot perform a path of length $> L$ such that each vehicle starts from and teminates at the depot and all the edges of a given set S ($S \subseteq E$) must be visited. The objective of the routing plan is to minimize the number of vehicles used. This paper discusses the challenge around this problem and applies the constructed framework to the resolution of this problem. The proposed model is generic; it allows to solve some variants of the problem where side constraints are required to be added.

General Terms

Applications

Keywords

Combinatorial Optimization, Constraint Optimization, Graph, Local Search

1. INTRODUCTION

The AROUND project [11] 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) [4] 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 this paper, we inspire from the AROUND project and propose a problem relating to the VRP problem, 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. This problem arises when distributing a fleet of vehicles for carrying out some works along streets. For instance, in the context of rescue after the natural disaster, we have to route a fleet of vehicles on an urban network in order to observe the situation of the disaster along all the streets of the urban area or to distribute resources for destroyed areas; cleaning service need to distribute a fleet of vehicles from a depot in order to clean streets of a district.

Three important constraints are taken into account:

1. Each vehicle starts from and terminates at the depot,
2. Each vehicle cannot travel a path whose length 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.

The RNC problem is also closely related to the chinese postman problem [6] where we have to find a cycle in a mixed graph (i.e., a graph 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 paths 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 path.

There may exist different side constraints in the real-world situation but we consider in this paper the most basic version of RNC problems which is stated in Section 2. To our best knowledge, the RNC problem has not been considered before and there are thus no previous works for solving this problem.

The `LS(Graph)` framework [12, 13] provides high-level abstractions for modeling and solving constrained optimum paths problems on graphs by local search where paths are known to be elementary. In real-life situations, many paths finding problems arise where edges and vertices can be repeated on the paths. Local search [10] 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 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 main contribution of this paper is to propose a generic framework by extending the `LS(Graph)` framework for solving constrained optimum paths problems where paths are not known to be simple. We also propose a novel routing problem on networks (RNC) which is proved to be NP-hard and we apply the framework for solving this problem. In critical situations, it is required to solve problems taking into account additional constraints. Here, we focus on the design and implementation of a framework allowing to easily model and solve the RNC problems with different side constraints by local search. Our first experimental results demonstrate the feasibility of the approach as well as the interest of our model.

The paper is organized as follows. Section 2 gives the definition of the RNC problem and discusses its complexity. A modeling approach is presented in Section 3. Section 4 presents local search model for solving the RNC problem. Section 5 gives some experimental results of the model on some random grid instances. Section 6 concludes the paper and states research directions for future work.

2. PROBLEM DEFINITION AND COMPLEXITY

2.1 Definitions and Notations

Given an undirected weighted graph $G = (V, E)$ representing a transportation network and a vertex $v_0 \in V$ representing the depot. c is a length function defined on edges: for each edge $e = (u, v) \in E$, $c(e)$ (or $c(u, v)$) is the length of e. A path on G is a sequence of vertices $v_1, v_2, ..., v_k$ in which $(v_i, v_{i+1}) \in E, \forall i = 1, 2, ..., k-1$.

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

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

Given an undirected weighted graph $G = (V, E)$, a vertex $v_0 \in V$ representing the depot, a set S_0 of edges of G ($S_0 \subseteq E$) and a value L, the RNC problems consists of finding a minimal cardinality set of paths starting from and terminating at v_0 whose length are less than or equal to L that covers all edges of S_0.

A feasible path is defined to be a path starting from and terminating at v_0 whose length is less than or equal to L. A feasible solution is a set of feasible paths that visit all edges of S_0. A feasible solution that has k feasible paths 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 $< G, c, v_0, L, S_0 >$.

2.2 Complexity

Definition Given an undirected weighted graph $G = (V, E)$, a vertex $v_0 \in V$, a set of edges $S_0 \subseteq E$, an integral value k and a value L, the problem of determining whether or not there exists a k-feasible solution is called $k-$RNC.

THEOREM 2.1. *1-RNC is NP-complete.*

The theorem 2.1 is proved from the fact that the Hamilton cycle problem which is NP-complete [6] can be reduced to this problem.

COROLLARY 2.2. *Given an undirected weighted graph $G = (V, E)$, two vertices $u, v \in V$, a value L and a set of edges $S_0 \subseteq E$, the problem of determining whether or not there exists a path from u to v whose length is less than or equal to L which passes all edges of S_0 is NP-complete.*

An equivalent reduced problem.

We can reduce an RNC problem instance $< G, c, v_0, L, S_0 >$ to an equivalent RNC problem instance $< G', c', v_0, L, S_0 >$ on smaller complete graph $G' = (V', E')$ as follows.

- $V' = \{v \in V \mid \exists u \in V : (u, v) \in S_0\} \cup \{v_0\}$
- $\forall u, v \in V' : c'(u, v) = c(u, v)$ if $(u, v) \in S_0$. Otherwise, $c'(u, v) =$ length of shortest path from u to v on G.

This reduction has a huge advantage for the programming complexity when the number of vertices having edges in S_0 is small. In this case, in order to solve the problem in a big graph G, we will solve it in the smaller one G'.

Theorem 2.1 shows that the RNC problem is NP-hard in general. Hence, exact methods for solving this problem induce an exponential computation time. There exists other related routing problems, for instance, Constrained Shortest Path Problems [5], [3], [1], Multiobjective Shortest Path Problems [14], [7], [9], etc. An integer programming formulation has been proposed to model these problems, for instance, [5], [3], [1]. A lagrangian relaxation combined with enumeration techniques are then applied to solve these problems. These techniques are specific and depend on particular constraints appearing in the problems. Applying them to the RNC problem where solutions are set of paths whose vertices, edges can be repeated seems to be sophisticated and is difficult to extend.

In this paper, we extend the `LS(Graph)` framework [12, 13] by constructing abstractions for easily modeling and solving the different constrained optimum non-simple paths problems and apply it to the resolution of the RNC problem.

3. MODELING APPROACH

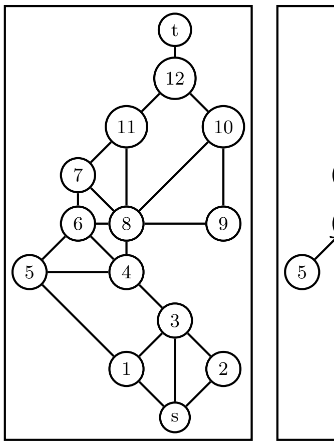
3.1 Modeling by using a sequence of spanning trees

The `LS(Graph)` framework [12] provides an abstraction denoted by RST(g, s, t) representing a dynamic spanning tree of a given graph g rooted at t which induces an elementary path from a given vertex s to t on g $(s, t \in V(g))$. An update over the spanning tree creates a new spanning tree which may induce a new path from s to t (see [12] for more details).

We give an example in Figure 1. Figure 1a is an undirected connected graph g and Figure 1b is a rooted spanning tree tr of g rooted at t in which that path induced by tr is $< s, 3, 4, 6, 7, 11, 12, t >$.

The main advantage of using rooted spanning tree for modeling paths instead of using explicit paths representation (i.e. a sequence of vertices) is the simplification of neighborhood computation. The tree structure contains rich information that induces directly path structure from a vertex s to the root. A simple update over that tree (i.e. an edge replacement) will induce a new path from s to the root.

But for the RNC problem and many other real-life applications, paths have not necessary to be elementary. For



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

Figure 1: Example of rooted spanning tree

example, a vehicle can depart from a depot, traverse some roads and return to the depot and it is allowed to pass visited roads. So it is required to model a path where vertices, edges can be repeated. Henceforth, we use the word "itinerary" to express paths where vertices, edges can be repeated which differs from "path" where vertices and edges can not be repeated. The basic idea here is to use a sequence of spanning trees: <RST$(g, s = x_0, x_1)$, RST(g, x_1, x_2), ..., RST$(g, x_{k-1}, x_k = t)$> to model a dynamic itinerary *VarItinerary*(s, t, g) from s to t on the graph g. Each instance *it* of *VarItinerary*(g, s, t) is a sequence $< tr_0, tr_2, ..., tr_{k-1} >$ where tr_i is an instance of RST(g, x_i, x_{i+1}), $\forall i = 0, 1, 2, ..., k-1$. 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 2 shows an example where $VarItinerary(g, s, t)$= <RST$(g, s, 5)$, RST$(g, 5, t)$ >. Figure 2a is an instance tr_1 of RST$(g, s, 5)$ which induces the path $p_1 =< s, 3, 4, 8, 7, 6, 5 >$ and Figure 2b is an instance tr_2 of RST$(g, 5, t)$ which induces the path $p_2 =< 5, 4, 8, 10, 12, t >$. Hence, the itinerary induced by $VarItinerary(g, s, t)$ is $p_1 + p_2 =< s, 3, 4, 8, 7, 6, 5, 4, 8, 10, 12, t >$.

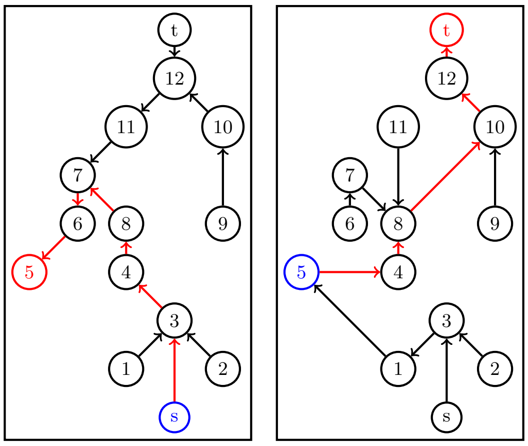
Property Each instance of $it = < tr_0, tr_1, ..., tr_{k-1} >$ of $VarItinerary(s, t, g)$ is an itinerary where vertices and edges are repeated at most k times.

3.2 Neighborhood

Given an instance $it = < tr_0, tr_1, ..., tr_{k-1} >$ of *VarItinerary*$(s, t, g) =$ <RST$(g, s = x_0, x_1)$, RST(g, x_1, x_2),..., RST-$(g, x_{k-1}, x_k = t)$ > 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 generated by taking a modification (local move) over *it*. In [12], we consider an edge replacement as a basic local move for RST(g, s, t). Given an instance tr of RST(g, s, t), we define the basic neighborhood of tr (see [12] for more details):

$$N(tr) = \{tr' = rep(tr, e', e) \mid e \in S_1, e' \in S_2\}$$

where $rep(tr, e', e)$ is an action that replaces the edge e' of the tree tr by the edge e conserving the tree property. The resulting tree tr' must also induce a different path from the



a. tr_1 b. tr_2

Figure 2: Example of itinerary

path induced by tr. S_1, S_2 are respectively the set of potential edges to be added and to be removed in the replacement.

We thus consider the first basic neighborhood of it:

$$N_1(it =< tr_0, tr_1, ..., tr_{k-1} >) =$$

$$\{< tr_0, tr_1, .., tr'_i, .., tr_{k-1} >| \; tr'_i \in N(tr_i), 0 \leq i \leq k-1\}$$

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. So we extend the RST(g, s, t) of [12] by considering another modification over this abstraction. Given an instance tr of RST(g, s, t). We define the actions $cs(tr, s')$ and $cr(tr, t')$ which respectively change the source s and the destination (or the root) t of tr by a new vertex s' and t' resulting new rooted spanning trees tr' and tr'' which induce new paths ($s', t' \in V(g)$). We denote $tr' = cs(tr, s')$, $tr'' = cr(tr, t')$.

We define a second basic neighborhood of it: $N_2(it =< tr_0, ..., tr_i, tr_{i+1}, ..., tr_{k-1} >) = \{< tr_0, ..., tr'_i, tr'_{i+1}, ..., tr_{k-1} >| \; tr'_i = cr(tr_i, y), tr'_{i+1} = cs(tr_{i+1}, y), 0 \leq i \leq k-2, y \in V(g)\}$. 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.

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[1] allows to avoid the computation overhead.

3.3 Overview of the abstractions

The implementation extends the `LS(Graph)` framework to support modeling constrained optimum paths problems where paths have not to be elementary (vertices and edges of the paths could be repeated). The core of the implementation is the `VarItinerary(s,t,g)` abstraction representing an itinerary variable (the itinerary begins at `s` and terminates at `t` on a graph `g`). Over this abstraction, basic constraints and functions have been designed and implemented. Fundamentally, constraints and objective functions appear on paths (itineraries) problems are those who relate to the total cost of the path (itineraries) and vertices, edges visited by paths (itineraries). We design and implement in this paper crucial constraints and objective functions for constrained itineraries problems which can be used to state more complex ones. For instance, `NBVisitedEdgesItinerary(I,S)` (`NBVisitedVerticesItinerary(I,S)`) is a graph objective representing the number of edges (vertices) of `S` which are visited by the list of itineraries `I`. `ItineraryCost(I)` represents the sum of weights of all the edges of the itinerary `I`. We can combine these graph objectives with arithmetic operators +, -, * and state constraints with basic relation operators <=, >=, ==. For instance, if we would like to state the constraint saying that the sum of lengths of two itineraries `I1` and `I2` must be less than or equal to `L`, we can easily do it in the following snippet[2]:

```
GraphConstraintSystem gcs(ls);
ItineraryCost c1(I1);
ItineraryCost c2(I2);
gcs.post(c1 + c2 <= L);
```

The following snippet state and post constraints over the total length of itineraries and constraint of covering edges which appear in the RNC problem:

```
GraphConstraintSystem gcs(ls);
forall(i in I.rng()){
  ItineraryCost c(I[i]);
  gcs.post(c <= L);
}
NBVisitedEdgesItinerary nb(I,S);
gcs.post(nb == S.getSize());
gcs.close();
```

We also design and implement generic search abstractions. For instance, `CSMaxTabuSearch(Model<Itinerary> m)` is a tabu search component specified over a model `m` for constrained maximum itineraries problems which finds a set of itineraries under constraints maximizing a given objective function. The tabu search features an aspiration criterion allowing forbidden solutions which are better than the best solution found so far. We now describe briefly this tabu search component which will be applied to the RNC problem. The tabu search schema given in Figure 3 is the same as that described in [2] in which `tbMin, tbMax, tinc, maxStable` are local search parameters. The tabu length `tbl` (line 7) varies between `[tbMin,tbMax]` and is set to `tbMin` at the beginning. `tinc` (line 7) is the change step of `tbl` and `nic` (lines 6 and 13) counts the number of local moves that do not improve the best solution found from each restart. The tabu length `tbl` is updated (line 7) whenever the search does not improve solution after `maxStable` local moves (line 6). If `tbl` is greater than its upper bound `tbMax` (line 8) then we perform restart (method `performRestart` in line 9) which generate randomly new solution, set `tbl` to its lower

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

[2]All code illustration presented in this paper is written in `COMET` programming language [8]: a high-level modeling programming language for constraint optimization.

```
1.  void search(){
2.    initSolution();
3.    updateBest();
4.    it = 1;
5.    while(it < maxIt){
6.      if(nic > maxStable){
7.        tbl = tbl + tinc;
8.        if(tbl > tbMax){
9.          performRestart();
10.       }else{
11.         updateTabuLists();
12.       }
13.       nic = 1;
14.     }
15.     if(localmove()){
16.       updateBest();
17.     }else{
18.       performRestart();
19.     }
20.     it++;
21.   }
22.}
```

Figure 3: Generic reactive tabu search schema

```
1.  bool localmove(){
2.    MinNeighborSelector N();
3.    exploreReplaceNeighborhood(N, true);
4.    exploreChangeDestinationNeighborhood(N, true);
5.    if(N.hasMove()){
6.      call(N.getMove());
7.      return true;
8.    }
9.    return false;
10.}
```

Figure 4: neighborhood exploration and move execution

```
1.  void exploreReplaceNeighborhood(Neighborhood N,
           bool firstImprovement){
2.   float eval = System.getMAXINT();
3.   VarPath sel_vp = null;
4.   Edge sel_ei = null;
5.   forall(i in _varpaths.rng()){
6.    VarPath vp = _varpaths[i];
7.    GTabuEdge tbIn = _mapTabuEdgeIn{vp};
8.    ReplacingEdgesMaintainPath rpl = _map{vp};
9.    forall(ei in rpl.getSet()){
10.    float d = F.getDeltaWhenUseReplacingEdge(vp,ei);
11.    if(!tbIn.isTabu(ei,it) ||
             d + F.getValue() < fgb){
12.     if(eval > d){
13.      sel_ei = ei;
14.      sel_vp = vp;
15.      eval = d;
16.     }
17.     if(firstImprovement && eval < 0)
18.      break;
19.    }
20.   }
21.   if(firstImprovement && eval < 0)
22.    break;
23.  }
24.  if(sel_ei != null)
25.   neighbor(eval,N){
26.    select(eo in getPreferredReplacableEdges(vp,
         sel_ei))
27.     vp.replaceEdge(eo,ei);
28.     GTabuEdge tbOut = _mapTabuEdgeOut{sel_vp};
29.     GTabuEdge tbIn = _mapTabuEdgeIn{sel_vp};
30.     tbOut.makeTabu(sel_ei);
31.     tbIn.makeTabu(eo,it);
32.  }
33.}
```

Figure 5: First Neighborhood exploration

bound `tbMin`, reset `nic`. If it is not the case, the tabu lists are updated with new tabu length `tbl` (line 11). Line 15 perfrom a local move. If no move is taken (i.e., tabu condition is reached), we perform restart (line 18). The core of the search is the neighborhood exploration in order to choose a desired neighboring solution and move execution (see method `localmove` in Figure 4). `MinNeighborSelector N` (line 2) is a `COMET` structure that maintains the best move and its evaluation submitted so far (i.e., by the neighborhood exploration methods in lines 3-4 which are detailed in Figures 5, 6). Lines 5-6 perform the move (if any). Two neighborhood structures described above are considered in this tabu search.

Figure 5 explores the first neighborhood. Lines 5-6 scan all `VarPath vp`[3] components of the solution (i.e., the set of `VarItinerary`). Line 8 retains a graph invariant `rpl` representing the set of *preferred replacing* edges of `vp`. All *preferred replacing* edges `ei` of `vp` are scanned (line 9) and line 10 evaluates the quality of the neighboring solution corresponding with `ei`[4] in term of the function to be minimized `F`. In the `CSMaxTabuSearch(Model<Itinerary> m)`, the function `F` to be minimized which control the search procedure is set to `alpha*c - beta*f` where `c, f` are respectively the constraint to be satisfied and the objective function to be maximized, `alpha, beta` are search parameters. Line 11 checks whether or not `ei` belongs to the tabu list of edges to be added `tbIn` corresponding to `vp` (see line 7) or this neighbor improves the best value of `F` found so far `fgb`. Lines 12-15 store new chosen solution. Lines 25-31 perform the move including the edge replacement (line 27) and make tabu two edges of the replacement (lines 28-31).

Figure 6 explores the second neighborhood. Similar to the exploration of the first neighborhood, lines 6-9 scan all `VarPath vp` which is not the last component of the `VarItinerary vi` to which it belongs[5]. Lines 10-12 evaluate

[3]`VarPath` is an abstraction which encapsulates RST(g, s, t) representing a dynamic path from s to t on g.

[4]A neighboring solution of the first neighborhood depends only on the *preferred replacing* edge to be used, not on the *preferred replacable* edge (see [12] for more details).

[5]we do not change the destination of the last `VarPath vp`

```
1.  void exploreChangeDestinationNeighborhood(
           Neighborhood N, bool firstImprovement){
2.   float eval = System.getMAXINT();
3.   Vertex sel_des = null;
4.   VarPath sel_vp = null;
5.   VarItinerary sel_vi = null;
6.   forall(i in _varpaths.rng()){
7.    VarPath vp = varpaths[i];
8.    VarItinerary vi = _mapI{vp};
9.    if(!vi.lastVarPath(vp)){
10.    Vertex des = vp.getDestination();
11.    forall(newDes in vp.getLUB().getVertices():
          newDes != des){
12.     float d = F.getChangeDestinationDelta(vi,
            vp,newDes);
13.     GTabuVertex tbv = _mapTabuVertex{vp};
14.     if(!tbv.isTabu(newDes,it) ||
            d + F.getValue() < fgb){
15.      if(eval > d){
16.       sel_des = newDes;
17.       sel_vp = vp;
18.       sel_vi = vi;
19.       eval = d;
20.      }
21.     }
22.     if(firstImprovement && eval < 0)
23.       break;
24.    }
25.    if(firstImprovement && eval < 0)
26.     break;
27.   }
28.  }
29.  if(sel_des != null)
30.    neighbor(eval,N){
31.      vi.changeDestination(sel_vp, sel_des);
32.      GTabuVertex tbv = _mapTabuVertex{vp};
33.      tbv.makeTabu(sel_des,it);
34.    }
35.}
```

Figure 6: Second Neighborhood exploration

the quality of the neighboring solution generated by change the destination of `vp` and the source of the next `VarPath` of `vp` in `vi` in term of the function `F`. Line 14 checks tabu condition and lines 15-20 store new best neighbor if it is discovered. Lines 29-34 perform the move including changing the destination of the selected `VarPath sel_vp` by a new selected destination `sel_des` and making this new destination tabu (lines 32-33).

4. LOCAL SEARCH-BASED MODELS FOR SOLVING THE RNC PROBLEM

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, `S0` is the set of edges that need to be covered and `L` is the maximum value allowed of the

```
1.  void model(){
2.    set{VarItinerary} sol();
3.    set{Edge} S = S0;
4.    while(S.getSize() > 0){
5.      VarItinerary I = greedy(S);
6.      sol.insert(I);
7.    }
8.  }

9.  VarItinerary greedy(set{Edge} S){
10.   LSGraphSolver ls();
11.   VarItinerary I(ls,depot,depot,g,k);
12.   ItineraryCost cost(I,g);
13.   NBVisitedEdgesItinerary nbVisitedE(I,S,g);

14.   Model<Itinerary> mod(I,cost <= L,nbVisitedE);
15.   CSMaxTabuSearch<Itinerary> se(mod);
16.   se.search(nbTrials);

17.   forall(e in I.getEdges())
18.     S.delete(e);

19.   return I;
20.}
```

Figure 7: First local search model

cost of each itinerary in the solution.

The model is a greedy constructive search which is depicted in Figure 7. At each step, we find greedily a feasible path that covers as many edges of the remaining uncovered edges `S` (`S` is initialized by `S0` (line 3) and is reduced after each iteration) as possible (see method `greedy` at line 5 of Figure 7) until `S` is empty. `sol` stores the solution which is updated when a new itinerary is discovered (lines 5-6). The method `greedy` is detailed in lines 9-20. Line 10 creates a `LSGraphSolver ls` which manages all graph variables, graph invariants, graph constraints and graph objectives of the model. Line 11 declares an object `I` representing *VarItinerary* which is composed by a sequence of `k` *VarPath*s starting and terminating at `depot`, rooted at random vertices except the last one. Line 12 initializes a graph objective `cost` representing the cost of `I` and line 13 initializes a graph objecitve `nbVisitedE` representing the number of edges of `S` visited by `I`. Line 14 creates a model with a decision variable `I`, a constraint `cost <= L` to be satisfied and an objective function `nbVisitedE` to be maximized. A search object which applies to the model `mod` and maximizes `nbVisitedE` is initialized in line 15. Line 16 performs a tabu search of `nbTrials` iterations. Lines 17-18 remove edges visited by the itinerary `I` from `S` for next steps.

We can see that the model is compositional in the sense that it is easy to state[6] and post new constraints to the `GraphConstraintSystem gcs` without changing the search. On the other hand, one can easily perform different heuristics and meta-heuristics on a given model thanks to built-in graph invariants supporting neighborhood computation[7].

5. EXPERIMENTS

of a given `VarItinerary vi` because this destination is also the destination of `vi` which is fixed.

[6]The `LS(Graph)` framework provides built-in fundamental abstractions that enable to state some constraints and ob-

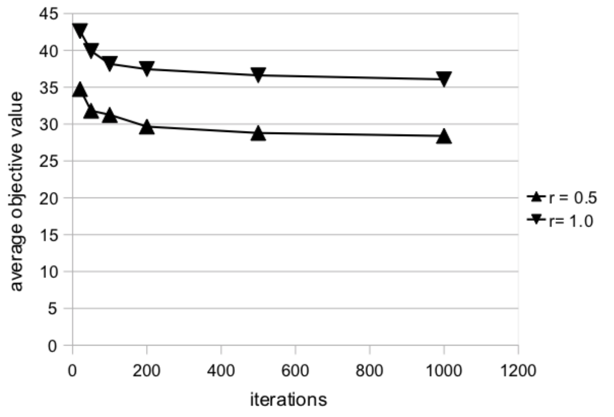


Figure 8: Grid 25x25: average objective value over iterations

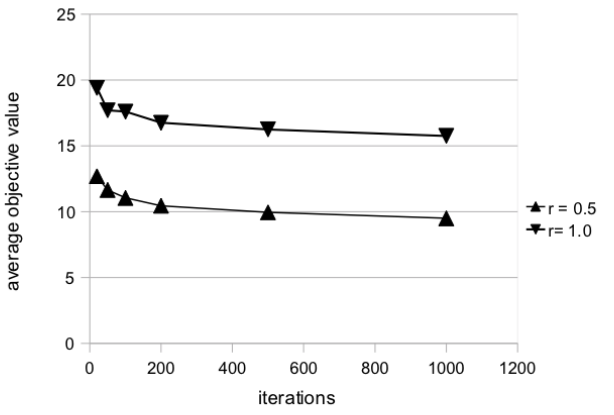


Figure 9: Grid 15x15: average objective value over iterations

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 by 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 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 3). 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 (the value `maxIt` in line 5 of Figure 3 and `nbTrials` in line 16 of Figure 7). 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. In total, we have 33600 executions.

Results.

The results are shown in Table 10 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 9, 8 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).

6. CONCLUSION

In this paper, we have extended the `LS(Graph)` framework by constructing abstractions which allow to model and solve different constrained optimum non-simple paths problems. We have also introduced a novel routing problem on networks, an instance of that problem class, which consists of finding paths for a fleet of identical vehicles with limited capacity from a given depot on a transportation network in order to carry out some works along streets, for instance, to collect the damage informations after natural disasters or to clean streets, etc. Two important constraints are considered: each vehicle must start from and terminate their path at the depot and respect their capacity; a set of streets of the network need to be visited. There exists different side constraints in real-world situations but we consider here a basic version of the problem with the above two constraints. This problem is proved to be NP-hard. A local search modeling

jective functions on graphs.

[7]On the above models in Figure 7, we used a built-in search component.

Graph	d	r	20 iters				100 iters				200 iters				B & B
			m	M	$\bar{f}$	$\bar{t}$(sec.)	m	M	$\bar{f}$	$\bar{t}$(sec.)	m	M	$\bar{f}$	$\bar{t}$(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	-

Figure 10: Results for 20, 100, 200 iterations

has been proposed by using the constructing abstractions. First experimental results show the feasibility and the interest of our approach. Our future work will focus on exploring other meta-heuristic algorithms and testing the model on extensive data sets, especially, in real urban networks. We will also consider different variants of this RNC problem with different constraints and objective function to be optimized.

7. REFERENCES

[1] J. E. Beasley and N. Christofides. An algorithm for the resource constrained shortest path problem. *Networks*, 19:379–394, 1989.

[2] 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.

[3] R. J. O. Carlyle, W. M. and R. Wood. Lagrangian relaxation and enumeration for solving constrained shortest-path problems. *Networks*, 52:256–270, 2008.

[4] G. B. Dantzig and R. Ramser. The truck dispatching problem. *Management Science*, 6:80–91, 1959.

[5] I. Dumitrescu and N. Boland. Algorithms for the weight constrained shortest path problem. *International Transactions in Operational Research*, 8(1):15–29, 2001.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability: A guide to the Theory of NP-Completeness*. W. H. Freeman, 1st ed., 1979.

[7] F. Guerriero and R. Musmanno. Label correcting methods to solve multicriteria shortest path problems. *Journal of Optimization Theory and Applications*, 111:589–613, 2001.

[8] P. V. Hentenryck and L. Michel. *Constraint-based local search*. The MIT Press, 2005.

[9] J. M. F. C. João C. N. Clímaco and M. M. B. Pascoal. A bicriterion approach for routing problems in multimedia networks. *Networks*, 41:206–220, 2003.

[10] W. Michiels, E. Aarts, and J. Korst. *Theoretical Aspects of Local Search*. Springer, 2007.

[11] MSI/IFI. Around project, url = http://www.ifi.auf.org/site/content/view/48/84/. 2006.

[12] Q. D. Pham, Y. Deville, and P. V. 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*, 2010.

[13] 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.

[14] A. Skriver and K. Andersen. A label correcting approach for solving bicriterion shortest-path problems. *Computer and Operations Research*, 27:507–524, 2000.