Noname manuscript No.
(will be inserted by the editor)

Solving the Quorumcast Routing Problem by Constraint Programming

Pham Quang Dung · Yves Deville

Received: date / Accepted: date

Abstract The quorumcast routing problem is a generalization of multicasting which arises in many distributed applications. It consists of finding a minimum cost tree that spans the source node r and at least q out of m specified nodes on a given undirected weighted graph. This paper proposes a complete and an incomplete approach, both based on the same Constraint Programming (CP) model, but with two different specific search heuristics based on shortest paths. Experimental results show the efficiency of the two proposed approaches. Our complete approach (CP model + complete search) is better than the state of the art complete algorithm and our incomplete approach (CP model + incomplete search) is better than the state of the art incomplete algorithm. Moreover, the proposed complete search is better than the standard First-Fail search in the same CP model.

Keywords Constraint Programming · Quorumcast routing · Constrained Optimum Trees · Graphs · Incomplete Search

1 Introduction

Multicasting is the problem of delivering a message from a source to a given subset of nodes, called *multicast* nodes, in a network. The quorumcast routing (QR) problem is a generalization of multicasting in which we have to send a message from a source to at least q out of n specified multicast nodes [2], [11], [8], [16]. This problem appears in many distributed applications, for example, distributed synchronization and updating a replicated resource (see [2] for more detail).

Pham Quang Dung
Hanoi University of Science and Technology. 1 Dai Co Viet road, Hanoi, Vietnam
Tel.: (+84)43 8692463
Fax: (+84)43 8692906
E-mail: dungpq@soict.hut.edu.vn

Yves Deville
Université catholique de Louvain B-1348 Louvain-la-Neuve, Belgium
Tel.: (++32)10 47 20 67
Fax: (++32)10 45 03 45
E-mail: yves.deville@uclouvain.be

The QR problem can be solved by complete (also called exact) approaches where the optimal solution is computed together with a proof of its optimality. One can also use incomplete approaches which compute an approximation to the optimal solution. Incomplete approaches are of course much more efficient, but do not ensure obtaining the optimal solution, which can be important in some applications.

Various incomplete approaches have been proposed in [2], [8], [16]. As far as we know, only one complete algorithm has been proposed for solving this problem [11]. A threshold was proposed and integrated in an exhaustive search for reducing the problem size. The proposed algorithm was evaluated on small graphs (up to 30 nodes).

Constraint Programming (CP) is a paradigm for solving combinatorial optimization problems using high-level formulations [14]. This approach has two main components: propagation and search. The propagation component uses constraints to prune values from the domain of variables that do not belong to any solution. The search component specifies a search tree and different heuristics can be used to search for the optimal solution. CP is a complete approach as the search component covers all the solutions. It is however possible to also use CP for incomplete approaches by restricting the search component to only a subset of the solutions.

In this paper, we propose a Constraint Programming model together with two search components. The search components are the main contributions of the papers; we develop a complete and an incomplete approach. Both search components are based on the shortest-path heuristics used in [2] for solving the QR problem. Our models are implemented in the Comet programming language [4], an object-oriented programming language supporting Constraint-Based Local Search, Constraint Programming, and Mathematical Programming. We experimentally show that our complete algorithm solves problems on larger graphs than [11] (graphs up to 60 nodes). Moreover, our incomplete algorithm (the CP model + incomplete search) outperforms other state of the art incomplete algorithms such as those described in [2] and our complete algorithm (the CP model + complete search) is more efficient than the state-of-the-art exact algorithm of [11]. We also show the extensibility of the proposed CP model when side constraints need to be added.

1.1 Problem formulation

Given a weighted undirected graph $G = (V, E, w)$, each edge $e \in E$ is associated with a positive cost $w(e)$. Given a source node $r \in V$, an integral value q, and a set $S \subseteq V$ of multicast nodes, the quorumcast routing problem consists of finding a minimum cost tree $T = (V', E')$ of G spanning r and at least q nodes of S. The graph $T = (V', E')$ should satisfy the following properties:

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

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

   is minimal over all subgraphs of G with properties 1, 2, and 3

1.2 Related work

The state-of-the-art complete algorithm was proposed in [11]. In that paper, a partial solution is defined to be a set of sub-trees that spans the source node and some multicast nodes. A partial solution will be extended by adding one edge at each step until a feasible solution is constructed (i.e., a tree that spans the source node and at least q multicast nodes). A *Confined Area Pruning (CAP)* scheme was introduced that allows to reduce that search space. Let C_{min} be the cost of the best solution found so far, and D_{max} be the longest distance from the source node to a multicast node[1]. A threshold value T is computed:

$$T = \begin{cases} \frac{C_{min}+D_{max}}{2} & \text{if } C_{min} > D_{max} \\ C_{min} & \text{otherwise} \end{cases}$$

If the cost of the shortest path from the source to a node $v \in V$ exceeds the threshold value T, then the node v cannot be in any optimal solution.

A particular case of the QR problem is the well-known Minimum Steiner Tree (MST) Problem [1] where $|S| = q$. The QR problem could then be solved by successively considering all the possible subsests Q of S with $|Q| = q$. The QR problem is thus reduced to $\binom{|S|}{q}$ ($|S|$ choose q) MST problems. Although there exist efficient exact algorithms for solving the MST problem [1], this approach is not relevant when $\binom{|S|}{q}$ is large. In the experimental section, we will show that our complete algorithm is able to solve QR problem where $\binom{|S|}{q}$ is large. Moreover, our proposed CP model and heuristic searches are simple to implement and extensible while the algorithm for the MST is very sophisticated to implement and difficult to extend with side constraints.

The Connected Subgraph in wildlife conservation problem [5] is also related to the MST problem, where each node of the given graph is associated with a cost and a profit. One has to find a connected subgraph that spans a set of given terminals, satisfying the constraint on the cost while maximizing the profit. In that paper, different ILP-based modeling approaches have been proposed and compared.

An incomplete approach was proposed in [2] where three heuristics were described: Minimal Cost Path Heuristic (MPH), Improved Minimum Path Heuristic (IMP), and Modified Average Distance Heuristic (MAD). The idea of the MPH heuristic is to construct the solution in a greedy way. It starts from a partial solution (a tree under construction) containing only the source node r. At each step (called a selection step), it selects the closest node v of S that does not belong to the partial solution and inserts all the nodes of the corresponding shortest path from v to the partial solution into the current partial solution, until the partial solution contains q nodes of S. The main idea of the IMP algorithm is to repeat the MPH several times but at each selection step, it does not consider the nodes of S that have been selected in any of the previous MPH calls. The MAD heuristic consists of two stages. The first stage selects q multicast nodes W' based on the idea of Kruskal's algorithm when solving the minimum spanning tree problem. The second stage reconnects the nodes of W' with the root using the MPH algorithm. Experimental results in [2] show that among the three heuristics, the IMP heuristic produces the best solutions. The idea of the MPH heuristic will be the basis of the two search components that we propose for our CP model.

[1] The distance between two nodes on a graph G is defined to be the cost of the shortest path between these nodes in G.

In [8], a multispace search heuristic was proposed for solving the particular QR problem where $S = V$. It gives better results than the IMP and the MAD heuristics on 12-node networks and 100-node networks. This approach is however no longer applicable when S is a strict subset of V.

In [16], the authors considered the QR problem with additional constraints on the total cumulative delay along the path from r to any destination node of Q and proposed a distributed heuristic algorithm for solving it. Experiments were conducted over graphs up to 200 nodes.

The minimum cost tree for multi-resource many cast (MRM) in networks [15] is a generalization of the QR problem where each terminal node is associated with a number of resource units and one has to find a minimum cost tree that spans the source and at least k resource units. When each terminal node is associated with exactly one resource unit, we obtain the QR problem. In that paper, four heuristic algorithms were proposed for solving the MRM problem: Least Unit-Cost First (LUF), Least Path-Cost First (LPF), Min-Cost Resource First (MCF), and Most Available First (MAF). In these heuristics, the tree is constructed incrementally: at each step, a terminal node v (v is not yet in the tree) is selected and all the nodes and edges of the shortest path from the source to v are inserted into the tree until the tree spans the required number of resource units. The selection of a terminal v is based on a heuristic value $h(v)$. In the LUF heuristic, $h(v)$ is the ratio between the cost of the shortest path from the source to v and the resource unit of v. In the LPF, $h(v)$ is the ratio between the cost of the shortest path from the source to v and the total resource units along this path. In the MCF heuristic, $h(v)$ is the cost of the shortest path from the source to v. In the MAF heuristic, $h(v)$ is the resource units of v. The LUF, LPF and MCF heuristics choose a terminal v having the smallest value of $h(v)$ while the MAF heuristic selects a terminal v having the largest value of $h(v)$. Our proposed heuristics are closely related to MCF as they are all are based on a shortest path measure for choosing the next node to integrate in the tree. The main difference is that MCF is a greedy constructive approach while our heuristic is integrated in a complete and incomplete search with backtracking. Our heuristic thus allows a better exploration of the search space. When applied to the QR problem, MCF reduces to the MPH algorithm of [2] which is less efficient than the IMP algorithm of the same paper [2]. In the experimental section, we show that our incomplete algorithm (the CP model + incomplete search) is better than IMP, hence better than the MCF approach applied on the QR problem.

The above incomplete algorithms can tackle large instances in reasonable time, but cannot guarantee optimality. A complete approach not only always finds the optimal solution, but also proves its optimality. Such a proof of optimality might be important and useful in many situations, especially when dealing with smaller instances. Moreover, when side constraints are required, a complete approach such as CP will be able to exploit these constraints to reduce the search space.

Different authors have proposed exact algorithms based on the Integer Programming approach for solving the problem of determining a subtree under constraints of a given graph [3], [10]. But these algorithms use sophisticated procedures for the search, applying specific techniques for the separation step in the branch-and-cut scheme. This does not allow of handling side constraints flexibly and easily.

Global constraints for spanning trees have been proposed in the constraint programming literature. The *Minimum Spanning Tree* constraint *MST(G,T,W)* [6] is defined on two graph variables G, T, and a vector W of scalar variables which is satisfied if T is a minimum spanning tree of G, where the weights of edges of G and T are specified by W. The *Weight-Bounded Spanning Tree* constraint *WBST(G,T,I,W)* [7] is defined on two graph variables G and T, a scalar variable I and a vector of scalar variables W which is satisfied if T is a spanning tree of G whose total weight is less than or equals to I, where the weights of edges of G and T are specified by W. The *weighted spanning tree* constraint [13] is a simpler form of *WBST(G,T,I,W)* which is defined on the neighbor representation of a graph G; each edge of G has a cost and a value I. This constraint states that there exists a spanning tree in G whose cost is at most I. None of these global constraints is available in existing constraint solvers; only the underlying principles are described in the literature. Moreover, in the Quorumcast Routing problem, the tree is not required to span the whole tree but only a subset of its nodes. This makes the global constraint much more complex. Only the *MST(G,T,W)* constraint [6], with non-fixed graphs G and T, and non-fixed weight W would be relevant. Such a constraint also requires having high level graph and tree domain variables, not available in existing solvers. Solving our QR problem with a global constraint on spanning tree was thus not possible.

1.3 Structure of the paper

The rest of this paper is organized as follows. In Section 2, we propose the CP model for this problem. The two search components will be presented in Section 3. Section 4 gives experimental results of the proposed model and Section 5 concludes the paper and sketches some future research directions.

2 CP model

In [12], a CP model was proposed for solving the Diameter Constrained Minimum Spanning Tree problem. The key idea in that paper for representing a spanning tree with a so-called root r of a given graph $G = (V, E)$ $(r \in V)$ is to use, for each node v, a variable $x(v)$ representing the successor of v on the unique path from v to r of the spanning tree (the solution) and a variable $y(v)$ representing the length of the path from v to r. By imposing the constraint $y(v) = y(x(v)) + 1, \forall v \in V \setminus \{r\}$, cycles are avoided and the set $\{x(v) \mid \forall v \in V\}$ represents a spanning tree of the given graph.

We extend this model and propose a model for representing a subtree of a given graph. To do this, we introduce a dummy node $\perp$ such that $\perp \notin V$.

2.1 Variables

For each node $v \in V$:

- The variable $x(v)$ represents the successor of v on the unique path from v to r on T. The domain of $x(v)$ is denoted by $D(x(v))$ and is defined to be the set of adjacent vertices of v in G plus $\perp$: $D(x(v)) = \{u \in V \mid (u, v) \in E\} \cup \{\perp\}$. If v is not in T, then $x(v) = \perp$. Moreover, for the root r, we have $x(r) = r$.
- The variable $y(v)$ represents the length of the path from v to r on T. It is undefined if $v \notin T$.

2.2 The CP model

$$min \sum_{v\in V\setminus\{r\}} w(v,x(v)) \tag{1}$$

s.t.

$$x(v) \neq \perp \Rightarrow y(v) = y(x(v)) + 1,\ \forall v \in V \setminus \{r\} \tag{2}$$

$$x(u) = \perp \Rightarrow x(v) \neq u,\ \forall u, v \in V \tag{3}$$

$$\sum_{v\in S}(x(v) \neq \perp) \geq q \tag{4}$$

$$x(r) = r \tag{5}$$

$$y(r) = 0 \tag{6}$$

In the objective function (1), we assume that $w(v, \perp) = 0, \forall v \in V$. The constraint (2) plays the role of eliminating cycles. Constraint (3) specifies that if a node u is not included in the solution, then it cannot be a successor of any other node v. The constraint (4) states that the number of multicast nodes must be at least q (called the *quorum* constraint). By convention, constraints (5) and (6) impose the successor of the root and the length of the path from the root to itself.

The above CP model could be simplified by replacing the constraint (2) by the constraints:

$$y(v) = y(x(v)) + 1,\ \forall v \in V \setminus \{r\} \tag{7}$$

and by posting the constraint $y(\perp) = -1$. However, our experimental results showed that this simplified model does not perform better than the model (1)–(6). In this model, the guarded constraints (2) are delayed until the guard becomes true. This avoids substantial time consuming calls to the propagation methods. Note also that the CP model (1)–(6) differs from that of [12] by the appearance of the $\perp$ value and the guarded constraints. This allows of modeling the problems in which a subtree under constraints of a given graph must be determined while the model of [12] can only be applied to the problems of determining a spanning tree of a given graph.

We can see that the CP model can easily be extended for modeling other constraints on trees. For instance, the constraint specifying that the degree of each node of the tree cannot exceed a given bound D can be modeled as:

$$\sum_{v\in V}(x(v) = u) \leq D - 1, \forall u \in V \setminus \{r\}$$

$$\sum_{v\in V\setminus\{r\}}(x(v) = r) \leq D$$

If each edge $e \in E$ is associated with a delay $d(e) > 0$, we can also easily model the constraint specifying that the total delay of each path from the source to a multicast node cannot exceed a given value L. To do this, we use a variable $y(v)$ for each node $v \in V$ representing the delay of the path from the source to v. We then replace the constraint (2) of the CP model by $x(v) \neq \perp \Rightarrow y(v) = y(x(v)) + d(v, x(v)),\ \forall v \in V\setminus\{r\}$ and add the constraint $x(v) \neq \perp \Rightarrow y(v) \leq L, \forall v \in S$ to the model.

```
1        Solver<CP> cp();
2        var<CP>{int} x[i in 1..n](cp,D[i]);
3        var<CP>{int} y[1..n](cp,0..n);

5        minimize<cp>
6          sum(i in x.rng(): i != r)(w[i,x[i]])
7        subject to{
8          forall(i in 1..n: i != r)
9            cp.post((x[i] != NULL) ~> (y[i] == y[x[i]] + 1));

11         forall(i in 1..n, j in 1..n){
12           cp.post((x[j] == NULL) <= (x[i] != j));
13         }

15         cp.post(sum(i in S)(x[i] != NULL) >= q);

17         cp.post(x[r] == r);
18         cp.post(y[r] == 0);
19       }
```

Fig. 1 The Model

The model in Comet [4] is given in Figure 1. Lines 2–3 declare the decision variables `x` and `y` in which `1..n` represents the nodes of the given graph. The variable `x[i]` represents the successor of the path from the node `i` to `r` in the solution. `D[i]` is the domain of `x[i]` which consists of the adjacent nodes of the node `i` plus the `NULL` value (which represents the $\perp$ value encoded by the value -1). The variable `y[i]` represents the length of the path from the node `i` to `r` in the solution. The domain of `y[i]` is `0..n`. The objective function is stated in line 6. The constraint (2) is stated in lines 8–9. The constraint (3) is stated in lines 11–13 and line 15 states the quorum constraint (4). In line 9, we used the Comet blocking implication as the expression `y[x[i]]` is meaningless when `x[i]=NULL`. The right hand side of the implication is only considered when `x[i]`$\neq$ `NULL`. In line 12, we use the Boolean encoding of an implication instead of the blocking implication "$\sim$>" in order to enable the propagation in two directions. When a variable `x[j]` is assigned to `NULL`, the value `j` is removed from the domain of all other variables `x[i]`. When a variable `x[i]` is assigned to `j`, the value `NULL` is removed from the domain of `x[j]`.

3 The Search

The search is a procedure that defines the search tree and specifies its traversal. At each node of the search tree, we have a partial solution: some variables are instantiated or assigned. When all the variables are instantiated, we then have a solution. A branch and bound procedure is also used to handle the optimization aspects.

At any step of the search, the partial solution P can be represented by a directed graph $G_P = (V_P, A_P)$ (as illustrated in Figure 2) in which the set of nodes $V_P \subseteq V$ is associated with a set of variables (each node v is associated with a variable $x(v)$) which are non-instantiated or are instantiated (but not to $\perp$):

$$V_P = \{v \in V \mid 1 < |D(x(v))| \vee D(x(v)) = \{u\} \wedge u \neq \perp\}$$

Henceforth, we use the words nodes and variables interchangeably in the discussion. If the variable $x(v)$ is instantiated, we say the node v is instantiated. We denote by BV_P the set of nodes of V_P that are already known to be included in the solution ($\perp$ has been removed from the domains of these nodes)

$$BV_P = \{v \in V_P \mid \perp \notin D(v)\}$$

Each node of BV_P is said *bold* node (see, for example, nodes r, 2, 3, 4, 5, 6, 7, 8, 9, 10, 16 in Figure 2). The set of arcs A_P is defined as follows:

$$A_P = \{(u, v) \mid u, v \in V_P \wedge v \in D(x(u))\}$$

The set A_P is partitioned into *bold* arcs set BA_P (the arcs that are known to be in the solution) and *dashed* arcs set DA_P (the arcs that are not known to be or not to be in the solution): (see Figure 2)

$$BA_P = \{(u, v) \in A_P \mid |D(x(u))| = 1\}$$

$$DA_P = \{(u, v) \in A_P \mid |D(x(u))| > 1\}$$

The set of BA_P induces a set of disjoint rooted trees F_P. We denote R_P the rooted tree of F_P with root r. We also use R_P to denote the set of its nodes if there is no ambiguity (in Figure 2, $R_P = \{r, 2, 3, 4\}$).

As usual, we will consider a depth-first traversal of the search tree. For the definition of the search tree, different strategies will be considered.

3.1 First-Fail Search

Our first search component is the standard labeling approach with a First-Fail heuristic. At each node of the search tree, an (uninstantiated) variable is selected. Then for each of the values in its domain, a child node is created with the variable assigned to this value. Our First-Fail search component (FF), depicted in Figure 3, is a generic heuristic that can thus be applied to any problem. It considers the non-instantiated variables in increasing order of the size of their domains for instantiation (line 2). It then tries all values of its domain (line 3). Line 4 assigns the selected value to the chosen variable. An increasing order of the different values in the domain of the variable is taken, but this does not influence the strategy.

An extension of the FF heuristic is to use the IMP heuristic [2] for finding the primal bound B^0 and post a constraint saying that the objective function must be smaller than B^0 before starting the search. This version is denoted by BFF.

3.2 Shortest-Paths Heuristic Search (SPH)

The First-Fail heuristic is efficient in many cases but it is generic: it cannot exploit the problem structure. We propose here a heuristic which combines First-Fail with a greedy heuristic based on Shortest-Paths. Such a search strategy quickly finds a high-quality solution. This high-quality solution can then prune the search space and improve the efficiency. The proposed heuristic is inspired by the MPH heuristic of [2].

The SPH heuristic is as follows. If R_P contains less than q multicast nodes, we select the multicast non-instantiated node v having the shortest path in G_P to R_P, such

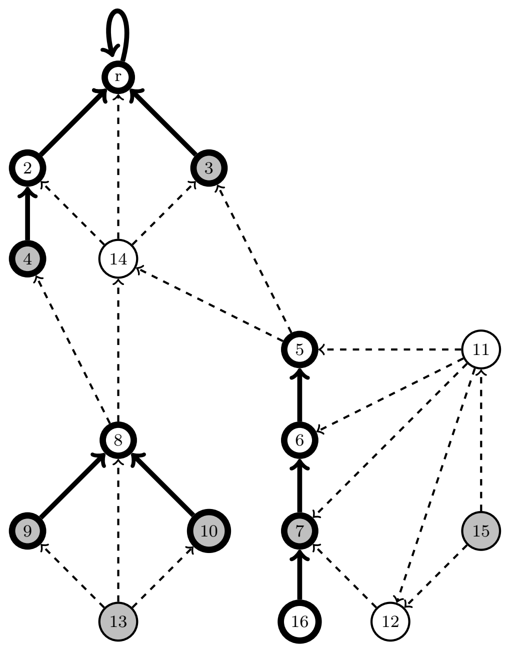

Fig. 2 Partial solution: grey nodes are multicast nodes, the value of q is 6

```
using{
   forall(i in x.rng(): !x[i].bound()) by (x[i].getSize())
      tryall<cp>(v in x[i].getMin()..x[i].getMax():
            x[i].memberOf(v)) by (w[i,v])
         label(x[i],v);
}
```

Fig. 3 The First-Fail Heuristic Search

path being denoted by $\mathcal{P}(v)$. The next variables to be instantiated are the variables corresponding to nodes on $\mathcal{P}(v)$. For each node $vi \in \mathcal{P}(v)$, the first value tried for $x(vi)$ is the successor of vi on $\mathcal{P}(v)$. This gives directly a high-quality solution without backtracking. All the other values are then considered to achieve a complete search.

The SPH heuristic search is depicted in Figure 4. The variable `C1` (line 3) counts the number of multicast nodes included in R_P in which the variable `RP` represents the set of nodes of R_P. The variable `S2` represents the set of multicast nodes that are not in R_P (see lines 4–7). At any step, if the number of multicast instantiated and non `NULL` nodes plus the number of multicast non-instantiated nodes is less than `q`, then the search backtracks because it cannot extend this to any feasible solution (lines 9–10). Lines 12–33 perform the search when the number of multicast nodes in R_P is less than `q`. Lines 12–14 compute the shortest paths from all nodes of $V_P \setminus R_P$ to R_P

in which `d[v]` (line 13) is the distance (i.e., the cost of the shortest path) from `v` to R_P and `pr[v]` (line 14) is the successor of `v` in this shortest path. Line 16 selects the multicast node `v` of $V_P \setminus R_P$ having the smallest distance to R_P. If the path from `v` to R_P does not exist (line 17), then the search backtracks because this cannot be extended to a feasible solution which contains at least `q` multicast nodes. Otherwise, lines 20–31 try to assign values to variables corresponding to nodes on the shortest path from `v` to R_P: for each node `vi`, the search first tries the successor `fvi` of `vi` for `x[vi]` (line 24). It then tries other values `val` for `x[vi]` (lines 27–28) to ensure the completeness of the search. Lines 35–37 apply the First-Fail heuristic when the number of multicast nodes in the current partial solution is greater than or equal to `q`. These instructions will consider all the possible values for the remaining uninstantiated variables, ensuring the completeness of the search. Thanks to the completeness of SPH, the resulting solution implicitly contains a proof of optimality.

3.3 Incomplete Shortest-Paths Heuristic Search (IS)

We now consider an incomplete search component, depicted in Figure 5. The differences of IS from SPH are:

1. line 16 of Figure 5 in contrast with line 16 of Figure 4
2. lines 21–25 of Figure 5 in contrast with lines 21–31 of Figure 4
3. lines 29–31 of Figure 5 in contrast with lines 35–37 of Figure 4

The first two differences of IS from SPH are that IS tries all multicast nodes $v \in V_P \setminus R_P$ (line 16 in Figure 5) instead of selecting the multicast node `v` of $V_P \setminus R_P$ having the smallest distance `d[v]` to R_P (line 16 in Figure 4). For each path from `v` to R_P (if it exists), the search assigns the successor `fvi` of `vi` to variable `x[vi]`, for all node `vi`, except the last one, of the considered path (see lines 20–25). It does not try other values for `x[vi]` as in the SPH (see lines 25–29 in Figure 4). This makes the search incomplete. The last difference of IS from SPH is that if R_P contains at least q multicast nodes then IS assigns all other variables to `NULL` (see lines 29–31 in Figure 5) because BA_P corresponds to a feasible tree, as shown in Proposition 1, while SPH performs a non-deterministic program for instantiating all non-instantiated variables because, with SPH, BA_P does not always correspond to a tree (lines 35–37 in Figure 4).

Proposition 1 *During the IS search, if the number of multicast nodes of R_P is greater or equal to q, then BA_P corresponds to a tree.*

Proof The IS search, at each step, finds and attaches a path $\mathcal{P}(v)$ from a node $v \in V_P \setminus R_P$ to R_P. All arcs of $\mathcal{P}(v)$ are attached to the rooted tree R_P to constitute R'_P which is another rooted tree with root r. For example, in Figure 6, $R_P = \{r, 2, 3, 4, 14, 8, 9, 10\}$, $v = 7$, $\mathcal{P}(v) = \langle 7, 6, 5, 3\rangle$ and $R'_P = \{r, 2, 3, 4, 14, 8, 9, 10, 7, 6, 5\}$. A bold arc $(u, v) \in BA_P$ having no common endpoints with R'_P may appear, thanks to propagation, only when (see, for example, the arc (9,8) in Figure 7):

1. the pruning is performed by the current bound of the objective function, and
2. the pruning is performed by the quorum constraint, and
3. u is a multicast node.

```
using{
  while(!bound(x)){
    int C1 = sum(v in S: RP.contains(v))(1);
    set{int} S2();
    forall(v in S: !RP.contains(v))
      if(x[v].bound() && x[v] != NULL || !x[v].bound())
        S2.insert(v);

    if(C1 + S2.getSize() < q){
      cp.fail();
    }else if(C1 < q){
      spa.computeShortestPaths(x);
      float[] d = spa.getDistances();
      int[] pr = spa.getSucc();

      selectMin(v in S2)(d[v]){
        if(d[v]==MAX_INT){
          cp.fail();
        }else{
          int vi = v;
          while(pr[vi] != vi){
            int fvi = pr[vi];
            try<cp>
              label(x[vi],fvi);
            |
            {
              tryall<cp>(val in x[vi].getMin()..x[vi].getMax():
                  val != fvi && x[vi].memberOf(val))
                label(x[vi],val);
            }
            vi = fvi;
          }
        }
      }
    }else{
      forall(i in x.rng(): !x[i].bound()) by (x[i].getSize())
        tryall<cp>(v in x[i].getMin()..x[i].getMax():
            x[i].memberOf(v)) by (w[i,v])
          label(x[i],v);
    }
  }
}
```

Fig. 4 The SPH Search

This pruning makes the domain of $x(u)$ a singleton: $D(x(u)) = \{v\}$ $(v \neq \bot)$. The number of multicast nodes without $\bot$ in their domain is always less than or equal to q. The second pruning is triggered only when the number of multicast nodes of V_P is equal to q and this pruning removes $\bot$ from the domain of all multicast nodes of V_P: in Figure 7, $\bot$ is removed from $D(10), D(13)$ and the nodes 10, 13 become bold nodes. Hence, if BA_P has more than one tree, then the number of of multicast nodes of V_P equals to q and some bold multicast nodes are outside R_P. Thus we have that if BA_P has more than one tree, then the number of multicast nodes of R_P is less than q. ∎

```
using{
  while(!bound(x)){
    int C1 = sum(v in S: RP.contains(v))(1);
    set{int} S2();
    forall(v in S: !RP.contains(v))
      if(x[v].bound() && x[v] != NULL || !x[v].bound())
        S2.insert(v);

    if(C1 + S2.getSize() < q){
      cp.fail();
    }else if(C1 < q){
      spa.computeShortestPaths(x);
      float[] d = spa.getDistances();
      int[] pr = spa.getSucc();

      tryall<cp>(v in S2) by (d[v]){
        if(d[v] == MAX_INT){
          cp.fail();
        }else{
          int vi = v;
          while(pr[vi] != vi){
            int fvi = pr[vi];
            label(x[vi],fvi);
            vi = fvi;
          }
        }
      }
    }else{
      forall(i in x.rng(): !x[i].bound()){
        label(x[i],NULL);
      }
    }
  }
}
```

Fig. 5 The Incomplete Search

4 Experiments

In [11], the proposed complete algorithm (denoted by Low) was tested on random instances of 30 nodes with different connectivities: 4, 8, 12. We re-implemented this state-of-the-art exact algorithm in C++ for the comparison[2]. We also re-implemented the state-of-the-art IMP heuristic algorithm of [2] in Comet for comparing it with our incomplete IS method. To evaluate the proposed CP models, we generated random instances based on the description given in [11] but with larger graphs: graphs with n = 30, 40, 50, 60 nodes and different connectivities $c \in \{4, 8, 12\}$. The values of q and the number of multicast nodes ($|S|$) are generated with eight cases: $\langle q, |S|\rangle \in \{\langle 2,3\rangle, \langle 3,5\rangle, \langle 4,7\rangle, \langle 5,9\rangle\}, \langle 5,20\rangle, \langle 7,20\rangle, \langle 5,25\rangle\}, \langle 7,25\rangle\}$. For each tuple $\langle n, c, q, |S|\rangle$, we randomly generated 10 instances. In total, we have 960 instances which are divided into two classes: the first class contains instances with $\langle q, |S|\rangle \in \{\langle 2,3\rangle, \langle 3,5\rangle, \langle 4,7\rangle, \langle 5,9\rangle\}$[3] where $\binom{|S|}{q}$ is small and the second class contains instances

[2] Comet is a programming environment based on a just-in-time compiler. A program written in Comet always runs slower than in C++.

[3] The values of $\langle q, |S|\rangle$ in this class are exactly as same as in [11].

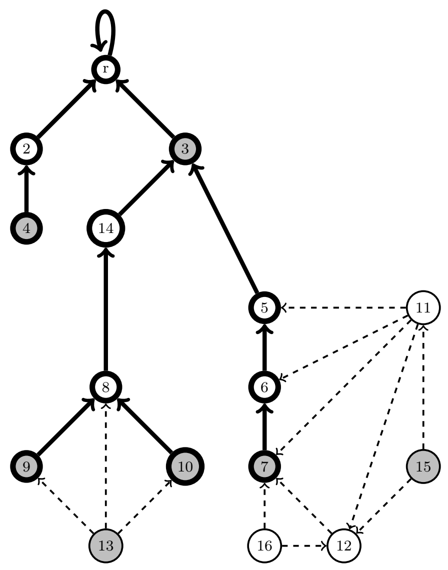

Fig. 6 Partial solution: BA_P contains only one rooted tree ($q = 6$)

with $\langle q, |S| \rangle \in \{\langle 5, 20\rangle, \langle 5, 25\rangle, \langle 7, 20\rangle, \langle 7, 25\rangle\}$ where $\binom{|S|}{q}$ is large (ranging from 15 10^3 to 480 10^3). Due to the stochastic components, for each instance, the First Fail with primal bound (BFF), and the Shortest Path (SPH) heuristics are executed 10 times. In order to show the interest of the SPH algorithm, the worst execution of the SPH and the best execution of the BFF among these will be reported for the comparison. For each instance, the FF algorithm is executed once. The time limit for each execution of all algorithms is 30 minutes. The experiments were performed on XEN virtual machines with 1 core of a CPU Intel Core2 Quad Q6600 @2.40GHz and 1GB of RAM.

4.1 Comparison of Exact Algorithms

We first compare the efficiency of Low, FF, BFF, and SPH in term of the number of instances (among 480 instances of each class) solved within a given time limit[4]. The results are given in Tables 1 and 2. Column 1 gives different values of the time limit. Columns 2–5 respectively give the percentages of instances solved within the corresponding time limit by Low, FF, BFF, and SPH algorithms. Figures 8 and 9 describe the evolution of this information. They show that our three algorithms FF, BFF, SPH solve more instances than Low for any value of time limit and SPH is better than FF and BFF algorithms. It shows that in 1000 seconds, the SPH algorithm solves more than 97% of the instances in the first class and more than 91% of the instances in the second class. Within 1800 seconds, the algorithm Low solves only 39.38% of the

[4] The algorithm is complete within the given time limit.

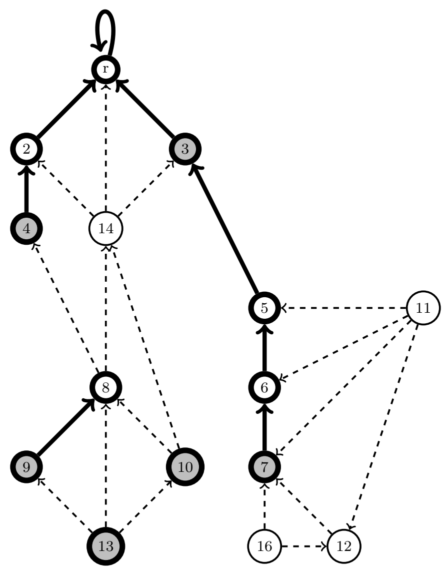

Fig. 7 Partial solution: BA_P contains more than one rooted tree ($q = 6$)

instances in the first class and 42.92% of instances in the second class. Note that the efficiency of Low relies strongly on the pruning of the CAP scheme for reducing the search space. Among 480 random instances of the first class, there are 189 instances in which the CAP scheme cannot reduce the search space. Moreover, the tables also show that in all values of time limit except the value 1s in the first class (see line 1 in Table 1), the percentage of instances solved by the SPH algorithm is always higher than that solved by the BFF algorithm.

Table 3 summarizes the average of execution times of the SPH algorithm for each group of instances, characterized by the tuple $\langle n, c, q, |S|\rangle$ (each group has 10 instances). Columns 1 and 4 are instance groups. Columns 2 and 5 present the average execution times (only instances where the SPH algorithm terminates within 1800 seconds are considered). Columns 3 and 6 gives the number of instances of each group which are solved within 1800 seconds. Figures 10 and 11 present the execution time information under a column format. We can see that in many instances of the second class, the SPH algorithm is very efficient (a few seconds). In these instances, as the value of $\binom{|S|}{q}$ is large, it would be irrelevant to apply an exact MST algorithm to each possible subset. We now characterize hard/easy instances. For the first class, generally, the instances are harder when one of the values of n,c,$\langle q, |S|\rangle$ increases[5] and the other values stay the same. The reasons are:

- When n increases, the number of variables of the CP model increases.
- When c increases, the sizes of domains of the variables in the CP model increase.

[5] In the first class, when q increases, $|S|$ also increases and vice versa.

time limit (s)	Low (%)	FF (%)	BFF (%)	SPH (%)
1	4.79	18.54	**24.58**	22.08
2	6.04	30.63	37.50	**37.92**
5	7.92	43.33	49.79	**51.67**
10	10.42	51.25	57.50	**63.33**
20	12.50	58.54	65.21	**70.21**
50	17.08	71.46	73.96	**79.17**
100	20.63	77.29	80.42	**86.46**
200	25.00	83.13	87.50	**91.25**
500	31.04	89.38	93.33	**95.21**
1000	35.63	93.33	96.46	**97.29**
1200	36.67	93.33	97.08	**97.71**
1500	38.13	94.17	97.71	**98.33**
1800	39.38	94.58	98.13	**98.75**

Table 1 Percentage of instances solved within a time limit (first instance class)

time limit (s)	Low (%)	FF (%)	BFF (%)	SPH (%)
1	5.00	7.92	10.83	**14.58**
2	6.46	17.50	21.67	**25.21**
5	10.83	28.96	33.75	**38.54**
10	12.71	38.96	45.00	**50.00**
20	14.58	47.29	53.96	**57.50**
50	19.17	58.54	65.21	**68.13**
100	24.38	65.21	71.04	**76.46**
200	30.00	74.38	78.96	**82.92**
500	33.75	82.08	86.04	**88.54**
1000	39.17	87.08	90.00	**91.67**
1200	40.21	88.75	90.63	**92.29**
1500	41.46	90.00	91.88	**93.54**
1800	42.29	90.63	92.50	**94.38**

Table 2 Percentage of instances solved within a time limit (second instance class)

– When $\langle q, |S| \rangle$ increases, the number of multicast nodes that the search must visit increases.

For the second class, we can see that for instances with the same values of n and c, the most difficult instances are those having large q and small $|S|$, i.e., $q = 7, |S| = 20$. The reason is when q is higher, the search must visit more multicast nodes. Moreover, when $|S|$ is higher, there are more choices for feasible solutions and the search quickly finds a feasible solution with very good quality. This yields more pruning.

instance group	$\bar{t}$ (s)	♯solved	instance group	$\bar{t}$ (s)	♯solved
n30-c4-q2-S3	0.65	10	n30-c4-q5-S20	0.85	10
n30-c8-q2-S3	0.95	10	n30-c8-q5-S20	1.03	10
n30-c12-q2-S3	0.72	10	n30-c12-q5-S20	1.70	10
n30-c4-q3-S5	1.02	10	n30-c4-q7-S20	6.78	10
n30-c8-q3-S5	0.83	10	n30-c8-q7-S20	11.76	10
n30-c12-q3-S5	0.97	10	n30-c12-q7-S20	25.56	10
n30-c4-q4-S7	3.38	10	n30-c4-q5-S25	0.53	10
n30-c8-q4-S7	4.21	10	n30-c8-q5-S25	0.63	10
n30-c12-q4-S7	2.53	10	n30-c12-q5-S25	0.90	10
n30-c4-q5-S9	7.74	10	n30-c4-q7-S25	1.53	10
n30-c8-q5-S9	15.75	10	n30-c8-q7-S25	2.59	10
n30-c12-q5-S9	9.73	10	n30-c12-q7-S25	7.04	10
n40-c4-q2-S3	1.90	10	n40-c4-q5-S20	5.57	10
n40-c8-q2-S3	0.55	10	n40-c8-q5-S20	2.56	10
n40-c12-q2-S3	0.65	10	n40-c12-q5-S20	4.30	10
n40-c4-q3-S5	2.69	10	n40-c4-q7-S20	93.73	10
n40-c8-q3-S5	1.99	10	n40-c8-q7-S20	113.39	10
n40-c12-q3-S5	3.54	10	n40-c12-q7-S20	135.77	10
n40-c4-q4-S7	13.28	10	n40-c4-q5-S25	3.60	10
n40-c8-q4-S7	21.57	10	n40-c8-q5-S25	1.33	10
n40-c12-q4-S7	39.96	10	n40-c12-q5-S25	2.34	10
n40-c4-q5-S9	57.00	10	n40-c4-q7-S25	41.02	10
n40-c8-q5-S9	58.22	10	n40-c8-q7-S25	22.74	10
n40-c12-q5-S9	91.95	10	n40-c12-q7-S25	43.10	10
n50-c4-q2-S3	2.20	10	n50-c4-q5-S20	8.82	10
n50-c8-q2-S3	0.99	10	n50-c8-q5-S20	30.87	10
n50-c12-q2-S3	1.07	10	n50-c12-q5-S20	59.84	10
n50-c4-q3-S5	2.70	10	n50-c4-q7-S20	229.17	10
n50-c8-q3-S5	6.97	10	n50-c8-q7-S20	553.63	8
n50-c12-q3-S5	4.52	10	n50-c12-q7-S20	569.39	8
n50-c4-q4-S7	41.86	10	n50-c4-q5-S25	5.77	10
n50-c8-q4-S7	63.68	10	n50-c8-q5-S25	9.87	10
n50-c12-q4-S7	34.05	10	n50-c12-q5-S25	15.99	10
n50-c4-q5-S9	212.44	10	n50-c4-q7-S25	90.48	10
n50-c8-q5-S9	253.73	9	n50-c8-q7-S25	251.55	10
n50-c12-q5-S9	174.77	10	n50-c12-q7-S25	214.59	9
n60-c4-q2-S3	1.89	10	n60-c4-q5-S20	29.45	10
n60-c8-q2-S3	2.13	10	n60-c8-q5-S20	117.25	10
n60-c12-q2-S3	5.90	10	n60-c12-q5-S20	113.45	10
n60-c4-q3-S5	10.80	10	n60-c4-q7-S20	284.36	7
n60-c8-q3-S5	17.74	10	n60-c8-q7-S20	1114.98	6
n60-c12-q3-S5	21.75	10	n60-c12-q7-S20	762.72	2
n60-c4-q4-S7	52.30	10	n60-c4-q5-S25	11.28	10
n60-c8-q4-S7	95.62	10	n60-c8-q5-S25	29.70	10
n60-c12-q4-S7	175.30	10	n60-c12-q5-S25	28.80	10
n60-c4-q5-S9	306.58	9	n60-c4-q7-S25	103.63	9
n60-c8-q5-S9	511.98	10	n60-c8-q7-S25	476.26	7
n60-c12-q5-S9	270.79	6	n60-c12-q7-S25	562.29	7

Table 3 Summary of the average of execution time of the SPH algorithm for 10 instances of each group

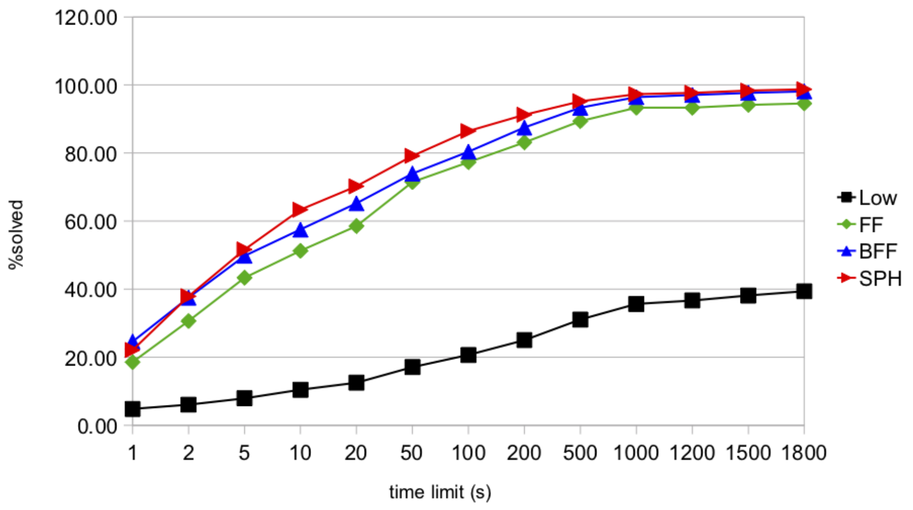

Fig. 8 Complete algorithms: Comparison between FF, BFF, SPH, and Low heuristics in term of number of instances solved in a given time limit (first instance class)

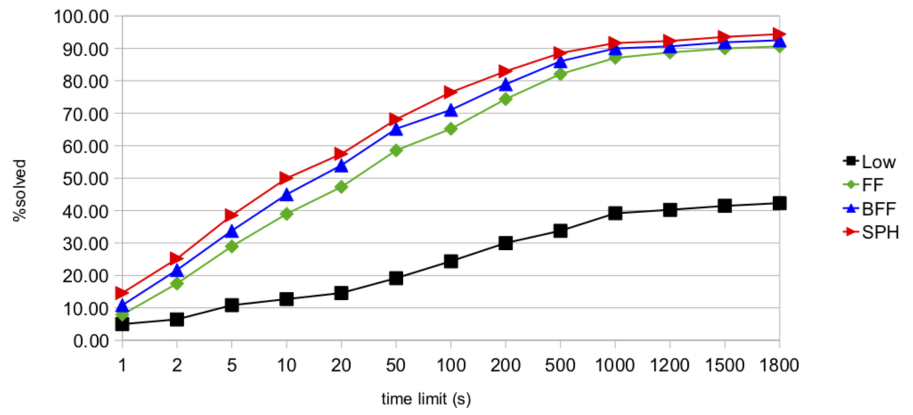

Fig. 9 Complete algorithms: Comparison between FF, BFF, SPH, and Low heuristics in term of number of instances solved in a given time limit (second instance class)

4.2 Comparison of Incomplete Algorithms

We now compare the proposed incomplete IS method with the state of the art IMP heuristic. The IMP heuristic has been reimplemented in Comet, based on the description in [2]. Due to the stochastic components, the IMP algorithms is executed 10 times, for 9600 total executions. The IS algorithm has not random factor, it is thus executed once for each instance. Experimental results show that our IS heuristic finds optimal solutions 95.42% of the time in the first instance class and also 95.42% of the time in the second class while the IMP heuristic finds optimal solutions 59.67% of the time in the first instance class and 39.78% of the time in the second instance class. Of course, neither IS nor IMP are able to prove the optimality of the computed solution.

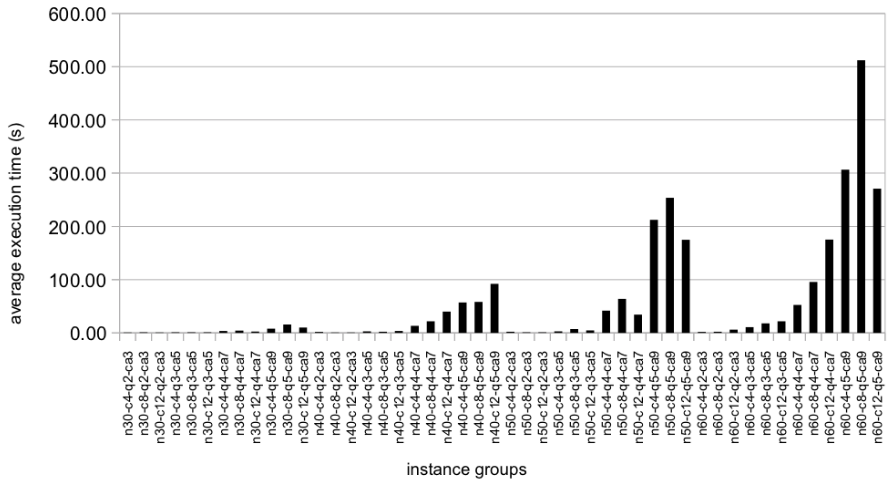

Fig. 10 Average of execution time of the SPH algorithm for 10 instances of each group in the first class

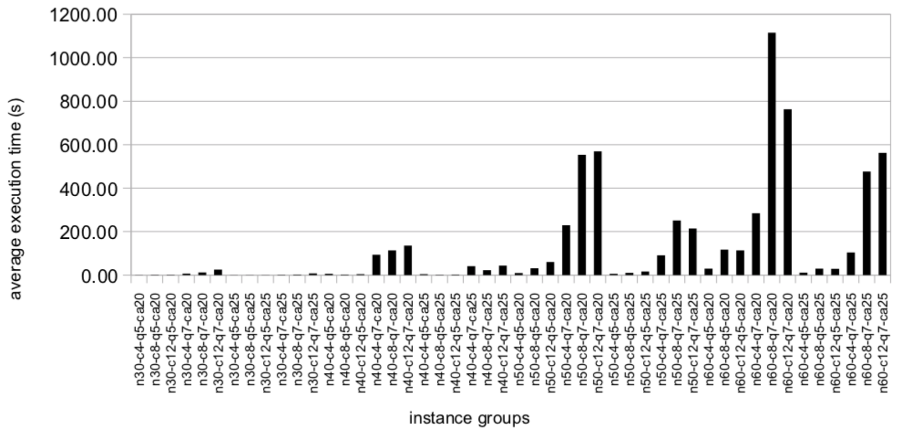

Fig. 11 Average of execution time of the SPH algorithm for 10 instances of each group in the second class

A complete method was used to assess the optimality of the results. For evaluating the quality of non-optimal solutions found by the IS algorithm, Tables 4 and 5 presents the experimental results on 22 instances where the IS algorithm cannot find optimal solutions. Columns 2–5 present the minimal, maximal, the average of the value of the objective function, and the average of execution time found by the IMP heuristic in 10 runs. The objective values found by the IS algorithm and their execution times are presented in columns 6–7. The last column shows the optimal values of the objective function. We see that the IS algorithm finds better solutions than the IMP algorithm in 15 out of 22 instances in the first instance class and 14 out of 22 instances in the second class while the IMP finds better solutions than the IS algorithm in only one instance in both two instance classes. Note that for each of the remaining 458 instances

instance	IMP				IS		opt^*
	m	M	μ	$\bar{t}$ (s)	m	$\bar{t}$ (s)	
g-n30-c8.ins10-q2-S3	128	146	138.8	0.13	128	0.58	122
g-n30-c12.ins3-q2-S3	108	108	108	0.16	108	0.54	107
g-n40-c12.ins1-q2-S3	117	117	117	0.20	117	0.6	110
g-n60-c12.ins8-q2-S3	195	195	195	0.32	**165**	0.88	164
g-n40-c12.ins1-q3-S5	155	156	155.6	0.20	155	0.6	149
g-n50-c8.ins2-q3-S5	150	150	150	0.22	**148**	0.66	137
g-n50-c12.ins8-q3-S5	132	132	132	0.22	132	0.75	129
g-n60-c8.ins10-q3-S5	142	142	142	0.28	**133**	0.88	132
g-n30-c4.ins8-q4-S7	241	278	255.8	0.13	**238**	0.52	208
g-n30-c8.ins10-q4-S7	174	174	174	0.15	**147**	0.5	142
g-n40-c4.ins1-q4-S7	321	321	321	0.17	**310**	0.7	293
g-n50-c4.ins7-q4-S7	258	258	258	0.29	258	0.72	253
g-n50-c12.ins7-q4-S7	189	189	189	0.25	**145**	0.84	137
g-n60-c4.ins9-q4-S7	281	281	281	0.31	**258**	1.05	253
g-n60-c8.ins10-q4-S7	164	164	164	0.30	**162**	1	153
g-n60-c12.ins8-q4-S7	207	207	207	0.36	**186**	1.32	183
g-n60-c12.ins9-q4-S7	164	164	164	0.34	**158**	1.05	157
g-n30-c12.ins5-q5-S9	**82**	83	82.8	0.20	83	0.52	82
g-n40-c12.ins7-q5-S9	143	143	143	0.18	**123**	0.59	119
g-n50-c8.ins7-q5-S9	230	230	230	0.26	**220**	2.76	218
g-n60-c12.ins5-q5-S9	171	172	171.7	0.34	**155**	1.73	152
g-n60-c12.ins7-q5-S9	158	158	158	0.28	**154**	4.39	153

Table 4 Results on instances where IS cannot find optimal solutions (first instance class)

instance	IMP				IS		opt^*
	m	M	μ	$\bar{t}$ (s)	m	$\bar{t}$ (s)	
g-n40-c4.ins1-q5-S20	219	219	219	0.17	**213**	1.16	199
g-n40-c4.ins2-q5-S20	226	226	226	0.17	**219**	0.83	202
g-n40-c4.ins5-q5-S20	223	223	223	0.17	**220**	1.3	218
g-n40-c4.ins10-q5-S20	137	137	137	0.17	137	0.7	130
g-n40-c12.ins8-q5-S20	77	77	77	0.16	**76**	1	75
g-n60-c4.ins7-q5-S20	167	167	167	0.30	167	1.39	157
g-n60-c8.ins4-q5-S20	124	132	131.2	0.29	124	2.52	118
g-n40-c4.ins8-q7-S20	274	274	274	0.17	**254**	1.7	246
g-n50-c4.ins1-q7-S20	313	313	313	0.21	313	14.63	299
g-n50-c4.ins5-q7-S20	243	243	243	0.21	**242**	2.46	235
g-n50-c12.ins3-q7-S20	186	186	186	0.21	**178**	55.81	173
g-n30-c4.ins2-q5-S25	79	79	79	0.13	79	0.51	77
g-n30-c4.ins4-q5-S25	180	180	180	0.13	**179**	0.53	166
g-n30-c8.ins7-q5-S25	**83**	**83**	**83**	0.14	85	0.75	82
g-n30-c12.ins8-q5-S25	96	96	96	0.13	**93**	0.78	91
g-n60-c12.ins10-q5-S25	101	101	101	0.28	101	1.67	100
g-n30-c4.ins3-q7-S25	249	249	249	0.12	**229**	3.94	223
g-n40-c4.ins1-q7-S25	250	250	250	0.16	**235**	6.19	226
g-n40-c12.ins5-q7-S25	140	140	140	0.17	**135**	32.18	132
g-n50-c4.ins5-q7-S25	243	243	243	0.22	**238**	4.12	235
g-n60-c4.ins1-q7-S25	302	302	302	0.29	**282**	4.05	271
g-n60-c12.ins9-q7-S25	157	163	160	0.29	157	75.64	156

Table 5 Results on instances where IS cannot find optimal solutions (second instance class)

of each instance class, the IS algorithm finds optimal solutions while the IMP does not. Table 4 also shows that the average execution time of the IS algorithm is relatively low. Note also that the maximum execution time of the IS algorithm is 8.45 seconds in the first instance class. In the second instance class, as the size of S is much larger, the execution time of the IS algorithm is higher: the maximal execution time of the IS algorithm is 231.6 seconds.

As expected, the incomplete IS approach is much more efficient than the complete SPH approach as the search space is smaller. What is more surprising is the high

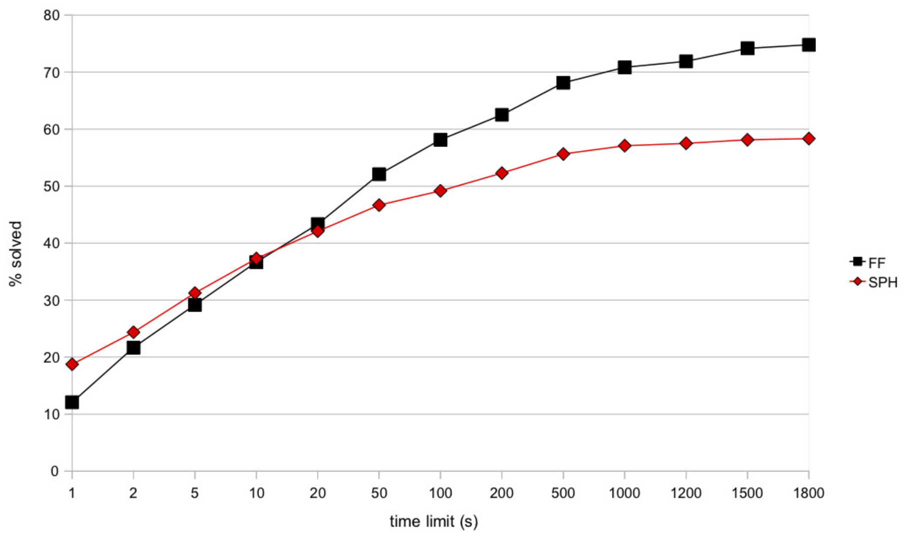

Fig. 12 Delay side constraints: Number of instances solved in a given time limit

quality of the solutions provided by IS. It obtains optimal solutions in 458 out of 480 instances of each class. Clearly, when no proof of optimality is required, the IS method is a good candidate.

4.3 Memory requirements

Table 6 summarizes the memory consumed by the considered algorithms in the second instance class. The last column gives the number of instances among 480 instances where the algorithm crashed because of lack of memory. Columns 2–5 presents the minimal, the maximal, the average, and the standard deviation of memory consumed (in MB) of the algorithms over 480 instances[6]. The memory consumed is measured in term of VmSize of the process executing the algorithms. We can see that the IMP algorithm consumed less memory than the algorithms using CP search (FF, BFF, SPH, IS) because the IMP does not use backtracking search. The exact algorithm Low of [11] crashed in 207 instances and the variation of memory requirement in non-crashed cases is high (from 2.9MB to 2965MB). On some difficult instances, the Low algorithm can be very efficient, but requires too much memory on other difficult instances.

4.4 Dealing with Side Constraints

This section illustrates on a concrete example the extensibility of the CP model. We introduce a delay side constraints, what is particularly realistic for this problem. Suppose that each edge is associated with a unit delay, and the solution must satisfy a constraint saying that the total delay of the path from the source to each node of the

[6] Only instances without memory crashed are considered

algorithm	min	max	avg	std. dev.	memory crashed
FF	51.6	56.6	55.3	1.26	0
BFF	53.2	58.3	56.9	1.30	0
SPH	52.3	56.7	55.4	0.88	0
Low	2.9	2965.3	499.0	824.78	207
IMP	30.7	31.1	30.9	0.086	0
IS	53.3	62.1	58.1	3.06	0

Table 6 Summary of memory requirement (in MB) of algorithms

tree must not exceed `maxDelay`. To do this, we simply insert the following snippet to the model in Figure 1 between lines 18 and 19:

```
forall(i in 1..n: i != r){
  cp.post((x[i] != NULL) ~> (y[i] <= maxDelay));
}
```

Experimental results with `maxDelay = 3` in the first instance class are presented in Figure 12. We can see that the SPH heuristic solves more instances than the FF heuristic in a time limit less than 10 s. But when the time limit is greater than 10 sec., the FF heuristic solves more instances than the SPH heuristic. In 30 minutes, the FF heuristic solves more than 74% of the instances while the SPH heuristic solves less than 59% of the instances. The reason is that SPH does not take into account side constraints and is thus only dedicated to the problem without side constraints. At each step, the SPH heuristic choose a node having the smallest distance to the partial tree. Without side constraints, this choice always leads to a solution, hopefully of high quality. With side constraints, this choice may violate delay constraints, hence requiring backtracking before finding a first solution. However, the FF heuristic is generic; it exploits all the constraints, including the side ones. It is thus more efficient than SPH on large problems with side constraints.

5 Conclusion

In this paper, we proposed a Constraint Programming model and two different search components for solving the quorumcast routing problem. These two search heuristics are the main contibution of this paper and are based on Shortest Path, inspired by the MPH algorithm [2]. The first search component, denoted SPH, provides a complete approach, while the IS search component only considers part of the search tree and is thus an incomplete method.

We showed that specific search heuristics exploiting the problem structure are more efficient than the generic First Fail search heuristic over the proposed CP model. The CP model with the generic First Fail heuristic is also shown to be better than the exact state-of-the-art algorithm of [11]. The proposed IS approach (the CP model+incomplete search) has been shown to outperform the state-of-the-art IMP heuristic. In addition to the efficiency, the proposed CP model can be easily extended to deal with other constraints on trees, for example, a constraint on the degrees of the nodes or on the delay of paths from the source to other nodes. We showed the feasibility of the CP model when delay side constraints are added with generic First Fail heuristic search while it requires very sophisticated and specific techniques for searching solutions when using the MIP approach. For not-too-hard instances, we can apply the CP solver as a black box with default first-fail heuristic search which is good while we cannot do this with MIP approach as it requires to state an exponential number of constraints in the modeling.

The CP models and the instances experimented in this paper are available on:

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

for future comparisons.

As future work, we intend to extend the proposed CP approach for the resolution of an extensively studied application on networks: Delay Bound Minimum Cost Multicast (DBMC) for multicast routing with QoS constraints application [9]. We will also study other side constraints on trees and develop other incomplete CP search for other classes of problems to compete with local search approaches.

References

1. E. Althaus, T. Polzin, and S. V. Daneshmand. Improving linear programming approaches for the steiner tree problem. In *Experimental and Efficient Algorithms, Lecture Notes in Computer Science, 2003, Volume 2647/2003*, pages 1–14, 2003.
2. S. Y. Cheung and A. Kumar. Efficient quorumcast routing algorithms. *In: Proceedings of INFOCOM'94*, pages 840–847, 1994.
3. M. Chimani, M. Kandyba, and P. M. I. Ljubic. Obtaining optimal k-cardinality trees fast. *ACM Journal of Experimental Algorithmics*, 14(2):5.1–5.23, 2009.
4. Comet. Comet user manual, dynadec, 2011. http://dynadec.com/.
5. B. N. Dilkina and C. P. Gomes. Solving connected subgraph problems in wildlife conservation. In *6. 7th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2010)*, pages 102–116, 2010.
6. G. Dooms and I. Katriel. The minimum spanning tree constraint. In *12th International Conference on Principles and Practice of Constraint Programming (CP2006)*, pages 211–225, 2006.
7. G. Dooms and I. Katriel. The "not-too-heavy spanning tree" constraint. In *4th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2007)*, pages 59–70, 2007.
8. B. Du, J. Gu, D. Tsang, and W. Wang. Quorumcast routing by multispace search. *Proceedings of IEEE Globecom1996*, pages 1069–1073, 1996.
9. S. Li, R. Melhem, and T. Znati. An efficient algorithm for constructing delay bounded minimum cost multicast trees. *J. Parallel Distrib. Comput.*, 64:1399–1413, 2004.
10. I. Ljubic, R. Weiskircher, U. Pferschy, G. W. Klau, P. Mutzel, and M. Fischetti. An algorithmic framework for the exact solution of the prize-collecting steiner tree problem. *Math. Program.*, 105(2-3):427–449, 2006.
11. C. P. Low. A fast search algorithm for the quorumcast routing problem. *Information Processing Letters*, 66:87–92, 1998.
12. T. F. Noronha, C. C. Ribeiro, and A. C. Santos. Solving diameter-constrained minimum spanning tree problems by constraint programming. *International Transactions in Operational Research*, 17:653–665, 2010.
13. J.-C. Régin. Simpler and incremental consistency checking and arc consistency filtering algorithms for the weighted spanning tree constraint. In *5th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2008)*, pages 233–247, 2008.
14. F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming.* Elsevier Science Inc., New York, USA, 2006.
15. Q. She, N. Kannasoot, J. P. Jue, and Y.-C. Kim. On finding minimum cost tree for multi-resource manycast in mesh networks. *Optical Switching and Networking*, 6:29–36, 2009.
16. B. Wang and J. C. Hou. An efficient QoS routing algorithm for quorumcast communication. *Computer Networks Journal*, 44(1):43–61, 2004.