RAIRO Operations Research
RAIRO Oper. Res. **35** (2001) 189-209

RESOURCE ALLOCATION IN A MOBILE TELEPHONE NETWORK: A CONSTRUCTIVE REPAIR ALGORITHM

Patrice Boizumault[1], Philippe David[1] and Housni Djellab[1, 2]

Abstract. To cope with its development, a French operator of mobile telephone network must periodically plan the purchase and the installation of new hardware, in such a way that a hierarchy of constraints (required and preferred) is satisfied. This paper presents the "constructive repair" method we used to solve this problem within the allowed computing time (1 min). This method repairs the planning during its construction. A sequence of repair procedures is defined: if a given repair cannot be achieved on a partial solution, a stronger repair (possibly relaxing more important constraints) is called upon. We tested our method on ten (both hand-made and real) problems. All our solutions were at least as good as thoses computed by hand by the engineer in charge with the planning.

Résumé. Afin de couvrir les besoins liés au développement de son réseau, un opérateur français de téléphonie mobile doit périodiquement planifier l'achat et l'installation de nouveaux matériels, tout en respectant un ensemble de contraintes (contraintes obligatoires ou préférences hiérarchisées). Cet article présente la méthode, baptisée "constructive repair", utilisée pour résoudre ce problème dans les délais impartis (1 min de temps de calcul). Cette méthode répare le planning durant sa construction. Une suite de procédures de réparation est définie : si une réparation donnée ne peut aboutir sur une solution partielle, une réparation plus forte (relâchant éventuellement des contraintes plus importantes) est appelée. Nous avons testé notre méthode sur dix problèmes (aussi bien réels que spécifiquement conçus "à la main" pour ces tests). Nos solutions sont toutes au moins aussi bonnes que celles imaginées par l'ingénieur responsable de la planification.

Keywords: Resource allocation, repair algorithms, constrained resolution time, CSP, telecommunications.

Received May, 1999.

[1] Département Informatique, École des Mines de Nantes, 4 rue Alfred Kastler, La Chantrerie, 44307 Nantes Cedex 3, France; e-mail: `{Patrice.Boizumault, Philippe.David}@emn.fr`

[2] *Currently at*: Eurodecision, 9a rue de la Porte de Buc, 78000 Versailles, France; e-mail: `housni.djellab@eurodecision.fr`

1. Introduction

Bouygues Telecom, a French operator of mobile telephone network, is growing rapidly. To cope with this development, new hardware must continually be added to the existing network. The addition of new hardware must firstly fulfil strong technical constraints related to compatibility and capacity. Moreover, both the choice of new hardware to buy, and the possibilities of connections to the network, must satisfy quality criteria stated by the operator. These criteria are expressed as preference constraints, hierarchically ordered.

The problem is, on a given time horizon, to plan the purchase and installation of hardware; this planning determines what hardware must be bought and what connections must be achieved, knowing that connections may evolve (hereafter, a connection change will be denoted "*reconnection*"). Bouygues Telecom asked us to develop an application, with an important requirement: their engineers should be able to use it very interactively, as a simulation tool. To do so, a solution (even if it does not satisfy all constraints) has to be computed within a given time[3].

In this paper, we show how this problem can be solved using constraint programming, and particularly repair techniques. Indeed, the "*time-out*" constraint, together with the important size of the problem, do not allow the use of systematic constructive methods. A hybrid technique (between constructive methods and local search algorithms) appeared to be a well suited approach for this problem: the planning is *repaired during its construction.*

In addition, repair techniques allowed us to cope with two specific features of our problem:

constraints hierarchy: each hierarchy level is associated with specific repair procedures which can possibly violate constraints up to this level;
"**reconnection**": a new planning may change some existing connections; moreover, a strong "meta-constraint" strictly limits the number of possible *reconnection*s.

The prototype we have developed has been validated by Bouygues Telecom, both on hand-made benchmarks and on real data. Every problem has been solved within one minute; moreover, each solution was at least as good as the solution computed "by hand".

Section 2 describes the resource allocation problem. In Section 3 we model our problem as a Constraint Satisfaction Problem with preference constraints. Section 4 presents related works (from systematic constructive methods to local search algorithms) and motivates how repair algorithms are well suited for solving this problem. Section 5 describes our resolution method, based upon local repair techniques *during the generation.* We finally conclude with some results and comments about this method.

[3]In our case, one minute.

2. The problem

In order to cope with the development of its mobile telephone network, Bouygues Telecom has to plan, on a given time horizon, the purchase and the installation of new transceivers and controllers. The whole period is divided into *np* elementary time periods (here, $np = 6$) during which the planning will be updated (see Fig. 1). For each time period, the number and the location of added transceivers is fixed; but they can vary from one time period to another.

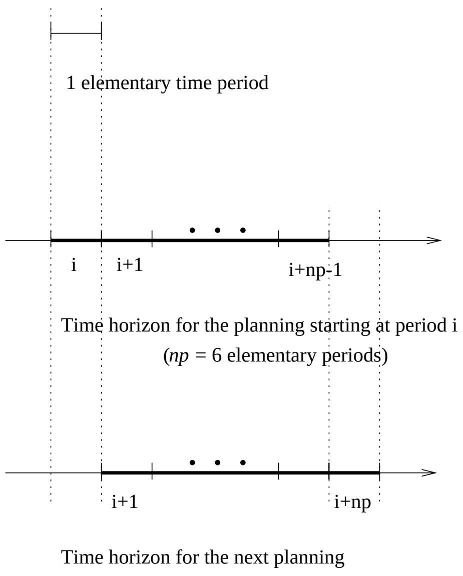
1 elementary time period
i
i+1
i+np-1
Time horizon for the planning starting at period i
(np = 6 elementary periods)
i+1
i+np
Time horizon for the next planning

Figure 1. Time management for successive plannings.

In this paper, we will not describe the whole mobile telephone network. Instead, we only focus on the components related to our problem: transceivers and controllers[4]. Each transceiver is controlled by exactly one controller while a controller can manage several transceivers (depending on its capacity). A controller must be assigned to every new transceiver installed on the network (see Fig. 2). This controller can be either an already installed one (*i.e.*, already in use), or a new one.

The problem consists in assigning a controller to each transceiver on a given time horizon and satisfying the following requirements:

1. a transceiver can be connected to a controller if technical constraints associated to the network are satisfied (Fig. 2b): capacity constraints, compatibility between providers, compatibility between release versions, ...;

[4]For the sake of confidentiality we cannot provide more details, neither about data, nor about constraints. However the problem we present is representative of the real life problem.

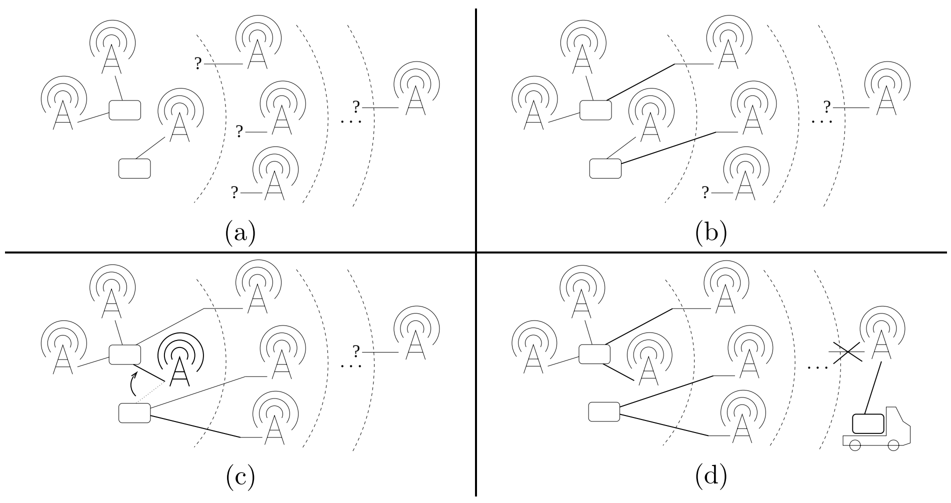

FIGURE 2. Overview of the network.

2. a transceiver can be reconnected to another controller (*reconnection*) if the technical constraints are satisfied (the bold transceiver in Fig. 2c). A *reconnection* enables to temporarily find "a place" for a transceiver. But the total number of *reconnection*s has been limited by BOUYGUES TELECOM. A reconnection does not imply a physical movement of a transceiver (or a controller); but it necessitates that technicians move to the transceiver location to modify its pluggins in order to establish the new connection;
3. finally, when a transceiver cannot be connected to a controller a new controller must be bought by BOUYGUES TELECOM (Fig. 2d). The delivery delay is several weeks; consequently, those purchases must be foreseen by the planning.

Moreover, the planning has to satisfy *at best* a set of quality criteria defined by BOUYGUES TELECOM. Those criteria could not be combined into a linear objective function. Instead, the quality of a planning is expressed by preference constraints with different levels of priority (some criteria are more important than other ones).

Achieving such a planning "by hand" requires one week for an engineer. BOUYGUES TELECOM asked us to develop a tool which could help them designing their planning by simulating an important number of possible scenarios. Consequently, the computation time for building a planning has to be low[5].

[5]For our application, the user requires a solution in less than one minute.

3. Modelling the problem as a CSP with preference constraints

In this section, we model our problem as a Constraint Satisfaction Problem (*CSP*) [17] with preference constraints.

3.1. Overview of the model

A CSP can be defined as a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$: a finite set $\mathcal{X}$ of n variables $X_1, X_2, ..., X_n$; the set $\mathcal{D}$ of their respective domains $domain(X_1), domain(X_2), ...,$ $domain(X_n)$ (where $domain(X_i)$ is the finite set of the possible values for X_i); and the set $\mathcal{C}$ of the constraints specifying which combinations of values are allowed.

The **"classical" CSP** is a satisfiability problem: solving such a CSP consists in assigning a value to every variable so that *all* constraints are satisfied.

In this case, the result provided by a constraint solver is either one or several solutions if they exist, or "No Solution".

But in most real life problems, the user cannot accept such an answer: he does want a solution, even if some constraints are not satisfied. The "classical" CSP model has therefore been extended to introduce **preference constraints**. In this model, the set $\mathcal{C}$ of the constraints is divided in two subsets: *required* constraints which must necessarily be satisfied, and *preference* constraints which should preferably be satisfied, but can be violated. Now, a solution must satisfy all the required constraints and the preferred ones in the best possible way with respect to a given comparator [4, 16].

Example. In a research laboratory, Michael, John and Alan have to schedule a meeting in order to present their current works. John and Alan want to present their work to Michael. Michael wants to give a talk to John and Alan. Each presentation requires an half-day. Michael, John and Alan have decided that the duration of the meeting could not exceed 4 half-days. Each of the three researchers has expressed a set of preference constraints that he expects to be verified by the schedule.

This problem can be modelled as a CSP: let Ma, Mj, Am, Jm be the four presentations (Michael to Alan, Michael to John, Alan to Michael, John to Michael respectively). As each talk lasts one half-day and the meeting could not exceed 4 half-days, those variables will have $[1, 2, 3, 4]$ as domain.

- integrity constraints are required:
  - someone who attends a presentation cannot present something at the same time:

$$\{Ma \neq Am, Ma \neq Jm, Mj \neq Am, Mj \neq Jm\}$$

  - no one can attend two presentations at the same time:

$$\{Am \neq Jm\}$$

- preference constraints (by decreasing order of importance):
  - Michael wants to know what John and Alan have done before presenting his work (priority 3):

$$\{Ma > Am, Ma > Jm, Mj > Am, Mj > Jm\}$$

  - Michael prefers not to present his work to Alan and John at the same time (priority 2):

$$\{Ma \neq Mj\}$$

  - Michael would like not to come the fourth half-day (priority 1):

$$\{Ma \neq 4, Mj \neq 4, Am \neq 4, Jm \neq 4\}\cdot$$

There exists no solution that verifies all the constraints. The assignment $\{Ma = 4, Mj = 3, Am = 1, Jm = 2\}$ violates only one constraint $Ma \neq 4$ with priority 1.

3.2. Variables and domains

Let $Trans = \{t_i\}$ be the set of the transceivers, and $Ctr = \{ctr_j\}$ be the set of the controllers; as some transceivers can be *reconnected*, time periods must be taken into account. $t_{i,p}$ will refer to transceiver t_i at time period p, and $ctr_{j,p}$ to controller ctr_j at time period p.

Each transceiver has some specific features: the administrative area it depends on, its provider, its release version, its actual (physical) location ... Below is the list of those features together with the associated notations:

- Administrative area: `area`$(t_{i,p})$
- Provider: `provider`$(t_{i,p})$
- Release version: `release`$(t_{i,p})$
- Whether the physical location is safe or not: `safeLocation`$(t_{i,p})$
- Identifier of the physical location: `location`$(t_{i,p})$
- Load of a transceiver: `load`$(t_{i,p})$

As well as transceivers, controllers have their own characteristics:

- Administrative area: `area`(ctr_j)
- Provider: `provider`(ctr_j)
- Release version of a controller for a given period p: `release`(ctr_j, p)
- Capacity: `capacity`(ctr_j)

The set of the controllers in use during period p will be denoted `controllers`(p).

To each transceiver $t_{i,p}$ is associated a variable $C_{i,p}$ which represents the controller allocated to t_i at time period p. The domain of such a variable $C_{i,p}$ is a subset of `controllers`(p) verifying certain basic technical constraints (*e.g.*, compatibility with the provider of $t_{i,p}$).

For the sake of brevity and simplicity, the period parameter p will sometimes be omitted. This means (notably in comparisons between features) that the time period is the same for all parts. For instance,

- $(C_{i,5} = ctr_j) \wedge (C_{i,6} = ctr_k)$
  transceiver t_i has been reconnected from controller ctr_j to controller ctr_k at period $p = 6$;
- $\texttt{release}(t_i) \leq \texttt{release}(C_i)$
  whatever any given period, the release version of transceiver t_i must be less than or equal to the release version of the controller C_i which manages it, during that given period.

The problem is made up of a few hundreds of transceivers t_i and a few tens of controllers ctr_j; there are about a few thousands of domain variables $C_{i,p}$ whose domains size are about a few tens.

The planning problem consists in assigning a value (controller) to every variable $C_{i,p}$ such that all required constraints are verified and preference constraints are satisfied at best.

3.3. Constraints

There are three kinds of constraints:

- required constraints which must be satisfied: technical constraints, capacity constraints, ...;
- preference constraints which characterize the quality of a solution; they constitute a hierarchy according to their levels of priority [1, 12, 19];
- meta-constraints which are related to the preference constraints by limiting the number of violations of these constraints. For instance, the total number of *reconnections* is strictly limited during a time period.

3.3.1. *Required constraints*

A transceiver can be connected to a controller only if technical constraints are satisfied:

- compatibility between providers is ensured:
  $\forall t_i$,
$$\texttt{provider}(C_i) = \texttt{provider}(t_i)$$
- compatibility between release versions is ensured:
  $\forall t_i$,
$$\texttt{release}(C_i) \geq \texttt{release}(t_i)$$
- capacity: each controller has a specific load; the total load of the transceivers connected to a controller cannot exceed the capacity of the controller:
  $\forall ctr_j$,
$$\sum_{t_i \mid C_i = ctr_j} \texttt{load}(t_i) \leq \texttt{capacity}(ctr_j)$$

3.3.2. *Preference constraints*

Unlike the required constraints, preference constraints have to be satisfied at best (a planning is allowed to violate some of them). Preference constraints are expressed as a hierarchy according to the priority levels defined by the user. Preference constraints characterize the quality of a solution.

A constraint with a given priority will be considered as more important than any number of constraints with lower priorities [1]. We present here a significant subset of the actual preference constraints:

6. **locality:** every transceiver should be connected to a controller which belongs to the same area:
   $\forall t_i$,

$$\texttt{area}(t_i) = \texttt{area}(C_i)$$

5. **controller purchase:** as far as possible, no new controller should be bought:
   $\forall p > 0$,

$$\mathrm{card}(\texttt{controllers}(p)) = \mathrm{card}(\texttt{controllers}(p-1))$$

4. **controller version upgrading**[6]: as far as possible, no controller should have its version upgraded:
   $\forall p > 0, \forall ctr_j$,

$$\texttt{release}(ctr_j, p) = \texttt{release}(ctr_j, p-1)$$

3. **safety:** not all the transceivers physically located at the same place should be connected to the same controller, except if this location is considered as "safe" by Bouygues Telecom:
   $\forall t_i \mid \text{not } \texttt{safeLocation}(t_i)$,

$$\begin{array}{rl} \exists & t_j \neq t_i \mid \texttt{location}(t_j) = \texttt{location}(t_i) \\ \Rightarrow \ \exists & t_k \mid \texttt{location}(t_k) = \texttt{location}(t_i) \ \wedge \ C_k \neq C_i \end{array}$$

2. **reconnection:** as far as possible, no transceiver already connected to a controller should be reconnected to another one:
   $\forall p > 0, \forall t_i$,

$$C_{i,p} = C_{i,p-1}$$

1. **safety margin:** every controller has its own safe loads; no load should exceed those safety margins (expressed as percentages):
   $\forall ctr_j$,

$$\sum_{t_i \mid C_i = ctr_j} \texttt{load}(t_i) \leq \texttt{safetyMargin}(ctr_j) \times \texttt{capacity}(ctr_j)$$

[6]Upgrading the version of a controller may increase its capacity or its functionalities.

3.3.3. *Meta-constraints*

Bouygues Telecom has restricted the allowed number of *reconnection* changes and *version upgrading* changes. These meta-constraints are related to preference constraints by limiting the number of violations of these constraints:

- upon every period, the ***number of reconnection***s is limited:
  $\forall p > 0$,

$$\mathrm{card}(\{t_{i,p} | C_{i,p} \neq C_{i,p-1}\}) \leq \texttt{maxReconnectionNumber}(p)$$

- upon every period, the ***number of version upgradings*** is limited as well:
  $\forall p > 0$,

$$\mathrm{card}(\{ctr_j | \texttt{release}(ctr_j, p-1) \neq \texttt{release}(ctr_j, p)\}) \leq \texttt{maxUpgradeNumber}(p)$$

In the same way as for required constraints, those meta-constraints must necessarily be satisfied.

4. Solution approach

The previous section models our problem as a CSP with preference constraints. In order to select an appropriate method for solving it, we must take into account two specificities of this planning problem:

- the computation time is bounded (less than one minute). So, an exhaustive planning has to be rapidly built, even if it does not respect at best the preference constraints;
- meta-constraints limit the number of *reconnections* and the number of version upgradings.

4.1. Four main approaches

Four main categories of methods can be used to solve this problem:

1. *systematic constructive methods,* based upon backtracking or branch and bound techniques. These methods are complete and could provide an optimal solution, but they may be very time consuming. Constraint programming can be classified in this category;
2. *local search methods* start from an initial solution (often randomly generated). From this initial solution, exchanges between components are achieved and results are evaluated. The exchange producing the best solution is retained and the procedure continues until a stopping test. The exchange process depends on the method (*e.g.*, simulated annealing [9, 10], tabu search [5, 6], repair-based methods [13]...) and the neighbourhood system used. The choice of a neighbourhood generating mechanism should be driven by the structure of the problem;

3. *greedy methods* construct a solution from scratch with no backtracking mechanism. These methods are obviously very fast, but they can be used only on problems with very specific properties;
4. *hybrid methods,* can be obtained by combining constructive methods (either systematic or not) and local search methods [11,14,15].

4.2. Discussion

Systematic constructive methods could not be used because of the requirement in computation time and of the size of the problem. An hybrid technique (combining constructive methods and local search algorithms) appeared to be well suited for this problem: no systematic backtrack is achieved; instead, the planning is *repaired during its construction.*

This method consists in two steps:

1. at each stage of the construction (instanciation of a variable $C_{i,p}$), we use a *guided local repair* method in order to revise the current partial solution;
2. the solution of step 1 clearly depends on the variables ordering; we therefore iterate the first step with another *assignment order.* The number of iterations depends on the computational time imposed by the user.

5. "Constructive repair"

The principle of our method is to repair the planning during its construction; we will therefore first present this approach, which could be called "constructive repair". Then, we give two examples of repair functions (Sect. 5.2).

As repairing algorithms are not complete, they naturally cannot ensure any kind of optimality; Section 5.5 discusses how various heuristics are successively tried in order to improve the quality of the solutions.

5.1. Constructive repair of a planning

At time period `p`, function `allocate(S, p)` tries to allocate a controller to each transceiver belonging to the set `S`. For each transceiver to be connected to the network, if no controller verifies all the constraints, repair functions are applied until one succeeds.

For each transceiver there are three cases:

1. there exists a controller satisfying all the constraints ("initial" success);
2. a controller can be selected consequently to a repair ("repair" success);
3. no controller can be allocated to this transceiver without violating at least one required constraint; in this case, all repair functions have failed to provide a controller for this transceiver ("failure")[7].

[7]It does not necessary mean that such a controller does not exist; constructive repair does not ensure completeness.

Finally, all the transceivers for which no repair function is able to provide a controller are gathered.

```
function allocate(S, p)
-- looks first for an "initial" success, else for a "repair" success;
-- returns {ti,p in S | for every controller available at period p,
--                      there exits at least one required constraint
--                      that is not satisfied}
1   --  WithoutCtr denotes the set of the transceivers for which
    --  no controller has been found;
2   WithoutCtr <- emptyset;
3   while S is not empty do
4       select a transceiver ti,p in S;
5       look for a controller in domain(Ci,p)
            such that all constraints are satisfied;
6       if there is no such controller
7           then repair(ti,p);
8               if failure then add ti,p to WithoutCtr;
9   end while
10  return WithoutCtr
end
```

The order the transceivers are selected (line 4) does not greatly affect the quality of the solution; we used several orders (highest load first, lowest load first, most recently in use first, random) with similar results on average; the reason is that all transceivers have approximately the same load.

`planifyFrom(p1)` performs the planning over the time-horizon (`np` time periods) starting from time period `p1`. At each time period `p`, the corresponding set `Sp` of transceivers to be connected to the network is processed calling `allocate(Sp, p)`. The reason why a transceiver cannot find a controller at a time period is that not enough compatible controllers are available at this time period.

If the purchase of a controller `ctr` is decided, this controller will be connected to the network only at period `p'= delivery-period(ctr)` (because of delivery delay depending on the provider). For periods anterior to `p'`, it is too late: the only thing we can do is to send a warning to the user. For periods posterior to `p'`, this controller will become available for every compatible transceiver (`addNewCtr(ctr, p')`) and a new allocation step is performed for non allocated transceivers, taking into account this new controller (line `13`)[8].

```
planifyFrom(p1)
1   for each time period p from p1 to p1 + np - 1
2      for each ti do
           domain(Ci,p) <- domain(Ci,p-1); Ci,p <- Ci,p-1
           -- at period p, each ti remains connected to the
```

[8]For the sake of clarity, we present here a slightly simplified version of the method. In the actual algorithm, line `13` is replaced by a call to function `improve(p)`, which tries to benefit from the new controller: it is added to the domains of *all* transceivers compatibles with it. `improve` then applies to *all* those transceivers (those in NoC, those in Sp, but also those assigned "by default" to the same controller as at period `p-1`) to try to perform some reallocations in order to increase the quality of the partial plan (*i.e.*, decrease the number of constraint violations).

```
              -- controller it was at period (p-1)
           end for
3          let Sp be the set of the ti to be connected at period p;
4          let NoC = allocate(Sp, p)
5          while NoC is not empty     -- there exist some ti without controller
6              choose any ti,p in NoC
7              let ctr be a new controller compatible with ti,p
8              let p' = delivery-period(ctr);
9              if p < p'
10               then send a warning: too late for ti,p
11                    remove ctr from NoC
12               else addNewCtr(ctr, p')
13                    NoC <- allocate(NoC, p)
               end if
           end while
        end for
end
```

Every new controller *ctr*, connected to the network at period *p*, becomes available for every compatible transceiver $t_{i,q}$ for time periods $q \geq p$. To achieve this, `addNewCtr(ctr, p)` extends the domains of the variables $C_{i,q}$ by adding the value *ctr*.

```
addNewCtr(ctr, p)
    for each transceiver ti,q compatible with ctr such that q >= p do
       add ctr to domain(Ci,q)
    endfor
end
```

The key point of our method is the successive calls to repair functions. Each repair tries to find a controller for a determined transceiver by locally modifying the current solution, and possibly violating some constraints; if a repair succeeds, a solution is obtained. If a repair fails, the next repair function is called, which may violate more important constraints. As successive repairs fail, more and more important constraints have to be relaxed in order to find a solution. A dozen of repair functions have been developed for this application. The number of iterations depends on the imposed computational time.

```
function repair(ti,p)
-- Each repair function revises the current partial solution.
1   repeat
2      select a repair function according to the hierarchy;
3      perform this repair for transceiver ti,p
4   until one repair succeeds;
5   if not, return failure
end function
```

We can now allocate the resources starting form period *p*. The number of iterations (heuristics tried) depends on the allowed computing time. The various heuristics are depicted in Section 5.2.

```
function repair-reallocate(ti,p)
-- returns {success,failure}
1   for all ctr in domain(Ci,p)
2   such that area(ctr) = area(ti,p)
3   and Ci,p <- ctr is safe
4     for all transceiver tj,p assigned ctr
5       for all ctr' <> ctr in domain(Cj,p)
6       such that area(ctr') = area(tj,p)
7       and Cj,p <- ctr' is safe
8         if load(ctr') + load(tj,p) <= capacity(ctr')
9         and load(ctr) + load(ti,p) - load(tj,p) <= capacity(ctr)
10           then assign ctr' to Cj,p
11                assign ctr to Ci,p
12                update data
13                return success
          end if
        end for
      end for
    end for
14  return failure
end function
```

FIGURE 3. First example of a repair function: `repair-reallocate`.

```
function planify(p)
    while current-computing-time <= allowed-computing-time do
       select a heuristic;
       planifyFrom(p)
    endwhile
    return the best computed solution
end function
```

5.2. Repair functions

In this section, we give two examples of repair functions: the first one deals with *reconnection* and the second one concerns *version upgrading.*

For instance, assume that for period p, no controller which is compatible with transceiver $t_{i,p}$ has enough capacity left to manage it. This would lead to failure. But, it may exist a transceiver $t_{j,p}$ connected to a controller *ctr*, belonging to `domain(`$C_{i,p}$`)` such that it exists another controller *ctr′* which can be allocated to $t_{j,p}$. A solution would consist in assigning *ctr′* to $t_{j,p}$ (*reconnection*) and in assigning *ctr* to $t_{i,p}$, if $t_{i,p}$ verifies the capacity constraint. The repair function outlined Figure 3 enables such *reconnections*; Figure 7 in Section 5.4 illustrates how this function repairs a partial plan.

Notice that if $\mathtt{C}_{\mathtt{j,p-1}}$ was assigned controller `ctr`, this repair adds one violation of constraint 5 (*reconnection*). On the other hand, lines 2–3 and 6–7 ensure that "same area" and "safe location" constraints cannot be violated.

```
function repair-upgrade(ti,p)
-- returns {success,failure}
  for all ctr in domain(Ci,p) s.t. area(ctr) = area(ti,p)
    if load(ctr) + load(ti,p) <= capacity(upgraded(ctr))
      then upgrade(ctr,p)
           assign ctr to Ci,p
           update data
           return success
    end if
  end for
  return failure
end function
```

Figure 4. Second example of a repair function: `repair-upgrade`.

Another repair may upgrade the version of a controller to increase its capacity so that it can be assigned to more transceivers (Fig. 4). As *version upgrading* is quite an important constraint, this repair is called upon much later than this presented above. Moreover, the total number of upgradings is strictly limited, which also limits the use of this repair.

Notice that safety is not necessarily checked, since safety constraints have a lower priority than version upgrading constraints. Another version of this repair function with and additional check to ensure safety would be called just before `repair-upgrade`.

5.3. Complexity

The complexity of the constructive repair method is closely related to the complexity of the repair functions. Fast repairs allow a low run time, whereas more sophisticated (and consequently more time consuming) repairs increase the quality of the search, and may therefore produce a better solution. The efficiency of constructive repair relies on this compromise.

5.3.1. *Complexity of repair functions*

In this section, we evaluate the complexity in the worst case of the function `repair`.

Let nt be the total number of transceivers, d be the total number of controllers and e be the total number of constraints; the number of time periods (np) is a constant (for our application $np = 6$); let $n = np \times nt$; the CSP has n variables with domains of maximum size d.

Every repair function has a low polynomial complexity. This is necessary because a solution has to be computed in a given time.

We first evaluate the complexity of the two repair functions described in Section 5.2; then, we deduce the complexity of the function `repair`.

Let us consider function `repair-upgrade` depicted in Figure 4. The worst case happens when there is no controller verifying the required property. So, there will be at most d (max size of a domain) iterations in the loop (line 1). The complexity of `repair-upgrade` is $O(d)$.

The function `repair-reallocate` described in Section 5.2 is also a search algorithm which proceeds by enumerating all the potential solutions. So, the worst case happens when no solution can be found. `repair-reallocate` is made up of three nested loops. There will be:

- at most d (max size of a domain) iterations in the first loop (line 1);
- at most nt iterations in the second loop (line 4): the worst case happens when all the transceivers are connected to the same controller;
- at most d iterations in the third loop (line 5).

The complexity of `repair-reallocate` is $O(nt \times d^2)$.

The worst case for function `repair` happens when all repair functions have to be applied. The complexity of `repair` is the sum (or maximum) of the complexities of all the repair functions. For our application, no repair function has a complexity greater than the complexity of `repair-reallocate`; so the complexity in the worst case of `repair` is $O(nt \times d^2)$.

5.3.2. *Complexity of general algorithm*

The function `allocate` tries to allocate a value (controller) to every variable in `S`:

- first (line `5`), all values (possibly d) are checked against all constraints; there may therefore be $O(e \times d)$ checks;
- if no value is found, `repair` is called upon, whose complexity is $O(nt \times d^2)$ (see above).

The complexity of `allocate` is therefore $O(\text{card}(\texttt{S}) \times (e \times d + nt \times d^2))$.

The function `planifyFrom` planifies the successive allocations for all time periods. For each period `p`, there is an initialization step, where all transceivers already connected at period `p-1` default to the same controller (line `2`); the second step allocates a controller to all new transceivers: those in `Sp` (lines `3-13`).

The complexity of `planifyFrom` is therefore

$$\sum_{p \in periods} (\textit{complexity}(\text{initialization}(p)) + \textit{complexity}(\text{connection}(Sp)))$$

$$= \sum_{p \in periods} \textit{complexity}(\text{initialization}(p)) + \sum_{p \in periods} \textit{complexity}(\text{connection}(Sp))$$

$\sum_{p \in periods} \textit{complexity}(\text{initialization}(p)) = nt$

Let us now compute $\alpha = \sum_{p \in periods} \textit{complexity}(\text{connection}(Sp))$

The worst case happens when each call to `allocate` only connects one `ti,p`:

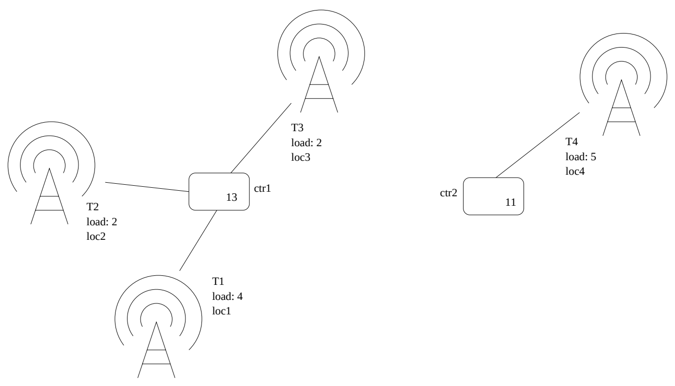

FIGURE 5. Initial state of the network.

card(Sp) calls to `allocate` are then required. This would mean that a new controller should be purchased for each transceiver to be inserted into the network, which never happens.

$\alpha = \sum_{p \in periods} \sum_{i=1}^{i=\text{card}(Sp)} \mathit{complexity}(\texttt{allocate}(\texttt{NoC}, p))$
Since $\text{card}(\texttt{NoC}) = i$, $\mathit{complexity}(\texttt{allocate}(\texttt{NoC}, p)) = i \times (e \times d + nt \times d^2)$
Let us denote $\beta = e \times d + nt \times d^2$

$$\begin{aligned}\alpha &= \sum_{p \in periods} \sum_{i=1}^{i=\text{card}(Sp)} i \times \beta \\ &= \beta \times \sum_{p \in periods} \sum_{i=1}^{i=\text{card}(Sp)} i \\ &\leq \beta \times \max_{p \in periods} \text{card}(Sp) \times nt, \quad \text{since} \sum_{p \in periods} \text{card}(Sp) = nt \\ &= (e \times d + nt \times d^2) \times \max_{p \in periods} \text{card}(Sp) \times nt\end{aligned}$$

As $\max_{p \in periods} \text{card}(Sp) < nt$, the complexity of `planifyFrom` is $O(nt^2 \times (e \times d + nt \times d^2))$.

5.4. Running an example

Let us now illustrate constructive repair with an example. The initial network is composed of two controllers and four transceivers (see Fig. 5). For the sake of simplicity, we will assume that compatibility constraints and administrative area constraints are always satisfied. Also, the safety margin constraint is 5% for both controllers: $\texttt{safetyMargin}(ctr_1) = \texttt{safetyMargin}(ctr_2) = 0.95$

- controller ctr_1 manages transceivers t_1, t_2 and t_3; its capacity is 13
  - $\texttt{load}(t_1) = 4$; $\texttt{location}(t_1) = \texttt{loc1}$; `loc1` is safe
  - $\texttt{load}(t_2) = 2$; $\texttt{location}(t_2) = \texttt{loc2}$; `loc2` is unsafe
  - $\texttt{load}(t_3) = 2$; $\texttt{location}(t_3) = \texttt{loc3}$; `loc3` is safe
- controller ctr_2 manages transceiver t_4; its capacity is 11
  - $\texttt{load}(t_4) = 5$; $\texttt{location}(t_4) = \texttt{loc4}$; `loc4` is safe

Now, assume we want to connect a new transceiver (say, t_5), whose `load` is 3, and located on `loc2`. We must allocate a controller to t_5:

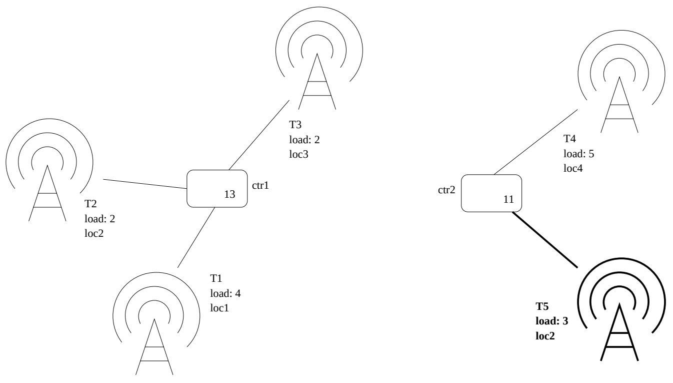

FIGURE 6. t_5 is connected.

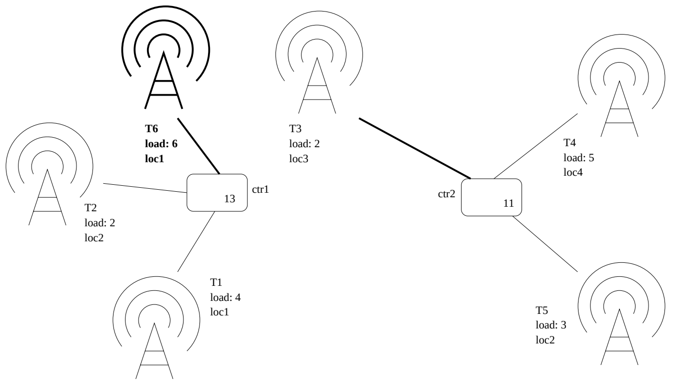

FIGURE 7. t_6 is connected.

- ctr_1 is first tried; all required constraint are satisfied, but the *safety* constraint is violated: `loc2` is `unsafe`;
- ctr_2 is then successfully tried, and therefore allocated to t_5: see Figure 6.

Another new transceiver, t_6 (`load` $= 6$, `location` $=$ `loc1`), has to be inserted into the network; neither ctr_1 nor ctr_2 can be allocated to it: they do not have enough capacity ($4+2+2+6 > 13$, $5+3+6 > 11$). We therefore try to "repair" the current planning. First, `repair-reallocate` is called:

- reconnecting t_1 to ctr_2 fails: not enough capacity ($5+3+4 > 11$);
- reconnecting t_2 to ctr_2 fails: `loc2` is unsafe (t_5 is located on `loc2` and is already managed by ctr_2);
- t_3 can be reconnected to ctr_2; ctr_1 now has enough capacity to be allocated to t_6 ($4+2+6 \leq 13$); therefore this repair is achieved (Fig. 7).

Finally, we will insert transceiver t_7 (`load` $= 4$, `location` $=$ `loc3`). No controller has enough capacity to manage it; moreover, `repair-reallocate` fails (notice it

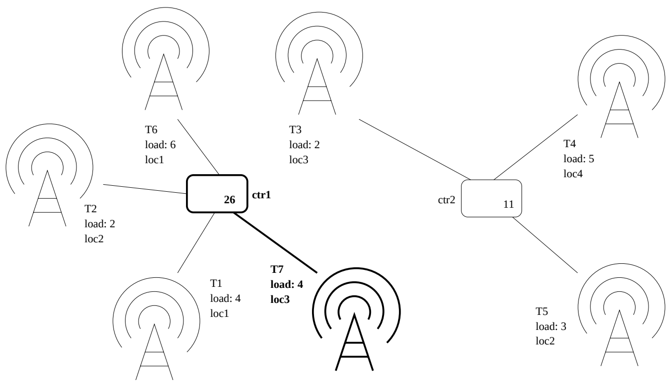

FIGURE 8. t_7 is connected.

could not succeed, since the total load of all transceivers exceeds the total capacity of all controllers). We then try `repair-upgrade` (assume both controllers can be upgraded, and upgrading one would double its capacity):

- ctr_1 is upgraded; its new capacity is 26 and it can now be allocated to t_7.

Notice that if `repair-upgrade` checks *safety* constraint, t_7 cannot be connected to ctr_1, since t_6 is also located on `loc3`, which is unsafe.

Last remark: if no upgrade were available, a new controller should be purchased.

5.5. Heuristics for variable ordering

Our method is not complete; hence, it cannot ensure optimality. The resulting planning clearly depends on the order in which the transceivers are assigned, even though this order is not related to the average quality. In order to tackle that problem, we try several orders: as long as the allowed computing time is not exceeded, the whole method is run with a different order[9]; we then keep the best solution. Of course, the longer the computing time, the higher the likelihood to improve the planning. There exist various elementary transformations of the transceivers order; Figure 9 shows three possible transformations.

- Insertion: one transceiver is randomly selected and inserted between two transceivers also randomly chosen.
- Swapping: swap two transceivers, randomly chosen.
- Circular rotation: choose one transceiver i at random and apply circular rotation between the transceivers before i and those after i (included).

We implemented the first two transformations. Simulation results indicate that the swapping transformation is better than the insertion transformation. Hence we used that one.

[9]The static orders (minimum load, maximum load, random) we have tested approximately lead to the same solution quality.

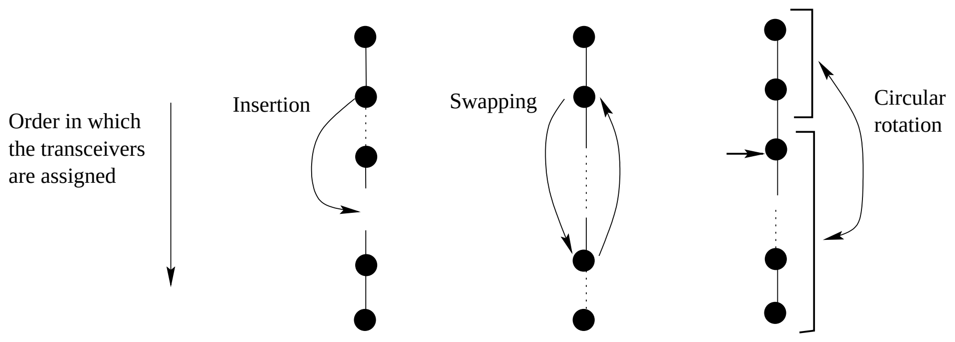

FIGURE 9. Three elementary transformations.

6. RESULTS

The prototype we developed in C++ has been tested with BOUYGUES TELECOM on ten problems (both hand-made problems and real problems), on a PC (pentium II-300 Mhz), with no specific compiler option. Real life problems deal with a few hundreds of transceivers and a few tens of controllers. From the CSP point of view, there are about thousands of variables whose domains size are about of a few tens.

In all cases, the solution computed during the first iteration was at least as good as the solution computed by hand by the engineer.

We then increased the computation time: in only one case, a better solution was found within one minute. Even with longer computing times (more than one hour), no further improvement was found, for that single problem as well as for the other nine ones. This behaviour is mainly due to the fact that:

1. each controller has an important load (the decision to buy a new controller is postponed as long as possible);
2. the quality of a solution is characterized by the hierarchy of constraints and the constructive repair algorithm looks for solutions according to this hierarchy.

BOUYGUES TELECOM currently uses our application to schedule the installation of new hardware for its mobile telephone network, with several advantages:

- they no longer need such a qualified engineer to achieve the planning;
- due to the low computation time, they can perform several simulations, and therefore be more demanding; the hand-made planning was too time-consuming: they could not afford searching more than one single solution;
- due to the quality of the solutions,
  - they save hardware when the solution provided by the application reduces the purchase of new controllers;
  - they save cost of labour when the number of reconnections is reduced.

7. Conclusion

This paper shows how repair algorithms were well suited for solving a resource allocation problem in a mobile telephone network. A strong request was that computation time should not be greater than one minute. A complete method could not guarantee a solution could be found within this time. We therefore used an incomplete method based upon repair techniques during the construction; the user can easily limit the computing time to any value he wants (including a few seconds); moreover, successive repairs easily cope with priority between preference constraints.

This application is currently used at Bouygues Telecom in order to plan the installation of new hardware on their mobile telephone network. This planning tool is also used as a simulation tool in order to find and investigate rather different solutions. The user can interactively change the order of the criteria describing the quality of a solution by modifying the priority levels of the constraints in the hierarchy. The constructive repair algorithm remains the same. Taking into account new kinds of constraints would require to extend the program. This adding can be incrementally performed by just giving the level of the new constraints and defining a repair function specific to them.

Repairing algorithms have also been successfully applied for scheduling an examination timetabling (with a bounded computing time) [2]. We think that this approach is well suited for optimization problems where computing time is critical as well as for devising *anytime algorithms*. With repairing algorithms, a good solution can be computed very quickly[10], which may be quite useful if there is no time for an iterative improvement phase. Moreover, the number and the levels of repair function can be adapted to every specific situation, in terms of quality of solution as well as in terms of computing time. We think that repairing algorithms will be of great help for solving dynamic resource allocation problems in particular in the field of computer network or mobile telephone network.

Acknowledgements. This work was partially supported by Bouygues Telecom. We wish to thank Laurent Bister, Corinne Cohen, Vincent Ducas and Virginie Gaborit for their helpful comments.

References

[1] A. Borning, M. Maher, A. Martindale and M. Wilson, Constraint hierarchies and logic programming, in *Proc. of ICLP'89.* Lisbon, Portugal (1989) 149-164.

[2] P. David, *A constraint-based approach for examination timetabling using local repair techniques*, Selected papers (extended version) from the Second International Conference on the Practice and Theory of Automated Timetabling (PATAT'97). *Lecture Notes in Comput. Sci.* **1408** (1998) 169-186.

[10]Generally, the first computed solution is already very good.

[3] S. de Givry, G. Verfaillie and T. Schiex, Bounding the optimum of constraint optimization problems, in *Proc. of the 3rd Int. Conference on Principles and Practice of Constraint Programming (CP'97)*. Schloss Hagenberg, Austria, *Lecture Notes in Comput. Sci.* **1330** (1997) 405-419.

[4] E.C. Freuder and R.J. Wallace, Partial constraint satisfaction. *Artificial Intelligence* **58** (19923) 21-70.

[5] F. Glover, Tabu search, I. *ORSA J. Comput.* **1** (1989) 190-206.

[6] F. Glover, Tabu search, II. *ORSA J. Comput.* **2** (1990) 4-32.

[7] N. Jussien and P. Boizumault, *Implementing constraint relaxation over finite domains using ATMS*, edited by M. Jampel, E. Freuder and M. Maher, Over-Constrained Systems. Springer-Verlag, *Lecture Notes in Comput. Sci.* **1106** (1996) 265-280.

[8] N. Jussien and P. Boizumault, Best-first search for property maintenance in reactive constraints systems, in *International Logic Programming Symposium.* MIT Press, Port Jefferson, NY, USA (1997) 339-353.

[9] S. Kirkpatrick, C.D. Gelatt and M.P. Vecchi, Optimization by simulated annealing. *Science* **220** (1983).

[10] S. Kirkpatrick, Optimization by simulated annealing: Quantitative studies. *J. Statist. Phys.* **34** (1984).

[11] F. Laburthe and Y. Caseau, SALSA, a language for search algorithms, in *Proc. of CP'98.* Springer, *Lecture Notes in Comput. Sci.* **1520** (1998) 310-324.

[12] F. Menezes and P. Barahona, Defeasible constraint solving, in *Over-Constrained Systems.* Springer, *Lecture Notes in Comput. Sci.* **1106** (1996) 151-170.

[13] S. Minton, M.D. Johnston, A.B. Philips and P. Laird, Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence* **58** (1992) 161-205.

[14] G. Pesant and M. Gendreau, A view of local search in constraint programming, in *Proc. of CP'96.* Springer, *Lecture Notes in Comput. Sci.* **1118** (1996) 353-366.

[15] A. Schaerf, Combining local search and look-ahead for scheduling and constraint satisfaction problems, in *Proc. of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97).* Morgan Kaufmann, Nagoya, Japan (1997) 1254-1259.

[16] T. Schiex, H. fargier and G. Verfaillie, Valued constrain satisfaction problems: Hard and easy problems, in *Proc. of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95).* Montréal, Canada (1995) 631-637.

[17] E. Tsang, *Foundations of constraint satisfaction.* Academic Press (1993).

[18] G. Verfaillie and T. Schiex, Solution reuse in dynamic constraint satisfaction problems, in *Proc. of the 12th National Conference on Artificial Intelligence (AAAI'94).* Seattle, WA, USA (1994) 307-312.

[19] M. Wilson and A. Borning, Hierarchical constraint logic programming. *J. Logic Programming* **16** (1993) 277-318.