Anytime Lower Bounds for Constraint Violation Minimization Problems

Bertrand Cabon[1], Simon de Givry[2], and Gérard Verfaillie[3]

[1] MATRA MARCONI SPACE, 31 av. des Cosmonautes, 31402 Toulouse Cedex 4, France,
Bertrand.Cabon@tls.mms.fr
[2] THOMSON LCR, Domaine de Corbeville, 91404 Orsay Cedex, France,
degivry@thomson-lcr.fr
[3] ONERA-CERT, 2 avenue Edouard Belin, BP 4025, 31055 Toulouse Cedex 4, France,
Gerard.Verfaillie@cert.fr

Abstract. *Constraint Violation Minimization Problems* arise when dealing with over-constrained *CSPs*. Unfortunately, experiments and practice show that they quickly become too large and too difficult to be optimally solved. In this context, multiple methods (limited tree search, heuristic or stochastic local search) are available to produce non-optimal, but *good* quality solutions, and thus to provide the user with anytime upper bounds of the problem optimum. On the other hand, few methods are available to produce anytime lower bounds of this optimum. In this paper, we explore some ways of producing such bounds. All of them are algorithmic variants of a *Branch and Bound* search. More specifically, we show that a new algorithm, resulting from a combination of the *Russian Doll Search* and *Iterative Deepening* algorithms, clearly outperforms five known algorithms and allows high lower bounds to be rapidly produced.

1 Constraint Violation Minimization Problems

In many decision tasks (planning, scheduling, resource allocation, design, diagnosis), the objective is to find a solution which satisfies a set of hard (or imperative) constraints, generally expressing physical limitations, and which, at the same time, satisfies as well as possible a set of soft (or relaxable) constraints, expressing costs, utililities, preferences, probabilities.

Such problems can be referred to as *Constraint Violation Minimization Problems*. *Max-CSPs* [6], *Weighted CSPs*, *Possibilistic CSPs* [15], *Fuzzy CSPs* [14, 4], and *Probabilistic CSPs*[14, 5] are some of the frameworks which have been defined to represent them. *Valued* and *Semiring CSPs* [16, 2, 1] are two more general frameworks, which include all of the above frameworks.

Although all of our work has been carried out in the general *Valued CSP* framework, for the sake of simplicity, we limit our presentation to *Weighted CSPs*. These problems can be defined as classical *CSPs*, with a weight associated with each constraint. The weight of a complete variable assignment is defined as the sum of the weights of the violated constraints, and the objective is to find a complete assignment whose weight is minimum. In *Max-CSPs*, all the constraints have the same weight, equal to 1, and the weight of a complete variable assignment is simply the number of violated constraints.

Remember that all of the algorithms which are presented in this paper can be easily extended to the *Valued CSP* framework, to take into account soft and hard constraints and to consider other violation aggregation functions.

2 Why produce anytime lower bounds?

It is known that the *Weighted CSP* and *Max-CSP* problems, which generalize the *Max-SAT* problem, are *NP-hard*. In addition, experiments, which have been carried out for example on randomly generated *Max-CSPs* [18], show that these problems are far more difficult to solve that the corresponding *CSPs*. Even small instances, involving only some tens of variables with ten values per variable, cannot be optimally solved in a reasonable time. The situation is naturally worse when one wants to solve instances which result from translating real-world problems and involve hundreds or thousands of variables.

In practice, when someone wants to solve such a problem, he uses any incomplete method (limited tree search, heuristic or stochastic local search) which is able to provide him with solutions of increasing quality, and thus with decreasing upper bounds of the problem optimum.

But some questions arise: What is the quality of the best solution found thus far? How far is its weight from the optimal weight? What gain can be expected from using more time or more resources to explore other solutions?

Producing lower bounds of the problem optimum can help to answer these questions. For example, if the lower bound equals the upper bound, optimality is proven and the search can stop. More generally, if the distance between the upper and lower bounds is small enough, the user or the system can decide to stop the search and to use time and ressources for other tasks. Conversely, if this distance remains large, he/it can decide to invest more time and ressources in finding better solutions or producing higher lower bounds.

In this paper we explore some ways of producing anytime increasing problem lower bounds, concurrently with the usual production of anytime decreasing problem upper bounds, in order to obtain a tighter and tighter bounding of the problem optimum.

Whereas any incomplete method can be used for producing upper bounds, producing lower bounds generally requires complete methods. This statement is consistent with the results of the *Complexity Theory* [12]: since producing an upper bound is an *NP* problem, a polynomial length certificate therefore exists; since producing a lower bound is a *Co-NP* problem, such a certificate may not exist (if $P \neq NP$).

After a quick overview of possible approaches for producing such lower bounds in the context of *Weighted CSPs* (see 3), we limit this paper to an exploration and a comparison of algorithmic variants of a *Branch and Bound* search, some of them well-known, others more recent (see 4). The main result is that a new algorithm, resulting from a combination of the *Russian Doll Search* [19] and *Iterative Deepening* [8] algorithms, clearly outperforms all the others and allows high lower bounds to be rapidly produced (see 5).

3 How produce anytime lower bounds?

3.1 Problem simplification

A first method to obtain a problem lower bound consists in solving completely a simplified problem: the optimum of the simplified problem, either is a lower bound of the original problem, or allows such a bound to be computed. Such a simplification is currently obtained by removing some constraints or by removing some forbidden tuples in some constraints. In [3], it is obtained by modifying the violation aggregation function to be minimized.

3.2 Objective simplification

Another method to obtain a problem lower bound consists in aiming at a simpler objective, like *Local Consistency*. For example, *Directed Arc Consistency* preprocessing [20] produces a problem lower bound. The sum of the weights of the constraints which must be relaxed, in order to obtain a classical *CSP*, whose *Arc Consistency* closure is non empty or, more generally, a classical *CSP* which satisfies any *Local Consistency* property, is also a problem lower bound[1].

3.3 Search simplification

A last method consists in performing a limited tree search, depending on the available time and ressources, and in exploiting the subproblem lower bounds, which are computed at each node of the tree, in order to produce a problem lower bound. This is the way we explore in this paper.

4 Using variants of a Branch and Bound search

We explored and compared six variants of a *Branch and Bound* search, each of them able to produce anytime problem lower bounds:

- *Recursive Best First* (*rbf*);
- *Iterative Deepening* (*id*);
- *Iterative Objective Relaxing* (*ior*);
- *Iterative Approximating* (*ia*);
- *Russian Doll Search* (*rds*);
- *Iterative Deepening* + *Russian Doll Search* (*idrds*).

For each algorithm, we give a well-known reference, a short informal description and the corresponding pseudo-code. Though these algorithms have been developped in various contexts and presented using various styles, we tried to present all the pseudocodes in a uniform way in the *Weighted CSP* context.

[1] It is due to the fact that *Consistency* implies *Local Consistency*.

In each pseudo-code, both global variables ub and lb record the anytime problem upper and lower bounds. ub and lb are respectively initialized to $+\infty$ and 0.

In an application context, at any time, these bounds can be used to help the user or the system to decide about the continuation of the search. Note however that, while the six presented algorithms produce an anytime problem lower bound, only *rbf* and *ia* produce concurrently an anytime problem upper bound. The other four (*id*, *ior*, *rds* and *idrds*) produce no problem upper bound before the last step of the algorithm.

Whereas *rbf* uses a *Best First* search, the other five (*id*, *ior*, *ia*, *rds* and *idrds*) use a *Depth First* search.

Although all of them have been developed and tested with a dynamic value ordering and all of them, except the last two (*rds* and *idrds*), with a dynamic variable ordering (see 5), for the sake of simplicity, the pseudo-codes have been written assuming static variable and value orderings.

During the search, a global array A stores the current assignment. At the beginning, A is empty. n is the number of problem variables. $\text{DS}(j)$ returns the domain size of the variable j.

$\text{FC}(A, i, j)$ returns a lower bound of the subproblem, limited to the variables after i (i included) and constrained by the assignment A of the variables between i and j (i and j included). A *Forward Checking* method [6, 16] is used to compute this bound.

In a few words, if $\text{BC}(A, i, j)$ returns the sum of the weights of the constraints which are violated by the assignment A of the variables between i and j, and if $\Delta\text{BC}(A, i, j, l, k)$ returns the increment in BC, which would result from extending the assignment A with the assignment of the value k to the variable l, then:

$$\text{FC}(A, i, j) = \text{BC}(A, i, j) + \sum_{l=j+1}^{n} min_{k \in [1..\text{DS}(l)]} \Delta\text{BC}(A, i, j, l, k)$$

It is easy to establish that $\text{FC}(A, i, j)$:

- is really a subproblem lower bound;
- is monotonic: $i' \leq i, j \leq j' \Rightarrow \text{FC}(A, i, j) \leq \text{FC}(A, i', j')$;
- is exact *i.e.*, equal to the subproblem optimum, when $j = n$.

It can be also observed that FC takes into account only the constraints between assigned variables ($i \leq l \leq j$) and the constraints between assigned and unassigned variables ($j + 1 \leq l \leq n$). It does not take into account the constraints between unassigned variables. This fact will be used by *rds* and *idrds*.

Note that the second parameter i of the function FC is useful only with *rds* and *idrds*. It has been introduced for the sake of homogeneity and is systematically set to 1 with *rbf*, *id*, *ior* and *ia*.

Although the *Depth First Branch and Bound* algorithm, currently used for producing an anytime problem upper bound, produces no problem lower bound before ending, as an introduction, we present its pseudo-code in Figure 1.

```
DFBB()
   ub ← +∞
   lb ← 0
   SEARCH-DFBB(1)
   lb ← ub

SEARCH-DFBB(j)
   for k = 1 to DS(j)
      A[j] ← k
      llb ← FC(A, 1, j)
      if llb < ub
      then if j = n
           then ub ← llb
           else SEARCH-DFBB(j + 1)
```

Fig. 1. *Depth First Branch and Bound*

4.1 Recursive Best First

What we refer to as *rbf* (for *Recursive Best First*) is the recursive version of the classical *Best First Branch and Bound*, proposed in [9] in order to avoid the exponential space requirements of the original version. Whereas a node is explored only once by the original version, it may be explored several times by the recursive version. But, as the classical version, the recursive version explores the nodes in a *Best First* order. Thus the lower bound associated with the current node is a problem lower bound.

Although we implemented the full recursive version, we present in Figure 2 a simpler version (called *Simple Recursive Best First* in [9]). Note that $\text{BEST}(j)$ returns the first child of the current node whose associated lower bound is minimum.

4.2 Iterative Deepening

What we refer to as *id* (for *Iterative Deepening*) is the first version of this algorithm, proposed in [8]. The search is limited to a depth h ($h \leq n$) and the minimum of the lower bounds associated with all the nodes at depth h is a problem lower bound. Increasing h from 1 to n produces an anytime problem lower bound (see Figure 3).

4.3 Iterative Objective Relaxing

What we refer to as *ior* (for *Iterative Objective Relaxing*) is the second version of the *Iterative Deepening* algorithm, proposed in [8]. The algorithm searches for an assignment whose weight equals the current problem lower bound. Initially, this lower bound equals 0. When the search fails, the minimum of the lower bounds associated with all the failing nodes is the new current problem lower bound. When it succeeds, the current problem lower bound is obviously the optimum (see Figure 4).

```
RBF()
    ub ← +∞
    lb ← 0
    SEARCH-RBF(1, +∞)

SEARCH-RBF(j, llb)
    for k = 1 to DS(j)
        A[j] ← k
        llb[j, k] ← FC(A, 1, j)
    k* ← BEST(j)
    if j = n
    then ub ← min(ub, llb[j, k*])
    while llb[j, k*] ≤ llb
        lb ← llb[j, k*]
        if j = n
        then stop
        llb' ← min(llb, min_{k≠k*} llb[j, k]))
        A[j] ← k*
        llb[j, k*] ← SEARCH-RBF(j + 1, llb')
        k* ← BEST(j)
    return llb[j, k*]
```

Fig. 2. *Recursive Best First*

```
ID()
    ub ← +∞
    lb ← 0
    for h = 1 to n
        lub ← +∞
        SEARCH-ID(1, h)
        lb ← lub

SEARCH-ID(j, h)
    for k = 1 to DS(j)
        A[j] ← k
        llb ← FC(A, 1, j)
        if llb < lub
        then if j = h
             then lub ← llb
                  if h = n
                  then ub ← lub
             else SEARCH-ID(j + 1, h)
```

Fig. 3. *Iterative Deepening*

```
IOR()
    ub ← +∞
    lb ← 0
    while ub = +∞
        lub ← +∞
        SEARCH-IOR(1, lb)
        lb ← lub

SEARCH-IOR(j, lb)
    for k = 1 to DS(j)
        A[j] ← k
        llb ← FC(A, 1, j)
        if llb ≤ lb
        then if j = n
            then ub ← lub ← llb
            else SEARCH-IOR(j + 1, lb)
        else lub ← min(lub, llb)
```

Fig. 4. *Iterative Objective Relaxing*

4.4 Iterative Approximating

What we refer to as *ia* (for *Iterative Approximating*) is referred to as A^*_ϵ in [13]. By replacing the backtrack condition of the classical *Depth First Branch and Bound* ($ub \leq$ FC$(A, 1, j)$) by a weaker condition ($\epsilon.ub \leq$ FC$(A, 1, j), 0 < \epsilon \leq 1$), we obtain at the end of the search a problem lower bound lb equal to $\epsilon.ub$. Increasing ϵ produces an anytime problem lower bound. For the sake of homogeneity with the other algorithms, we chose $\epsilon = \frac{i}{n}$, with i increasing from 1 to n (see Figure 5).

4.5 Russian Doll Search

Russian Doll Search, we refer to as *rds*, has been recently proposed in [19]. Assuming a static variable ordering, one searches for an optimum of the problem limited to the variables after i (i included). The optimum found is obviously a problem lower bound and decreasing i from n down to 1 provides an anytime problem lower bound.

More importantly, the successive optima are recorded (in the global array rds) and used in order to improve the lower bound associated with each node of the future searches: because they do not take into account the same constraints, FC$(A, 1, j)$ (resulting from *Forward Checking*) and $rds[j + 1]$ (resulting from previous searches) can be safely added to obtain a higher subproblem lower bound FC$(A, i, j) + rds[j + 1]$ (see Figure 6).

4.6 Iterative Deepening + Russian Doll Search

Iterative Deepening + *Russian Doll Search*, we refer to as *idrds*, results from a combination of *Iterative Deepening* and *Russian Doll Search*. With the original *Russian Doll*

```
IA()
    ub ← +∞
    lb ← 0
    for i = 1 to n
        SEARCH-IA(1, i)
        lb ← ub.i/n

SEARCH-IA(j, i)
    for k = 1 to DS(j)
        A[j] ← k
        llb ← FC(A, 1, j)
        if llb < ub.i/n
        then if j = n
             then ub ← llb
             else SEARCH-IA(j + 1, i)
```

Fig. 5. *Iterative Approximating*

```
RDS()
    ub ← +∞
    lb ← 0
    rds[n + 1] ← 0
    for i = n downto 1
        lub ← +∞
        SEARCH-RDS(i, i)
        lb ← rds[i] ← lub

SEARCH-RDS(j, i)
    for k = 1 to DS(j)
        A[j] ← k
        llb ← FC(A, i, j) + rds[j + 1]
        if llb < lub
        then if j = n
             then lub ← llb
                  if i = 1
                  then ub ← lub
             else SEARCH-RDS(j + 1, i)
```

Fig. 6. *Russian Doll Search*

Search, one searched for an optimum of the problem limited to the variables after i (i included). With the new algorithm, one only searches for a lower bound of this optimum, using an *Iterative Deepening* approach. The depth h of the search increases from 1 to n. For each value of h, i decreases from $n-h+1$ down to 1.

As previously, these successive lower bounds are recorded (in the global array $lbrds$) and used in order to improve the lower bound associated with each node of the future searches: as with *rds* and for the same reasons, $\text{FC}(A, 1, j)$ and $lbrds[j+1]$ can be safely added to obtain a subproblem lower bound (see Figure 7). Note that, because $lbrds[j+1] \le rds[j+1]$:

$$\text{FC}(A,i,j) \le \text{FC}(A,i,j) + lbrds[j+1] \le \text{FC}(A,i,j) + rds[j+1]$$

```
IDRDS()
  ub ← +∞
  lb ← 0
  lbrds[n + 1] ← 0
  for h = 1 to n
     for i = n − h + 1 downto 1
        lub ← +∞
        SEARCH-IDRDS(i, i, h)
        lbrds[i] ← max(lub, lbrds[i + 1])
        lb ← max(lub, lb)

SEARCH-IDRDS(j, i, h)
  for k = 1 to DS(j)
     A[j] ← k
     llb ← FC(A, i, j) + lbrds[j + 1]
     if llb < lub
     then if j = i + h − 1
          then lub ← llb
               if h = n
               then ub ← lub
          else SEARCH-IDRDS(j + 1, i, h)
```

Fig. 7. *Iterative Deepening + Russian Doll Search*

5 Experiments

For the first four algorithms (*rbf*, *id*, *ior* and *ia*), we used the following classical dynamic variable ordering: among the variables of minimum current domain size, choose the first of maximum degree. For the last two (*rds* and *idrds*), which only support a static variable ordering, we used a maximum degree ordering. For all the algorithms, *Forward Checking* allows us to use a dynamic minimum *Inconsistency Count* value ordering [6, 16]: choose for the variable l the first value k such that $\Delta\text{BC}(A, i, j, l, k)$ is minimum.

5.1 Randomly Generated Problems

We carried out first experiments on *Max-CSPs*, randomly generated according to the four parameter model described in [17]: a number n of variables, a domain size d constant for all the variables, a graph connectivity c, and a constraint tightness t constant for all the constraints.

The results we show have been obtained with $n = 30$, $d = 10$ and $c = 0.5$. Figures 8, 9 and 10 are respectively associated with $t = 0.5$, $t = 0.7$ and $t = 0.9$, with 100 problems for each t's value. All these problems are inconsistent and cannot be optimally solved within a reasonable time. The number of violated constraints in optimal solutions increases with t. Note that, unlike what occurs with classical *CSPs*, complexity also increases with t and decreases only when t becomes very close to 1 [19, 11]. Each figure shows the mean evolution, within 300 seconds, of the problem lower bound, computed by each of the six algorithms.

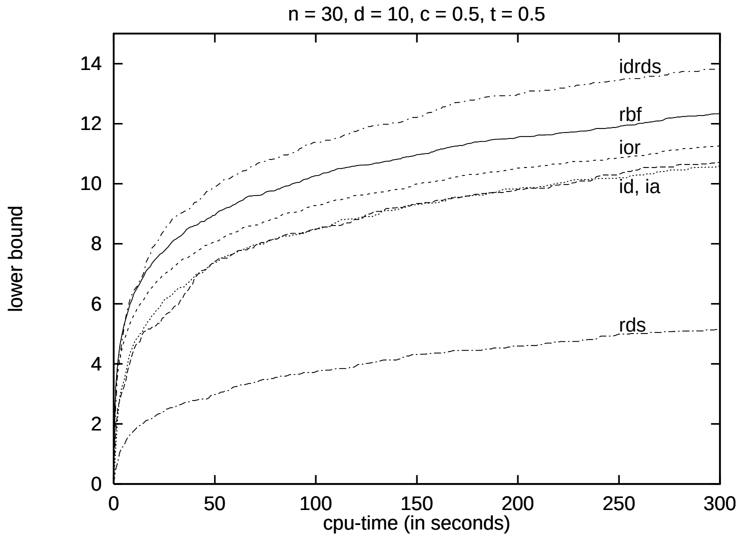

Fig. 8. Randomly generated problems. Tightness = 0.5

The three figures display the same traits:

- *rds* is not the right option for producing anytime lower bounds;
- *rbf*, *id*, *ior*, and *ia* have a similar behavior;
- *idrds* is systematically the best option.

The similarity between *rbf*, *id*, *ior*, and *ia*, as well as the relative efficiency of *idrds*, increases with constraint tightness. With $t = 0.9$, the mean lower bound produced by *rbf*, *id*, *ior*, and *ia*, within 300 seconds is about 50. Within the same time, the mean lower bound produced by *idrds* exceeds 100.

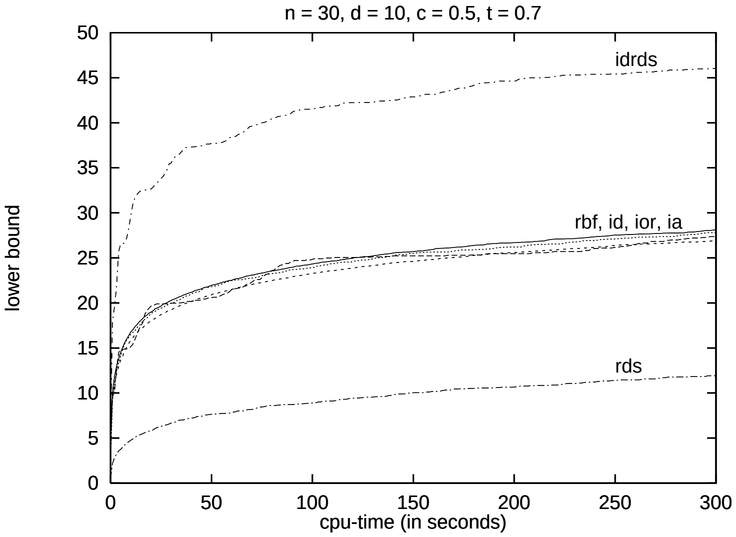

Fig. 9. Randomly generated problems. Tightness = 0.7

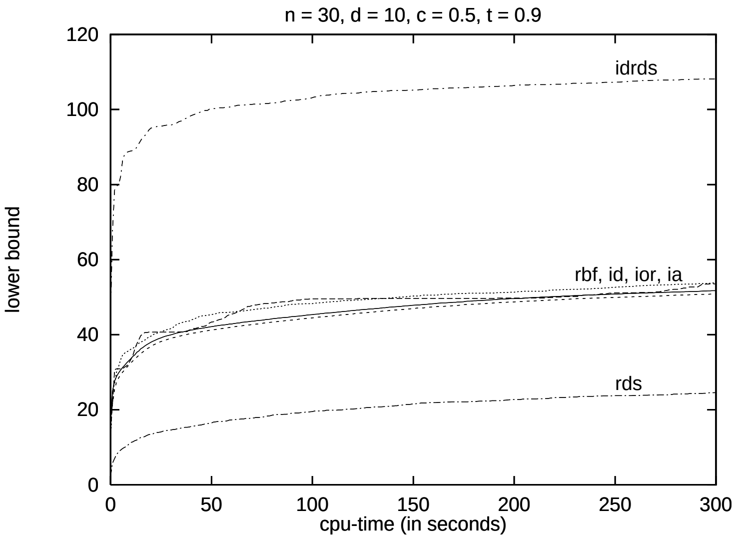

Fig. 10. Randomly generated problems. Tightness = 0.9

Figure 11 gives an idea of the quality of the optimum bounding, which can be obtained by using concurrently an *idrds* algorithm for producing lower bounds and a *Simulated Annealing* [7] algorithm for producing upper bounds: t is the constraint tightness, ub the mean upper bound provided by *Simulated Annealing*, lb the mean lower bound provided by *idrds* within 300 seconds, and $q = \frac{lb}{ub}$ a measure of the optimum bounding quality. One observes that this quality increases with constraint tightness.

t	0.5	0.7	0.9
ub	31.24	73.88	135.08
lb	13.82	46.04	108.14
q	44.24%	62.32%	80.06%

Fig. 11. Optimum bounding quality

5.2 Real World Problems

This behavior is made still clearer with larger, more difficult problems, like the *CELAR* instances of the *Radio Link Frequency Assignment Problem* (see ftp://ftp.cert.fr/pub/lemaitre/LVCSP/Pbs/CELAR/). Figure 12 shows the evolution, within 70000 seconds, of the *CELAR6* instance lower bound, produced by *ior* and *idrds*. This instance involves 100 variables, with up to 44 values per variable, and 1222 constraints, whose weights equal 1000, 100, 10, or 1. The objective is to minimize the sum of the weights of the unsatisfied constraints and the optimum equals 3389[2]. Within 70000 seconds, *ior* does not manage to produce a lower bound higher than 104. Note that *rbf*, *id*, and *ia* produce similar results. On the contrary, *idrds* produces a lower bound equal to 1314 within 325 seconds and equal to 2338 within 64809 seconds.

5.3 Explanation

Explaining this behavior may help to develop more efficient algorithms. These are some preliminary explanations:

- although *rds* is an efficient algorithm, these problems are too hard for it; one observes that *rds* is generally stalled, when searching for the optimum of subproblems involving a subset of the variables; it provides consequently poor lower bounds; this behavior is strengthened by the maximum degree variable ordering: the first considered variables are the least constrained; note that choosing the inverse order leads to worse results: solving subproblems is strongly penalized by a bad variable ordering;

[2] This optimum has been proven by producing a lower bound, equal to the best upper bound found (see [3]).

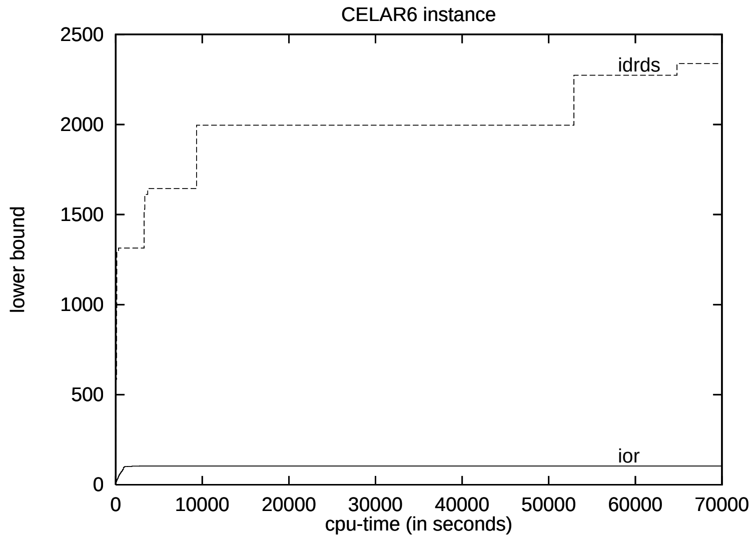

Fig. 12. Radio Link Frequency Assignment Problem

– the similar behavior of *rbf*, *id*, *ior*, and *ia* was unexpected; it may be due to the fact that these four algorithms use the same *Forward Checking* method for computing a subproblem lower bound at each node of the search; because *Forward Checking* takes into account only the constraints between assigned variables and between assigned and unassigned variables, the resulting lower bounds are poor, particularly when few variables are assigned; note that preliminary experiments with the same algorithms, improved by combining *Forward Checking* and *Directed Arc Consistency Counts* [20, 10], do not provide significantly better results;
– as to the efficiency of *idrds*, it may be due to an excellent cooperation between *id* and *rds*:

  • at the beginning of the search, with low values for h, the computed lower bounds, stored in the array $lbrds$, are poor, but searches are strongly limited; then progressively, as lower bounds increase, searches are less and less limited;
  • the main difference from *rds* lies in the fact that *idrds* is not stalled in the optimal solving of subproblems involving a subset of the variables; it rapidly considers the whole problem;
  • the main difference from *id* and the similar algorithms lies in the use of the results of the previous searches to improve the lower bound associated with each node of the current search;
  • finally, the fact that the efficiency of *idrds* increases with constraint tightness is certainly due to increasing subproblem lower bounds and a consequently increasing improvement of the lower bound associated with each node.

6 Conclusion

In this paper, we presented six variants of a *Branch and Bound* search, which are able to produce an anytime lower bound of the optimum of a *Weighted CSP*. Used concurrently with any incomplete algorithm, able to produce an anytime upper bound of the problem optimum, they can provide the user or the system with an increasingly tighter bounding of the optimum. Among these six algorithms, the last, resulting from a combination of *Russian Doll Search* and *Iterative Deepening*, is definitely the most efficient.

But, the results in Figure11, concerning the quality of the optimum bounding, recall that a lot of work remains to be done before getting really usable lower bounds. A combination of the *limited tree search* methods presented in this paper with the *problem simplification* methods recalled in 3.1 might be a promising way to explore.

7 Acknowlegments

This work was supported by ONERA-CERT. We thank Michel Lemaître, Thomas Schiex, Lionel Lobjois, and David Martinez for helping us to improve the first versions of this paper.

References

1. S. Bistarelli, H. Fargier, U. Montanari, F. Rossi, T. Schiex, and G. Verfaillie. Semiring-based CSPs and Valued CSPs: Basic Properties and Comparison. In M. Jampel, E. Freuder, and M. Maher, editors, *Over-Constrained Systems (LNCS 1106, Selected papers from the Workshop on Over-Constrained Systems at CP-95, reprints and background papers)*, pages 111–150. Springer, 1996.
2. S. Bistarelli, U. Montanari, and F. Rossi. Constraint Solving over Semirings. In *Proc. of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 624–630, Montréal, Canada, 1995.
3. S. de Givry, G. Verfaillie, and T. Schiex. Bounding the Optimum of Constraint Optimization Problems. In *Proc. of the 3rd International Conference on Principles and Practice of Constraint Programming (CP-97)*, Schloss Hagenberg, Austria, 1997.
4. H. Fargier, D. Dubois, and H. Prade. Problèmes de satisfaction de contraintes flexibles: une approche égalitariste. *Revue d'Intelligence Artificielle*, 9(3):311–354, 1995.
5. H. Fargier and J. Lang. Uncertainty in Constraint Satisfaction Problems: a Probabilistic Approach. In *Proc. of the European Conference on Symbolic and Quantitavive Approaches of Reasoning under Uncertainty (ECSQARU-93)*, pages 97–104, Grenade, Spain, 1993.
6. E. Freuder and R. Wallace. Partial Constraint Satisfaction. *Artificial Intelligence*, 58:21–70, 1992.
7. S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by Simulated Annealing. *Science*, 220, 1983.
8. R. Korf. Depth-First Iterative Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27:97–109, 1985.
9. R. Korf. Linear-space Best-first Search. *Artificial Intelligence*, 62:41–78, 1993.
10. J. Larrosa and P. Meseguer. Expoiting the Use of DAC in MAX-CSP. In *Proc. of the 2nd International Conference on Principles and Practice of Constraint Programming (CP-96, LNCS 1118)*, pages 308–322, Cambridge, MA, USA, 1996.

11. J. Larrosa and P. Meseguer. Phase Transition in MAX-CSP. In *Proc. of the 12th European Conference on Artificial Intelligence (ECAI-96)*, pages 190–194, Budapest, Hungary, 1996.
12. C. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994.
13. J. Pearl. *HEURISTICS, Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Publishing Company, 1984.
14. A. Rosenfeld, R. Hummel, and S. Zucker. Scene Labeling by Relaxation Operations. *IEEE Transactions on Systems, Man, and Cybernetics*, 6(6):173–184, 1976.
15. T. Schiex. Possibilistic Constraint Satisfaction Problems or "How to handle soft constraints ?". In *Proc. of the 8th International Conference on Uncertainty in Artificial Intelligence (UAI-92)*, Stanford, CA, USA, 1992.
16. T. Schiex, H. Fargier, and G. Verfaillie. Problèmes de satisfaction de contraintes valués. *Revue d'Intelligence Artificielle*, 11(3):339–373, 1997.
17. B. Smith. Phase Transition and the Mushy Region in Constraint Satisfaction Problems. In *Proc. of the 11th European Conference on Artificial Intelligence (ECAI-94)*, pages 100–104, Amsterdam, The Netherlands, 1994.
18. G. Verfaillie and S. de Givry. Algorithmic problems and solutions in the Valued Constraint Satisfaction Problem framework. In *Proc. of the EUFIT-97 session on "Valued Constraint Satisfaction"*, Aachen, Germany, 1997.
19. G. Verfaillie, M. Lemaître, and T. Schiex. Russian Doll Search for Solving Constraint Optimization Problems. In *Proc. of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 181–187, Portland, OR, USA, 1996.
20. R. Wallace. Directed Arc Consistency Preprocessing. In *Proc. of the ECAI-94 Workshop on Constraint Processing (LNCS 923)*, pages 121–137. Springer, 1994.