Bound arc consistency for weighted CSPs

Christine Gaspin, Thomas Schiex, Matthias Zytnicki

INRA Toulouse – BIA

Abstract. WCSP is a soft constraint framework with a wide range of applications. Most current complete solvers can be described as a depth-first branch and bound search that maintain some form of local consistency during the search. However, the known consistencies are unable to solve problems with huge domains because of their time and space complexities. In this paper, we adapt a weaker form of arc consistency, well-known in classic CSPs, called the *bound arc consistency* and we provide several algorithms to enforce it.

1 Introduction

The weighted constraint satisfaction problem (WCSP) is a well-known extension of the CSP framework with many practical applications. Recently, several generalizations of the CSP's arc consistency have been proposed for soft constraints, like AC* in [1]. Unfortunately, the time complexity always increases by a factor of d (the size of the largest domain) and the memory space is at least proportional to d. This makes these consistencies useless for problems with long domains like RNA detection or temporal constraints with preferences. We present here an extension of the *bound arc consistency*, first described for classic CSPs in [2]. Its time and space complexities are better than the complexities of AC* by an order of d.

Bound arc consistency (BAC*) is based on a interval representation of the sets of values and it can treat efficiently "easy" constraints, such as precedence

$$f(v_1, v_2) = \begin{cases} v_2 - v_1 - d & \text{if } v_2 - v_1 - d > 0, \\ 0 & \text{otherwise.} \end{cases}$$

that often show up in problems with long domains (like scheduling). We also propose several extensions of this consistency that take into account the semantics of the function, like monotonicity or convexity and we define *∅-inverse consistency* that can boost the cost propagation on some conditions.

Finally, we compare BAC* with AC* on the problem of non-coding RNA detection and show the superiority of our consistency for this kind of problems.

2 Preliminaries

Valuation structures are algebraic objects that specify costs [3]. For WCSP [4], it is defined by a triple $\mathcal{S} = \langle E, \oplus, \leq \rangle$ where

– $E = [0..k] \subseteq \mathbb{N}$ is the *set of costs*, k can possibly be ∞;
– $\oplus$, the *addition* on E, is defined by $\forall (a,b) \in \mathbb{N}^2, a \oplus b = \min\{a+b, k\}$,
– $\leq$ is the usual operator on $\mathbb{N}$.

It is useful to define the *subtraction* $\ominus$ of costs:

$$\forall (a,b) \in \mathbb{N}^2, a \ominus b = \begin{cases} a-b & \text{if } a \neq k, \\ k & \text{otherwise.} \end{cases}$$

A binary WCSP is a tuple $\mathcal{P} = \langle \mathcal{S}, \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where:

– $\mathcal{S}$ is the valuation structure,
– $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of n variables,
– $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ is the set of the finite domains of each variable and the size of the largest one is d,
– $\mathcal{C} = \{c_1, \ldots, c_e\}$ is the set of e constraints.

A constraint $c \in \mathcal{C}$ can be either:

– a unary constraint: $c : D(x_i) \to E$ (we call it c_i), or
– a binary constraint: $c : D(x_i) \times D(x_j) \to E$ (we call it c_{ij}).

We will restrict ourselves to *binary* WCSP, where no constraint has an arity greater than 2. Results can easily be extended to higher arity constraints. Furthermore, we assume the existence of a unary constraint c_i for every variable, and a *zero*-arity constraint (i.e. a constant), noted $c_\varnothing$ (if no such constraints are defined, we can always define *dummy* ones: c_i is the null function over $D(x_i)$, $c_\varnothing = 0$).

Given a pair (v_i, w_j) (resp. a value v_i), $c_{ij}(v_i, w_j) = k$ (resp. $c_i(v_i) = k$) means that the constraint forbids the corresponding assignment. Another cost means the pair (resp. the value) is permitted by the constraint with the corresponding cost. The cost of an assignment $t = (v_1, \ldots, v_n)$, noted $\mathcal{V}(t)$, is the sum over all the cost functions:

$$\mathcal{V}(t) = \bigoplus_{i,j} c_{ij}(v_i, v_j) \oplus \bigoplus_{i} c_i(v_i) \oplus c_\varnothing$$

An assignment t is *consistent* if $\mathcal{V}(t) < k$. The usual task of interest is to find a consistent assignment with minimum cost. This is a NP-hard problem. Observe that, if $k = 1$, a WCSP reduces to classic CSP.

3 Some local properties

3.1 Existing local consistencies

WCSPs are usually solved with a branch-and-bound tree of which each node is a partial assignment. To accelerate the search, local consistency properties are widely used to transform the sub-problem at each node of the tree to an equivalent, simpler one. The simplest local consistency property is the *node consistency* (NC*, cf. [1]).

Definition 1. *A variable x_i is node consistent if:*

- $\forall v_i \in D(x_i), c_\varnothing \oplus c_i(v_i) < k$ *and*
- $\exists v_i \in D(x_i), c_i(v_i) = 0$ *(this value v_i is called the* unary support *of x_i).*

A WCSP is node consistent if every variable is node consistent.

This property can be enforced in time and space $\mathcal{O}(nd)$. Another famous stronger local consistency is the *arc consistency* (AC*, cf. [1]).

Definition 2. *The* neighbours $N(x_i)$ *of a variable x_i is the set of the variables x_j such that there exists a constraint that involves x_i and x_j. More formally:*

$$\forall x_i \in \mathcal{X}, N(x_i) = \{x_j \in \mathcal{X} : c_{ij} \in \mathcal{C}\}$$

A variable x_i is arc consistent if:

- $\forall v_i \in D(x_i), \forall x_j \in N(x_i), \exists w_j \in D(x_j), c_{ij}(v_i, w_j) = 0$ *(this value w_j is called the* support *of x_i in v_i w.r.t. c_{ij}) and*
- *x_i is node consistent.*

A WCSP is arc consistent if every variable is arc consistent.

On a binary WCSP, arc consistency can be enforced in time $\mathcal{O}(n^2d^3)$ and in space $\mathcal{O}(ed)$. The algorithm uses the operations ProjectUnary and Project described in Alg. 1 to enforce the supports of the values and the unary supports respectively.

Algorithm 1: Operations enforcing AC*

Procedure *ProjectUnary(x_i)* [Find the unary support of x_i]
  $min \leftarrow \min_{v_i \in I(x_i)}\{c_i(v_i)\}$;
  if $(min = 0)$ **then return** ;
  $c_\varnothing\_raised \leftarrow$ true ;
1 **foreach** $v_i \in I(x_i)$ **do** $c_i(v_i) \leftarrow c_i(v_i) \ominus min$;
  $c_\varnothing \leftarrow c_\varnothing \oplus min$;
  if $(c_\varnothing \geq k)$ **then raise** *exception* ;

Procedure *Project(x_i, v_i, x_j)* [Find the support of v_i w.r.t. c_{ij}]
  $min \leftarrow \min_{w_j \in I(x_j)}\{c_{ij}(v_i, w_j)\}$;
2 **foreach** $w_j \in I(x_j)$ **do** $c_{ij}(v_i, w_j) \leftarrow c_{ij}(v_i, w_j) \ominus min$;
  $c_i(v_i) \leftarrow c_i(v_i) \oplus min$;

Example 1. Fig. 1(a) represents an instance of a small problem. It contains two variables (x_1 and x_2) with two possible values for each one (a and b), a unary constraint for each variable (the costs are written in the circles) and a binary constraint (the costs are written on the edge that connects a pair of values; if there is no edge between two values, the cost is 0). k is arbitrarily set to 4 and $c_\varnothing$ is set to 0. As the cost of $x_1 = a$ is equal to k (first point of the definition

of NC*), this value is discarded (cf. FIG. 1(b)). Then, we notice that x_2 has no unary support (second point of the definition of NC*) and we project a cost of 1 to $c_\varnothing$ (cf. FIG. 1(c)). The instance is NC*. To enforce AC*, we project 1 from the binary constraint to $x_1 = a$ as this value has no support (cf. FIG. 1(d)). Finally, we project 1 from $c_1(b)$ to $c_\varnothing$, as seen on FIG. 1(e).

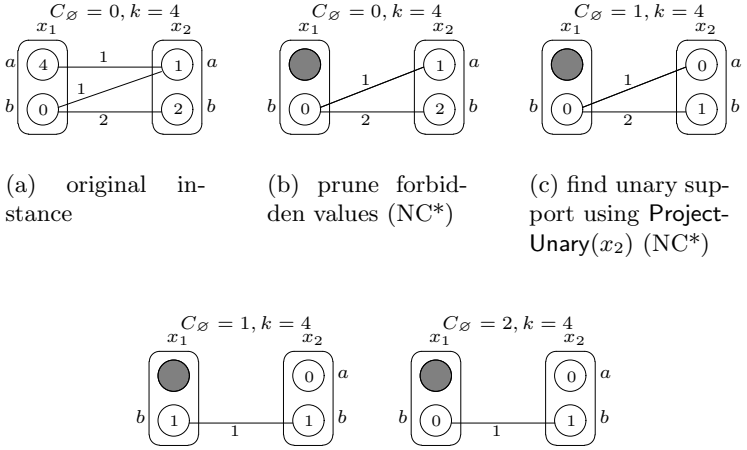
In practice, to reach the $\mathcal{O}(ed)$ space complexity, the algorithm uses extra costs differences data structures as suggested in [5]. For each value v_i of each variable involved in each binary constraint c_{ij}, we create a new cost difference $\Delta_{ij}^{v_i}$, initialized to 0. It stores the cost that has been projected to $c_i(v_i)$ by the binary constraint c_{ij}. Thus the line **2** can be replaced by

$$\Delta_{ij}^{v_i} \leftarrow \Delta_{ij}^{v_i} \oplus min \ ;$$

and every occurrence of "$c_{ij}(v_i, w_j)$" should be replaced by "$c_{ij}(v_i, w_j) \ominus (\Delta_{ij}^{v_i} \oplus \Delta_{ij}^{w_j})$". Similarly, we use another cost difference in ProjectUnary for each variable: Δ_i. It stores the cost that has been projected from c_i to $c_\varnothing$. The line **1** can be replaced by

$$\Delta_i \leftarrow \Delta_i \oplus min \ ;$$

and every occurrence of "$c_i(v_i)$" should be replaced by "$c_i(v_i) \ominus (\Delta_i)$".



(a) original instance

(b) prune forbidden values (NC*)

(c) find unary support using ProjectUnary(x_2) (NC*)

(d) find support for $x_1 = b$ using Project(x_1, b, x_2) (AC*)

(e) find unary support using ProjectUnary(x_1)

Fig. 1. Steps to enforce AC*

3.2 Bound arc consistency

We present here a consistency which is weaker than AC*. It can be enforced with lower time and space complexities and it is called *bound arc consistency* (BAC*).

Definition 3. *To apply bound arc consistency, we need to change the definition of a WCSP: the domains are now* intervals $\mathcal{I}$. *Each variable x_i can take all the values in $I(x_i) = [lb_i..ub_i]$ (lb_i is the* lower bound *of the interval of x_i and ub_i is its* upper bound*). A variable x_i is bound node consistent (BNC*) if:*

- $(c_\varnothing \oplus c_i(lb_i) < k) \wedge (c_\varnothing \oplus c_i(ub_i) < k)$ *and*
- $\exists v_i \in I(x_i), c_i(v_i) = 0$.

A variable x_i is bound arc consistent if:

- $\forall x_j \in N(x_i), \exists (w_j, w'_j) \in I^2(x_j), c_{ij}(lb_i, w_j) = c_{ij}(ub_i, w'_j) = 0$ *and*
- *it is bound node consistent.*

A WCSP is bound arc consistent if every variable is bound arc consistent.

The intervals initially range over all the possible values. We shall suppose that all the values of the variables are sorted by an arbitrary order and $\forall x_i \in \mathcal{X}, lb_i = \min\{D(x_i)\}, ub_i = \max\{D(x_i)\}$. Changing the representation of the set of the values to intervals alters the expressivity of the framework: it is not possible to describe that a value which is inside an interval has been deleted. But this allows us to decrease the space complexity as a domain is now represented by only two values. The Alg. 2 provides an algorithm to enforce this consistency.

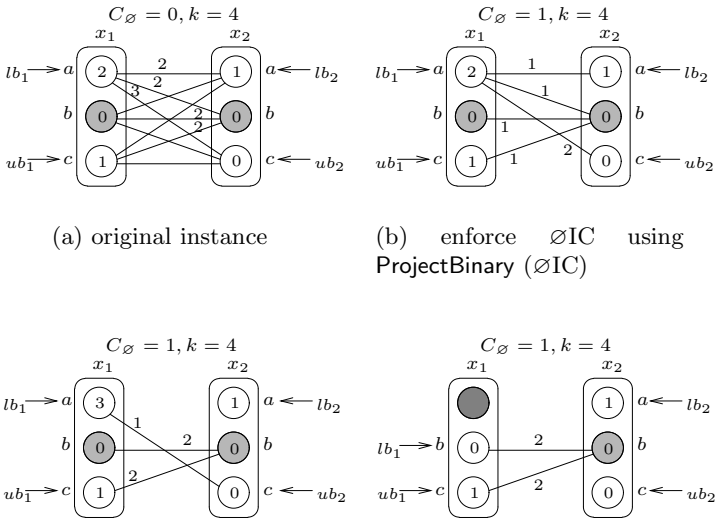
Example 2. Fig. 2(b) describes another problem. The values are supposed to be sorted by the lexicographic order ($a \prec b \prec c$), thus $lb_1 = a$ and $ub_1 = c$ for x_1 and the same for x_2. After a call of $\mathsf{Project}(x_1, a)$, we get Fig. 2(c). As $c_\varnothing \oplus c_i(lb_1)$ is equal to k, $x_1 = a$ is discarded and the lower bound of x_1 is updated to lb_1 (cf. Fig. 2(d)). This instance is BAC* but not AC* because $x_2 = b$ has no support. This proves that BAC* is strictly weaker than AC*.

Theorem 1. *Algorithm 2 enforces BAC* in time $\mathcal{O}(ed^2 + knd)$ and in space $\mathcal{O}(n + e)$.*

Proof. **Correction:** We will consider the following invariants:

1. on line **2**, all variables are BNC*,
2. if x_i is not in Q, then $\forall x_j \in N(x_i), lb_i, ub_i, lb_j$ and ub_j have a support w.r.t. c_{ij}.

First, $\mathsf{ProjectUnary}(x_i)$ finds the unary support of x_i and $\mathsf{SetBNC*}(x_i)$ loops until it finds the allowed bounds of x_i, so this function enforces BNC*. At the beginning of the algorithm, as the variables may not have this property, we call $\mathsf{SetBNC*}(x_i)$ for each variable x_i. Thus the second invariant is respected at the beginning of the algorithm.



(a) original instance

(b) enforce ∅IC using ProjectBinary (∅IC)

(c) project to lb_1 using Project (BAC*)

(d) move lb_1 (BAC*)

Fig. 2. Steps to enforce BAC* with ∅IC

This invariant may be broken by a projection from a binary constraint to a bound of an interval; this may either lead to the fact that one of the bound is now forbidden, or that a unary support (which was this bound) has disappeared. This is why SetBNC* is called on x_j and all its neighbours (lines **5** and **6**) after the projections of the line **4**.

The first invariant could also be broken when $c_\varnothing$ increases: a bound can now have a unary cost greater that $k - c_\varnothing$. This event can occur after the lines **5** and **6**. This explains the **if** beginning at line **7**.

Concerning the second invariant, it is true at the beginning of the algorithm as all the variables are enqueued. Afterwards, Project(x_i, v_i, x_j) finds the support of v_i w.r.t. c_{ij}, so SetBSupport(x_i, x_j) finds the supports of the bounds of x_i w.r.t. c_{ij}. Thus the line **4** enforces the second invariant.

This invariant can only be broken by SetBNC* and anytime this function is called, the corresponding variable is enqueued. Finally, at the end of the algorithm, the instance is BNC* (thanks to the first invariant) and every bound has a support w.r.t. to each constraint in which it is involved (thanks to the second invariant): the problem is now BAC*.

Time complexity: Thanks to [1], we know that Project and ProjectUnary take time $\mathcal{O}(d)$. Thus SetBSupport also takes time $\mathcal{O}(d)$ and the complexity of the line **1** is $\mathcal{O}(nd)$.

Algorithm 2: Algorithm enforcing BAC*

```
   Procedure SetBAC*()                                              [ Enforce BAC* ]
1      foreach x_i ∈ 𝒳 do SetBNC*(x_i) ;
       Q ← 𝒳; c_∅_raised ← false ;
2      while (Q ≠ ∅) do
           x_j ← Q.pop() ;
3          foreach x_i ∈ N(x_j) do
4              SetBSupport(x_i, x_j) ; SetBSupport(x_j, x_i) ;
5              if (SetBNC*(x_i)) then Q ← Q ∪ {x_i} ;
6          if (SetBNC*(x_j)) then Q ← Q ∪ {x_j} ;
7          if (c_∅_raised) then
               c_∅_raised ← false ;
8              foreach x_i ∈ 𝒳 do
9                  if (SetBNC*(x_i)) then Q ← Q ∪ {x_i} ;

   Function SetBNC*(x_i): boolean                                   [ Enforce NC* ]
       changed ← false ;
10     while (bi_i ≤ bs_i) ∧ (c_∅ ⊕ (c_i(bi_i) ⊖ Δ_i) ≥ k) do
11         bi_i ← bi_i + 1 ; changed ← true ;
12     while (bi_i ≤ bs_i) ∧ (c_∅ ⊕ (c_i(bs_i) ⊖ Δ_i) ≥ k) do
13         bs_i ← bs_i − 1 ; changed ← true ;
       ProjectUnary(x_i) ;
       return changed ;
   Procedure SetBSupport(x_i, x_j)   [ Find the supports of the bounds of x_i w.r.t. c_ij ]
       Project(x_i, bi_i, x_j) ; Project(x_i, bs_i, x_j) ;
```

Each variable can be pushed in at most $\mathcal{O}(d)$ times into Q, thus the overall complexity of the line **6** is $\mathcal{O}(nd^2)$. The program enters in the loop of line **3** at most $\mathcal{O}(ed)$ times (given a constraint c_{ij}, the program can enter $\mathcal{O}(d)$ times because of x_i and $\mathcal{O}(d)$ times because of x_j) thus the overall complexity of lines **4** and **5** is $\mathcal{O}(ed^2)$. The line **7** can be true at most k times (otherwise the problem is detected as inconsistent) and the overall complexity of the line **9** is $\mathcal{O}(k \times n \times d)$. To sum up, this algorithm takes time $\mathcal{O}(nd^2 + ed^2 + knd) = \mathcal{O}(ed^2 + knd)$. However, as the **while** on line **2** can be true at most $\mathcal{O}(nd)$ times, the **foreach** on line **8** cannot loop more than $\mathcal{O}(n^2d)$ times and the complexity of the line **9** is not greater than $\mathcal{O}(n^2d^2)$. So the actual time complexity is $\mathcal{O}(ed^2 + \min\{k, nd\} \times nd)$, and if $k > nd$ then it is $\mathcal{O}(n^2d^2)$.

Space complexity: For each binary constraint, we need 4 cost differences (one for each bound of each variable) and for each variable x_i, a cost difference Δ_i. Including the space for Q, the overall space complexity is $\mathcal{O}(e + n)$.

3.3 Strengthening BAC*

We may want to enforce a stronger local consistency that takes into account the constraint costs involving values *inside* the intervals. To keep a reasonable space

complexity, this cost will be projected directly to $c_\varnothing$. Thus we add to the BAC* property the *$\varnothing$-inverse consistency* ($\varnothing$IC):

Definition 4. *The constraint c_{ij} is $\varnothing$-inverse consistent if*

$$\exists (v_i, w_j) \in D(x_i) \times D(x_j), c_{ij}(v_i, w_j) = 0$$

(this pair (v_i, w_j) is called the binary support of $c_\varnothing$). A WCSP is $\varnothing$-inverse consistent if every constraint is $\varnothing$-inverse consistent.

Remark that $\varnothing$IC is a generalization to a higher arity of the second point of the NC* property.

When BAC* finds a support w_j for lb_i w.r.t. c_{ij}, it projects the cost $c_{ij}(lb_i, w_j)$ to the unary constraint c_i. The constraint is now $\varnothing$IC (the binary support is (lb_i, w_j)), but this property is more relevant when enforced first: it directly increases the $c_\varnothing$.

Example 3. Let us resume with the problem on FIG. 2(a). If no cost is mentionned on an edge, it is by default 1. We can see on this instance that for any value of x_1 and for any value of x_2, the binary constraint yields to a cost not less than 1. In this case, BAC* would project some binary costs to the bounds but $\varnothing$IC directly projects all of this costs to $c_\varnothing$ (cf. FIG. 2(b)); this guarantees an increase of the lower bound.

Algorithm 3: Algorithm enforcing BAC* with $\varnothing$IC

Procedure *SetBSupport(x_i, x_j)* [Add $\varnothing$IC to the previous procedure]
  ProjectBinary(x_i, x_j) ;
  Project(x_i, bi_i, x_j) ; Project(x_i, bs_i, x_j) ;

Procedure *ProjectBinary(x_i, x_j)* [Find the binary support of c_{ij}]
  $min \leftarrow \min_{\substack{v_i \in I(x_i) \\ w_j \in I(x_j)}} \{c_{ij}(v_i, w_j) \ominus (\Delta_{ij}^{v_i} \oplus \Delta_{ij}^{w_j} \oplus \Delta_{ij})\}$;
  if $(min = 0)$ **then return** ;
  $c_\varnothing\_raised \leftarrow$ true ;
  $\Delta_{ij} \leftarrow \Delta_{ij} \oplus min$;
  $c_\varnothing \leftarrow c_\varnothing \oplus min$;
  if $(c_\varnothing \geq k)$ **then raise** *exception* ;

ALG. 3 shows the differences with the previous algorithm to enforce BAC* with $\varnothing$IC.

Theorem 2. ALG. *3 takes time $\mathcal{O}(ed^3 + knd)$ and space $\mathcal{O}(n + e)$.*

Proof. **Correction:** We add an invariant to the ones listed in the previous proof:

3. if x_i is not in Q, then $\forall x_j \in N(x_i), c_{ij}$ has a binary support.

Note that the prerequisite is the same as in the first invariant. This comes from the fact that, once a binary support has been enforced, only the application of SetBAC* can break it. As this invariant is enforced in the same time as the first invariant, the same reasoning applies.

Time complexity: The procedure ProjectBinary takes time $\mathcal{O}(d^2)$. Thus the overall complexity of the algorithm becomes $\mathcal{O}(nd^2 + ed^3 + knd) = \mathcal{O}(ed^3 + knd)$.

Space complexity: As we just store the cost difference, we only need $\mathcal{O}(e)$ extra space to remember the cost that has been projected from a constraint directly to $c_\varnothing$. The overall space complexity remains the same.

It could be possible to decrease the time complexity in d by using an appropriate structure that contains the sorted costs of a constraint. But this would increase the space complexity by a factor at least of d^2, which is unacceptable. Another possibility to have a faster algorithm is to use the semantics of the constraints to find the minimum of the function in less than $\mathcal{O}(d^2)$ time, when possible, to decrease the complexity. We need a definition to describe easily the cost propagation:

Definition 5. *Given a binary constraint c_{ij}, $c_{ij}(v_i, w_j)$ is a* border *cost if $v_i = lb_i$ or $v_i = ub_i$ or $w_j = lb_j$ or $w_j = ub_j$. It is an* interior *cost otherwise.*

Given a unary constraint c_i, $c_i(v_i)$ is a border *cost if $v_i = lb_i$ or $v_i = ub_i$. It is an* interior *cost otherwise.*

Theorem 3. *If the minimum of the binary cost functions can be found in $\mathcal{O}(d)$ time, the complexity of BAC* with ∅IC becomes $\mathcal{O}(ed^2 + knd)$ with no memory space increase.*

Proof. The main difficulty is that the costs of the constraint can be projected either to the unary constraints (BAC*) or to $c_\varnothing$ (∅IC). In the latter case, the minimum is still attained by the same tuple as all costs have uniformly decreased. In the former case, the actual minimum may be a border cost and each of them must be checked. There are $4(d-1)$ border costs and finding the minimum amoung interior cost, by assumption, takes $\mathcal{O}(d)$ time. ProjectBinary now takes time $\mathcal{O}(d)$ and thus the complexity of the whole algorithm is $\mathcal{O}(ed^2 + knd)$.

This result is particularly interesting for semi-convex functions (well-known in temporal constraints with preferences) w.r.t. *a single variable*, because the minimum cost is reached by a value on the edge of the cost matrix and so can be found in $\mathcal{O}(d)$ time.

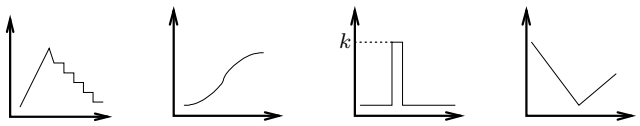
Definition 6. *A function c_i (resp. c_{ij}) is* semi-convex *[6] iff: $\forall e \in E$, the set*

$$\{v_i \in D(x_i) : c_i(v_i) > e\} \text{ (resp. } \{(v_i, w_j) \in D(x_i) \times D(x_j) : c_{ij}(v_i, w_j) > e\}\text{)}$$

is an interval.

Informally speaking, semi-convex functions have only one peak. An example of semi-convex function is described Fig. 3(a). The unary semi-convex functions

encompass monotonic functions (cf. FIG. 3(b)) and anti-functional constraints [7] (cf. FIG. 3(c)). The function on FIG. 3(d) is not semi-convex. An example of semi-convex function w.r.t. a single variable is $x, y \mapsto x^2 - y^2$. It is semi-convex w.r.t. x but not to y.



(a) semi-convex (b) semi-convex (c) anti-functional (d) not semi-convex

Fig. 3. Characteristics of some functions

If the costs functions are semi-convex w.r.t. *every variable*, like $x, y \mapsto x + y$, the minima can be found in constant time because they are located in the corner of the cost matrices and we have the following result:

Theorem 4. *If the minimum of unary and binary cost functions can be found in constant time, the complexity of BAC* with $\varnothing IC$ becomes $\mathcal{O}(ed+kn)$ with no memory space increase.*

Proof. To find the binary support of c_{ij} in ProjectBinary rapidly, we need to compute nine minima and compare them: the minimum of the interior of c_{ij}, the minimum of the four borders (excluding the corners) $c_{ij}(lb_i, .) \ominus \Delta_{ij}^{lb_i}$, $c_{ij}(ub_i, .) \ominus \Delta_{ij}^{ub_i}$, $c_{ij}(., lb_j) \ominus \Delta_{ij}^{lb_j}$ and $c_{ij}(., ub_j) \ominus \Delta_{ij}^{ub_j}$, and the minimum of the four corners $c_{ij}(lb_i, lb_j) \ominus \Delta_{ij}^{lb_i} \ominus \Delta_{ij}^{lb_j}$, $c_{ij}(lb_i, ub_j) \ominus \Delta_{ij}^{lb_i} \ominus \Delta_{ij}^{ub_j}$, $c_{ij}(ub_i, lb_j) \ominus \Delta_{ij}^{ub_i} \ominus \Delta_{ij}^{lb_j}$ and $c_{ij}(ub_i, ub_j) \ominus \Delta_{ij}^{ub_i} \ominus \Delta_{ij}^{ub_j}$. Thus, ProjectBinary and SetBSupport run in constant time.

The same idea applies to ProjectUnary. The domain should be split in three parts (the interior and the two bounds) and the minimum can be found and projected to $c_\varnothing$ in constant time with the cost differences Δ_i. Now we can notice that the conditions at lines **10** and **12** are true, given a variable, at most d times, so the overall complexity of lines **11** and **13** is $\mathcal{O}(nd)$.

Let us sum up the overall complexities:

- the line **4** takes $\mathcal{O}(ed)$,
- the line **5** takes $\mathcal{O}(ed + nd)$,
- the line **6** takes $\mathcal{O}(nd)$,
- the line **9** takes $\mathcal{O}(kn + nd)$,

This proves our theorem.

4 Discussion

Comparison with 2B-consistency: The definition of 2B-consistency, as defined in [2] for numeric non-binary CSP (NCSP) is:

Definition 7. *$x \in \mathcal{X}$ is 2B-consistent if $\forall c : D(x) \times D(x_1) \times \ldots \times D(x_r) \in \mathcal{C}$ if:*

- $\exists (v_1, \ldots v_r) \in D(x_1) \times \ldots \times D(x_r), c(lb, v_1, \ldots, v_r)$ *and*
- $\exists (v_1, \ldots v_r) \in D(x_1) \times \ldots \times D(x_r), c(ub, v_1, \ldots, v_r)$.

A NCSP is 2B-consistent iff every variable is 2B-consistent.

Obviously, a WCSP such that $k = 1$ which is BAC* is 2B-consistent.

Besides, it is possible to express a WCSP in classic CSP by *reifying* the costs [8].

Definition 8. *Consider the WCSP $\mathcal{P} = \langle \mathcal{S}, \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ Let $\mathcal{P}' = \langle \mathcal{X}', \mathcal{D}', \mathcal{C}' \rangle$ be the classic CSP such that:*

- *the set $\mathcal{X}'$ of variables is $\mathcal{X}$ augmented with a cost variable x_E per constraint: x_E^{ij} for the binary constraint c_{ij}, x_E^i for the unary constraint c_i;*
- *the domain of x is $D(x)$ if x is in $\mathcal{X}$, E if x is a cost variable x_E; the set of the domains is $\mathcal{D}'$;*
- *the set $\mathcal{C}'$ of constraints contains:*
  - *the reified constraints c'_{ij} defined by the set of tuples*
  $$\{(v_i, w_j, e) : v_i \in D(x_i), w_j \in D(x_j), e = c_{ij}(v_i, w_j)\}$$
  - *the reified constraints c'_i defined by the set of tuples*
  $$\{(v_i, e) : v_i \in D(x_i), e = c_i(v_i)\}$$
  - *an extra constraint c'_E that applies on the cost variables x_E*
  $$\sum_{c_{ij} \in \mathcal{C}} x_E^{ij} + \sum_{c_i \in \mathcal{C}} x_E^i < k$$

The problem $\mathcal{P}'$ has a solution iff $\mathcal{P}$ has a solution. The aim of enforcing a property is usually to find inconsistencies as soon as possible. This leads to a definition of the *strength* of a consistency:

Definition 9. *A property $\mathcal{T}$ is at least as strong as another property $\mathcal{T}'$ iff for any problem $\mathcal{P}$, when the enforcement of $\mathcal{T}'$ finds an inconsistency, then $\mathcal{T}$ finds an inconsistency too.*

Consider now the little WCSP defined by three variables (x_1, x_2 and x_3) and two binary constraints ($c_{1,2}$ and $c_{1,3}$). $D(x_1) = \{a, b, c, d\}$, $D(x_2) = D(x_3) = \{a, b, c\}$ (we suppose $a \prec b \prec c \prec d$) and the costs of the binary constraints are described FIG. 4. We set k to 2.

The reader can check the reified problem is 2B-consistent. BAC* would detect an unconsistency by projecting the costs to x_1 and reducing little by little its domain. This shows that BAC* is at least not comparable with 2B-consistency for reified WCSPs. The existence of a more accurate comparison between these consistencies is still an open problem.

$$
\begin{array}{cc}
 & (x_1) \\
 & \begin{array}{cccc} a & b & c & d \end{array} \\
\begin{array}{c} a \\ (x_2)\ b \\ c \end{array} & \begin{bmatrix} 1 & 0 & 2 & 1 \\ 1 & 0 & 2 & 1 \\ 1 & 0 & 2 & 1 \end{bmatrix}
\end{array}
\qquad
\begin{array}{cc}
 & (x_1) \\
 & \begin{array}{cccc} a & b & c & d \end{array} \\
\begin{array}{c} a \\ (x_3)\ b \\ c \end{array} & \begin{bmatrix} 1 & 2 & 0 & 1 \\ 1 & 2 & 0 & 1 \\ 1 & 2 & 0 & 1 \end{bmatrix}
\end{array}
$$

Fig. 4. Two cost matrices

Comparison with AC*: BAC* coupled with ∅IC can be strictly weaker than AC* even for semi-convex functions. Consider for example the matrix cost in FIG. 5. It represents the costs of a binary semi-convex function with domain $[a..c]$. All the bounds have a support and thus the constraint is BAC* and ∅IC. But the values b have no support and thus this instance is not AC*.

$$
\begin{array}{c}
 \\
\begin{array}{c} a \\ b \\ c \end{array} \\
\end{array}
\begin{array}{c}
\begin{array}{ccc} a & b & c \end{array} \\
\begin{bmatrix} 0 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix}
\end{array}
$$

Fig. 5. A cost matrix

The advantage of BAC with ∅IC is that projecting the minimum of a constraint requires only one operation. For the same cost propagation, AC* must project from the binary constraints to the unary constraints and to the unary constraints to $c_\varnothing$. Moreover, if AC* does not project all the binary costs on the same variable, $c_\varnothing$ may even not increase with the same amount.

To take advantage of the efficiency of BAC* with ∅IC and the strength of AC*, both consistencies can be combined in the same algorithm. Initially, the set of values is represented by intervals. When they are smaller than a given value, intervals are transformed into domains and holes are possible. This needs only minor changes in the code in SetBSupport and SetBNC*.

Extension of BAC* for piecewise functions: BAC* can also be extended to efficiently handle piecewise monotonic function. It is called *piecewise bound arc consistency*:

Definition 10. *To apply piecewise bound arc consistency (PBAC*), an interval $I(x_i)$ becomes a set of p_i intervals $I^1(x_i), \ldots, I^{p_i}(x_i)$ with $\forall q \in [1..p_i], I^q(x_i) = [lb_i^q..ub_i^q]$. We also have $lb_i^1 = lb_i$, $ub_i^{p_i} = ub_i$ and $\forall q \in [1..p_i - 1], ub_i^q + 1 = lb_i^{q+1}$. A variable x_i is piecewise bound node consistent (PBAC*) if:*

- $\forall q \in [1..p_i], (c_\varnothing \oplus c_i(lb_i^q) < k) \wedge (c_\varnothing \oplus c_i(ub_i^q) < k)$ *and*
- $\exists v_i \in \bigcup_{q \in [1..p_i]} I^q(x_i), c_i(v_i) = 0$.

A variable x_i is piecewise bound arc consistent if:

– $\forall q \in [1..p_i], \forall x_j \in N(x_i), \exists (w_j, w'_j) \in I^2(x_j), c_{ij}(lb_i^q, w_j) = c_{ij}(ub_i^q, w'_j) = 0$,
– *it is piecewise bound node consistent.*

A WCSP is piecewise bound arc consistent if every variable is piecewise bound arc consistent.

Even for continuous function, dividing the long intervals into several smaller ones could notably improve the cost propagation.

5 Experimental results

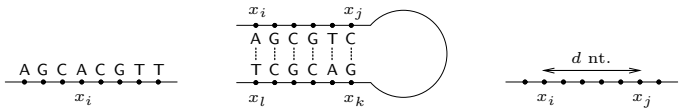
We have applied BAC* to the problem of non-coding RNA (ncRNA) detection. RNA sequences can be considered as oriented texts (left to right) over the four letter alphabet {A, C, G, U}. An RNA molecule can fold on itself through interactions between the nucleotides G–C, C–G, A–U and U–A. Such a folding gives rise to characteristic structural elements such as helices (a succession of paired nucleotides), and various kinds of loops (unpaired nucleotides surrounded by helices).

Thus, the information contained both in the sequence itself and the structure can be viewed as a biological signal to exploit and search for. These common structural characteristics can be captured by a signature that represents the structural elements which are conserved inside a set of related RNA molecules.

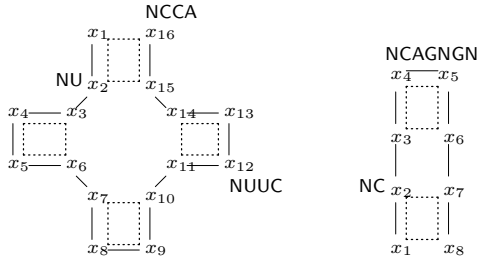
We call *motif* the elements of the secondary structure that define a RNA family. To a first approximation, a motif can be decomposed into *strings* (cf. Fig. 6(a)) and *helices* (cf. Fig. 6(b)). Two elements can be separated by *spacers* (cf. Fig. 6(c)). These elements of description are modeled by soft constraints and the costs are given by the usual pattern matching algorithms (for strings and helices) or analytic function (for spacers).

Our aim is to find all the occurrences in the sequence that match the given motif, and the cost of these solutions. We have tried to detect the structure of tRNA [10] (cf. Fig. 6(d)), modeled by 16 variables, 15 spacers, 3 strings and 4 helices as well as an IRE motif [11] (cf. Fig. 6(e)) modeled by 8 variables, 7 spacers, 2 strings and 2 helices on parts of the genome of *Saccharomyces Cerevisiæ* of different sizes and on the whole genome of *Escherichia coli*. For tRNA, we used two different models, the first being much tighter than the second.

For each soft constraint, there is an hard constraint that prunes all the unconsistent values faster through bound arc consistency for classic CSPs. As the helix is a 4-ary constraint, we used a generalized bound arc consistency to propagate the costs. ∅IC has been enforced for spacers (which are semi-convex functions) but not for strings nor for helices. We used a 2.4Ghz Intel Xeon with 8 GB RAM to solve these instances. The results on our comparison between our algorithm and the classic AC* are displayed on Fig. 7. For each instance of the problem, we write its size (10k is sequence of 10.000 nucleotides and the genome of *Escherichia coli* contains more than 4.6 millions nucleotides) and the number of solutions. We also show the number of nodes explored and the time in seconds



(a) constraint $strings(x_i, \mathsf{ACGT})$ (b) constraint $helix(x_i, x_j, x_k, x_l)$ (c) constraint $spacer(x_i, x_j, d)$



(d) tRNA (e) IRE

Fig. 6. A few motifs

tRNA, tight definition						
Size / ♯ solutions	10k / 16	50k / 16	100k / 16	500k / 16	1M / 24	ecoli / 140
AC* (nodes/time)	23 / 29	35 / 545	-	-	-	-
BAC* (nodes/time)	32 / 0	39 / 0	51 / 0	194 / 1	414 / 2	1867 / 7
tRNA, loose definition						
Size / ♯ solutions	10k / 84	50k / 84	100k / 84	500k / 111	1M / 164	ecoli / 702
AC* (nodes/time)	215 / 401	495 / 7041	-	-	-	-
BAC* (nodes/time)	347 / 0	1036 / 1	1775 / 2	8418 / 4	17499 / 8	83476 / 34
IRE						
Size / ♯ solutions	10k / 0	50k / 0	100k / 0	500k / 1	1M / 4	ecoli / 8
AC* (nodes/time)	0 / 3	0 / 57	0 / 223	-	-	-
BAC* (nodes/time)	0 / 0	0 / 0	0 / 0	20 / 0	44 / 2	237 / 8

Fig. 7. Number of nodes explored and time in seconds spent to solve several instances of the ncRNA detection problem

spent. A "-" means the instance could not be solved due to memory reasons despite all the memory optimizations.

The reason of the superiority of BAC* over AC* is twofold. First, AC* needs to store all the unary cost for every variable to project cost from binary constraints to unary constraint. Thus, the space complexity of AC* is at least $\mathcal{O}(nd)$. For very long domains (in our experiment, greater than 50.000 values), the computer cannot allocate sufficient memory and the program is aborted. For the

same kind of projection, BAC* only needs to store the costs of the bounds of the domains, leading to a space complexity of $\mathcal{O}(n)$. A similar conclusion would have been drawn after a comparison between BAC* and Max-CSP algorithms like PFC-MRDAC (cf. [12]).

Second, the *distance* constraints dramatically reduce the size of the domains. Concretely, when a single variable is assigned, and when all the distance costs have been propagated, all the other domains have a size that is a constant with respect to d. As BAC* behaves particularly well with this kind of constraints, the instance becomes quickly tractable.

6 Conclusions and future work

In this paper we have presented a new local consistency for weighted CSPs, called *bound arc consistency*. It is specially devoted to problems with large domains and time and space complexities are lower than the well-known arc consistencies. Several extensions have been proposed for constrains with good characteristics, like semi-convex functions, and $\varnothing$IC seems particularly efficient for this kind of functions. Finally, we showed that maintaining BAC* is much better than AC* for the problem of ncRNA detection. In the future, we will try to implement better heuristics for boosting the search.

References

1. Larrosa, J.: Node and arc consistency in weighted CSP. In: Proc. AAAI'02. (2002)
2. Lhomme, O.: Consistency techniques for numeric CSPs. In: Proc. IJCAI 1993. (1993) 232–238
3. Schiex, T., Fargier, H., Verfaillie, G.: Valued constraint satisfaction problems: Hard and easy problems. In: Proc. IJCAI 1995. (1995)
4. Larrosa, J., Schiex, T.: Solving Weighted CSP by Maintaining Arc-consistency. Artificial Intelligence **159** (2004) 1–26
5. Cooper, M., Schiex, T.: Arc consistency for soft constraints. Artificial Intelligence **154** (2004) 199–227
6. Khatib, L., Morris, P., Morris, R., Rossi, F.: Temporal constraint reasoning with preferences. In: Proc. IJCAI 2001. (2001) 322–327
7. Hentenryck, P.V., Deville, Y., Teng, C.M.: A generic arc-consistency algorithm and its specializations. Artificial Intelligence **57** (1992) 291–321
8. Petit, T., Régin, J.C., Bessière, C.: Meta-constraints on violations for over constrained problems. In: Proc. ICTAI'00. (2000) 358–365
9. Bessière, C., Régin, J.C.: Refining the basic constraint propagation algorithm. In: Proc. IJCAI 2001. (2001) 309–315
10. Gautheret, D., Major, F., Cedergren, R.: Pattern searching/alignment with RNA primary and secondary structures: an effective descriptor for tRNA. Comp. Appl. Biosc. **6** (1990) 325–331
11. Gorodkin, J., Heyer, L.L., Stormo, G.D.: Finding the most significant common sequence and structure motifs in a set of RNA sequences. Nucleic Acids Research **25** (1997) 3724–3732
12. Larrosa, J., Meseguer, P., Schiex, T.: Maintaining reversible DAC for Max-CSP. Artificial Intelligence **17** (1999) 149–163