Backtracking Through Biconnected Components of a Constraint Graph

Jean-François Baget
LIRMM
161, rue Ada
34392 Montpellier, Cedex 5
France
E-mail: baget@lirmm.fr

Yannic S. Tognetti
LIRMM
161, rue Ada
34392 Montpellier, Cedex 5
France
E-mail: tognetti@lirmm.fr

Abstract

The algorithm presented here, BCC, is an enhancement of the well known *Backtrack* used to solve constraint satisfaction problems. Though most backtrack improvements rely on propagation of local informations, BCC uses global knowledge of the constraint graph structure (and in particular its biconnected components) to reduce search space, permanently removing values and compiling partial solutions during exploration. This algorithm performs well by itself, without any filtering, when the biconnected components are small, achieving optimal time complexity in case of a tree. Otherwise, it remains compatible with most existing techniques, adding only a negligible overhead cost.

1 Introduction

Constraint satisfaction problems (CSPs), since their introduction in the early 60's, have flourished in many branches of Artificial Intelligence, and are now used in many "real-life" applications. Since the satisfiability of a CSP is a NP-complete problem, much effort have been devoted to propose faster algorithms and heuristics. Backtrack can be seen as the backbone for those improvements: this algorithm first extends a partial solution by assigning a value to a variable, and undo a previous assignment if no such value can be found.

Backtracking heuristics try to restrict search space, their goal can be to maximize the probability of a "good guess" for the next assignment (variable and value orderings), or to recover more efficiently from a dead-end (backjumping and its variants). *Filtering techniques* use propagation of some local consistency property ahead of the current partial solution, effectively reducing variable domains; they can be used either in a preprocessing phase, or dynamically during exploration. *Structure-driven* algorithms emerged from the identification of tractable classes of CSP, such as trees [Mackworth and Freuder, 1985]; they often transform the CSP into one that can be solved in polynomial time [Gottlob *et al.*, 1999].

The algorithm presented, BCC, does not fit easily in this classification. It accepts any preprocessed ordering of the variables, as long as it respects some properties related to the biconnected components of the constraint graph. Then it exploits the underlying tree structure to reduce *thrashing* during the backtrack, storing partial solutions and removing permanently some values. Though no kind of local consistency property is used, and no look-ahead is performed, BCC performs well when the graph contains small biconnected components, achieving optimal time complexity in case of a tree.

After basic definitions about CSPs, we present in Sect. 3 the version of *Backtrack* used as the kernel of BCC. Sect. 4 is devoted to definitions of the variable orderings compatible with BCC. The algorithm is presented in Sect. 5, along with a proof of its soundness and completeness and an evaluation of its worst-case complexity. In Sect. 6, we compare BCC with popular backjumping or filtering techniques, pointing out the orthogonality of these approaches and showing that a mixed algorithm is possible. Finally, we discuss limitations of our algorithm, and suggest a method to overcome them.

2 Definitions and Notations

A *binary constraint network* $\mathcal{R}$ over a set of symbols $\mathcal{D}$ consists of a set of *variables* $V = \{x_1, \ldots, x_n\}$ (denoted $V(\mathcal{R})$), each x_i associated to a finite *value domain* $D_i \subseteq \mathcal{D}$, and of a set of *constraints*. A constraint R_{ij} between two variables x_i and x_j is a subset of $D_i \times D_j$. A variable x_i is called *instantiated* when it is assigned a value from its domain. A constraint R_{ij} between two instantiated variables is said *satisfied* if $(\text{value}(x_i), \text{value}(x_j)) \in R_{ij}$. This test is called a *consistency check* between x_i and x_j. A *consistent instantiation* of a subset $X \subseteq V(\mathcal{R})$ is an instantiation of all variables in X that satisfies every constraint between variables of X. The CONSTRAINT SATISFACTION PROBLEM (CSP), given a constraint network $\mathcal{R}$, asks whether there exists a consistent instantiation of $V(\mathcal{R})$, called a *solution* of $\mathcal{R}$.

Without loss of generality, we can consider that constraints are always defined over two different variables, and that there is at most one constraint (either R_{ij} or R_{ji}) between two variables. So the *constraint graph* of a network (associating each variable to a node and connecting two nodes whose variables appear in the same constraint) is a non oriented simple graph without loops. In this paper, we will consider some total order $\leq_V$ on $V(\mathcal{R})$, inducing an orientation of this graph: edges can be seen as oriented from the smallest to the greatest of their ends. According to this orientation, we can define the *predecessors* $\Gamma^-(x)$ and the *successors* $\Gamma^+(x)$ of a node x. Note that $|\Gamma^-(x)|$ is the *width* of x [Freuder, 1982].

3 Back to Backtrack

As the `Backtrack` algorithm (BT) is the kernel of BCC, we considered as vital to make it as efficient as possible, according to its usual specifications: 1) variables are examined in a fixed, arbitrary ordering; and 2) no information on the CSP is known or propagated ahead of the current variable.

3.1 Algorithm Kernel

Algorithm 1: `BackTrack`($\mathcal{R}$)

```
Data   : A non empty network R.
Result : true if R admits a solution, false otherwise.
computeOrder (N);
level ← 1;
failure ← false;
while ((level ≠ 0) and (level ≠ |V(R)| + 1)) do
    currentVariable ← V[level];
    if (failure) then
        if (hasMoreValues (currentVariable)) then
            failure ← false;
            level ← nextLevel (currentVariable);
        else level ← previousLevel (currentVariable);
    else
        if (getFirstValue (currentVariable)) then
            level ← nextLevel (currentVariable);
        else
            failure ← true;
            level ← previousLevel (currentVariable);
return (level = |V(R)| + 1);
```

As in [Prosser, 1993], BT is presented in its derecursived version: we will later easily control the variable to study after a successful or failed instantiation, without unstacking recursive calls. For the sake of clarity, it is written to solve a decision problem, though a solution could be "read in the variables". This algorithm should consider any variable ordering, so `computeOrder` only builds the predecessor and successor sets according to the implicit order on the variables, and *sorts the predecessors* sets (for soundness of *partial history*). To respect our specifications, if x is the ith variable according to $\leq_V$, `previousLevel`(x) must return $i-1$ and `nextLevel`(x) $i+1$. According to this restriction, our only freedom to improve BT is in the implementation of the functions `getFirstValue` and `hasMoreValues`.

3.2 Computing Candidates

Suppose BT has built a consistent instantiation of $\{x_1, \ldots, x_{i-1}\}$. Values that, assigned to x_i, would make the instantiation of $\{x_1, \ldots, x_i\}$ consistent, are the *candidates* for x_i. Then BT is sound and complete if `getFirstValue` indicates the first candidate, returning `false` when missing (a *leaf dead-end* at x_i, [Dechter and Frost, 1999]), and successive calls to `hasMoreValues` iterate through them, returning `false` when they have all been enumerated (*internal dead-end*). Testing whether a value is a candidate for x is done in at most width(x) consistency checks.

The iterative method: The most natural method to implement `getFirstValue` (resp. `hasMoreValues`) is to iterate through the domain of x_i, halting as soon as a value pass a consistency check with every y in $\Gamma^-(x_i)$, starting at the first domain value (resp. after the last successful value). Both return `false` when the whole domain has been checked. Though in the worst case, the cost of is $\Theta(|D_i|$ width$(x_i))$, values are checked only when needed: this method is good when the CSP is underconstrained or when width(x_i) is small.

The partition refinement method: Suppose $\Gamma^-(x_i) = \{y_1, \ldots, y_k\}$, and note Δ_0 the domain D_i of x_i. Then compute $\Delta_1, \ldots, \Delta_k$ such that Δ_j contains the values of Δ_{j-1} satisfying the constraint R_{ji}. Then Δ_k is the set of candidates for x_i. Though worst case complexity (when constraints are "full") is the same, it is efficient in practice: if $0 < p \leq 1$ is the *constraint tightness* of a random network [Smith, 1996], then the expected size of Δ_{j+1} is $(1-p) \times |\Delta_j|$; so the expected time to compute Δ_k is $|D_i|/p$. Though some computed candidates may not be used, this method performs well when constraints tightness and variables width are high. The method `hasMoreValues` only iterates through this result.

Storing refinement history: A *partial history* is a trace of the refinements computed in the *current branch* of the backtrack tree: so having computed $\Delta_1 \ldots \Delta_k$ and stored these sets as well as the values of y_j used to obtain them, if we backtrack up to y_p and come back again later to x_i, we only have to refine domains from Δ_{p+1} to Δ_k, replacing the previous values. This mechanism is implemented in Alg. 2.

Algorithm 2: `getFirstValue`(x)

```
Data   : A variable x, belonging to a network R where BT has computed
         a consistent instantiation of all variables preceding x. We also
         suppose that x has at least one ancestor.
Result : Stores refinement history, and returns true unless the computed
         set of candidates is empty.
last ← 1;
w ← width(x);
y ← Γ⁻[0];
while (last ≤ w and Δ[last] ≠ ∅ and usedVal[last] = y.value) do
    y ← Γ⁻[last];
    last++;
while ((last ≤ w) and (Δ[last-1] ≠ ∅)) do
    y ← Γ⁻[last-1];
    usedVal[last] ← y.value;
    Δ[last] ← ∅;
    for (v ∈ Δ[last-1]) do
        if ((y.value, v) ∈ constraint(y, x)) then
            Δ[last] ← Δ[last] ∪ {v};
    last++;
success ← (last = w+1) and (Δ[last-1] ≠ ∅);
if (success) then value ← Δ[last-1][0];
return success;
```

Each variable x is assigned a field `last` (denoted, in an OOP style, x.`last` or simply `last` when there is no ambiguity), a field `value` pointing to the current value of x, a vector containing the *sorted* variables in $\Gamma^-(x)$, and two vectors of size width(x)+1: Δ contains the successive refinements, and `usedVal` contains the values of the ancestors used for successive refinements of the Δ_j. Note that Δ[k] contains all values of Δ[k-1] consistent with the assignment of `usedVal`[k] to the variable `ancestors`[k-1]. Δ[0] is initialized with the domain of x. The partial history mechanism adds the same benefit as *Backmarking* (BM) [Gaschnig, 1979]. Though more memory expensive ($\mathcal{O}(|D_i| \times$ width$(x_i))$ space, for each variable x_i) than the usual version of BM, it is more resistant to the jumps forward and backward that will be explicited further in this paper.

4 Variables Orderings Compatible with BCC

Let us now define the particular variable orderings that will be compatible with BCC. Let $V_1, \ldots, V_k$ be ordered subsets of $V(\mathcal{R})$ such that $V(\mathcal{R}) = \cup_{i=1}^{k} V_i$ (their intersection is not necessarily empty), then predset(V_i)= $\cup_{j=1}^{i-1} V_j$. We call an ordering of $V(\mathcal{R})$ *compatible* with this decomposition if, for every subset V_i, for every variable $x \in V_i$, if $x \notin$ predset(V_i), then x is greater than every variable of predset(V_i). Given such an ordering, the *accessor* of V_i is its smallest element.

4.1 Connected Components

A non empty graph G is *connected* if, for every pair of nodes, there is a walk from one node to the other. A connected component of G is a maximum connected subset of $V(G)$.

Now suppose a network $\mathcal{R}$ whose constraint graph admits $k \geq 2$ connected components, namely $V_1, \ldots, V_k$. We call a *CC-compatible ordering* of the variables of $\mathcal{R}$ an ordering of $V(\mathcal{R})$ compatible with some arbitrary ordering of these connected components. Should we launch BT on this network, and should `computeOrder` store such an ordering, a well-known kind of *thrashing* can occur: if BT fails on component V_i (*i.e.* it registers a dead-end at the accessor of V_i), then it will keep on generating every consistent instantiation of the variables in $V_1, \ldots, V_{i-1}$ to repeat the same failure on V_i.

Since the V_i represent *independent subproblems* (a global knowledge on the graph), the usual method is to launch BT on each connected component, stopping the whole process whenever some V_j admits no solution. It could also be implemented in a simple way with a slight modification of the function `previousLevel`: if x is the accessor of a connected component, `previousLevel`(x) must return 0.

Though very simple, this example illustrates well the method we use later: introduce in the variables ordering some "global" properties of the graph, then use them during backtrack. With a CC-compatible ordering, we benefit from the independence of the sub-problems associated to the connected components; with BCC-compatible orderings, we will benefit from "restricted dependence" between these sub-problems.

4.2 Biconnected Components

A graph G is *k-connected* if the removal of any $k-1$ different nodes does not disconnect it. A *k-connected component* of G is a maximum k-connected subgraph of G. A *biconnected component* (or *bicomponent*), is a 2-connected component. Note we consider the complete graph on two vertices biconnected. The *graph of bicomponents* (obtained by representing each bicomponent as a node, then adding an edge between two components if they share a node) is a forest called the *BCC tree* of G. It is computed in linear time, using two depth-first search (DFS) traversals of G [Tarjan, 1972]. Fig. 1 represents a constraint graph and its associated BCC tree.

If we consider the BCC tree as a rooted forest, a *natural ordering* of its nodes (representing bicomponents) is a total order such that children are greater than their parent. For the BCC tree in Fig. 1, choosing A and B as roots, a DFS could give the order $\{A, F, B, G, E, H, I, D, C\}$, and a BFS (breadth-first search) $\{A, F, B, G, H, C, E, I, D\}$. Both DFS and BFS traversals result in a natural ordering.

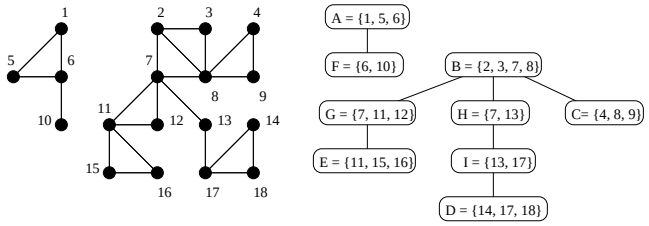

Figure 1: A constraint graph and its BCC tree.

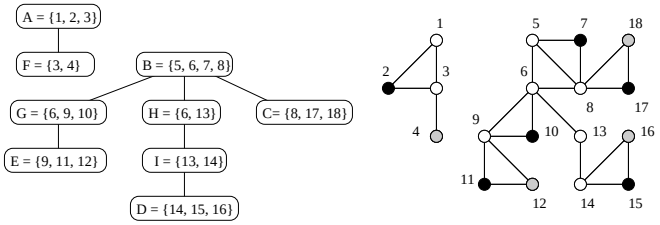

Figure 2: A BCC(DFS, BFS) ordering.

A *BCC-compatible ordering* of the variables of a network $\mathcal{R}$ is an ordering compatible with a natural ordering of its BCC tree (it is also CC-compatible). Such an ordering, if computed by a DFS traversal of the BCC tree, and by a BFS traversal inside the components, would be denoted by a BCC(DFS, BFS) ordering. It can be computed in linear time.

Nodes of the graph in Fig. 2 have been ordered by a BCC(DFS, BFS) ordering. Accessors of a bicomponent have been represented in white. The *second variable* of a bicomponent is the smallest variable in this component, its accessor excepted. By example, in Fig. 2, 9 is the second variable of the component $\{6, 9, 10\}$. Given a BCC-compatible ordering, the *accessor of a variable* is defined as the accessor of the smallest (according to the natural ordering) bicomponent in which it belongs. A second variable is always chosen in the neighborhood of its accessor. Tab. 1 gives the accessors determined by the BCC(DFS, BFS) ordering illustrated in Fig. 2.

Variable	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Accessor	1	1	1	3	5	5	5	5	6	6	9	9	6	13	14	14	8	8

Table 1: Accessors determined by the ordering of Fig. 2.

Suppose a natural ordering of a BCC tree nodes. A *leaf variable* is the greatest variable appearing in a leaf of the BCC tree. In Fig. 2, leaf variables are coloured in grey. We say that a bicomponent C *covers* a leaf variable y if y is the greatest node appearing in the subtree rooted by C (including C itself). Now, we define the *compilers* of a leaf variable y as the accessors of the bicomponents covering y. For the ordering in Fig. 2, compilers are: cmp$(4) = \{1, 3\}$, cmp$(12) = \{6, 9\}$, cmp$(16) = \{6, 13, 14\}$ and cmp$(18) = \{5, 8\}$.

5 The Algorithm BCC

The kernel of BCC is BT, as described in Sect. 3. The call to `computeOrder` orders the variables in a BCC-compatible way, and stores all information on accessors, leaf variables and compilers. This preprocessing takes linear time. We suppose BT is running on a network $\mathcal{R}$.

5.1 Theoretical Foundations

Theorem 1 *If x is the second variable of some bicomponent, any call to* `previousLevel`*(x) can safely (and optimally) return the index of its accessor y.*

Moreover, no solution of $\mathcal{R}$ contains the variable y instantiated with its current value.

A call to `previousLevel`(x) means that we have detected some dead-end (either internal or leaf) at variable x. The first part of this theorem asserts that we can backjump right to the accessor of x, without missing any possible solution, and that it could be dangerous to backjump any further (formal definitions about safe and optimal backjumps can be found, by example, in [Dechter and Frost, 1999]). A dead-end found at variable 13 in Fig. 2 would result in a backjump to variable 6. The second part of the theorem means that we can *permanently remove* the current value of y from its domain.

Proof of Theorem 1: The first part is proved by showing that `previousLevel`(x) performs a particular case of *Graph-Based Backjumping* (GBBJ) [Dechter, 1990]. The accessor y of a second variable x is the only variable in $\Gamma^-(x)$. GBBJ performs a safe backjump to the greatest predecessor of x, we have no other choice than y. It also adds to y a "temporary neighborhood" containing all other predecessors of x (Sect. 6.1). This set being empty, not maintaining it is safe. ◁

Second part relies on the ordering. The following lemma locates the leaf dead-ends responsible for the dead-end at x.

Lemma 1 *Let L be the set of variables that registered a dead-end after the last successful instantiation following x. Then variables in L belong to the subtree T of bicomponents rooted by the smallest component containing x (the* BCC subtree *of x, the accessor of x is also called the accessor of T).*

Sketch of proof: A failure is "propagated upward" in the same bicomponent until a second variable (since variables in the same component, the accessor apart, are numbered consecutively by the compatible ordering). At this point, failure is propagated to the accessor (the backjump in the first part of the theorem), and this accessor belongs to a parent component (a consequence of the natural ordering). ◁

Now suppose there is a solution such that y is assigned the value v, and we prove it is absurd. Let us now consider the subnetwork $\mathcal{R}'$ of $\mathcal{R}$, containing all the variables in the the BCC subtree of x, ordered in the same way as in $\mathcal{R}$. Let us reduce the domain of y in $\mathcal{R}'$ to $\{v\}$. Then $\mathcal{R}'$ admits a solution, and BT will find the first one for the given ordering. Let us now examine the first leaf dead-end (at variable x_e) that caused to bypass this solution in the BT on $\mathcal{R}$ (thanks to Lem. 1, it is a variable of $\mathcal{R}'$). Since we had just before a part of the solution of $\mathcal{R}'$, the domain of x_e has been emptied by a variable outside $\mathcal{R}'$ (or it would not have been a solution of $\mathcal{R}'$). So there is a constraint between x_e and some variable x'_e in a biconnected component outside of $\mathcal{R}'$: this is absurd, since there would be an elementary cycle $x_e, \ldots, x, y, \ldots, r, \ldots, x'_e, x_e$ where r appears in some ancestor of both the component of y and the one of x_e. Then x_e and x'_e would be in the same bicomponent. ◁

Since no variable smaller than y is responsible for the dead-end registered at x, the backjump is optimal. □

Theorem 2 *We suppose* `previousLevel` *has been implemented to match specifications of Th. 1. If x is a leaf variable numbered i, any call to* `nextLevel`*(x), before returning $i+1$, can mark, for every variable y being a compiler of x, that the current value v of y has been compiled and that $i+1$ is the* forward jump *(FJ) for that value (eventually erasing previous FJ for that value).*

Thereafter, every time a call to `nextLevel` *should return the index of a second variable z, if the current value v of its accessor t has been compiled,* `nextLevel` *can safely (and optimally) return instead the forward jump for v.*

Sketch of proof: This is the dual of Th. 1 second part: we had shown that when some failure propagates to the accessor x of a BCC subtree T, then no modification outside T can give a consistent instantiation of T, while keeping x current value. Now, when we have found a consistent instantiation of T, then no modification outside T can make all instantiations of T inconsistent, unless modifying the current value of x. □

A call to `nextLevel` on a leaf means that a whole BCC subtree has been successfully instantiated. In Fig. 2, a call to `nextLevel`(12) means we have a consistent instanciation of the BCC subtree rooted by the bicomponent G covering 12. Before returning 13, this call will store in the accessors of G and E their current values $v(6)$ and $v(9)$, and mark that the FJ for these compiled values is 13. Now we go to 13 and begin to instantiate the subtree containing 6 (already instantiated, and even compiled) and 13, ..., 16. If there is a dead-end at 13, we should backjump to 6 (see Th. 1), and permanently remove its current value $v(6)$ (we must also remove it from its compiled values). Otherwise, it means we can successfully instantiate this subtree until leaf 16, then make a call to `nextLevel`. This call updates all information for compilers of 16, and in particular in 6: the FJ for the compiled value $v(6)$ is now 17. Now suppose we have some dead-end at 17. According to Th. 1, we can backjump to 8. Now suppose further failures make us backtrack to 5, that we successfully instantiate, and that we also instantiate 6 with its compiled value. Should a call to `nextLevel` return 9, it would mean that we found a consistent instantiation of the whole component $\{5, 6, 7, 8\}$. Then we know there is a consistent instantiation of $\{5, \ldots, 16\}$, and we can safely jump forward to 17.

5.2 Implementation

Functions `previousLevel` and `nextLevel` (Alg. 3, 4) naturally encode the above theorems. `nextIndex` and `previousIndex` assume the previous role of `nextLevel` and `previousLevel` (returning $i \pm 1$).

Algorithm 3: `previousLevel`(x)

```
Data   : A variable x.
Result : The index of the variable to be examined by BCC after a dead-end.
if (accessor(x) = x) then return 0;
if (isSecondVariable? (x)) then
    accessor ← accessor(x);
    val ← accessor.value ;
    for (i = 0; i ≤ width(accessor); i++) do
        accessor.Δ[i] ← accessor.Δ[i] \ {val};
    return getIndex (accessor(x));
return previousIndex (x);
```

Algorithm 4: `nextLevel`(x)

```
Data   : A variable x.
Result : The index of the variable to be examined by BCC after a success.

index ← nextIndex(x);
if (isLeaf?(x)) then
    for c ∈ compilers(x) do
        c.compiledValues ← c.compiledValues ∪ {c.value};
        addKey/Value(c.FJ, (c.value, index));

if (index ≠ |V(R)| + 1) then
    nextVariable ← V[index];
    if (isSecondVariable?(nextVariable)) then
        accessor ← accessor(nextVariable);
        if (accessor.value ∈ accessor.compiledValues) then
            index ← getKey/Value(accessor.FJ, accessor.value);

return index;
```

5.3 Worst-Case Complexity

Lemma 2 *Using the BCC algorithm, any BCC subtree is entered at most once for each value in its accessor's domain.*

Proof: A consequence of Th. 1 and 2. Should we enter a BCC subtree T, either we find a consistent instantiation of T, and the current value for its accessor x is compiled (so every time `nextLevel` should return the second variable y for the smallest bicomponent containing x, it jumps instead to the next subtree); or we have registered some dead-end at y (thanks to Lem. 1, this failure came from T), and the current value for the accessor is permanently removed. □

Corollary 1 *When $\mathcal{R}$ is a tree, BCC runs at most in $\mathcal{O}(d^2 n)$, where $n = |V(\mathcal{R})|$ and d is the greatest domain size.*

Proof: Bicomponents are complete graphs with two nodes (accessor and second variable y). They are entered on y, and there will be at most d consistency checks on y. There is n bicomponents, each one entered at most d time (Lem. 2). □

This result is the same as the one given in [Freuder, 1982], obtained by first achieving arc consistency ($\mathcal{O}(d^2 n)$), then by a backtrack-free search. Preprocessing required for BCC is linear, and all possible dead-ends can be encountered, but BCC benefits as much from these failures as from successes. This complexity is optimal [Dechter and Pearl, 1988]. Th. 3 extends this result to any constraint graph.

Theorem 3 *Let n be the number of bicomponents, k be the size of the largest one, and d be the size of the largest domain, then BCC runs in the worst case in $\mathcal{O}(d^k n)$.*

Though this worst-case complexity is the same as [Freuder, 1982], BCC should be more efficient in practice: it backtracks along bicomponents instead of generating all their solutions.

6 BCC and Other Algorithms

Though BCC is efficient when the constraint graph contains numerous bicomponents, it is nothing more than a BT inside these components. We study in this section how to improve BCC with some well-known BT add-ons, and point out that BCC reduces the search tree in an original way.

6.1 Backjumping Schemes

To be sound, BCC must perform a very limited form of Backjumping (BJ) (see Th. 1). The natural question is: what if we use more performant backjumps?

Gaschnig's Backjumping (GBJ) [Gaschnig, 1979] is only triggered in case of a leaf dead-end. It jumps back to the first predecessor of x that emptied its domain. This backjump is safe and optimal in case of a leaf dead-end.

Graph-based Backjumping (GBBJ) [Dechter, 1990] is triggered on any dead-end x, and returns its greatest predecessor y. Safety is ensured by adding to y a temporary set (jumpback set) of predecessors consisting of all the predecessors of x. It is optimal when only graph information is used.

Conflict Directed Backjumping (CDBJ) [Prosser, 1993] integrates GBJ and GBBJ. It backjumps at any dead-end to the greatest variable that removed a value in the domain of x. The jumpback set contains only the predecessors of x that removed some value in x. CDBJ is safe and optimal.

Theorem 4 *Let* `previous`*(x) be a function implementing a safe Backjump scheme XBJ, always returning the index of a variable y belonging to a bicomponent containing x. Then if y is the accessor of x, and if no further backjump from y will return in the component of x, the current value of y can be permanently removed.*

The obtained algorithm BCC+XBJ is sound and complete.

Idea of proof: Restrictions enable to paste proof of Th. 1. □

Corollary 2 *The algorithms BCC+GBJ, BCC+GBBJ and BCC+CDBJ are sound and complete.*

Sketch of proof: See that BCC+GBBJ and BCC+CDBJ respect the specifications in Th. 4, and that a backjump on an accessor y makes us remove a value only if the jumpback set of y only contains its predecessors. Note also that GBJ does not always backjump in the same component (a second variable can backjump to the last element of the parent component, not necessarily the accessor), so BCC specific backjump must be kept to make BCC+GBJ sound and complete. □

Finally, we note that these BJ schemes recover more quickly from a failure than BT, but nothing ensures that this failure will not be repeated over and over. BCC's behaviour cannot be obtained by such a BJ mechanism.

6.2 Filtering Techniques

Different levels of arc consistency can be used in algorithms to prune values that become inconsistent ahead of the last instantiated variable. A constraint R_{ij} is said arc-consistent if, $\forall\ \delta_i \in D_i, \exists\ \delta_j \in D_j\ /\ (\delta_i, \delta_j) \in R_{ij}$, and conversely.

Maintaining Arc-Consistency (MAC) [Sabin and Freuder, 1994] is believed to be the most efficient general CSP algorithm. This family of algorithms rely on maintaining some kind of arc-consistency ahead of the current variable.

Theorem 5 *Let x be the variable currently studied by MAC, and y be a variable whose domain was emptied by the AC phase. Let $C_1, \ldots, C_k$ be the bicomponents belonging to the branch of the BCC tree, between the component C_1 of x and the component C_k of y. Let $z_2, \ldots, z_k$ be their acessors. Then all values from their domains that were not temporarily removed by the AC phase can be* definitively *removed by BCC.*

Sketch of proof: Consider the accessor z_k of the component containing y. If AC propagation has emptied the domain of

y, consider the filtered domain D'_k of z_k. This failure means that, should we extend our current instantiation of all variables until x with any assignment of $\delta \in D'_k$ to z_k, this instantiation does not extend to a solution. Thanks to Th. 1, no instantiation of z_k with the value δ extends to a solution. □

Finally, having proven that BCC and MAC combine their effort to produce a good result, we point out that MAC alone cannot replace BCC's efficiency: perhaps MAC will never consider a value that had to be remove by BCC, but should it consider it, nothing will restrain MAC to fail again on it.

7 Conclusion and Future Works

We have presented here a variation on BT called BCC, whose efficiency relies on the number of bicomponents of the constraint graphs. Early tests have shown that, the number of components increasing, BCC alone quickly compensates its inefficiency inside a component and its results are more than a match for other BT improvements. Moreover, we have shown that BCC itself could be added some well-known BT improvements, such as BJ, FC or MAC, effectively combining the best of two worlds. So an improved BCC could be the perfect candidate for networks whose constraint graph contains many bicomponents.

However, we must confess that such constraint graphs are not so common... So, apart from an original theoretical result expressing that a network whose constraint graph is a tree does not need an AC phase nor a backtrack-free search to be solved in $\mathcal{O}(d^2n)$, what can BCC be used for?

Let $\mathcal{R}$ be a binary constraint network of size n, and x_i and x_j be two of its variables. Then we can transform $\mathcal{R}$ into an equivalent network of $n-1$ variables, by fusioning x_i and x_j into a single variable x_{ij}. The domain of x_{ij} is composed of the pairs of compatible values in the constraint R_{ij} (it is $D_i \times D_j$ when there is no such constraint). Any constraint R_{ki} is then transformed in the following way: if $(\delta_k, \delta_i) \in R_{ki}$, then all pairs $(\delta_k, (\delta_i, \delta))$ such that (δ_i, δ) belongs to the domain of x_{ij} are possible values for the constraint between x_k and x_{ij}. The same construction is performed for constraints incident to x_j. The obtained network is equivalent, but has now $n-1$ variables, and its maximum domain can have now size d^2 (incident constraints can have size d^3). We can iteratively fusion k different nodes, obtaining a network of size $n-k+1$, but where a variable domain can have size d^k.

If a constraint graph admits a k-separator (a set of k nodes whose removal disconnects the graph), then, by fusioning these k variables as indicated above, we create at least two bicomponents sharing the obtained variable. Suppose that we can find in some way separators of size $\leq$ k, the fusion of these separators creating p bicomponents of size $\leq q$. Then BCC (without any improvement), will run in time $\mathcal{O}(d^{kq}p)$. This decomposition may be quite efficient when k is small and the separators cut the graph in a non degenerate way, keeping the size of components comparable. This decomposition method (where polynomial cases are obtained when k and q are both bounded by a constant) is still to be compared to the ones studied in [Gottlob *et al.*, 1999].

To implement and test this decomposition method, we are currently looking for an heuristic that, given an undirected simple graph with weighted edges, find a "small" separator of the graph such that:

(1) removal of the separator creates a graph whose greatest connected component is as small as possible;

(2) product of the weight on edges that belong to the separator is as small as possible.

The weight on edges will be initialized by the constraint tightness. We believe that networks designed by a human being, who decomposes a problem into subproblems slightly related to each other, will be good candidates for this heuristic.

Acknowledgements

We would like to thanks Christian Bessière for his bibliographical suggestions, Michel Chein and Michel Habib who helped making the concept of "semi-independant subgraphs" evolve into the much more formal one of k-connected components, as well as the anonymous referees, for their precious comments and advices.

References

[Dechter and Frost, 1999] R. Dechter and D. Frost. Backtracking Algorithms for Constraint Satisfaction Problems. ICS technical report, University of California, 1999.

[Dechter and Pearl, 1988] R. Dechter and J. Pearl. Network-Based Heuristics for Constraint Satisfaction Problems. *Artficial Intelligence*, 34(1):1–38, 1988.

[Dechter, 1990] R. Dechter. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence*, 41(3):273–312, January 1990.

[Freuder, 1982] E. C. Freuder. A Sufficient Condition for Backtrack-Free Search. *Journal of the ACM*, 29(1):24–32, January 1982.

[Gaschnig, 1979] J. Gaschnig. Performance measurement and analysis of certain search algorithms. Research report CMU–CS–79–124, Carnegie-Mellon University, 1979.

[Gottlob *et al.*, 1999] G. Gottlob, N. Leone, and F. Scarcello. A Comparison of Structural CSP Decomposition Methods. In *Proc. of IJCAI'99*, pages 394–399, 1999.

[Mackworth and Freuder, 1985] A.K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25, 1985.

[Prosser, 1993] P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9(3):268–299, 1993.

[Sabin and Freuder, 1994] D. Sabin and E. C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proc. of ECAI'94*, pages 125–129, 1994.

[Smith, 1996] B. Smith. Locating the Phase Transition in Binary Constraint Satisfaction Problems. *Artificial Intelligence*, 81:155–181, 1996.

[Tarjan, 1972] R. E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. of Computing*, 1:146–160, 1972.