Maintien de la Consistance d'Arc dans Dynamic Backtracking

Narendra Jussien, Romuald Debruyne et Patrice Boizumault
École des Mines de Nantes
4 rue Alfred Kastler – BP 20722
F-44307 Nantes Cedex 3
email: {jussien,rdebruyn,boizu}@emn.fr

Résumé

La plupart des algorithmes de recherche complets pour les Problèmes de Satisfaction de Contraintes (CSP) sont basés sur l'algorithme *Backtrack Standard*. Deux améliorations majeures de cet algorithme de base ont été proposées : premièrement, intégrer un mécanisme de propagation des contraintes comme dans l'algorithme `mac` qui maintient la consistance d'arc durant la recherche ; deuxièmement, procéder à des retours arrières intelligents afin d'éviter de tomber plusieurs fois dans les mêmes impasses en mémorisant des nogoods comme dans *Conflict-directed BackJumping* (`cbj`) ou dans *Dynamic Backtracking* (`dbt`). Des intégrations de mécanismes de propagation des contraintes dans des algorithmes de retours arrières intelligents ont déjà été proposées comme `mac-cbj` qui maintient la consistance d'arc dans `cbj`. Cependant, Bessière et Régin ont montré que `mac-cbj` était très rarement meilleur que `mac`. Mais nous pensons que les mauvais résultats de `mac-cbj` sont plus liés au fait que `cbj` ne parvient pas à éviter le thrashing[1] qu'au coût de la gestion des nogoods.

Cet article décrit et étudie `mac-dbt` qui maintient la consistance d'arc dans `dbt`. Des expérimentations montrent que `mac-dbt` est capable de résoudre de très grands problèmes et qu'il demeure très stable quand la taille des problèmes augmente. De plus, `mac-dbt` a de meilleurs résultats que `mac` sur les problèmes aléatoires structurés que nous avons générés.

1 Introduction

La plupart des algorithmes de recherche complets pour les Problèmes de Satisfaction de Contraintes (CSP) sont basés sur l'algorithme *Backtrack Standard* (`sb`): une recherche en profondeur d'abord réalisant des retours arrières chronologiques. Différents

1. On dit qu'il y a trhashing quand on refait a de nombreuses reprises le même travail d'instanciation à cause du principe du backtrack qui ne revient pas nécessairement sur les causes de l'échec.

algorithmes de retours arrières intelligents ont été proposés : *Conflict-directed BackJumping* (`cbj`) [Pro95], *Dynamic Backtracking* (`dbt`) [Gin93], *Partial order Dynamic Backtracking* (`pdb`) [GM94], *Generalized Dynamic Backtracking* (`gpb`) [Bli98], etc. Dans ces algorithmes, de l'information (plus précisément des nogoods) est mémorisée lorsque des inconsistances sont détectées afin de ne pas réexplorer les mêmes impasses de l'espace de recherche durant le reste du calcul.

Des mécanismes de propagation des contraintes ont été intégrés dans `sb` donnant jour à forward checking `fc` et plus récemment à l'algorithme de *Maintien de la Consistance d'Arc pendant la recherche* (`mac`) [SF94]. `mac` est considéré jusqu'à aujourd'hui comme l'un des meilleurs algorithmes pour la résolution des CSP [BR96].

Diverses tentatives d'intégration de mécanismes de propagation des contraintes dans les algorithmes de retours arrières intelligents ont été faites : par exemple, Prosser a proposé `mac-cbj` qui maintien la consistance d'arc dans `cbj` [Pro95]. Toutefois, Bessière et Régin [BR96] ont fait cesser toute recherche dans cette voie en montrant que `mac-cbj` était très rarement meilleur que `mac`. Leur conclusion était qu'il n'est pas rentable d'utiliser du temps et de l'espace pour réaliser des retours arrières intelligents parce que la puissance brute de `mac` permet d'effectuer la recherche plus rapidement.

Selon nous, les mauvaises performances de `mac-cbj` sont plus liées au fait que `cbj` ne parvient pas à éviter le thrashing qu'au coût de la gestion des nogoods. Quand un retour arrière doit être opéré, `cbj` revient à l'instanciation partielle pertinente en oubliant l'exploration de l'espace de recherche réalisée depuis cette affectation: comme `sb`, `cbj` a un comportement multiplicatif sur des sous-problèmes indépendants. `dbt` quant à lui utilise des nogoods non seulement pour réaliser des retours arrières intelligents mais aussi pour devenir additif sur des sous-problèmes indépendants [Gin93].

Dans [BR96] est présenté un autre point poussant à penser qu'il faut limiter l'utilisation des nogoods : il est toujours possible de trouver une bonne heuristique sur l'ordre d'instanciation telle qu'un algorithme basé sur le Backtrack Standard soit aussi efficace qu'un algorithme de retours arrières intelligents. D'après nos études, l'utilisation d'une heuristique sophistiquée permet de réduire le nombre de problèmes sur lesquels l'algorithme fera du thrashing, mais ne permet pas de rendre l'algorithme additif sur des sous problèmes indépendants : il existe toujours des problèmes sur lesquels l'heuristique ne peut pas éviter le thrashing.

Bien que de nombreux travaux ont porté sur `dbt`, rien à notre connaissance n'a été publié sur le maintien de la consistance d'arc dans `dbt`. Pas même une association entre *Forward Checking* et *Dynamic Backtracking* (`fc+dbt`) n'a encore été complètement décrite [VS94].

L'objectif de cet article est de décrire et d'étudier `mac-dbt` qui maintient la consistance d'arc dans `dbt`. Nous commençons par rappeller les principes de `dbt` avant de décrire comment intégrer la propagation de contraintes dans ce dernier. Nous présentons ensuite des expérimentations dans lesquelles `mac-dbt` est comparé à `dbt`, `fc-dbt` et `mac`. Ces expérimentations montrent que `mac-dbt` est capable de résoudre de très grand problèmes et qu'il reste très stable quand la taille des problèmes augmente. De plus `mac-dbt` a de meilleurs résultats que `mac` sur les problèmes aléatoires structurés que nous avons générés.

2 Améliorations de l'algorithme Backtrack Standard

Pour améliorer la rapidité de la résolution, les algorithmes de retours arrières intelligents mémorisent pour chaque impasse un **nogood**, c'est-à-dire un sous-ensemble de l'instanciation courante responsable de l'impasse. Mémoriser cette information permet d'éviter de retomber à nouveau dans les mêmes impasses. *Dependency Directed Backtracking* (`ddb`) [SS77] fut le premier algorithme à utiliser ce type d'amélioration, mais il a hélas un grave inconvénient : sa complexité spatiale est exponentielle parce que le nombre de nogoods qu'il mémorise augmente de manière monotone.

Pour éviter ce problème, des algorithmes comme `cbj` et `dbt` suppriment les nogoods qui ne sont plus valides par rapport à l'instanciation courante. En agissant ainsi, la complexité spatiale demeure polynomiale.

2.1 Nogoods et explications de retraits

Considérons un CSP (V, D, C). Un **nogood** est une instanciation partielle globalement inconsistante de valeurs a_i à des variables v_i (Aucune solution ne peut contenir un nogood) [2]:

$$C \vdash \neg\,(v_1 = a_1 \wedge \ldots \wedge v_k = a_k) \tag{1}$$

Pour chaque nogood, une variable v_j peut être choisie et la précédente formule réécrite de la façon suivante :

$$C \vdash \bigwedge_{i \in [1..k] \setminus j} (v_i = a_i) \rightarrow v_j \neq a_j \tag{2}$$

La partie gauche de l'implication constitue une **explication** du retrait de la valeur a_j du domaine de la variable v_j (notée $\texttt{expl}(v_j \neq a_j)$).

Quand le domaine d'une variable v_j devient vide durant le filtrage, un nouveau nogood est déduit à partir des explications de ses valeurs supprimées :

$$C \vdash \neg \left(\bigwedge_{a \in d(v_j)} \texttt{expl}(v_j \neq a) \right) \tag{3}$$

Il y a généralement plusieurs explications possibles pour le retrait d'une valeur. On pourrait vouloir mémoriser l'ensemble de toutes ces explications comme dans `ddb` mais comme nous l'avons vu cela conduirait à une complexité spatiale exponentielle.

Une autre approche consiste à *oublier* (supprimer) les nogoods qui ne sont plus valides [3] par rapport à l'instanciation courante des variables. En agissant ainsi, la complexité spatiale reste polynomiale. `cbj`, `dbt` (et ses extensions `pdb`, `gpb`) ne mémorisent par conséquent qu'**une** seule explication pour le retrait d'une valeur. Dans le pire des cas, l'espace requis pour la gestion des nogoods est en $O(n^2 d)$ où n est le nombre de variables et d la taille maximum des domaines du CSP. En effet, la taille de chaque explication est dans le pire des cas $(n-1)$ et il y a au plus $n \times d$ explications : une pour chaque valeur de chaque domaine.

2. Le *nogood* est une conséquence logique de l'ensemble des contraintes C.

3. Un nogood est dit valide si toutes les affectations qui le compose sont encore valides dans l'état courant de la recherche [BM96].

2.2 Du Backtrack Standard à Dynamic Backtracking

Quand un échec apparaît, `sb`, `cbj` et `dbt` doivent identifier l'affectation qui doit être remise en cause (suspectée d'être coupable de l'échec).

`sb` considère toujours l'affectation la plus récente comme étant la coupable. Ce choix peut-être totalement sans effet sur l'échec courant, conduisant à l'exploration de parties de l'arbre de recherche dont on sait déjà qu'elles sont des impasses.

Dans `cbj` un ensemble de conflits est associé à chaque variable : CS_{v_i} (pour la variable v_i) contient l'ensemble des variables instanciées dont la valeur est en conflit avec la valeur de v_i. Quand une impasse est identifiée durant l'affectation de v_i, `cbj` considère la plus récente des variables dans CS_{v_i} comme étant un coupable. Mais contrairement à `dbt`, avec `cbj` un retour arrière a lieu : on redonne aux ensembles de conflits et aux domaines des variables futures les valeurs qu'ils avaient à ce point de saut. Ainsi, `cbj` oublie une très grande quantité d'information qui aurait pu être utile, ce qui conduit à du thrashing.

`dbt`, tout comme `cbj`, choisit la variable la plus récente dans le nogood (l'ensemble de conflits de `cbj`) afin de défaire l'affectation. Cependant, grâce aux explications, `dbt` ne supprime que l'information dépendant de cette affectation et évite ainsi le thrashing : l'information utile est conservée. En effet, il n'y a pas de véritable retour arrière dans `dbt` et comme dans une méthode de réparation, seules les affectations responsables de la contradiction sont défaites. Par conséquent, `sb` et `cbj` ont tous les deux un comportement multiplicatif vis-à-vis de sous problèmes indépendants alors que `dbt` est additif.

Notons que `sb` peut également être considéré comme sélectionnant l'affectation la plus récente d'un nogood, mais le nogood en question est alors l'ensemble des affectations de l'instanciation courante (dont on ne peut retirer d'information vraiment judicieuse).

2.3 Dynamic Backtracking

Dans ce qui suit, le domaine courant d'une variable correspond à l'ensemble de ses valeurs qui ont une explication de retrait vide ($a \in d(v_i) \iff expl(v_i \neq a) = \emptyset$).

La fonction `dbt` (fig. 1) réalise la boucle principale qui essaye d'affecter des valeurs aux variables tant qu'une solution n'a pas été trouvée. V désigne l'ensemble des variables non instanciées et $\bar{V}$ l'ensemble des variables à qui on a déjà affecté une valeur. La fonction `affecterValeurEtTester(`$\bar{V}$`,`v_i`,`a`)` détermine si la nouvelle instanciation (incluant la nouvelle affectation $v_i = a$) est consistante ; si elle ne l'est pas, cette fonction retourne un nogood expliquant l'échec. Afin de restaurer un état cohérent, la fonction `dbt-traiterContradiction` *saute* vers une autre instanciation partielle consistante. Les domaines et les nogoods sont mis à jour grâce aux explications (voir la fonction `miseAJourDomaines` en figure 5).

`testerContraintesEnArriere` (fig. 3) teste *a posteriori* si les contraintes sont vérifiées pour la nouvelle instanciation partielle courante. Si elles ne le sont pas, cette fonction retourne une contrainte violée par l'instanciation courante. A partir de cette contrainte, `affecterValeurEtTester` (fig. 2) génère un nogood (line 9) qui ne contient que les affectations impliquées dans l'échec.

La fonction `dbt-traiterContradiction` (fig. 4) est un mécanisme de traitement de contradictions. L'affectation qui doit être remise en cause est déterminée à la ligne 5 de la figure 4 et un *retour arrière* (ou plus exactement un *saut*) est effectué en supprimant les nogoods qui ne sont plus valides, ce qui est fait par la fonction `miseAJourDomaines` (fig. 5).

```
dbt(csp)
début
    V̄ ← ∅
    tant que V ≠ ∅ faire
        choisir vᵢ ∈ V
        choisir a ∈ d(vᵢ)
        E ← affecterValeurEtTester(V̄,V,vᵢ,a)
        si E n'est pas un succès alors
            dbt-traiterContradiction(E,V̄,V)
        finsi
    fintantque
fin
```

FIG. 1 – *Fonction* ***dbt***

```
affecterValeurEtTester(V̄,V,vᵢ,a)
début
    affecter a à vᵢ
    V̄ ← V̄ ∪ {(vᵢ, a)}
    V ← V \ {vᵢ}
    c ← testerContraintesEnArriere(V̄)
    si c est un succès alors
        retourner succès
    sinon
        retourner {(vᵢ, a)} ∪ {(vₖ, b) | vₖ ∈ vars(c)}
    finsi
fin
```

FIG. 2 – *Fonction* ***affecterValeurEtTester***

```
testerContraintesEnArriere(V̄)
début
    pour tout c ∈ C et vars(c) ⊂ V̄ faire
        si c n'est pas vérifiée alors
            retourner c
        finsi
    finpour
    retourner succès
fin
```

FIG. 3 – *Fonction* ***testerContraintesEnArriere***

En fait, `dbt` ne réalise pas vraiment de retours arrières. Quand un échec se produit, on ne remet en cause que la plus récente des affectations qui ont causé la contradiction. En particulier, toutes les affectations qui n'interviennent pas dans la contradiction demeurent inchangées. C'est la raison pour laquelle `dbt` a un comportement additif sur des sous-problèmes indépendants.

```
dbt-traiterContradiction(E, V̄, V)
début
  si E = ∅ alors
    échec
  sinon
    (v_j, b) ← plusRécenteResponsable(E)
    miseAJourDomaines({(v_k, c) | (v_j, b) ∈ expl(v_k ≠ c)})
    expl(v_j ≠ b) ← E \ {(v_j, b)} /* d(v_j) ← d(v_j) \ {b} */
    V̄ ← V̄ \ {(v_j, b)}
    V ← V ∪ {v_j}
    si d(v_j) = ∅ alors
      dbt-traiterContradiction(⋃_{a∈d(v_j)} expl(v_j ≠ a), V̄, V)
    finsi
  finsi
fin
```

FIG. 4 – *Fonction* ***dbt-traiterContradiction***

```
miseAJourDomaines(back)
début
  pour tout (v_i, a) ∈ back faire
    expl(v_i ≠ a) ← ∅ /* d(v_i) ← d(v_i) ∪ {a} */
  finpour
fin
```

FIG. 5 – *Fonction* ***miseAJourDomaines***

3 Intégration de la propagation de contraintes

L'intégration d'un mécanisme de propagation des contraintes dans `dbt` est plus difficile qu'une intégration dans forward checking [SV94].

Premièrement, quand un échec apparaît, on ne peut plus générer des nogoods comme précédemment (les affectations qui interviennent dans la violation de la contrainte). Les effets de la propagation (des suppressions de valeurs) doivent être pris en compte : il est nécessaire de conserver des explications pour les suppressions effectuées par l'algorithme de filtrage.

Deuxièmement, une autre difficulté survient quand on désaffecte une variable. Restaurer dans les domaines les valeurs qui n'ont plus d'explication valide de leur retrait n'est pas suffisant puisqu'il peut exister d'autres explications valides pour ces valeurs supprimées. En effet, il peut exister plusieurs raisons justifiant le retrait d'une valeur et puisque nous n'en mémorisons qu'une, chaque restauration doit être confirmée par l'algorithme de propagation. Cette phase de confirmation des restaurations peut être comparée à celle pratiquée dans les algorithmes de maintien de la consistance d'arc dans les CSP dynamiques [Bes91].

```
mac-dbt(csp)
début
    V̄ ← ∅
    tant que V ≠ ∅ faire
        choisir vᵢ ∈ V
        choisir a ∈ d(vᵢ)
        E ← affecterValeurEtTesterAC(V̄ ,V ,vᵢ ,a)
        si E est un succès alors
            V̄ ← V̄ ∪ {(vᵢ, a)}
            V ← V \ {vᵢ}
        sinon
            mac-dbt-traiterContradiction(E)
        finsi
    fintantque
fin
```

FIG. 6 – *Fonction* ***mac-dbt***

```
mac-dbt-traiterContradiction(E)
début
    si E = ∅ alors
        échec
    sinon
        (vⱼ, b) ← plusRécenteResponsable(E)
        V̄ ← V̄ \ {(vⱼ, b)}
        V ← V ∪ {vⱼ}
        E′ ← miseAJourDomainesAC({(vₖ, c) | (vⱼ, b) ∈ expl(vₖ ≠ c)})
        si E′ n'est pas un succès alors
            mac-dbt-traiterContradiction(E′)
        finsi
        si E \ (vⱼ, b) est encore valide alors
            expl(vⱼ ≠ b) ← E \ {(vⱼ, b)}
            E′ ← supprimerValeurEtTesterAC(vⱼ ,b)
            si E′ n'est pas un succès
                mac-dbt-traiterContradiction(E′)
            finsi
        finsi
    finsi
fin
```

FIG. 7 – *Fonction* ***mac-dbt-traiterContradiction***

3.1 Implémentation de `mac-dbt`

Nous allons tout d'abord proposer nos solutions aux deux problèmes que nous venons d'évoquer, puis nous détaillerons l'implémentation avant d'en discuter la complexité.

3.1.1 Génération des nogoods durant le filtrage

La boucle principale de l'algorithme demeure inchangée (voir fonction `mac-dbt` de la figure 6 comparée à la fonction `dbt` de la figure 1) si ce n'est un appel à la fonction `affecterValeurEtTester` qui est remplacé par un appel à la fonction `affecterValeurEtTesterAC` décrite en figure 11. Cette fonction affecte la valeur a à la variable v_i en supprimant toutes les autres valeurs de son domaine et en propageant ces retraits au moyen de la fonction `propagSuppress` (voir fig. 8).

Tant qu'il existe des retraits de valeurs à propager dans la file de propagation, `propagSuppress` retire un élément de cette file afin de le propager sur la contrainte concernée. L'idée importante est qu'il faut expliquer (donner une explication) chaque re-

```
propagSuppress(Q)
début
    tant que Q ≠ ∅ faire
        (v_i, a) ← retirerFile(Q)
        pour tout c_ij faire
            E ← explicationAC(c_ij, v_i, a, Q)
            si E n'est pas un succès alors
                retourner E
            finsi
        finpour
    fintantque
    retourner succès
fin
```

FIG. 8 – *Fonction* ***propagSuppress***

```
explicationAC(c_ij, v_i, a, Q)
début
    pour tout b ∈ d(v_j) support faire
        nbSupports(b)--
        si nbSupports(b) = 0 alors
            expl(v_j ≠ b) ← ⋃_{a' ∈ supports(b)} expl(v_i ≠ a') + c_ij
            ajouterFile(v_j, b, Q)
            si d(v_j) = ∅ alors
                retourner ⋃_{a' ∈ d(v_j)} expl(v_j ≠ a')
            finsi
        finsi
    finpour
fin
```

FIG. 9 – *Fonction* ***explicationAC***

```
supprimerValeurEtTesterAC(v_j, b)
début
    initialiserFile(Q)
    ajouterFile(v_j, b, Q)
    retourner propagSuppress(Q)
fin
```

FIG. 10 – *Fonction* ***supprimerValeurEtTesterAC***

```
affecterValeurEtTesterAC(v_i, a)
début
    initialiserFile(Q)
    pour tout b ∈ d(v_i), b ≠ a faire
        ajouterFile(v_i, b, Q)
    finpour
    retourner propagSuppress(Q)
fin
```

FIG. 11 – *Fonction* ***affecterValeurEtTesterAC***

```
miseAJourDomainesAC(back)
(1)  début
(2)     pour tout (v_i, a) ∈ back faire
(3)        expl(v_i ≠ a) ← ∅
(4)     finpour
(5)     initialiserFile(Q)
(6)     pour tout (v_i, a) ∈ back faire
(7)        si il existe c_ij sur laquelle (v_i, a) n'a pas de support alors
(8)           ajouterFile(v_i ,a,Q)
(9)        finsi
(10)    finpour
(11)    retourner propagSuppress(Q)
(12) fin
```

FIG. 12 – *Fonction* `miseAJourDomainesAC`

trait de valeur opéré durant la propagation d'une contrainte. La fonction `explicationAC` (fig. 9) montre comment réaliser cela pour un mécanisme de propagation des contraintes basé sur `ac4`. Quand le nombre de supports d'une valeur devient nul, cette valeur doit être supprimée. Une explication pour cette suppression peut être obtenue à partir des explications des retraits de tous ses supports (ligne 5 de l'algorithme).

3.1.2 Désaffecter sans réel retour arrière

La fonction `miseAJourDomaines` de la figure 5 doit être modifiée pour obtenir un état arc-consistant après la désaffectation d'une variable. Ce qui conduit à la fonction `miseAJourDomainesAC` de la figure 12.

Les modifications (opérées à partir de la fonction `miseAJourDomaines`) commencent à la ligne 5 où la consistance d'arc des valeurs restaurées est vérifiée afin d'obtenir un état arc consistant. Si une valeur restaurée ne vérifie pas la consistance d'arc, elle est supprimée et son retrait est propagé par la fonction `propagSuppress`.

3.1.3 Traitement des nouvelles contradictions

Tout d'abord, désaffecter une variable comme on le fait aux lignes 5–8 (fig. 7) peut ne pas être aussi direct que dans l'algorithme d'origine puisque restaurer peut ne pas être suffisant pour obtenir un état consistant à cause du mécanisme de propagation des contraintes. C'est pourquoi on peut être amené à traiter une nouvelle contradiction : d'où l'appel récursif à `mac-dbt+traiterContradiction` à la ligne 10.

De plus, le traitement de contradictions peut conduire à la désaffectation de variables, rendant invalide l'explication du retrait de la valeur désaffectée à la plus récente variable impliquée dans la contradiction. D'où le test de la ligne 12 avant de supprimer la valeur car sinon on ne pourrait plus garantir la complétude de l'algorithme car une valeur pourrait rester avec une explication non valide.

3.1.4 Complexités

Comme mentionné précédemment, l'espace total requis pour les nogoods est dans le pire des cas en $O(n^2d)$. L'utilisation d'algorithmes de consistances locales permet d'obtenir des explications plus courtes parce que les affectations impliquées dans les explications sont plus judicieuses que dans le cas de `dbt`. En pratique, l'espace requis pour mémoriser les explications est moins important que l'espace occupé par la représentation du problème ($O(ed^2)$). De plus, `mac-dbt` n'utilise pas de mémorisation des états précédents sur la pile des appels, ce qui représente un gain important par rapport aux approches classiques. Enfin, nous n'avons jamais été gênés dans nos expérimentations pas un quelconque problème de mémoire, même sur de très grandes instances.

Du point de vue de la complexité temporelle, la gestion des explications demande bien évidement un peu de temps cpu supplémentaire comparé à `mac` mais cela ne change en rien la complexité dans le pire des cas. Le surcoût temporel est modeste et les explications permettent d'éviter du thrashing tout en fournissant des explications pour les échecs.

4 Discussion

4.1 Consistance d'arc dans les CSP dynamiques

Il existe des points communs entre `mac-dbt` et les techniques de maintien de la consistance d'arc dans les CSP dynamiques [Bes91, Deb96] puisqu'un système de maintien des déductions est là aussi utilisé. Cependant, il existe de réelles différences entre le système de **justifications** utilisé par DnAC-4 (et DnAC-6) et les **explications** utilisées par `mac-dbt`. De plus, le but des algorithmes DnAC-* est de maintenir la consistance d'arc dans les CSP dynamiques et non pas de résoudre des CSP statiques. Même si nous considérons une affectation comme une contrainte unaire restreignant un domaine, les idées de DnAC-* ne peuvent pas être utilisées pour résoudre un CSP statique sans un travail colossal (dans le cadre de la recherche d'une solution, les retraits de valeurs ne peuvent pas être justifiés par une seule contrainte mais par une conjonction de contraintes contrairement au cas du maintien de la consistance d'arc dans les CSP dynamiques. De plus, les algorithmes DnAC-* ne peuvent être utilisés dans ce but sans préciser comment construire la succession d'ajouts/retraits de contraintes à réaliser pour résoudre le problème, comment déduire l'ensemble des contraintes/affectations responsables d'un échec, spécifier qu'elle affectation doit être remise en cause quand un échec survient, etc.).

4.2 Extensions

L'idée forte de `mac-dbt` est l'association d'une explication à chaque retrait d'une valeur. Cette idée peut être étendue à des ensembles de valeurs : une première étude de l'utilisation des explications pour des intervalles à été menée pour les CSP numériques [JL98].

Tout mécanisme de propagation des contraintes capable de fournir des explications peut être intégré dans `dbt` de la même manière que nous l'avons fait pour la consistance d'arc. Cela pourrait être appliqué à des consistances *fortes* pour CSP binaires (voir [DB97, Deb98] pour un état de l'art) mais aussi pour des contraintes non binaires.

`mac-dbt` peut également être utilisé sur différents types de problèmes, par exemple pour améliorer sensiblement la résolution des problèmes de scheduling [JG99].

5 Experimentations

Nos expérimentations ont été réalisées avec une implémentation de `mac-dbt` dans laquelle la propagation des contraintes est faite à la `ac6`. `dbt` et `fc-dbt` ont été implé-

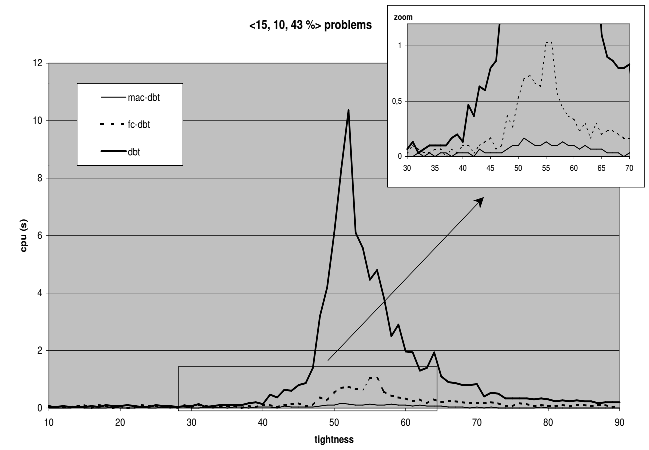

FIG. 13 – *Résultats de l'étude comparative de* **dbt**, **fc-dbt** *et* **mac-dbt**

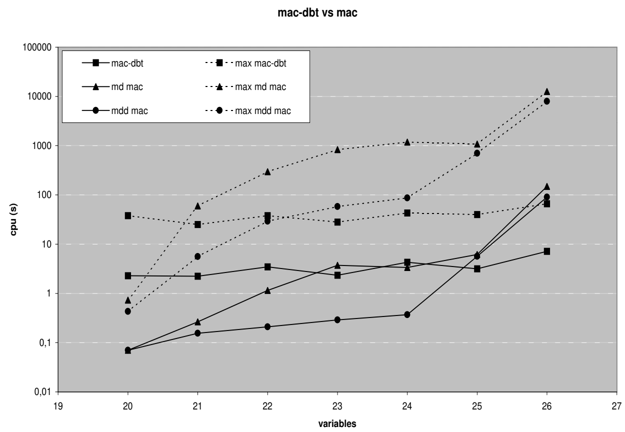

FIG. 14 – *Résultats de l'étude comparative de* **mac-dbt** *et* **mac**

mentés en utilisant les explications de retraits développées pour `mac-dbt` afin d'obtenir des versions comparables de ces algorithmes.

5.1 `dbt` and `fc-dbt` versus `mac-dbt`

Dans notre premier ensemble d'expérimentations, nous comparons les trois algorithmes basés sur *dynamic backtracking* : `dbt`, `fc-dbt` et `mac-dbt`.

La figure 13 présente les résultats obtenus sur des problèmes aléatoires [4] de 15 variables ayant chacune un domaine de 10 valeurs et 45 contraintes (densité de 43%). Les résultats présentés portent sur des problèmes ayant une tightness comprise entre 10% et 90%.

La figure montre que plus la consistance locale utilisée dans `dbt` est forte, plus la recherche est rapide. Sur ces problèmes, l'intérêt de `mac-dbt` est évident.

5.2 `mac-dbt` comparé à `mac`

Nous comparons à présent `mac-dbt` à la version `mac7ps` [Rég95] de `mac`. Nous allons dans le sens des expérimentations de [BS96] en ce sens que les problèmes générés ont une structure [5] : la transition de phase est utilisée pour générer des problèmes sous-contraints qui contiennent un sous-problème difficile (choisis près de la transition de phase) caché en leur sein. Cette structure ne peut pas être identifiée en utilisant un prétraitement et la structure n'apparaît que durant la recherche.

Nos premières expérimentations ont été faites sur de grandes instances : une série de problèmes de 200 variables ayant chacune 10 valeurs dans leur domaine respectif et comportant 2500 contraintes. Ces problèmes renferment un sous-problème difficile de 15 variables comportant 43 contraintes. Bien que `mac-dbt` résoud chacune de ces instances en quelques secondes, il apparaît que sur certains de ces problèmes `mac` n'a pas été en mesure de trouver une solution après plusieurs jours de calcul.

Nous avons par conséquent essayé de trouver des problèmes sur lesquels les performances de `mac` et `mac-dbt` sont plus proches. La figure 14 présente les résultats obtenus sur un type particulier de problèmes : les CSP structurés que nous avons générés comporte de 20 à 26 variables, chacune ayant 15 valeurs dans son domaine. Chaque CSP contient deux sous-problèmes de 8 variables ayant une densité de 57% et une dureté de 76%. Des contraintes supplémentaires de dureté 5%ont été ajoutées afin de relier tout couple de variable qui ne sont pas dans le même sous problème. Bien que cette construction peut sembler artificielle, la plupart des problèmes réels comportent des structures, qui sont d'ailleurs bien souvent cachées.

La figure 14 présente les résultats en moyenne ainsi que le temps cpu maximum requis (à la fois pour `mac-dbt` et `mac`) sur les 500 problèmes générés en utilisant `min dom/deg` et `mindom` pour heuristiques sur l'ordre d'instanciation des variables (resp. `mdd mac` et `md mac` sur la figure 14). On peut remarquer que `mac-dbt` reste stable quand le nombre de variables augmente alors que `mac` devient d'un coût prohibitif. De plus, même si au début les résultats en moyenne pour `mac` avec `min dom/deg` pour heuristique

4. D. Frost, C. Bessiere, R. Dechter, and J.C. Regin. Random uniform CSP Generators, 1996, http://www.ics.uci.edu/~dfrost/csp/generator.html

5. Comme on peut l'imaginer, les résultats en moyenne sur des CSP aléatoires sans structure présentent un léger surcoût temporel pour `mac-dbt` à cause du système de maintien de déductions dont on ne tire pas parti sur ce type de problèmes.

sont meilleurs que ceux de `mac-dbt`, on peut noter que sur les problèmes comportant 26 variables, les résultats en moyenne de `mac` sont moins bons que ceux de `mac-dbt` sur les mêmes problèmes. Aucun résultat n'est donné sur des problèmes de plus grande taille parce que sur certaines instances générées comportant 27 variables `mac` demande plus de 4 heures avant de fournir une réponse.

6 Conclusion et futurs travaux

Cet article décrit l'intégration de la consistance d'arc dans `dbt` en proposant `mac-dbt`. `mac-dbt` est à `dbt` ce que `mac` est à `sb`. Cette intégration s'appuie principalement sur la mémorisation d'explications pour les suppressions de valeurs.

`mac-dbt` montre de très bons résultats comparé à `dbt` ou `fc-dbt` : la propagation des contraintes améliore sensiblement la rapidité. Des expérimentations ont montré que `mac-dbt` est capable de résoudre de très grand problèmes, qu'il reste stable quand la taille des problèmes augmente et qu'il a même de meilleures performances que `mac` sur des problèmes structurés même si dans certains cas `mac-dbt` peut explorer une part plus importante de l'espace de recherche qu'un algorithme basé sur Backtrack Standard [Bak94].

Nos travaux courants portent notamment sur l'adaptation de ces idées à d'autres consistances locales (comme celle utilisée dans `quick` [Deb98]). Nous pensons fermement que les explications qui sont à la base de `mac-dbt`, peuvent être d'une très grande utilité dans la communauté CSP. Nous avons présenté ici leur utilisation pour développer un nouvel algorithme de recherche. Une autre technique de recherche nouvelle utilisant des explications a été présentée dans [JL00] montrant de très bons résultats sur les problèmes de scheduling. Les explications pourraient également être d'une grande utilité dans d'autres domaines comme pour expliquer les inconsistances lors du débogage en programmation par contraintes ou pour permettre le retrait de contraintes et ainsi introduire l'interactivité lors de la résolution de problèmes surcontraints.

Références

[Bak94] Andrew B. Baker. The hazards of fancy backtracking. In *12th National Conf. on Artificial Intelligence, AAAI94*, pages 288–293, Seattle, WA, USA, 1994.

[Bes91] Christian Bessière. Arc consistency in dynamic constraint satisfaction problems. In *Proceedings AAAI'91*, 1991.

[Bli98] C. Bliek. Generalizing partial order and dynamic backtracking. In *Proceedings of AAAI*, 1998.

[BM96] Roberto J. Bayardo Jr. and Daniel P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *AAAI'96*, 1996.

[BR96] Christian Bessière and Jean-Charles Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problem. In *CP'96*, Cambridge, MA, 1996.

[BS96] Roberto Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve exceptionnaly hard SAT instances. In *CP'96*, 1996.

[DB97] Romuald Debruyne and Christian Bessière. From restricted path consistency to max-restricted path consistency. In *CP'97*, pages 312–326, Linz, Austria, October 1997.

[Deb96] Romuald Debruyne. Arc-consistency in dynamic CSPs is no more prohibitive. In 8^{th} *Conference on Tools with Artificial Intelligence (TAI'96)*, pages 299–306, Toulouse, France, 1996.

[Deb98] Romuald Debruyne. *Local consistencies for large CSPs.* PhD thesis, Université de Montpellier II, December18 1998. In French.

[Gin93] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[GM94] Matthew L. Ginsberg and David A McAllester. Gsat and dynamic backtracking. In *International Conference on the Principles of Knowledge Representation (KR94)*, pages 226–237, 1994.

[JG99] Narendra Jussien and Christelle Guéret. Improving branch and bound algorithms for open shop problems. In *Conference of the International Federation of Operational Research Societies (IFORS'99)*, Beijing, China, August 1999.

[JL98] Narendra Jussien and Olivier Lhomme. Dynamic domain splitting for numeric csp. In *European Conference on Artificial Intelligence*, pages 224–228, Brighton, United Kingdom, August 1998.

[JL00] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. In *Seventh National Conference on Artificial Intelligence – AAAI'00*, Austin, TX, USA, August 2000.

[Pro95] Patrick Prosser. MAC-CBJ: maintaining arc-consistency with conflict-directed backjumping. Research Report 95/177, Department of Computer Science – University of Strathclyde, 1995.

[Rég95] Jean-Charles Régin. *Développement d'outils algorithmiques pour l'Intelligence Artificielle. Application à la chimie organique.* Thèse de doctorat, Université de Montpellier II, 21 December 1995. In French.

[SF94] Daniel Sabin and Eugene Freuder. Contradicting conventional wisdom in constraint satisfaction. In Alan Borning, editor, *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science.* Springer, May 1994. (PPCP'94: Second International Workshop, Orcas Island, Seattle, USA).

[SS77] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[SV94] Thomas Schiex and Gérard Verfaillie. Nogood Recording fot Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.

[VS94] Gérard Verfaillie and Thomas Schiex. Dynamic backtracking for dynamic csps. In Thomas Schiex and Christian Bessière, editors, *Proceedings ECAI'94 Workshop on Constraint Satisfaction Issues raised by Practical Applications*, Amsterdam, August 1994.