Backtracking adaptatif

Olivier Lhomme
IBM France
om.lhomme@gmail.com ou olivier.lhomme@fr.ibm.com

Abstract

L'algorithme de backtracking diffère de la recherche en profondeur d'abord par l'utilisation, à chaque noeud, d'une procédure de filtrage capable, dans certains cas, de détecter l'absence de solution parmi les descendants du noeud. L'exploration du sous-arbre correspondant peut ainsi etre évitée. Pour un domaine d'application particulier, il existe en général différentes procédures de filtrage. Nous proposons dans ce papier de doter l'algorithme de backtracking de la capacité de choisir, à chaque noeud, entre plusieurs algorithmes de filtrage, et en démontrons l'intérêt.

1 Introduction

L'algorithme de backtracking [1] est une composante essentielle de nombreux solveurs de problèmes combinatoires. Son principe est d'étendre la solution partielle courante, et d'explorer les différentes extensions possibles par une recherche en profondeur d'abord. Un prédicat, dit de *pruning*, capable dans certains cas de détecter qu'aucune solution complète ne peut étendre la solution partielle courante, permet à l'algorithme de backtracking d'éviter l'exploration de sous-arbres inutiles.

La capacité d'un algorithme de pruning à détecter de tels culs-de-sac est évidemment très dépendante de son temps d'exécution, ce qui signifie que le choix d'un algorithme de pruning est toujours affaire de compromis entre la taille de l'arbre de recherche exploré par l'algorithme de backtracking et le temps passé à chaque noeud de l'arbre de recherche. Par ailleurs, même si un algorithme de pruning particulier offre le meilleur compromis pour un arbre de recherche, un autre algorithme de pruning peut offrir un bien meilleur compromis localement à un sous-arbre. Cette remarque suggère de sélectionner un algorithme de pruning potentiellement différent à chaque noeud de l'arbre de recherche. Nous avons developpé cette idée, et montré qu'un choix dynamique judicieux peut améliorer les performances du backtracking de plusieurs ordres de grandeurs.

Etant donné un ensemble de prédicats de pruning, nous pouvons considérer tous les algorithmes de backtracking qui appliquent à chaque noeud l'un quelconque des ces prédicats de pruning. Nous pouvons définir le temps de calcul minimum parmi tous ces algorithmes de backtracking pour résoudre un problème donné. Ce temps de calcul théoriquement minimum peut servir de référence pour comparer la performance des algorithmes. Par exemple, étant donnés deux prédicats de pruning, le second bénéficiant d'une plus grand capacité à détecter les culs-de-sac, mais étant k fois plus lent que le premier, nous voulons savoir quel gain un algorithme de backtracking optimal peut apporter comparé aux deux algorithmes classiques avec un prédicat de pruning fixé. Nous montrons que pour certains problèmes, ce gain peut être, *à la fois*, d'un facteur exponentiel par rapport au backtracking avec le pruning faible et d'un facteur k par rapport au backtracking avec le pruning fort.

Malheureusement, un algorithme de backtracking optimal semble être irréalisable en pratique : il nécessiterait de connaître à l'avance le prédicat de pruning à appliquer à chaque noeud. Dans ce papier nous montrons que l'on peut concevoir des algorithmes de backtracking dont le coût en temps de calcul est proche de celui, purement théorique, de l'algorithme de backtracking optimal. Nous introduisons le principe de *backward-activated pruning* : au lieu d'élaguer un arbre de la racine vers les feuilles, l'idée est d'effectuer l'élagage lors d'un backtrack, c'est-à-dire des feuilles vers la racine. Nous montrons que le coût de calcul du backtracking avec le backward-activated pruning est au plus à un facteur $3 + \log_2(d-1)$ de l'exploration optimale (où d est la profondeur de l'arbre de recherche). De plus nous introduisons une propriété d'incrémentalité *bottom-up* pour les algorithmes de pruning, qui rend le backtracking avec backward-activated pruning plus efficace, avec un coût au plus à un facteur 3 du coût optimal. Les expérimentations confirment que le principe de backward-activated pruning améliore fortement l'algorithme de backtracking.

Ce papier est organisé de la manière suivante : la section 2 introduit le backtracking optimal ; la section 3 présente le backtracking avec backward-activated pruning ; la section 4 présente une analyse théorique, dont les preuves et détails sont donnés en annexe ; la section 5 résume quelques expérimentations et la section 6 aborde les travaux connexes et les perspectives.

2 Backtracking et Exploration Optimale

2.1 Préliminaires

L'algorithme de backtracking (Figure 1) commence par appliquer le prédicat *prune(c)* sur la solution partielle courante c. Si le prédicat *prune(c)* est capable de prouver que c ne peut être étendue en une solution complète, il renvoie *true*, et, donc, il est inutile d'explorer les extensions de c. Autrement, Backtracking est appelé récursivement sur les extensions possibles de c, générées par la fonction *children(c)*.

```
procedure Backtracking(c)
  if prune(c) then return
  for s ∈ children(c) do
    Backtracking(s)
```

FIG. 1 – Backtracking

Commençons par quelques définitions :

- Nous utiliserons parfois le terme *noeud* pour une solution partielle (ou complète), par référence au noeud dans l'arbre de recherche généré par l'algorithme de backtracking.
- Nous associerons un *opérateur de pruning* P au prédicat *prune* : soit N un noeud, $P(N)$ est défini comme suit :

$$P(N) = \begin{cases} \emptyset \text{ si } \textit{prune}(N) \text{ est vrai} \\ N \text{ si } \textit{prune}(N) \text{ est faux} \end{cases}$$

- Un opérateur de pruning a un *coût*, C, qui typiquement représente le temps d'exécution de l'algorithme de pruning. Nous supposerons que ce coût est constant.
- Une propriété essentielle d'un opérateur de pruning P est la *soundness* : il ne peut pas couper une branche qui conduit à une solution.
- Nous devrons comparer les capacités de filtrage des opérateurs de pruning : étant donnés deux opérateurs de pruning P_1 et P_2, nous dirons que P_2 est *plus fort que* P_1 si, pour chaque noeud N nous avons $P_1(N) = \emptyset \Rightarrow P_2(N) = \emptyset$.
- Nous étendons la fonction *children* utilisée dans l'algorithme de backtracking au symbole $\emptyset$: $children(\emptyset) = \emptyset$; nous supposerons que les solutions complètes n'ont pas de noeud fils.
- Nous supposerons qu'un opérateur de pruning P est toujours *monotone* : pour tout noeud fils N' d'un noeud N tel que $P(N) = \emptyset$, nous avons $P(N') = \emptyset$.

2.2 Backtracking Optimal

Dans cette section, nous étudions l'utilisation de plusieurs prédicats de pruning ayant différentes puissances de filtrage et différents temps d'exécution.

Considérons deux opérateurs de pruning P_1 et P_2, et deux versions de l'algorithme de backtracking : BT_1 qui utilise l'opérateur de pruning P_1, et BT_2 qui utilise P_2. Supposons que P_2 est plus fort que P_1, mais aussi que le coût de P_1 est inférieur à celui de P_2 (autrement, P_2 domine toujours P_1, et il n'y aucune raison de considérer P_1). Pour simplifier, nous supposerons que $C_2 = kC_1$ où k est un nombre entier supérieur à 1. Le coût total de l'algorithme BT_i pour explorer l'arbre de racine N peut être défini récursivement : l'opérateur P_i est appliqué sur le noeud N, ce qui coûte C_i ; puis, la récurrence sur les descendants s'applique. Nous obtenons :

$$Cost_{BT_i}(N) = C_i + \sum_{s \in children(P_i(N))} Cost_{BT_i}(s)$$

Notons les deux cas terminaux de la récurrence : (1) lorsque le noeud N est filtré, nous avons $P_i(N) = \emptyset$, et, donc, $children(P_i(N)) = \emptyset$; (2) lorsque N is une solution, N ne peut pas être filtré et n'a pas de descendant, donc nous avons $children(P_i(N)) = children(N) = \emptyset$.

Maintenant, imaginons que l'algorithme de backtracking, au lieu de toujours appliquer le même opérateur de pruning, applique parfois P_1 et parfois P_2, et considérons tous les algorithmes de cette forme possibles. Pour un problème donné, nous voudrions être aussi proche que possible du coût d'exploration minimal parmi tous ces algorithmes de backtracking. Nous appelerons *backtracking optimal*, et noterons BT_{opt}, un algorithme de backtracking idéal, capable, à chaque noeud, de sélectionner P_1 ou P_2 de telle façon que le coût total d'exploration soit minimal. Ce coût total de l'algorithme BT_{opt} pour explorer l'arbre de racine N peut être défini récursivement :

$$Cost_{BT_{opt}}(N) =$$

$$min \begin{cases} C_1 + \sum_{s \in children(P_1(N))} Cost_{BT_{opt}}(s) \\ C_2 + \sum_{s \in children(P_2(N))} Cost_{BT_{opt}}(s) \end{cases}$$

Une conséquence directe de la définition de BT_{opt} est la suivante :

Proposition 2.1 *Pour tout arbre de racine N, pour tout algorithme de backtracking A qui applique opérateur P_1 à certains noeuds et l'opérateur P_2 aux autres noeuds, $Cost_{BT_{opt}}(N) \leq Cost_A(N)$.*

```
procedure BackwardPruning(c)
1     credit := credit + C1
2     if credit < C2
3        if prune1(c) then return disablePrune2
4     else
5        credit := credit - C2
6        if prune1(c) then return enablePrune2
7        if prune2(c) then return enablePrune2
8     for s ∈ children(c) do
9        status := BackwardPruning(s)
10       if status = enablePrune2
11          if prune2(c) then return enablePrune2
12    return disablePrune2
```

FIG. 2 – Backward pruning

De plus, pour BT_1 et BT_2, nous avons la proposition suivante :

Proposition 2.2

(i) BT_{opt} *peut gagner un facteur exponentiel comparé à* BT_1.

(ii) BT_{opt} *peut gagner un facteur* C_2/C_1 *comparé à* BT_2.

(iii) De plus, BT_{opt} *peut gagner à la fois (i) et (ii) sur la* même *instance de problème.*

La preuve de cette proposition est donnée en annexe.

La proposition 2.2 fait apparaître tout l'intérêt d'un backtracking optimal pour l'exploration d'un arbre. Malheureusement, il nécessite de connaître à l'avance le prédicat de pruning à appliquer à chaque noeud, ce qui ne semble pas être possible dans le cas général. Dans la section suivante, nous présentons des algorithmes s'approchant d'une exploration optimale.

3 Backward-Activated Pruning

Dans la section 3.1, nous introduisons un algorithme de backtracking appelé *backward pruning*, puis, dans la section 3.2, nous introduisons une version dichotomique du backward pruning qui améliore sa complexité.

3.1 Backward Pruning

La figure 2 présente un algorithme, *backward pruning*, qui utilise deux algorithmes de pruning différents, *prune*$_1$ faible puissance de filtrage et rapide, et *prune*$_2$ puissance de filtrage plus forte et plus coûteux. Le filtrage faible est classiquement appliqué à chaque ouverture d'un nouveau noeud, alors que le filtrage fort est appliqué au backtrack (*backward-activated*), lorsqu'il a été détecté comme étant utile dans un sous-arbre.

Dans la fonction BackwardPruning, la variable globale *credit*, initialisée à 0, contrôle l'exécution périodique de l'algorithme de filtrage fort : à chaque appel du prédicat *prune*$_1$, *credit* is incrémenté de C_1, et lorsque *credit* atteint C_2, le coût de l'appel du prédicat *prune*$_2$, *credit* est remis à 0 (rappelons que nous supposons que C_2/C_1 est entier), et *prune*$_2$ est exécuté. Remarquons que *prune*$_2$ est appelé soit directement à ce niveau, soit, dans le cas où *prune*$_1$*(c)* est vrai, après backtracking.

Lorsque le crédit n'est pas suffisant pour appliquer le filtrage fort (lignes 2–3), seul le filtrage faible est appliqué. Dans ce cas, si le noeud courant est détecté comme étant un cul-de-sac, un backtrack a lieu : la fonction retourne la valeur *disablePrune2*. Lorsque le crédit est suffisant pour appliquer le filtrage fort (lignes 4–7), le filtrage faible est appliqué en premier, suivi du filtrage fort. Si l'un de ces filtrages détecte que le noeud courant est un cul-de-sac, un backtrack a lieu, la fonction retournant la valeur *enablePrune2*. Dans ce cas, le filtrage fort est aussi appliqué sur le père de noeud lors du backtrack (lignes 10–11). Les lignes 8 à 11 développent le noeud et explorent ses descendants.

Cette implémentation relativement simple de l'idée d'appliquer le filtrage fort des feuilles vers la racine sera utile pour l'analyse théorique de la section 4. Une implémentation plus réaliste éviterait l'application du filtrage fort lorsqu'elle est inutile[1].

3.2 Backward Pruning Dichotomique

Pour améliorer la complexité de l'algorithme backward pruning, nous en présentons une version dichotomique (figure 3). L'algorithme backward pruning applique le filtrage fort en remontant d'une feuille vers la racine jusqu'à trouver le plus récent ancêtre qui ne peut être élagué. Pour trouver ce même ancêtre, la version dichotomique du backward pruning utilise la méthode *findLevel*. Remarquons que cette fonction *Divide&Conquer-Backward Pruning* stocke tous les ancêtres du noeud courant dans un tableau global *node* indexé par leur profondeur dans l'arbre de recherche (voir ligne 1). Ce tableau est utilisé par la méthode *findLevel*. Une variable globale *dknp* (pour "deepest known as non prunable") maintient le niveau i le plus profond, dans la pile des appels récursifs, qui est connu comme non élaguable (*prune*$_2$*(node[i])* est faux). La méthode dichotomique commence par ce niveau le plus profond au lieu de 0, ce qui nous donnera une meilleure borne pour la complexité. Lors du premier appel à *Divide&Conquer-BackwardPruning*, le paramètre *level* doit être égal à 0, et la variable *dknp* doit être initialisée à -1.

4 Analyse théorique

La section 4.1 présente les résultats de l'analyse théorique des algorithmes de backward pruning : nous mon-

[1] Par exemple, lorsque le filtrage fort a déjà été appliqué sur le noeud c et n'a pu l'élaguer, il est inutile de l'appliquer à nouveau. Ou encore, lorsque le dernier fils du noeud c est élagué par le filtrage fort, on peut éviter de l'appliquer sur le noeud c et simplement backtracker comme si le filtrage fort avait élagué c.

```
procedure Divide&Conquer-BackwardPruning(c,level)
    node[level] := c
    credit := credit + C1
    if credit < C2
        if prune1(c) then return level-1
    else
        credit := credit - C2
        if prune1(c) then return findLevel(dknp+1,level-1)
        if prune2(c) then return findLevel(dknp+1,level-1)
        else dknp := level
    for s ∈ children(c) do
        returnedLevel := BackwardPruning(s,level+1)
        if returnedLevel < level return returnedLevel
        else dknp := min(level, dknp)
    return level-1

procedure findLevel(min,max)
    if min > max then
        dknp := min-1
        return dknp
    mid := floor((min+max)/2)
    if prune2(node[mid]) then return findLevel(min,mid-1)
    else return findLevel(mid+1,max)
```

FIG. 3 – Divide & Conquer Backward Pruning

trons que le coût du backward pruning est au plus $2 + \alpha$ fois le coût d'une exploration optimale, où α est le minimum entre la profondeur de l'arbre de recherche et le rapport C_2/C_1. Ensuite, nous en dérivons deux résultats importants en pratique :

1. Nous définissons une propriété d'incrémentalité *bottom-up* pour un algorithme de pruning, et nous montrons que pour un algorithme de pruning fort qui est incrémental, le coût du backward pruning est au plus à un facteur 3 de celui d'une exploration optimale.
2. Lorsque l'algorithme de pruning fort n'est pas incrémental, nous montrons que la version dichotomnique du backward pruning est nettement plus efficace que la version standard : son coût est au plus à un facteur $3 + \log_2(\alpha - 1)$ de celui d'une exploration optimale, où α est défini comme ci-dessus.

Les différents algorithmes de backtracking ne visitent pas exactement le même ensemble de noeuds : certains noeuds sont élagués par le filtrage fort mais pas par le filtrage faible. Par conséquent, suivant l'algorithme de filtrage utilisé en un noeud, ses sous-arbres peuvent être visités ou non. La comparaison des différents algorithmes de backtracking nécessite une analyse détaillée des relations entre ces algorithmes, cette analyse ainsi que les preuves des résultats théoriques sont données en annexe 1.

4.1 Résultats théoriques

Nous noterons BT_{bp} pour l'algorithme de backward pruning, et BT_{dcp} sa version dichotomique. Tout d'abord, la version standard du backward pruning peut être comparée au backtracking optimal :

Théorème 4.1 *$cost(BT_{bp}) \leq (\alpha + 2)cost(BT_{opt})$, où α est tel que :*
(1) $\alpha <$ la profondeur maximale de l'arbre exploé par BT_{bp} ; (2) $\alpha \leq C_2/C_1$.

Le surcoût du backward pruning comparé à BT_{opt} vient des séquences de backtracks successifs dus au filtrage fort. A chaque backtrack, le filtrage fort est appelé ; ce comportement est responsable du facteur $(\alpha + 2)$ présent dans la borne de complexité du théorème 4.1 (les détails se trouvent dans la preuve du théorème 4.1). Pour certains algorithmes de filtrage forts, il est possible d'exploiter les séquences de backtracks successifs pour en améliorer le coût total. Nous définissons l'incrémentalité bottom-up comme suit :

Définition 4.2 *Soit F un noeud de l'arbre de recherche, et soit C un noeud fils de F. Un algorithme de filtrage fort $prune_2$ de coût C_2 est bottom-up incremental pour un algorithme de filtrage faible $prune_1$ de coût C_1 si et seulement si il peut être implémenté de telle façon que le coût de l'appel $prune_2(C)$ suivi de l'appel $prune_2(F)$ est au plus $C_2 + C_1$ (au lieu de $2C_2$).*

En corollaire, l'appel à $prune_2$ sur d noeuds d'une séquence de $d-1$ backtracks successifs coûte $C_2+(d-1)C_1$ et non dC_2. Par conséquent, lorsque le filtrage fort peut être rendu bottom-up incremental, on obtient une bien meilleure borne que celle du théorème 4.1 :

Théorème 4.3 *Lorsque P_2 est bottom up incremental : $cost(BT_{bp}) \leq 3cost(BT_{opt})$*

Si l'algorithme de filtrage fort n'est pas bottom-up incremental, il peut être intéressant d'utiliser la version dichotomique du backward pruning, comme le suggère le théorème suivant :

Théorème 4.4 *$cost(BT_{dcp}) \leq (3 + \log_2(\alpha - 1))cost(BT_{opt})$, où α est tel que :*
(1) $\alpha \leq$ la profondeur maximale de l'arbre exploré par BT_{dcp}, (2) $\alpha \leq C_2/C_1$

Ces théorèmes ainsi que la proposition 2.2 nous permettent de comparer le backward pruning aux algorithmes de backtracking BT_1 et BT_2 :

Corollaire 4.5
(i) Backward pruning peut gagner un facteur exponentiel en la profondeur de l'arbre comparé à BT_1, et un facteur C_2/C_1 comparé à BT_2.
(ii) Backward pruning (la version dichotomique) coûte au plus $(3 + \log_2(\alpha - 1))$ fois BT_1 ou BT_2, où α est défini au théorème 4.4.
(iii) De plus, si P_2 est bottom up incremental : Backward pruning coûte au plus 3 fois BT_1 ou BT_2.

5 Résultats expérimentaux

Nous comparons les différents algorithmes sur une famille d'arbres de recherche binaires qui simulent des résolutions de problèmes combinatoires typiques, où généralement les arbres ne sont pas équilibrés, les feuilles des arbres apparaissant à différentes profondeurs.

Notre modèle d'arbre de recherche définit la forme de l'arbre en fonction de 4 paramètres : d_{max}, la profondeur maximale de l'arbre, une profondeur d_0, avec $0 < d_0 \leq d_{max}$, et deux probabilités p_1 et p_2. Un arbre de recherche binaire est généré de telle façon que, à la profondeur d, l'opérateur de pruning P_i ait une probabilité $p_i(d)$ d'élaguer un noeud, où $p_i(d)$ est défini par deux simples interpolations linéaires (voir la figure 4) :

- à la profondeur 0, $p_i(0) = 0$,
- à la profondeur d_0, $p_i(d_0) = p_i$,
- à la profondeur d_{max}, $p_i(d_{max}) = 1$,
- à la profondeur d, $0 < d < d_0$, $p_i(d) = p_i * d/d_0$,
- à la profondeur d, $d_0 < d < d_{max}$, $p_i(d) = \frac{1*(d-d_0)+p_i*(d_{max}-d)}{(d_{max}-d_0)}$.

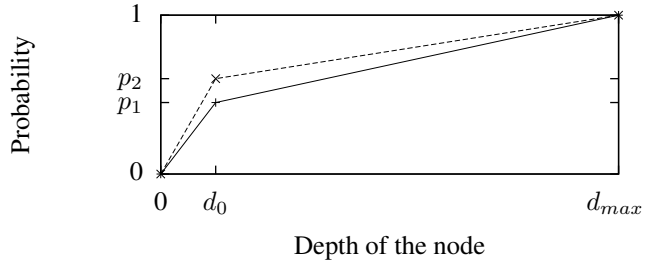

FIG. 4 – Probabilités d'élaguer un noeud en fonction de la profondeur du noeud

Pour comparer les différents algorithmes, nous normalisons leur coûts pour explorer l'arbre en les divisant par le coût du backtracking optimal. Par exemple, un rapport de 5 signifie que l'algorithme est 5 fois plus lent que l'algorithme le meilleur possible.

Les résultats sont présentés pour $d_{max} = 150$, $d_0 = 33$ et $p_1 = 0.45$. Ces valeurs ont été choisies pour que les arbres générés soient d'une taille raisonnable. Nous faisons varier p_2 de 0.45 à 1 par incrément de 0.05, puisque nous avons $p_2 \geq p_1$ (P_2 effectue un filtrage plus fort que P_1). Remarquons que, lorsque $p_2 = p_1 = 0.45$, P_2 ne peut élaguer aucun noeud qui n'est pas élagué par P_1. Lorsque $p_2 = 1$, P_2 élague toutes les branches de l'arbre à une profondeur au plus de d_0.

Le coût du filtrage P_1, C_1 est pris égal à 1, et les expérimentations ont été faites pour trois valeurs du coût du filtrage P_2 : $C_2 = 10, 100, 1000$. La figure 5 pésente les résultats pour la version dichotomique du backward pruning (BT_{dcp}), le backward pruning (BT_{bp}), le backward pruning avec un opérateur de pruning incremental ($Inc.BT_{bp}$), BT_1 et BT_2. Un point d'une des courbes de la figure 5 est une moyenne géométrique pour dix arbres de recherche générés aléatoirement. Les axes des ordonnées représentent les coût normalisés en échelle logarithmique, les axes des abscisses étant les valeurs de la probabilité p_2.

Les résultats sont en accord avec l'analyse théorique. Notons que $Inc.BT_{bp}$ a un bon comportement expérimental avec un coût d'environ 2 fois celui du backtracking optimal (la borne donnée par le théorème 4.3 est d'un facteur 3).

6 Discussion

6.1 Travaux connexes

De nombreux travaux destinés à améliorer l'algorithme de backtracking on été proposés, en particulier dans un contexte où il s'agit d'affecter des valeurs à un ensemble de variables soumises à des contraintes. Ces travaux peuvent généralement se classer en deux catégories (voir [3]) : les approches *look-ahead* et les approches *look-back*. Les approches *look-ahead*, qui exploitent des informations sur les variables non encore affectées, sont au coeur des solveurs SAT (qui utilisent des variations de l'algorithme DPLL [4, 5]), des solveur MIP (qui appliquent la programmation linéaire sur des relaxations continues du problème [6]), et des solveurs CP (qui utilisent diverses techniques de propagation [7]). Les approches *look-back* exploitent des informations sur la recherche déjà effectuée, en analysant le graphe de contraintes afin de déterminer de meilleur points de backtrack, ou d'apprendre de nouvelles contraintes entre les variables [8]. Notons aussi [9], qui présente un algorithme pour calculer un conflit minimal. Le backward pruning peut être considéré comme une nouvelle forme de méthode *look-back* dans laquelle l'analyse du graphe de contraintes est remplacée par l'exécution d'un algorithme au backtrack, un filtrage fort dans notre cas. De plus, le backward pruning, n'étant pas restreint à l'effectation de valeurs à des variables, est un peu plus général.

Le travail le plus proche du backward pruning est Quickshaving [10]. A chaque backtrack, Quickshaving vérifie les précédentes décisions de recherche qui ont conduit à un échec. On peut voir ces vérifications additionnelles comme une forme de filtrage fort. QuickShaving est une amélioration du backtracking dans la mesure où il peut être exponentiellement plus rapide au prix d'un faible surcoût. Cependant, contrairement au backward pruning, QuickShaving n'est pas toujours une amélioration du backtracking avec un filtrage fort et peut être exponentiellement plus lent. En fait, QuickShaving n'a pas l'équivalent des propriétés d'optimalité du backward pruning.

Différentes approches pour combiner un filtrage faible

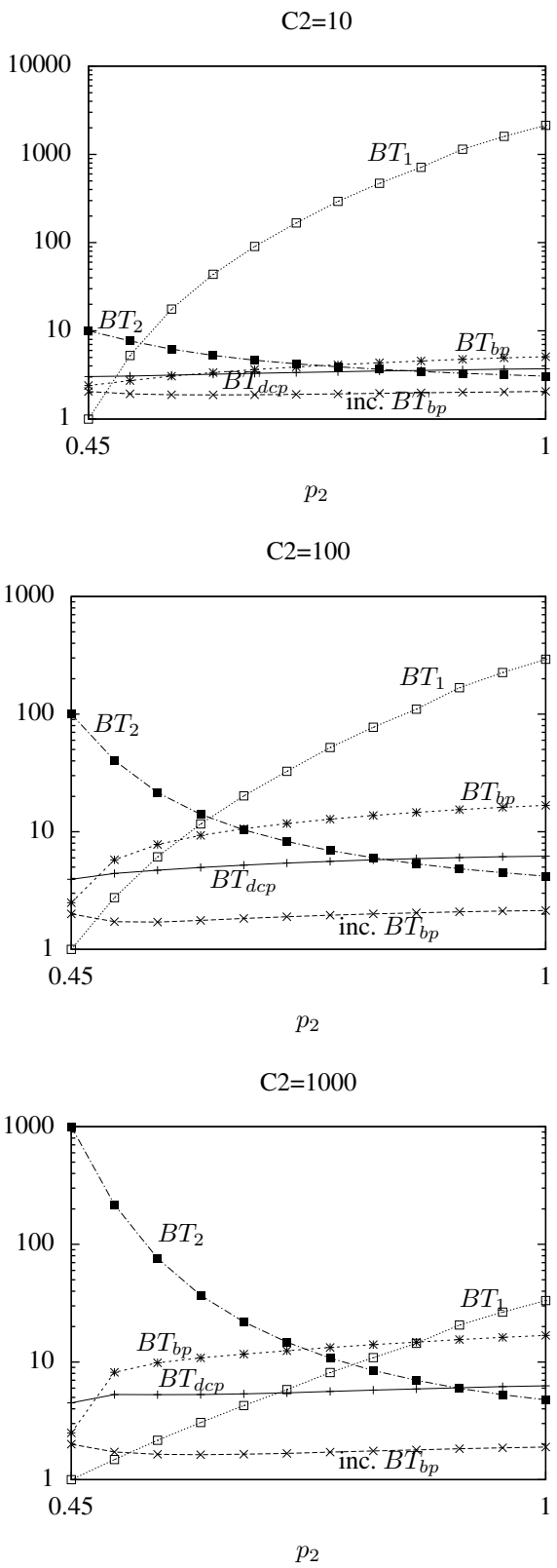

FIG. 5 – Coûts normalisés (axe des y en échelle logarithmique) pour p_2 variant de p_1 à 1.

et un filtrage fort ont été proposées dans la littérature. En MIP, le filtrage fort prend la forme d'un ajout de contraintes additionelles qui doivent être vérifiées par toute solution entière (*cut*). Générer de tels cuts est coûteux, et, en pratique, des règles heuristiques permettent de décider si un nouveau cut doit être gardé ou non. En CP, mentionnons d'abord [11] : deux techniques de pruning pour les réseaux de contraintes binaires sont analysées afin de déterminer des cas où elles donnent toutes deux le même résultat ; pendant la recherche, si l'on s'aperçoit que l'on se trouve dans un de ces cas, le pruning faible est appliqué, autrement, le pruning fort est appliqué. Dans [12], pour les problèmes continus, l'appel au pruning fort est évité lorsqu'une méthode d'extrapolation, peu coûteuse, prédit que le filtrage fort sera inutile. Finalement, [13] et [14] proposent des heuristiques spécifiques pour choisir entre deux techniques de pruning. Aucune de ces approches ne garantit de propriété d'optimalité proche de celles de backward pruning.

Les résultats de complexité donnés dans ce papier sont une forme de *competitive analysis*. La *competitive analysis* a été introduite par [15] pour analyser des problèmes online, c'est-à-dire des problèmes dans lesquels il faut satisfaire une séquence de requêtes sans connaître les futures requêtes. La performance d'un algorithme online est comparée à un algorithme offline, qui lui connaît à l'avance la séquence de requêtes. En fait, le backtracking optimal, comme défini en section 2.2, peut se voir comme un algorithme offline : il connaît à l'avance le meilleur pruning à appliquer à chaque noeud. Dans le vocabulaire de la *competitive analysis*, nos résultats de complexité (proposition 2.2, théorèmes 4.1, 4.3 et 4.4) sont la détermination des *competitive ratios* des différents algorithmes de backtracking.

6.2 Perspectives

Dans ce papier, nous avons supposé que deux prédicats de pruning étaient donnés, l'un plus fort et plus coûteux que l'autre. Discutons de ces hypothèses :

- Si les deux prédicats ont un coût similaire, il n'y a aucune raison d'utiliser le backward pruning, le backtracking avec le prédicat de pruning fort est optimal ou proche de l'optimal.
- Lorsque les puissances d'élagage des deux prédicats ne peuvent pas être comparées, c'est-à-dire lorsque aucun n'est plus fort que l'autre, le framework du backward pruning peut encore s'appliquer. Appelons $prune_2$ le plus coûteux des deux, nous pouvons définir $prune'_2$ comme suit : d'abord, appliquer $prune_1$, et ensuite appliquer $prune_2$. Ensuite, nous considérons les deux prédicats de pruning $prune_1$ et $prune'_2$.
- Considérons le cas d'un seul algorithme de pruning : dans ce cas, la question à chaque noeud se ramène à décider si l'algorithme de pruning est appliqué ou

non avant l'expansion du noeud. Le backward pruning peut s'appliquer en définissant $prune_1$ qui renvoie toujours faux, $prune_2$ étant l'algorithme de pruning donné.
- Nous pouvons aussi considérer plus de deux algorithmes de pruning. Ce cas peut se traiter par une petite extension du backward pruning.
- Ce papier s'est focalisé sur le cas général du backtracking où l'on ne sait rien du problème. Souvent, le problème est d'affecter des valeurs à des variables. Dans ce cas, au lieu d'un prédicat de pruning, on peut faire appel à une procédure de pruning *look ahead*, qui peut élaguer des descendants. Le principe du backward pruning fonctionne encore, cependant certaines adaptations des algorithmes et des résultats de complexité sont nécessaires, mais sont hors du cadre de ce papier.

Le backward pruning étant un schéma générique, une perspective générale est d'étudier spécifiquement différent domaines d'application. Par exemple, les théorèmes 4.1, 4.3 et 4.4 sont vrais lorsque les temps d'exécution des algorithmes de pruning sont constants. Lorsque ce n'est pas le cas, ces résultats peuvent encore être utilisés à condition que l'on dispose de bornes sur les temps d'exécution. Pour un problème donné, cependant, le temps d'exécution des algorithmes peut fortement varier, et une étude théorique spécifique peut être nécessaire pour obtenir des bornes précises sur le temps d'exécution du backward pruning, ainsi que pour spécialiser l'idée du backward pruning pour ce problème. Par exemple, nous avons intégré une variante du backward pruning dans la méthode de recherche par defaut d'un solveur de contraintes industriel (CP Optimizer dans IBM ILOG CPLEX Optimization Studio [2]). Dans un solveur de contraintes, les temps d'exécution des algorithmes de pruning sont très variables, ce qui a demandé quelques adaptations du schéma théorique décrit dans ce papier.

7 Conclusions

Ce papier apporte quelques éléments de réponse à la question générale de l'amélioration des algorithmes de recherche combinatoire. De nombreux solveurs combinatoires modernes, s'ils tirent leur efficacité d'une interaction complexe de plusieurs composants, reposent fortement sur l'algorithme de backtracking. Nous avons introduit le backward pruning, un algorithme de backtracking qui exploite l'idée contre-intuitive d'élaguer l'arbre de recherche en partant des feuilles de l'arbre. Nous montrons, théoriquement et expérimentalement, que le backward pruning améliore les performances pour une large classe de problèmes, ce qui laisse supposer qu'il pourrait remplacer avec succès le backtracking dans un certain nombre d'applications.

De plus, grâce à la borne théorique donnée par le "backtracking optimal", nous savons que les algorithmes de backward pruning ne sont pas très loin de la meilleure façon possible d'explorer un arbre de recherche.

Remerciements

Je remercie Jean-François Puget, Philippe Refalo et Paul Shaw pour leurs relectures et commentaires constructifs.

Annexe 1 : preuves et analyse détaillée

Proof. [of Proposition 2.2] Backtracking is used to solve a problem characterized by its size n. Typically, this size corresponds to the maximal depth of the search tree generated by backtracking. The maximal saving of BT_{opt} compared to BT_1 happens when the optimal exploration consists of applying P_2 to the root node, which proves that no solution exists, whereas BT_1 needs to develop a tree with branching factor $r \geq 2$ and of depth n : BT_{opt} costs C_2 and BT_1 costs $C_1 r^n$. Therefore, as C_1 and C_2 are constant, we have (i).

The gain of BT_{opt} compared with BT_2 is maximal when P_2 does not prune more nodes than P_1 : in this case, BT_{opt} is equivalent to BT_1 and is C_2/C_1 times more efficient than BT_2. Thus, we have (ii).

Finally, we can see that BT_{opt} may save an exponential running time compared to BT_1 and a factor C_2/C_1 in running time compared to BT_2 on the same problem, by considering a search tree where the root node has two children A and B such that : (1) for exploring A, BT_1 needs to visit 2^n nodes, and BT_2 needs to visit 1 node ; (2) for exploring B, BT_1 and BT_2 need to visit M nodes, where M is a big number compared to C_2/C_1, but is small compared to 2^n : $C_2/C_1 \ll M \ll 2^n$. The running times of BT_1, BT_2 and BT_{opt} are respectively equal to $C_1(2^n + M) \approx 2^n C_1$, $C_2(1 + M) \approx MC_2$ and $C_2 + C_1 M \approx MC_1$. Thus, we have (iii). □

In order to prove Theorems 4.1, 4.3 and 4.4, we need to introduce some additional definitions and lemmas. Let us call $Tree_1$, $Tree_2$, $Tree_{opt}$, $Tree_{bp}$ and $Tree_{dcp}$ the search trees respectively generated by BT_1, BT_2, BT_{opt}, BT_{bp} and BT_{dcp}.

Définition 7.1 *We can associate a label to each node N of a search tree, which denotes the unique path from the root to N specified by the rank of every child along the path : $label(N) = (r_0, r_1, \ldots, r_d)$.*

For instance, a label $(2, 1, 4)$ corresponds to the fourth child of the first child of the second child of the root.

Then, we define the concept of *position* of a backtracking search in a search tree. As a backtracking algorithm may backtrack several times to the same node, the label of

a node alone is not enough to specify exactly the position in a search tree.

Définition 7.2 *A position in a search tree is denoted by a couple (N, k), where N is a node of the search tree, and k is the number of children of N that have been already visited.*

We will need to compare two positions in a tree, possibly in two different trees (generated by different backtrack algorithms for the same problem). First, we extend the concept of label to position :

Définition 7.3 *Let $A = (N_A, k)$ be a position, let $label(N_A) = (r_0, r_1, \ldots, r_d)$, then $extendedLabel(A) = (r_0, r_1, \ldots, r_d, k, +\infty)$*

Now, we can define an order on positions, which simply corresponds to the order in which the positions are visited with a left to right depth first traversal of the tree (i.e., a backtracking algorithm) :

Définition 7.4 *Let A and B be two positions in a search tree. We say that A is smaller that B, and we note $A < B$, if and only if $extendedLabel(A)$ is lexicographically smaller than $extendedLabel(B)$.*

We will need to have a measure of the cost spent by a backtracking algorithm to go from a position to another one. This will be possible thanks to the following *distance*, which expresses the number of nodes opened by a left to right depth first traversal of the tree between the two positions (in other words the number of recursive calls to the backtracking procedure). Note that a node N is opened at position $(N, 0)$.

Définition 7.5 *Let A and B be two positions in a search tree such that $A \leq B$, $distance(A, B, Tree)$ denotes the number of positions C in the tree $Tree$ such that : $C > A$, $C \leq B$ and $C = (*, 0)$.*

Example : let $A = (N, k)$ be a position such that k is equal to the total number of children of node N. That is to say that all the children of node N have been just explored. Let $B = (F, i)$ be a position such that N is the i-th child of node F. In other words, B is the position just after A, and is reached after a simple backtrack from A. Then, by definition of distance, and since $i \neq 0$, we have : $distance(A, B, Tree) = 0$. That is to say that the distance when backtracking is 0. Then, consider the position $C = (N', 0)$ where N' is the $i + 1$-th child of F. We have $distance(A, C, Tree) = distance(B, C, Tree) = 1$.

Définition 7.6 *Let A and B be two positions in the search tree generated by backtracking algorithm BT such that $A \leq B$, $cost(A, B, BT)$ denotes the cost spent by BT for going from position A to position B.*

Note that, as BT_1 applies P_1 to each node opened between A and B, $cost(A, B, BT_1) = C_1 * distance(A, B, Tree_1)$.

All the nodes which are explored by BT_{opt} are also explored by BT_1. Furthermore, when a node is expanded into its children, the children are the same in BT_{opt}, BT_{bp}, BT_{dcp}, BT_1 or BT_2. Therefore, as a position is charaterized by a node and the rank of one of its children, a position in the tree $Tree_{opt}$ always exists in the tree $Tree_1$. Obviously, the converse is not true : a position in $Tree_1$ may not exist in $Tree_{opt}$.

The following lemma is needed for the proof of Theorem 4.1 :

Lemme 7.7 *Let $A = (N_A, *)$ and $B = (N_B, *)$ be two positions in $Tree_1$, and such that $A < B$. Assume that $P_2(N_A) \neq \emptyset$ and $P_2(N_B) \neq \emptyset$ (neither N_A nor N_B can be pruned by P_2), Then, we have :*

(i) A and B also exist in $Tree_{opt}$,

(ii) if $distance(A, B, Tree_1) \geq C_2/C_1$, then, $cost(A, B, BT_{opt}) \geq C_2$,

(iii) if $distance(A, B, Tree_1) < C_2/C_1$, then, $cost(A, B, BT_{opt}) = cost(A, B, BT_1) = distance(A, B, Tree_1) * C_1$.

Proof. P_2 cannot prune N_A nor N_B. By monotonicity of P_2, no ancestor of N_A or N_B can be pruned by P_2. Thus, we know that positions A and B also exist in the search tree generated by BT_{opt}, which proves (i). Furthermore, BT_{opt} does not apply P_2 on any ancestor of N_A or N_B, otherwise it would not be optimal. Let us consider two cases.
Case (1) : BT_{opt} applies P_2 at a position such that this pruning changes the distance : $distance(A, B, Tree_{opt}) < distance(A, B, Tree_1)$. Then, as no ancestor of N_A or N_B can be pruned by P_2, this position must be strictly between the positions A and B, and, thus, the cost for BT_{opt} to go from A to B is at least C_2.
Case (2) : BT_{opt} does not apply P_2 at a position such that this pruning changes the distance, thus, $distance(A, B, Tree_{opt}) = distance(A, B, Tree_1)$. As BT_{opt} spend at least C_1 at each open node, we have $cost(A, B, BT_{opt}) \geq distance(A, B, Tree_{opt}) * C_1 = cost(A, B, BT_1)$. By optimality, BT_{opt} cannot cost more than BT_1, then, it costs exactly $cost(A, B, BT_1)$.

Let us prove point (ii) of the lemma. We have $distance(A, B, Tree_1) \geq C_2/C_1$. Thus $distance(A, B, Tree_1) * C_1 \geq C_2$, thus, both in case (1) and case (2) we have $cost(A, B, Tree_{opt}) \geq C_2$, which proves (ii).

Now, let us prove (iii) : we have $distance(A, B, Tree_1) < C_2/C_1$. Then, $cost(A, B, BT_1) < C_2$. By optimality of BT_{opt}, we have $cost(A, B, BT_{opt}) \leq cost(A, B, BT_1)$. Thus, BT_{opt} cannot apply P_2 between A and B because this would cost at least C_2, therefore case (1) is not possible. Thus, we

are in case (2) and $cost(A, B, BT_{opt}) = cost(A, B, BT_1)$ which proves (iii). □

We now formalize BT_{bp} as a sequence of positions :

Définition 7.8 *For simplifying the formalization, we consider that the root node has a father, called top, which has only one child, the root node. Furthermore, let us assume that no pruning operator can prune top : $P_2(top) \neq \emptyset$ and $P_1(top) \neq \emptyset$. This additional node amounts to add two fake positions, position $start = (top, 0)$ and position $end = (top, 1)$, as first and last element to the sequence of real positions visited by a backtracking algorithm. Let $S_0, \ldots, S_w$ be the sequence of positions visited by BT_{bp} for exploring the whole tree starting at $start$ and ending at end. We have : $S_0 = start, S_w = end$. Note that, consistently with our definition of distance, we have $distance(start, S_1, Tree_{bp}) = 1$ and $distance(S_{w-1}, end, Tree_{bp}) = 0$.*

We define $Seq_{bp} = \{A_i\}_{i=0\ldots v}$ a subsequence of the above sequence $\{S_i\}_{i=0\ldots w}$ where $A_0 = S_0 = start$, $A_v = S_w = end$, and such that for $i \in [1, v-1], A_i$ is the first position at a distance of at least $k = C_2/C_1$ after A_{i-1} which cannot be pruned by P_2. Formally : $A_i = (N_{A_i}, *)$ is the smallest position in $\{S_i\}_{i=0\ldots w}$ such that $distance(A_{i-1}, A_i, Tree_{bp}) \geq k$ and $P_2(N_{A_i}) \neq \emptyset$.

This subsequence $Seq_{bp} = \{A_i\}_{i=0\ldots v}$ formalizes the behavior of BT_{bp} in the following way : let us start from a position A_i of the subsequence, where $i \in [0, v-1]$, and assume the global variable *credit* equals 0 when the algorithm is at this position. After $k = C_2/C_1$ applications of P_1, and k increases of *credit* by C_1, *credit* reaches the value C_2. Then, BT_{bp} reset *credit* to 0 and, possibly after a backtrack due to a pruning from P_1, applies P_2 on a position called B. If P_2 cannot prune the node at position B, we take $A_{i+1} = B$. If P_2 can prune the node at position B, BT_{bp} backtracks, reapplies P_2 on the father of B, and continues to backtrack until P_2 can no longer prune the current position. We call A_{i+1} this position.

The last position A_v may be different : it handles the case when the credit does not reach the value C_2 and BT_{bp} ends.

We are now able to prove Theorems 4.1, 4.3, and 4.4.

Proof. [of Theorem 4.1] Let $Seq_{bp} = \{A_i\}_{i=0\ldots v}$, let $k = C_2/C_1$. We first compute the cost of BT_{bp} for going from A_i to A_{i+1}, for $i \in [0, v-2]$, and, then, the cost of BT_{bp} for going from A_{v-1} to A_v :

1. The cost of BT_{bp} for going from A_i to A_{i+1}, for $i \in [0, v-2]$ is :
   $cost(A_i, A_{i+1}, BT_{bp}) = k * C_1 + (n^i_{bp} + 1) * C_2 = (n^i_{bp} + 2) * C_2$, where n^i_{bp} is the number of backtracks due to P_2 to find the position A_{i+1}, i.e., the number of backward prunings.
   Furthermore, as $distance(A_i, A_{i+1}, Tree_{bp}) \geq k$, we also have $distance(A_i, A_{i+1}, Tree_1) \geq k$. Thus, thanks to lemma 7.7 (ii) $cost(A_i, A_{i+1}, BT_{opt}) \geq C_2$. Thus we have : $cost(A_i, A_{i+1}, BT_{bp}) \leq (n^i_{bp} + 2) * cost(A_i, A_{i+1}, BT_{opt})$.
2. It remains to compute the cost for going from A_{v-1} to A_v. We consider two cases :
   (a) Either we have $distance(A_{v-1}, A_v, Tree_{bp}) \geq k$, thus, P_2 is applied on A_v, and the same argument as above applies, we have : $cost(A_{v-1}, A_v, BT_{bp}) \leq (n^{v-1}_{bp} + 2) * cost(A_{v-1}, A_v, BT_{opt})$.
   (b) Or we have $distance(A_{v-1}, A_v, Tree_{bp}) < k$. As credit is equal to 0 at position A_{v-1}, credit does not reach value C_2 and P_2 is not applied between A_{v-1} and A_v. Thus, backward pruning is equivalent to BT_1 between A_{v-1} and A_v : $cost(A_{v-1}, A_v, BT_{bp}) = cost(A_{v-1}, A_v, BT_1)$. Furthermore, A_v is the last position visited by BT_{bp}, i.e., $A_v = (top, 1)$, and we know that P_2 cannot prune the node top. Thus, we can apply lemma 7.7 (iii), from which we get : $cost(A_{v-1}, A_v, BT_{opt}) = cost(A_{v-1}, A_v, BT_1)$. Thus, $cost(A_{v-1}, A_v, BT_{bp}) = cost(A_{v-1}, A_v, BT_{opt})$.

Finally, by summing the costs for each part A_i, A_{i+1}, and by taking α equal to the maximal number of backtracks due to P_2, we get : $cost(BT_{bp}) \leq (\alpha + 2) cost(BT_{opt})$, where $\alpha = max_{i \in [0, v-1]} n^i_{bp}$.

Clearly $\alpha < depth(Tree_{bp})$, which proves the first bound of the theorem. The second bound of the theorem comes from $\alpha \leq C_2/C_1$. This can be seen by considering the position B where BT_{bp} applies P_2 for the first time after A_i. Position B is at distance C_2/C_1 from A_i. BT_{bp} does not backtrack higher than the most recent common ancestor of N_{A_i} and N_B (by monotonicity of P_2 since this is an ancestor of N_{A_i}, and thus P_2 cannot prune this ancestor). The number of ancestors of N_B, including B itself, that are not ancestor of N_{A_i} is at most C_2/C_1 since $distance(A_i, B, Tree_{bp}) = C_2/C_1$. □

Proof. [of Theorem 4.3] We do not give the complete proof, it is quite similar to the previous one. Instead, we simply recompute the cost of BT_{bp} for going from A_i to A_{i+1}. As P_2 is bottom up incremental, the cost for applying $n^i_{bp} + 1$ times P_2 while backtracking is equal to $C_2 + n^i_{bp} * C_1$.

Thus, the cost of BT_{bp} for going from A_i to A_{i+1} is : $cost(A_i, A_{i+1}, BT_{bp}) = k * C_1 + C_2 + n^i_{bp} * C_1 = (n^i_{bp}) * C_1 + 2 * C_2$.

As we have seen in the previous proof, $n^i_{bp} \leq C_2/C_1$. Thus, $cost(A_i, A_{i+1}, BT_{bp}) \leq 3 * C_2$.

By summing, we get $cost(BT_{bp}) \leq 3 * cost(BT_{opt})$. □

Proof. [of Theorem 4.4] The point is to note that BT_{dcp} can also be characterized by the sequence $Seq_{bp} = \{A_i\}_{i=0...v}$. The only difference with BT_{bp} is that, for reaching position A_{i+1} starting from A_i, BT_{bp} needs to apply at most $\lfloor \log_2(d-1) \rfloor + 2$ times pruning operator P_2, where d is the depth of the search tree (we assume $d > 1$, otherwise the theorem is trivially true). The cost of BT_{dcp} for going from A_i to A_{i+1} is : $cost(A_i, A_{i+1}, BT_{dcp}) \leq kC_1 + C_2\lfloor \log_2(d-1) + 2 \rfloor = C_2(3 + \lfloor \log_2(d-1) \rfloor)$. Indeed, BT_{dcp} calls the divide-and-conquer search between *dknp*+1 and *level*. Furthermore, we have that *level-dknp* $\leq C_2/C_1$. Thus, $cost(A_i, A_{i+1}, BT_{dcp}) \leq C_2 * (3 + \lfloor \log_2(C_2/C_1 - 1) \rfloor)$. By summing, we prove the theorem. □

Références

[1] Golomb, S.W., Baumert, L.D. : Backtrack programming. J. ACM **12** (1965) 516–524

[2] IBM : IBM ILOG CPLEX Optimiation Studio 12.2 User's manual. IBM (2010)

[3] Dechter, R. : Enhancement schemes for constraint processing : Backjumping, learning, and cutset decomposition. Artificial Intelligence **41** (1990) 273–312

[4] Davis, M., Putnam, H. : A computing procedure for quantification theory. J. ACM **7** (1960) 201–215

[5] Davis, M., Logemann, G., Loveland, D. : A machine program for theorem-proving. Commun. ACM **5** (1962) 394–397

[6] Wolsey, L.A. : Integer programming. Wiley (1998)

[7] Haralick, R., Elliot, G. : Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence **14** (1980) 263–313

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

[9] Junker, U. : Quickxplain : preferred explanations and relaxations for over-constrained problems. In : AAAI'04 : Proceedings of the 19th national conference on Artifical intelligence, AAAI Press (2004) 167–172

[10] Lhomme, O. : Quick shaving. In Veloso, M.M., Kambhampati, S., eds. : AAAI, AAAI Press / The MIT Press (2005) 411–415

[11] Sakkout, H.E., Wallace, M., Richards, B. : An instance of adaptive constraint propagation. In Freuder, E.C., ed. : CP. Volume 1118 of Lecture Notes in Computer Science., Springer (1996) 164–178

[12] Lebbah, Y., Lhomme, O. : Accelerating filtering techniques for numeric csps. Artif. Intell. **139** (2002) 109–132

[13] Stergiou, K. : Heuristics for dynamically adapting propagation. In Ghallab, M., Spyropoulos, C.D., Fakotakis, N., Avouris, N.M., eds. : ECAI. Volume 178 of Frontiers in Artificial Intelligence and Applications., IOS Press (2008) 485–489

[14] Szymanek, R., Lecoutre, C. : Constraint-level advice for shaving. In de la Banda, M.G., Pontelli, E., eds. : ICLP. Volume 5366 of Lecture Notes in Computer Science., Springer (2008) 636–650

[15] Sleator, D.D., Tarjan, R.E. : Amortized efficiency of list update and paging rules. Commun. ACM **28** (1985) 202–208