Recherche à focalisation progressive

Nicolas Prcovic
I3S,
930, Route des Colles, BP145
06903 Sophia-Antipolis
email: nprcovic@essi.fr

Bertrand Neveu
CERMICS,
Route des lucioles, BP93,
06902 Sophia-Antipolis
email: Bertrand.Neveu@sophia.inria.fr

Résumé

Dans la résolution d'un problème de satisfaction de contraintes par un algorithme complet de recherche arborescente, les heuristiques de choix de valeur sont censées guider la recherche vers une solution. Des résultats expérimentaux montrent l'efficacité d'une heuristique dynamique quand le problème a suffisamment de solutions pour que les premiers choix ne soient pas cruciaux. Quand ces choix peuvent mener à des sous-problèmes incohérents, l'heuristique devient moins efficace. Un nouvel algorithme, appelé recherche à focalisation progressive, a alors été conçu. Il entrelace la recherche dans différents sous-arbres dans le but d'obtenir une information qui lui permette de se focaliser progressivement sur le sous-arbre le plus prometteur. Cet algorithme est comparé à la recherche entrelacée (IDFS) et s'est avéré efficace pour des problèmes aléatoires au pic de complexité.

1 Introduction

Pour résoudre un problème de satisfaction de contraintes (CSP) sur domaines finis à l'aide d'une méthode complète, les algorithmes de recherche arborescente basés sur la recherche en profondeur d'abord (DFS) sont les plus communément utilisés. Diverses améliorations ont été proposées pour limiter l'explosion combinatoire. Ce sont principalement des techniques de filtrage des domaines, de retours-arrière avec sauts (backjumping) et l'utilisation d'heuristiques de choix de variable. Pour les plus grands espaces de recherche explorables par une méthode complète, les résultats suivants sont généralement admis pour les CSP binaires. Le meilleur niveau de filtrage est le maintien de cohérence d'arc(MAC) [1] [SF94], les retours-arrière avec sauts ne sont pas utiles [BR96] et, en l'absence de connaissances spécifiques sur le problème à résoudre, les heuristiques de choix de variable générales comme *min-domain+degree* ou *min-domain/degree* [BR96] sont aujourd'hui reconnues être les plus efficaces. Pour l'obtention d'une seule solution, les heuristiques de choix de valeur peuvent aussi être utilisées, mais elles sont considérées moins importantes pour réduire l'arbre de recherche. Dans cet article, nous rappellerons d'abord quelles sont les principales heuristiques de choix de valeur. Puis nous analyserons les effets des heuristiques de choix de valeur quand elles sont combinées avec l'algorithme MAC. Nous montrerons que pour des CSP avec peu de solutions près du pic de complexité, les premiers choix sont cruciaux et qu'il devient alors important de trouver une bonne heuristique pour ces choix. Nous avons pour cela conçu un nouvel algorithme, la recherche à focalisation progressive, qui alterne des recherches dans différents sous-arbres avant de se focaliser sur le plus prometteur. Nous le comparons expérimentalement avec la classique recherche en profondeur d'abord et avec la recherche entrelacée (IDFS) [Mes97].

2 Etudes précédentes sur les heuristiques de choix de valeur

2.1 Les heuristiques de choix de valeur existantes

Dans une recherche arborescente standard, deux choix doivent être faits à chaque étape. Tout d'abord, on choisit la prochaine variable à instancier. Le problème est alors décomposé en k sous-problèmes, un pour chaque valeur du domaine de cette variable. Le choix de valeur détermine quel sous arbre sera exploré en premier par DFS.

Les effets de ces deux choix sur la recherche sont complémentaires. Le choix de variable a un effet sur l'arbre de recherche lui-même et le choix de valeur sur les branches de l'arbre qui seront explorées. Le but principal de l'utilisation d'une heuristique de choix de variable est de construire un arbre le plus petit possible, c'est-à-dire de limiter le nombre de branches et d'avoir les branches conduisant à des échecs les plus courtes possibles [SG98]. Les heuristiques hybrides *min-domain+degree* et *min-domain/degree* se basant sur la taille des domaines et le degré des variables jouent sur ces deux effets.

Le but d'une heuristique de choix de valeur est d'éviter de faire un mauvais choix, c'est-à-dire de sélectionner un sous-arbre sans solution. Elles utilisent pour cela une information partielle qui différencie les valeurs. Cette information peut être calculée une fois pour toutes avant la recherche ou dynamiquement au cours de la recherche.

Quand l'heuristique est dynamique, la principale information utilisée est la taille des domaines des variables futures, c'est-à-dire non encore instanciées. Ces tailles varient dynamiquement quand l'algorithme de recherche comme MAC effectue un filtrage des domaines. Cette information peut être utilisée de diverses manières.

- LVO-MC de Frost et Dechter [FD95] : la valeur sélectionnée est celle qui a la plus grande somme des nombres de valeurs compatibles dans tous les domaines des variables futures.

1. Dans cet article, quand nous nous référons à MAC, il s'agit seulement du niveau de filtrage (cohérence d'arc) effectué à chaque affectation de variable et non du caractère binaire de l'arbre de recherche développé par cet algorithme.

– plus grand support de Larrosa et Meseguer (highest support) [LM95] : somme des tailles normalisées des domaines des variables futures en prenant en compte les incohérences.
– plus grande taille de domaine (max-domain-size) [Fro97]: les valeurs sont ordonnées dans l'ordre décroissant de la taille du plus petit domaine des futures variables.
– ES estimation du nombre de solutions [DP88] : le nombre de solutions potentielles est estimé en résolvant une relaxation arborescente du graphe de contraintes restant.
– MP Solutions potentielles (max-promises) de Geelen [Gee92]. C'est un cas simple de l'heuristique précédente : la relaxation du problème restant est le problème formé des contraintes reliant la variable courante aux variables futures. La valeur choisie est celle pour laquelle le produit des tailles de domaines compatibles est maximum.

Pour calculer cette information sur la taille des domaines des variables futures, il faut propager l'effet de l'instanciation de la variable courante avec toutes les valeurs de son domaine pour pouvoir choisir la meilleure valeur selon le critère de l'heuristique. En d'autres termes, il faut effectuer à l'avance la propagation du Forward Checking dans des branches qui ne seront peut être pas explorées par la suite et cela peut être coûteux.

Pour pallier cette difficulté, quelques heuristiques statiques ont été définies.

– static least-conflicts (SLC) [Fro97] : les valeurs sont ordonnées selon le nombre de conflits auxquels elles prennent part. Ces calculs sont effectués sur les domaines initiaux et cet ordre statique est utilisé pendant toute la recherche.
– static max-promises (SMP) : de la même manière, pour chaque variable v, on calcule avant la recherche pour chaque valeur dans le domaine de v, le produit des nombres de valeurs compatibles dans les domaines des autres variables et on ordonne le domaine de v selon ces produits.
– champ moyen (mean field) : un prétraitement basé sur la théorie du champ moyen ordonne les domaines avant la recherche [CVMB96].

La plupart des résultats publiés utilisent pour la résolution l'algorithme du Forward Checking [HE80]. Ils montrent qu'une heuristique dynamique a un surcoût important et ne devient bénéfique que quand le problème est suffisamment difficile (nécessitant plus d'un million de tests de contraintes dans [FD95]).

Mais, aujourd'hui, les problèmes résolus le sont avec un algorithme maintenant un niveau de cohérence comme la cohérence d'arc et l'on peut espérer que le surcoût effectué à chaque nœud pour calculer une heuristique dynamique devienne négligeable.

2.2 Justification de l'heuristique des solutions potentielles

Les diverses expérimentations effectuées jusqu'à présent n'ont pas mis en évidence la supériorité d'une heuristique par rapport aux autres. L'ensemble de ces heuristiques donnent des résultats comparables. Parmi ces heuristiques, nous avons décidé d'étudier plus en détail le comportement de l'heuristique des solutions potentielles, qui a un fondement théorique.

Cette heuristique donne le nombre de solutions dans un sous-arbre, s'il n'y avait pas de contraintes entre les variables futures. En absence d'information sur ces contraintes, on peut supposer que ce nombre est proportionnel à une estimation du nombre de solutions dans un sous-arbre. En effet, dans le modèle de CSP aléatoires que nous avons utilisé dans nos tests (voir ci-après), les paires de valeurs interdites par une contrainte sont indépendantes des paires interdites par une autre contrainte. Ainsi, si nous considérons un algorithme comme le Forward Checking, où il n'y a pas de propagation due aux contraintes entre les variables futures, chaque paire de valeurs restant dans les domaines des variables futures a la même probabilité de satisfaire toutes les contraintes entre ces variables.

Nous avons montré dans une étude précédente [PN99] que le comportement de cette heuristique dynamique s'améliorait quand la profondeur de recherche augmentait. Elle est plus informée et se trompe moins souvent. Dans la partie suivante, nous allons étudier plus en détail son efficacité.

3 Effets de l'heuristique des solutions potentielles

Pour étudier les effets de l'heuristique de choix de valeur des solutions potentielles, nous avons effectué quelques expérimentations sur des CSP aléatoires binaires. Nous nous sommes restreints à des tests dans des zones où les CSP ont des solutions et où les problèmes ne sont pas trop faciles (un nombre significatif de retours-arrière se produit). Quand MAC résout sans heuristique un problème sans aucun retour-arrière, il n'y a évidemment aucun besoin d'heuristique de choix de valeur. Une heuristique est aussi inutile pour les problèmes sans solution. Frost avait remarqué dans [FD95] que l'utilisation d'une heuristique de choix de valeur pouvait avoir un effet quand on effectue des sauts avec CBJ, mais comme il a déjà été vu dans [BR96], de tels sauts n'arrivent en pratique jamais avec MAC.

3.1 Résultats expérimentaux sur des CSP aléatoires

Nous avons réalisé des tests en utilisant la CSP-LIB de Hulubei [Hul99], une bibliothèque C++ pour la résolution de CSP binaires en domaines finis. Nous avons implanté les heuristiques de choix de valeur des solutions potentielles statique (SMP) et dynamique (MP). Pour étudier l'importance du premier choix, nous avons aussi réalisé des tests où l'heuristique n'était utilisée que pour ce premier choix, tous les autres choix étant effectués selon l'ordre naturel des domaines. Nous avons appelé H1 cette heuristique du premier choix dans le tableau présentant les résultats et HO l'algorithme sans heuristique de choix de valeur.

Les tests ont été effectués sur des problèmes aléatoires, en utilisant le générateur de problèmes aléatoires de la CSP-LIB. Une classe de problèmes est définie par 4 paramètres :

- le nombre de variables,
- la taille des domaines,
- le nombre de contraintes, déterminé par le paramètre densité D,
- le nombre de paires de valeurs interdites par contrainte, déterminé par le paramètre dureté T.

Les tests ont été effectués avec des problèmes ayant 50 variables, des domaines de taille 15. Nous avons fait varier les paramètres de densité et de dureté pour obtenir des problèmes difficiles, mais qui pouvaient être résolus en un temps raisonnable (au maximum quelques minutes par instance de problème). Pour chaque ensemble de paramètres, nous avons réalisé des tests avec les heuristiques H1, SMP et MP, et les avons comparés avec des tests sans heuristique(H0). Quant à l'heuristique de choix de variable, nous avons utilisé *min-domain/degree* qui sélectionne la variable ayant le plus petit rapport taille des domaines sur degré (nombre de contraintes entre cette variable et les variables futures). Le niveau de filtrage a été AC3, fourni par la bibliothèque.

La partie gauche du tableau 1 présente la profondeur du plus haut retour-arrière apparu sur un ensemble de 100 problèmes sans heuristique(H0), avec les heuristiques de premier niveau H1, statique SMP et dynamique MP. Un résultat $i(k)$ signifie que les i-1 premiers choix ont été de bons choix pour les 100 instances de problème, il n'a pas été nécessaire de revenir sur eux pour trouver une solution et que k problèmes sur 100 ont dû revenir à ce niveau i. Cela peut être vu comme une sorte de mesure de la distance de la classe de problèmes considérée au pic de complexité correspondant à la zone de transition de phase entre problèmes avec et sans solutions [SD96]. Quand la profondeur i pour l'algorithme H0 est assez grande, cela signifie que pour toutes les instances, bien qu'il ne fussent pas guidés par une heuristique de choix de valeur, ces $i-1$ premiers choix n'ont jamais causé d'erreur. On est donc alors dans une classe de problèmes avec de nombreuses solutions. Quand, au contraire, des retours-arrière remontent jusqu'au premier niveau pour certaines instances, le premier choix a conduit à un sous-arbre sans solution et on se trouve dans une classe de problèmes proche du pic de complexité.

La partie droite du tableau 1 présente le temps cpu utilisé sur un Pentium III 500 pour la résolution des 100 instances de la classe traitée.

classe de problèmes	H0	H1	SMP	MP	H0	H1	SMP	MP
D=0.89,T=0.10	4(3)	4(1)	7(7)	7(3)	19h03	16h36	2h27	2h29
D=0.75 T=0.12	4(9)	4(4)	6(9)	6(1)	19h	16h20	3h16	2h36
D=0.52 T=0.16	4(2)	4(3)	6(7)	7(9)	2h13	2h10	39m	21m
D=0.41 T=0.20	2(2)	3(5)	4(1)	5(11)	5h23	3h26	1h01	57m
D=0.40 T=0.20	4(16)	4(3)	5(2)	5(1)	1h12	39m	13m	11m
D=0.32 T=0.25	2(20)	2(2)	2(1)	3(6)	5h37	3h07	1h38	1h23
D=0.31 T=0.25	2(2)	3(5)	4(4)	4(3)	1h06	42m	17m10	10m34
D=0.30 T=0.25	4(5)	4(3)	5(1)	6(2)	10m05	5m40	2m00	2m14

TAB. 1 – *Profondeur du plus haut retour-arrière, avec le nombre de problèmes à cette profondeur, et temps cpu utilisé pour résoudre 100 instances de problèmes ayant 50 variables, des domaines de taille 15, une densité D et une dureté T*
.

Nous pouvons voir que l'heuristique MP est très efficace pour des problèmes assez éloignés du pic de complexité. Le gain en temps obtenu est un facteur entre 4 et 8. Le faible surcoût dû au calcul de l'heuristique est largement compensé par la diminution du nombre de nœud générés : la version dynamique de l'heuristique est presque toujours meilleure que la statique. Nous pouvons aussi remarquer en analysant les résultats de H1 qu'une part significative des gains est due au premier choix, même si ce choix n'a pas été remis en cause durant la recherche.

3.2 Inefficacité au pic de complexité

Une seconde série de tests a été réalisée plus près du pic de complexité avec des domaines de taille 8 et 15. Alors, le facteur de gain dû à l'utilisation de l'heuristique

MP est tombé entre 1.4 et 1.1 (voir par exemple les classes de problèmes S=15, D=0.20, T=0.36 et S=8, D=0.44, T=0.15). Comme nous l'avons déjà remarqué, le sous-arbre ayant le plus grand nombre de solutions potentielles a la plus forte probabilité de contenir des solutions. Mais, ce sous-arbre a aussi la probabilité d'avoir la plus grande taille et quand l'heuristique se trompe, le dommage est plus important.

En restant dans la zone des problèmes avec solutions et en s'approchant du pic de complexité, une telle erreur arrive plus haut dans l'arbre et peut avoir des effets désastreux. Ceci peut expliquer l'inefficacité de l'heuristique près du pic, bien que les cas d'erreurs soient moins nombreux que sans heuristique. En analysant en détail les résultats près du pic, par exemple pour une taille des domaines de 15, une densité de 0.20 et une dureté de 0.36, l'heuristique se trompe 18 fois pour l'affectation de la première variable, alors que sans heuristique, une telle erreur au premier niveau de la recherche intervient 50 fois sur l'ensemble des 100 instances de problèmes de cette classe. Mais le temps le plus long pour résoudre une instance est de 14mn avec heuristique et de 2 mn sans heuristique. On peut aussi remarquer que ce premier choix est critique pour le temps global de résolution des 100 instances. Même si l'heuristique guide la recherche vers une solution dans un sous-arbre qui en contient, elle n'a aucun effet sur les sous-arbres sans solutions. Comme ces sous-arbres, une fois choisis, doivent être explorés entièrement, le temps passé dans leur exploration est la majeure partie du temps total.

classe de problèmes	H0	H1	SMP	MP	H0	H1	SMP	MP
S=15 D=0.25 T=0.31	1(38)	1(16)	1(16)	1(16)	9h52	8h32	6h47	7h12
S=15 D=0.20 T=0.36	1(50)	1(18)	1(18)	1(18)	2h20	2h04	2h06	2h01
S=15 D=0.14 T=0.44	1(44)	1(12)	1(12)	1(12)	18m33	11m40	10m08	8m27
S=8 D=0.50 T=0.13	1(51)	1(26)	1(26)	1(26)	2h29	1h49	1h47	1h48
S=8 D=0.44 T=0.15	1(49)	1(27)	1(27)	1(27)	1h37	1h28	1h22	1h27
S=8 D=0.435 T=0.15	1(30)	1(9)	1(9)	1(9)	46m	35m	31m	31m
S=8 D=0.315 T=0.20	1(39)	1(18)	1(18)	1(18)	16m17	14m47	14m12	13m49

TAB. 2 – *Résultats au pic de complexité pour des problèmes avec 50 variables, une taille des domaines S, une densité D et une dureté T.*

Dans les problèmes suffisamment éloignés du pic de complexité dans la zone des problèmes avec solution, il y a toujours eu dans l'ensemble des 100 problèmes traités une différence de niveau pour le plus haut retour-arrière intervenu avec et sans heuristique. Ceci peut expliquer l'efficacité de l'heuristique dans ce cas. L'heuristique dynamique, qui est plus informée par les choix précédents dans la branche courante est alors plus efficace.

Nous avons vu au contraire que l'heuristique dynamique des solutions potentielles n'était pas suffisante pour améliorer substantiellement les résultats dans la zone de complexité : elle fait trop souvent un choix désastreux au premier niveau. Comme les choix cruciaux sont les premiers et comme l'heuristique ne peut pas bénéficier d'une information descendante par la branche courante, une idée pour l'améliorer est de remonter de l'information des sous-arbres, avant de choisir celui à explorer en premier. Dans la partie qui suit, nous allons présenter l'algorithme de recherche à focalisation progressive, qui exploite cette idée.

4 Recherche à focalisation progressive

Notre but est de trouver une meilleure heuristique de choix de valeur que MP pour les premières variables. Avant de présenter la recherche à focalisation progressive, nous allons décrire les différentes étapes qui nous ont conduits à son élaboration.

4.1 Estimation de la longueur moyenne des branche grâce à l'estimateur de Knuth

Il existe une méthode plus générale que MP pour amasser des informations sur la recherche : l'estimateur de Knuth [Knu75], déjà utilisé par exemple dans [LL98] pour prédire le comportement d'un algorithme de Branch and Bound. Cette méthode consiste à produire un échantillon d'essais aléatoires. Chaque essai génère une branche de l'arbre de recherche, chaque choix de valeur étant fait au hasard. Ensuite, la méthode calcule des statistiques sur un paramètre à estimer à partir de l'échantillon produit. L'estimateur a la propriété intéressante suivante : il tend vers la vraie valeur du paramètre estimé au fur et à mesure que la taille de l'échantillon croît.

Pour estimer quelles sont les meilleures valeurs pour les d premières variables, nous pouvons générer un échantillon d'essais pour chacune des combinaisons de valeurs des d premières variables puis les ordonner en fonction des résultats. Il semble raisonnable de considérer que le sous-arbre commençant à la profondeur d qui a la moyenne de longueur de branche la plus grande aura le plus grand nombre de solutions. Appelons $D_{i,j}$, la profondeur de la $j^{\text{ème}}$ branche du $i^{\text{ème}}$ sous-arbre. Nous pouvons mesurer la longueur moyenne $D_i(b) = \frac{\sum_{j=1}^{j=b} D_{i,j}}{b}$ de l'échantillon de b branches générées pour le $i^{\text{ème}}$ sous-arbre commençant à la profondeur d. Chaque sous-arbre correspond à une instanciation des d premières variables et ces sous-arbres peuvent être ordonnés selon le critère $D_i(b)$.

Ainsi, l'estimateur de Knuth peut permettre d'atteindre un ordre de valeurs qui soit meilleur que MP si l'échantillon est suffisamment grand. Cependant, nous ne pouvons augmenter les performances de la recherche que si le coût du précalcul de l'ordre des valeurs est compensé par une réduction du temps de recherche suffisante due au meilleur ordre des valeurs. A tout le moins, il faut que la taille de l'échantillon soit suffisamment grande pour que l'ordre des valeurs soit strictement meilleur que celui de MP. La principale difficulté est de trouver la taille adéquate. Si elle est trop petite, l'ordre des valeurs sera moins bon que celui de MP et la recherche sera moins efficace. Si elle est trop grande, le temps du précalcul risque de dépasser la réduction du temps de recherche.

4.2 L'algorithme de recherche à focalisation progressive

Au lieu d'effectuer un échantillonnage complet d'abord et la recherche arborescente ensuite, nous proposons de mélanger l'échantillonnage et la recherche. On élude ainsi le problème de l'optimalité de la taille de l'échantillon. L'idée de base est de mettre à jour chaque longueur moyenne de branche pendant la recherche arborescente. Le nouveau critère est alors $D_i(b_i)$, la longueur moyenne des b_i branches générées dans le sous-arbre i avant et pendant la recherche. Le sous-arbre i tel que $D_i(b_i) = max_j(\{D_j(b_j)\})$ est exploré jusqu'à ce que $D_i(b_i)$ devienne inférieur à un autre $D_j(b_j)$ ou que le sous-arbre i soit complètement exploré. Puis, la recherche continue dans le sous-arbre contenant le plus grand $D_i(b_i)$. L'avantage de continuer à mettre à jour $D_i(b_i)$ pendant la recherche est de permettre à l'algorithme de "changer d'avis" s'il s'aperçoit que le sous-arbre qu'il a choisi s'avère moins prometteur qu'il le semblait auparavant. Quand b_i croît, chaque $D_i(b_i)$ devient de plus en plus stable et la recherche finit par rester coincée dans un sous-arbre jusqu'à son exploration complète. Ce mécanisme permet de réduire la phase d'échantillonnage, laquelle n'a plus besoin d'atteindre un ordre des valeurs de qualité suffisante avant de commencer la recherche. Il faut noter qu'il y a une différence entre calculer $D_i(b_i)$ en choisissant des branches au hasard et le calculer en générant des branches durant la recherche arborescente. La longueur moyenne des branches générées pendant une recherche arborescente est biaisée à cause de l'heuristique de choix de valeur qui tend à choisir les branches les plus longues en premier. Evidemment, au fur et à mesure que la recherche progresse, la moyenne des longueurs mesurées tend vers la vraie valeur. Néanmoins, les valeurs $D_i(b_i)$ du sous-arbre choisi après l'échantillonnage vont avoir tendance à croître artificiellement en début de recherche, maintenant peut-être la recherche coincée dans un sous-arbre qui ne le mérite pas.

L'algorithme effectuant cette recherche est facile à définir. Une fois que chacun des nœuds de profondeur d a été généré, nous créons une pile pour chaque nœud et nous l'y empilons. Ainsi, nous pouvons effectuer une recherche en profondeur d'abord en utilisant la pile qui correspond au plus grand $D_i(b_i)$. Après chaque échec dans le sous-arbre i, la valeur courante de $D_i(b_i)$ est mise à jour : si elle n'est plus la plus grande, alors on change pour la pile qui a la valeur la plus grande.

Une idée plus radicale est d'abandonner la phase d'échantillonnage et de commencer la recherche en l'entrelaçant parmi tous les sous-arbres à la profondeur d. Pendant la recherche, puisque la moyenne de longueur des branches deviendra de plus en plus significative, nous pourrons progressivement altérer cet entrelacement en nous focalisant peu à peu sur les sous-arbres les plus prometteurs : le sous-arbre présentant la moyenne de longueur de branche la plus grande sera visité plus souvent jusqu'à ce qu'on retombe graduellement dans le mécanisme précédent, où la recherche est maintenue dans un sous-arbre tant qu'il a la plus grande valeur de $D_i(b_i)$. Ce nouvel algorithme est le même que celui que nous avons explicité ci-dessus, sauf qu'à la place de $D_i(b_i)$, la nouvelle mesure permettant de changer de sous-arbre est $E_i(b_i) = \frac{K.n+\sum_{j=1}^{j=b_i} D_{i,j}}{K+b_i}$, où K est une constante à fixer et n est le nombre de variables du CSP. $E_i(b_i)$ est une moyenne biaisée dans la mesure où elle calcule la longueur moyenne des branches comme si une phase d'échantillonnage en avait généré K de longueur n. Comme nous l'expliquerons en détail plus bas, l'effet engendré par la mesure de $E_i(b_i)$ à la place de $D_i(b_i)$ est un entrelacement artificiel et temporaire de la recherche sur les différents sous-arbres à sélectionner. Cet algorithme, que nous appelons recherche à focalisation progressive (en anglais : Progressive Focusing Search, PFS), est explicité en fig. 1. Notons que le problème pour calculer la moyenne du mélange entre des branches générées au hasard d'une part et avec heuristique d'autre part a disparu : seules les longueurs des branches les plus à gauche de chaque sous-arbre sont prises en compte, ce qui rend la comparaison entre leurs moyennes équitable.

Voici les propriétés de la mesure $E_i(b_i)$ et leurs conséquences sur la recherche arborescente :

- Comme $\forall D_{i,b_i} : D_{i,b_i} \leq n$, nous avons $\forall b_i : E_i(b_i) > D_i(b_i)$.
  $E_i(0) = n$ et $E_i(b_i)$ converge vers $D_i(b_i)$ quand b_i croît et dépasse largement K. Cela signifie que l'exploration du sous-arbre i fait augmenter b_i et diminuer $E_i(b_i) - D_i(b_i)$. Donc, $E_i(b_i)$ a tendance à diminuer même si $D_i(b_i)$ reste constant. Ceci fait alterner la recherche entre les sous-arbres de manière artificielle.

```
PFS(d):
Insérer les noeuds de profondeur d dans la file de priorité Q
s ← enlève_max(Q)
TANTQUE sommet(pile[s]) n'est pas une solution
   noeud ← dépile(pile[s])
   SI noeud = Echec
      mettre à jour E_s
      insérer(s, Q)
      s ← enleve_max(Q)
   SINON
      empile(les_fils(noeud), pile[s])
   SI pile[s] = ∅
      SI Q = ∅
         retourne Echec
      SINON
         s ← enlève_max(Q)
retourne sommet(pile[s])
```

FIG. 1 – *Recherche à focalisation progressive. Les sous-arbres à la profondeur d sont ordonnés dans la file de priorité Q en fonction de leur valeur E_s.* les_fils(noeud) *retourne la liste des fils de* noeud.

- $E_i(b_i+1) = \frac{K.n+\sum_{j=1}^{j=b_i+1} D_{i,j}}{K+b_i+1} = \frac{K+b_i}{K+b_i+1} E_i(b_i) + \frac{D_{i,b_i+1}}{K+b_i+1}$.

  Ainsi, $E_i(b_i+1) < E_i(b_i)$ si et seulement si $E_i(b_i) > D_{i,b_i+1}$.
  En début de recherche, si $K \gg b_i$ alors $K.n \gg \sum_{j=1}^{j=b_i} D_j$ et $E_i(b_i) \simeq n > D_{i,b_i+1}$. Donc, $E_i(b_i)$ va diminuer presque toujours et on changera souvent de sous-arbre. Après un certain temps, quand $K \ll b_i$, nous aurons $K.n \ll \sum_{j=1}^{j=b_i} D_j$ et $E_i(b_i) \simeq D_i(b_i)$. Donc, $E_i(b_i)$ décroîtra si et seulement si $D_{i,b_i+1} < D_i(b_i)$. Cela signifie que $E_i(b_i)$ est plus susceptible de diminuer en début de recherche que plus tard. Ainsi, l'entrelacement sera progressivement atténué tout au long de la recherche.
- Si $K = 0$ alors $E_i(b_i) = D_i(b_i)$. Si $K \gg 1$ alors $E_i(b_i) \simeq n$. Le réglage de K permet donc de changer la vitesse de convergence de $E_i(b_i)$ vers $D_i(b_i)$, c'est-à-dire, la vitesse de focalisation vers le sous-arbre présentant la plus grande valeur de $D_i(b_i)$.

4.3 Comparaison avec IDFS

PFS peut être comparée avec la recherche en profondeur d'abord entrelacée (en anglais :Interleaved Depth First Search, IDFS)[Mes97]. IDFS a été définie afin de prendre en compte la qualité croissante de l'heuristique de choix de valeur en fonction de la profondeur. Son comportement peut être opposé à celui de DFS. Quand elle rencontre un échec, DFS revient d'abord sur le *dernier* choix alors que IDFS revient sur le *premier* choix. La justification de ce comportement est simple : les choix effectués à de petites profondeur sont moins sûrs donc ce sont les plus profonds qui doivent en priorité être conservés. Il y a deux versions de IDFS. IDFS pur applique exactement le comportement que nous venons de présenter. L'arbre de recherche est exploré de telle manière que, quelle que soit la profondeur, aucun sous-arbre ne sera revisité avant que tous les autres de même profondeur aient été visités aussi. IDFS limité restreint l'entrelacement à une profondeur fixée et à un nombre limité de sous-arbres. Il simule exactement l'ordre d'examen des branches qui serait effectué par une recherche arborescente parallèle, la profondeur d'entrelacement et le nombre de sous-arbres actifs correspondant respectivement à la profondeur de distribution et au nombre de processus engagés. Dans le reste de l'article, lorsque nous nous référerons à IDFS, il s'agira implicitement de sa version limitée.

Au début de la recherche, PFS se comporte en quelque sorte comme IDFS, dans le sens où il génère une branche dans chaque sous-arbre et n'en générera pas une autre dans le même avant qu'il ait visité tous les autres sous-arbres. Quand la recherche arborescente commence, chaque $E_i(0) = n$. Ainsi, le premier sous-arbre est sélectionné, une de ses branches est générée jusqu'à la profondeur $D_{1,1}$ (où un échec survient) et $E_1(1)$ prend une valeur inférieure à n. En conséquence, la recherche doit être continuée dans le second sous-arbre et le processus se répète jusqu'à ce qu'une branche ait été générée dans chacun des sous-arbres. Ensuite, le sous-arbre j ayant la plus grande valeur de $E_j(1)$ est sélectionné et $E_j(2)$ est d'autant plus susceptible d'être la plus petite valeur que K est grand.

Quand $K \gg b_i$, si $b_i > b_j$, $E_i(b_i)$ est souvent inférieur à $E_j(b_j)$.
Si $b_i = b_j$, $(E_i(b_i) > E_j(b_j)) \equiv (D_i(b_i) > D_j(b_j))$. Donc, PFS avec un grand K entrelacera autant la recherche que IDFS mais pas exactement dans le même ordre : un sous-arbre qui aura généré moins de branches que les autres aura plus tendance a être choisi avant ces derniers mais si lui et les autres ont généré le même nombre de branches alors c'est celui qui aura la plus grande valeur de $D_i(b_i)$ qui sera choisi. Une PFS avec un K très grand peut être vue comme une IDFS qui réordonne les sous-arbres selon leur valeur de $D_i(b_i)$ après avoir généré une branche dans chacun d'eux. Cela entraîne qu'une PFS avec un très grand K devrait générer un petit peu moins de branches et de nœuds qu'une IDFS. Maintenant, nous allons voir pourquoi une valeur de K plus faible et une focalisation progressive peuvent conduire à de meilleures performances.

PFS peut être vue comme une recherche arborescente dont le comportement imite IDFS au début mais qui se focalise progressivement sur le sous-arbre le plus prometteur, imitant ainsi une DFS après un certain temps. Il a été montré en théorie et en pratique [Mes97, MW98, PN99] que IDFS pouvait avoir de meilleures performances que DFS lorsque la qualité de l'heuristique de choix de valeur augmentait avec la profondeur, ce qui est le cas de l'heuristique dynamique des solutions potentielles (MP). Si nous décidons d'utiliser l'heuristique de longueur moyenne des branches (LMB) pour les d premières variables et MP pour les $n - d$ variables restantes, alors :

– Au début de la recherche, LMB est pauvrement informée et incertaine donc la qualité de MP la dépasse. A ce stade de la recherche, il est donc fondé d'utiliser IDFS plutôt que DFS.
– Les estimations de LMB convergeant vers de vraies valeurs, il y aura un moment dans la recherche où la qualité de LMB dépassera celle de MP. Alors, la qualité de l'heuristique de choix de valeur MP utilisée à la profondeur $d + 1$ ne sera plus meilleure que celle, LMB, utilisée entre les profondeurs 1 et d. Il deviendra néfaste d'entrelacer la recherche à la profondeur d. Ainsi, l'utilisation de DFS donnera de meilleurs résultats à partir de ce moment là.

Au lieu d'un changement progressif de IDFS vers DFS, nous pourrions envisager un changement brusque dès que LMB deviendrait meilleur que MP. Malheureusement,

	(0.2, 0.09)	(0.2, 0.1)	(0.2, 0.105)	(0.9, 0.02)	(0.8, 0.025)
DFS	317	4218	11835	3417	9813
IDFS(1)	376	4349	9595	2076	9641
$\text{PFS}_{1000}(1)$	314	4315	9561	2038	7446
$\text{PFS}_{10}(1)$	242	4418	8668	2298	7258
$\text{PFS}_{1}(1)$	**237**	4854	7998	2875	7665
IDFS(2)	345	3818	6860	1676	4962
$\text{PFS}_{1000}(2)$	334	**3781**	6755	1561	4721
$\text{PFS}_{10}(2)$	331	3923	**5408**	**1220**	**3956**
$\text{PFS}_{1}(2)$	333	4111	5662	1262	4790

TAB. 3 – *Expérimentation avec des problèmes aléatoires de 100 variables, de taille de domaine 4 et différents paramétrages pour(p_1,p_2). Nous avons toujours indiqué le temps* total *en secondes des 100 instances de problèmes aléatoires.*

nous n'avons pas d'élément nous permettant de savoir quand aura lieu ce basculement. C'est pourquoi un changement progressif est un moyen d'obtenir un algorithme stable. L'efficacité de PFS repose sur la vitesse de focalisation. En conséquence, il faudra prendre soin du réglage de K.

5 Résultats expérimentaux

Les expérimentations ont été effectuées avec des CSP aléatoires tels que définis dans [Pro94]. Les quatre paramètres du CSP aléatoire (n, le nombre de variables, m, la taille de chaque domaine, $p1$, la densité du graphe de contraintes et $p2$, la dureté des contraintes) ont été choisis de façon à rester à gauche du pic de complexité. Pour chaque paramétrage, nous avons exécuté différents algorithmes sur les mêmes 100 instances de problèmes dont chacune avait au moins une solution. Tous les algorithmes maintenaient la cohérence d'arc, utilisaient l'heuristique de choix de variable dynamique *min-domain* et l'heuristique de choix de valeur MP. Les algorithmes étaient : DFS, IDFS(d) et $\text{PFS}_K(d)$, où d est la profondeur d'entrelacement. Chaque algorithme a été testé avec $d = 1$ et $d = 2$ et K a été réglé sur 1, 10 et 1000. La profondeur d'entrelacement est nécessairement faible car le nombre de sous-arbre croît exponentiellement avec la profondeur. Pour K, les 3 valeurs choisies correspondent à sa valeur minimale (1), une valeur suffisamment grande pour qu'elle soit largement supérieure au nombre de branches générées dans chaque sous-arbre (1000) et à une valeur intermédiaire (10) fixée arbitrairement qui sert à savoir si un entrelacement progressif est meilleur que pas ou peu d'entrelacement ($K = 1$) ou un entrelacement perpétuel ($K = 1000$). Les résultats sont présentés dans le tableau 3.

Nous pouvons remarquer que :

- IDFS a eu de meilleures performances que DFS sur les problèmes difficiles.
- Comme prévu, PFS_{1000}(d) a toujours été légèrement meilleur que IDFS(d).
- La profondeur 2 a été meilleure que la profondeur 1 pour l'entrelacement lorsque les problèmes sont devenus plus difficiles.
- PFS a surpassé d'autant plus DFS (jusqu'à 64% de temps gagné) et IDFS (jusqu'à

27% de temps gagné) que les problèmes ont été difficiles.

Pour les problèmes faciles, le nombre de retours-arrière obtenu avec $d = 2$ était toujours inférieur à celui avec $d = 1$; par contre, le nombre total de nœuds générés était lui toujours supérieur avec $d = 2$. Ceci peut être expliqué par les raisons suivantes. Quand les problèmes sont faciles, les retours-arrière ne surviennent qu'à des niveaux assez profonds et la recherche oscille dans un intervalle de profondeurs assez restreint. Avec une DFS, la première branche coûte la génération des nœuds de la racine jusqu'à un niveau assez profond tandis que les branches suivantes comportent beaucoup moins de nœuds. L'entrelacement provoque la génération de toutes les premières branches (longues) de chaque sous-arbres au début de la recherche. C'est pourquoi, à nombre de branches égales, DFS a généré moins de nœuds que IDFS pendant le début de la recherche. Comme le nombre de branches générées pour les problèmes faciles est peu élevé, il ne suffit pas à IDFS ou PFS de générer moins de branches quand $d = 2$ pour faire mieux qu'avec $d = 1$.

Les performances d'IDFS par rapport à DFS sur les problèmes difficiles confirment qu'éviter de choisir entre les valeurs des premières variables grâce à l'entrelacement est une bonne option quand la qualité de l'heuristique de choix de valeur augmente avec la profondeur. Les performances de PFS par rapport à IDFS montrent bien qu'il existe un moment de la recherche où il faut se focaliser sur un sous-arbre plutôt que continuer à les entrelacer car LMB sur les d premiers choix est devenue meilleure que MP qui est utilisée sur les choix suivants.

6 Conclusion

Nous avons étudié les effets de plusieurs heuristiques de choix de valeur dans le cadre de la recherche arborescente où la cohérence d'arc est maintenue. Nous avons observé expérimentalement que la version dynamique de l'heuristique des solutions potentielles avait en général de meilleures performances que sa version statique quand les problèmes avaient beaucoup de solutions. Cependant, pour des problèmes difficiles proches du pic de complexité, aucune version de l'heuristique n'améliorait subtantiellement une recherche sans heuristique. Ceci nous a conduits à essayer une heuristique de choix de valeur potentiellement meilleure pour les premières variables, car ces premiers choix sont cruciaux pour les problèmes difficiles. Cette heuristique fait une estimation de la longueur moyenne des branches (LMB) dans un sous-arbre. Nous avons défini un nouvel algorithme, la recherche à focalisation progressive, qui améliore les estimations en cours de recherche. Il commence par entrelacer la recherche dans différents sous-arbres afin d'obtenir des estimations valables et se focalise progressivement vers le sous-arbre le plus prometteur. Nos expérimentations montrent un gain de temps conséquent sur des problèmes difficiles par rapport à DFS et IDFS lorsque ces dernières n'utilisent que l'heuristique des solutions potentielles

D'un point de vue plus général, nos résultats montrent que les méthodes de recherche arborescente doivent être adaptées aux heuristiques de choix de valeurs qu'elles utilisent. C'est à l'heuristique de choix de valeur de commander le choix du type de recherche arborescente et non l'inverse. DFS est adapté aux heuristiques de choix de valeur statiques puisque celles-ci ne s'améliorent pas quand la profondeur augmente. Par contre, IDFS devient un bon choix quand on dispose d'une heuristique dynamique dont la qualité augmente avec la profondeur. Quand une heuristique de choix de valeur avec apprentissage comme LMB est utilisée, la recherche doit commencer comme une IDFS et se terminer comme une DFS.

Même si les valeurs de K pour PFS que nous avons testées ont toutes conduit à une amélioration par rapport à DFS et IDFS sur des problèmes difficiles, le choix de la valeur de K mène à des performances assez inégales. Il serait donc très utile de trouver un moyen de fixer K à sa meilleure valeur possible de façon automatique. Nous envisageons donc d'étudier cet aspect soit en calculant K a priori selon les caractéristiques intrinsèques du CSP, soit en ajustant K en cours de recherche suivant des critères qui restent à déterminer.

Références

[BR96] C. Bessière and J.C. Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *Proc of CP'96*, volume 1113 of *LNCS*, pages 61–75, 1996.

[CVMB96] B. Cabon, G. Verfaillie, D. Martinez, and P. Bourret. Using Mean Field Methods for Boosting Backtrack Search in Constraint Satisfaction Problems. In *Proceedings of the* 12^{th} *European Conference on Artificial Intelligence*, pages 165–169, 1996.

[DP88] R. Dechter and J. Pearl. Network–based heuristics for constraint–satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[FD95] D. Frost and R. Dechter. Look-ahead Value Ordering for Constraint Satisfaction Problems. In *Proceedings of the* 14^{th} *International Joint Conference on Artificial Intelligence*, 1995.

[Fro97] Daniel H. Frost. *Algorithms and Heuristics for Constraint Satisfaction Problems.* PhD thesis, University of California - Irvine, 1997.

[Gee92] P. A. Geelen. Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems. In *Proceedings of the* 10^{th} *European Conference on Artificial Intelligence*, 1992.

[HE80] R. M. Haralick and G. L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.

[Hul99] T. Hulubei. The CSP Library. http://www.cs.unh.edu/˜tudor/csp/, University of New Hampshire, 1999.

[Knu75] D. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129):121–136, 1975.

[LL98] L. Lobjois and M. Lemaître. Branch and Bound Algorithm Selection by Performance Prediction. In *Proc. of the 15th National Conference on Artificial Intelligence, AAAI'98*, pages 353–358, 1998.

[LM95] J. Larrosa and P. Meseguer. Optimization-based Heuristics for Maximal Constraint Satisfaction. In *Principles and Practice of Constraint Programming CP'95*, volume LNCS 976, pages 103–120, 1995.

[Mes97] P. Meseguer. Interleaved Depth-First Search. In *Proceedings of the* 15^{th} *International Joint Conference on Artificial Intelligence*, pages 1382–1387, 1997.

[MW98] P. Meseguer and T. Walsh. Interleaved and Discrepancy Based Search. In *Proceedings of the* 13^{th} *European Conference on Artificial Intelligence*, 1998.

[PN99] N. Prcovic and B. Neveu. Ensuring a Relevant Visiting Order of the Leaf Nodes during a Tree Search. In *Principles and Practice of Constraint Programming, CP'99*, volume LNCS 1713, pages 361–374, 1999.

[Pro94] P. Prosser. Binary Constraint Satifaction Problems : Some are harder than others. In *11th European Conference on Artificial Intelligence*, pages 95–99, 1994.

[SD96] B. M. Smith and M. E. Dyer. Locating the Phase Transition in Binary Constraint Satisfaction Problem. *Artificial Intelligence*, 81:155–181, 1996.

[SF94] D. Sabin and E. C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of the* 10^{th} *European Conference on Artificial Intelligence*, pages 125–129, 1994.

[SG98] B. M. Smith and S. A. Grant. Trying Harder to Fail First. In *Proceeedings of the* 13^{th} *European Conference on Artificial Intelligence*, 1998.