Heuristiques de choix de voisinage pour les recherches à voisinage variable dans les WCSP

Nicolas Levasseur, Patrice Boizumault, Samir Loudni

Heuristiques de choix de voisinage pour les recherches à voisinage variable dans les WCSP

Nicolas Levasseur **Patrice Boizumault** **Samir Loudni**

GREYC, UMR60-72
Campus Côte de Nacre, boulevard du Maréchal Juin, BP 5186
14032 CAEN Cedex
{nicolas.levasseur,patrice.boizumault}@info.unicaen.fr loudni@iutc3.unicaen.fr

Résumé

Le formalisme des Weighted CSP [5, 11] est un cadre général pour modéliser et résoudre des problèmes d'optimisation sous contraintes. VNS/LDS+CP [6] est une méthode hybride basée sur un schéma de recherche locale à voisinages de taille variable (VNS) [7]. Dans ce type de méthodes, le choix des voisinages à explorer joue un rôle primordial. Les très rares heuristiques développées dans le cadre CSP, comme `ConflictVar`, s'appuient sur les variables en conflit. Dans cet article, nous proposons de nouvelles heuristiques de choix de voisinage pour les WCSPs. Ces heuristiques, outre la notion de conflit, exploitent à la fois la topologie du graphe de contraintes et les poids qui leur sont associés. Les expérimentations réalisées avec VNS/LDS+CP montrent que nos heuristiques guident plus efficacement la recherche que `ConflictVar`.

1 Introduction

Le formalisme des Weighted CSP [5, 11] est un cadre général pour modéliser des problèmes d'optimisation sous contraintes. Les WCSP sont résolus soit par des méthodes arborescentes, des méthodes locales ou des méthodes hybrides. Ces dernières, en combinant les avantages des deux autres approches, offrent un bon compromis entre le temps de résolution et la qualité des solutions générées. VNS/LDS+CP [6] est une méthode hybride basée sur un schéma de recherche locale à voisinages de taille variable (VNS) [7] et dont l'exploration est effectuée par une recherche arborescente tronquée de type LDS [3]. Dans ces méthodes, les heuristiques de choix de voisinage jouent un rôle primordial dans la recherche en déterminant les sous-espaces les plus prometteurs à explorer. A notre connaissance, les très rares heuristiques développées dans le cadre CSP, comme par exemple `ConflictVar`, s'appuient sur les variables en conflit (une variable est en conflit si et seulement si elle apparaît dans au moins une contrainte insatisfaite). Dans cet article, nous proposons de nouvelles heuristiques de choix de voisinage pour les WCSPs. Ces heuristiques, outre la notion de conflit, exploitent à la fois la topologie du graphe de contraintes et les poids qui leur sont associés. Les expérimentations réalisées avec VNS/LDS+CP sur les instances réelles (CELAR) et structurées générées aléatoirement (GRAPH) montrent que nos heuristiques guident plus efficacement la recherche que `ConflictVar`. Après avoir brièvement rappelé le cadre des WCSP (Section 2), nous présentons les méthodes de recherche hybrides et décrivons la problématique du choix de voisinage (Section 3). Les nouvelles heuristiques de choix de voisinage que nous proposons sont décrites en Section 4. Enfin, les résultats des expérimentations effectuées sont décrits en Section 5.

2 Problème de Satisfaction de Contraintes Pondérées

Un WCSP (Weighted Constraint Sastisfaction Problem) est défini par un quadruplet $(\mathcal{X}, \mathcal{D}, \mathcal{C}, k_\top)$, avec $\mathcal{X} = \{x_1, ..., x_n\}$ l'ensemble des variables (de taille n), $\mathcal{D} = \{d_1, ..., d_n\}$ les domaines finis associés (la taille maximale des domaines est notée d) et $\mathcal{S}(k_\top)$ une structure de violation. $\mathcal{S}(k_\top)$ est le triplet ([0,1,...,$k_\top$], $\oplus, \geq$) où $k_\top$ est un entier naturel dans $[1, .., \infty]$, $\oplus$ est défini par : $a \oplus b = \min(k_\top, a + b)$ et $\geq$ est la relation d'ordre total définie sur les entiers. $\mathcal{C}$ est l'ensemble (de taille e) des contraintes. Chaque contrainte $c \in \mathcal{C}$, est définie pour un sous-ensemble $\mathcal{X}_c \subseteq \mathcal{X}$ de variables

liées à la contrainte. $\mathcal{A}_{\downarrow\mathcal{X}_c}$ est l'ensemble des affectations de ces variables. A chaque $c \in \mathcal{C}$ est associée une fonction $f_c : \prod_{x_i \in \mathcal{X}_c} d_i \mapsto [0, k_\top]$, qui retourne 0 quand c est satisfaite. Une affectation de x_i à la valeur a est notée : $(x_i = a)$. Une affectation *complète* (ou solution) $\mathcal{A} = \{a_1, ..., a_n\}$ est une affectation de toutes les variables ; à l'inverse, nous parlons d'affectation *partielle*. Le coût d'une affectation complète $\mathcal{A} = \{a_1, ..., a_n\}$ est noté : $\mathcal{V}(A) = \sum_{c \in \mathcal{C}} f_c(\mathcal{A}_{\downarrow\mathcal{X}_c})$. L'objectif est de trouver une affectation complète de coût minimal : $\min_{\mathcal{A} \in d_1 \times d_2 \times ... \times d_n} \mathcal{V}(\mathcal{A})$.

3 Méthodes hybrides

Les méthodes de recherche arborescentes sont souvent utilisées pour obtenir des solutions optimales et effectuer des preuves d'optimalité. Mais en raison de l'espace de recherche qui est souvent trop grand pour être exploré entièrement, ces méthodes peuvent s'avérer trop gourmandes en temps de calcul.

Grâce à leur façon opportuniste d'explorer l'espace de recherche, les recherches locales peuvent produire des solutions de bonne qualité en des temps de calcul raisonnables. Malheureusement, ces méthodes ne peuvent garantir l'optimalité des solutions obtenues et ne sont pas toujours capables de sortir facilement des optima locaux.

Les algorithmes hybrides [2] offrent un réel compromis entre ces deux approches. Plus précisément, ils sont capables de combiner efficacement les avantages de la *propagation de contraintes* (associée généralement aux méthodes de recherche complète) avec *l'exploration opportuniste* des recherches locales.

Les *hybridations imbriquées*, où recherche complète et recherche locale sont étroitement liées durant la résolution, sont de loin les hybridations les plus couramment utilisées et les plus fructueuses. La première catégorie de ces hybridations appartient à la famille des recherches locales et utilise des mécanismes complets pour rendre les *voisinages étendus* (LNS [12]) plus attractifs. La seconde catégorie appartenant à la famille des recherches complètes utilise des mécanismes issus des recherches locales pour améliorer les solutions partielles à des points de choix de l'arbre de recherche [10].

3.1 VNS/LDS+CP

VNS/LDS+CP [6] est une *hybridation imbriquée* de la famille des recherches locales. Nous avons utilisé une *recherche à voisinages variables* (VNS), qui généralise le principe des voisinages étendus (LNS) en *ajustant dynamiquement* la taille du voisinage, chaque fois que la solution courante est un *optimum local*. L'algorithme 1 décrit VNS/LDS+CP.

Algorithm 1: VNS/LDS+CP

```
   function VNS(X, C, k_init, k_max, δ_max)
   begin
1      s ← genRandomSol(X)
2      k ← k_init
3      iter ← 1
4      while (k < k_max) ∧ (not TimeOut) do
5          X_unaffected ← Hneighborhood(N_k, s)
6          A ← s\{(x_i = a) s.t. x_i ∈ X_unaffected}
7          s' ← NaryLDS(A, X_unaffected, δ_max, V(s), s)
8          if V(s') < V(s) then
9              s ← s'
10             k ← k_init
11         else k ← k + 1
12     return s
   end
```

L'algorithme part d'une solution initiale s générée aléatoirement. Un sous-ensemble de k variables (avec k la dimension du voisinage) sont sélectionnées dans le voisinage N_k (i.e. l'ensemble des combinaisons de k variables parmi $\mathcal{X}$)(ligne 5). Une affectation partielle $\mathcal{A}$ est alors générée à partir de la solution courante s en désaffectant les k variables sélectionnées ; les autres variables (i.e. non sélectionnées) gardent leur affectation dans s (ligne 6). $\mathcal{A}$ est alors reconstruite en utilisant une recherche arborescente tronquée de type LDS aidée par une propagation de contraintes (CP) basée sur un calcul de minorants. Si LDS trouve une solution voisine s' de meillleure qualité que la solution courante s (ligne 8), alors celle-ci devient la solution courante et k est réinitialisée à k_{init} (lignes 9-10). Sinon, k est incrémentée de 1 afin de s'échapper de ce minimum local (ligne 11). En effet, plus la dimension du voisinage est grande, plus l'espace de recherche est grand et a des chances de contenir de bonnes solutions permettant d'améliorer la solution courante. Toutefois, l'exploration de (très) grands voisinages peut rapidement devenir prohibitive en temps. C'est pour cette raison que nous avons retenu LDS. L'algorithme s'arrête dès que l'on a atteint la dimension maximale du voisinage à considérer k_{max} ou le TimeOut (ligne 4).

L'efficacité de VNS/LDS+CP dépend fortement de la gestion du voisinage par VNS, du choix des variables à désaffecter et de l'algorithme de reconstruction de ces variables. Dans [6], les auteurs ont montré l'intérêt d'utiliser la recherche de type VNS par rapport à LNS, ainsi que l'importance de la phase de reconstruction à base de LDS+CP.

3.2 ConflictVar

L'heuristique de choix de voisinage joue un rôle primordial dans la recherche puisqu'elle détermine les sous-espaces de recherche à explorer afin de trouver

Algorithm 2: ConflictVar

```
   function ConflictVar (A, X, C, k)
   begin
1      X_unaffected ← ∅
2      X_conflicted ← getConflict(A, C)
3      while #(X_unaffected) ≠ k do
4          if X_conflicted ≠ ∅ then
5              x ← randomPick(X_conflicted)
6              X_conflicted ← X_conflicted \ {x}
7          else x ← randomPick(X \ X_unaffected)
8          X_unaffected ← X_unaffected ∪ {x}
9      return X_unaffected
   end
```

des solutions de meilleure qualité. A notre connaissance, peu d'heuristiques indépendantes du problème existent. Parmi celles-ci nous pouvons citer `PGLNS` [9], qui en se basant sur la propagation et les conséquences des filtrages, permet d'éviter de sélectionner des variables dont les valeurs vont être déduites à partir des variables restant affectées ; et `ConflictVar` définie dans le cadre de la satisfaction, qui est une heuristique de choix de voisinage basée sur la notion de conflit ; une variable est en conflit lorsqu'elle figure dans au moins une contrainte violée.

Pour une dimension de voisinage k donnée, `ConflictVar` sélectionne aléatoirement k variables à désaffecter parmi celles en conflit ($\mathcal{X}_{conflicted}$), puis parmi celles qui ne le sont pas (si $k > \#(\mathcal{X}_{conflicted})$). Ce mécanisme essentiellement basé sur un choix aléatoire des variables à désaffecter, permet une meilleure diversité de la recherche mais aussi de s'échapper rapidement des minima locaux. Le pseudo-code de `ConflictVar` est détaillé dans l'algorithme 2 ; la fonction `getConflict` retourne les variables en conflit.

`ConflictVar` est une heuristique simple et facile à mettre en œuvre car indépendante du problème à résoudre. Néanmoins, celle-ci présente deux inconvénients majeurs : elle ne tient pas compte de la topologie du graphe de contraintes, ainsi que des poids (i.e. importance) de ces dernières. En effet, considérons l'exemple de la Figure 1 qui représente le graphe des contraintes de la scène 6 du CELAR, et supposons que $k = 8$, $\mathcal{A}$ soit une affectation complète et que les contraintes en pointillés soient satisfaites, les autres étant violées. `ConflictVar` pourrait choisir les variables grisées à désaffecter. Or, aucune des contraintes portant sur ces variables n'est complètement désaffectée, et comme ces variables ont des degrés (nombre de contraintes auxquelles elles participent) importants, il paraît difficile de les réaffecter (ou reconstuire) sans générer beaucoup d'inconsistances, et donc de trouver une meilleure solution.

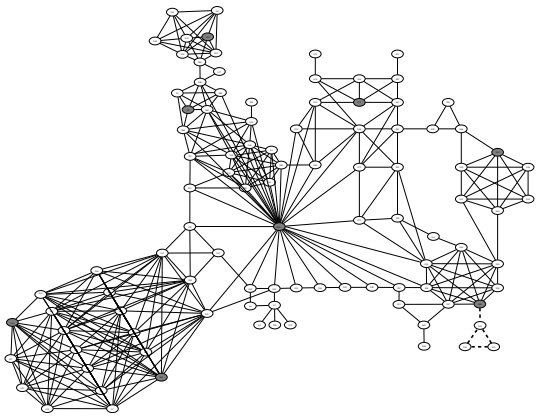

FIG. 1 – Instance Scen06 du CELAR.

Algorithm 3: N-ary optimisation LDS

```
   function NaryLDS (A, X_f, δ, UB, bestS)
   begin
1      if X_f = ∅ then
2          UB ← V(A)
3          return A
4      x_i ← select-variable (X_f)
5      j ← 0
6      while (d_i ≠ ∅) ∧ (j ≤ δ) do
7          a ← select-value (d_i)
8          if Filtering (A ∪ {(x_i = a)}, X_f \ {x_i}, UB) then
9              bestS ← NaryLDS(A ∪ {(x_i =
                   a)}, X_f \ {x_i}, (δ − j), UB, bestS)
10         cancelFiltering (A, (x_i = a), X_f)
11         d_i ← d_i \ {a},
12         j ← j + 1
13     return bestS
   end
```

3.3 Limited Discrepancy search

LDS (Limited Discrepancy Search) est une méthode de recherche partielle arborescente introduite par Harvey et Ginsberg [3] permettant de résoudre itérativement les problèmes de satisfaction de contraintes binaires. Soit h une heuristique dans laquelle on a une grande confiance, le principe de LDS est de suivre l'heuristique h lors du parcours de l'arbre de recherche, mais en considérant que h peut se tromper un petit nombre (δ_{max}) de fois. On s'autorise donc δ_{max} écarts (*discrepancies*) à l'heuristique h lors du parcours de l'arbre de recherche. Celui-ci est parcouru itérativement pour des valeurs croissantes de discrepancies. Dans [6], LDS a été étendu aux problèmes d'optimisation n-aires et n'effectue que la dernière itération (avec le nombre de discrepancies autorisées égal à la valeur maximale). Le pseudo-code de l'algorithme 3 détaille cette extension, avec $\mathcal{A}$ une affectation partielle, $\mathcal{X}_f$ les variables futures (c'est à dire, non encore affectées) et δ le nombre restant d'écarts à l'heuristique.

Si toutes les variables sont affectées (ligne 1), on met à jour $\mathcal{UB}$ (le coût de la meilleure solution connue) (ligne 2) et la solution courante devient la meilleure solution connue (lignes 3 et 9). Sinon, on sélectionne la prochaine variable à affecter (ligne 4) ; et soit d_i le domaine de x_i ordonné par h. Dans la boucle while, un écart de j correspond au choix de la $(j+1)$ième valeur. Tant qu'il reste suffisament d'écarts ($j \leq \delta$) et de valeurs dans le domaine d_i ($d_i \neq \emptyset$), la $(j+1)$ième valeur du domaine courant est sélectionnée (ligne 7) et le nombre restant d'écarts autorisés est décrémenté à $(\delta - j)$. Si le filtrage produit aucun domaine vide (ligne 8), on explore alors le sous-espace (ligne 9). Sinon (ou lorsque l'exploration du sous-espace est finie), on effectue un retour-arrière (ou backtrack) en annulant les modifications effectuées lors du filtrage (ligne 10). Puis le domaine de la variable est réduit (ligne 11). La variable est ensuite affectée à la valeur suivante de son domaine réduit (si celui-ci n'est pas vide) (ligne 6).

4 Heuristiques de choix de voisinage

Dans cette section, nous proposons plusieurs nouvelles heuristiques de choix de voisinage étendant `ConflictVar` pour les WCSP. Celles-ci expoitent à la fois la topologie du graphe de contraintes et les poids des contraintes. Pour cela, nous introduisons dans un premier temps, les définitions suivantes.

Définition 1 ***Variables voisines****, soient x et y deux variables, x est dite voisine de y si et seulement si il existe une contrainte portant sur x* ***et*** *sur y. On note $vois(x)$ l'ensemble des variables voisines de x.*

Définition 2 ***Degré d'une variable****, soit x une variable, le degré de x noté $deg(x)$ est égal au cardinal de $vois(x)$. Pour une affectation complète $\mathcal{A}$, on note également $degConflit(x)$ le nombre de variables en conflit voisines de x.*

Définition 3 ***Degré de liberté d'une variable****, soient $\mathcal{A}$ une affectation complète, $\mathcal{X}_{unaffected}$ un ensemble de variables à désaffecter et x une variable incluse dans $\mathcal{X}_{unaffected}$, le degré de liberté de x est égal au nombre de variables voisines de x incluses dans $\mathcal{X}_{unaffected}$. x est dite de degré de liberté maximal, si $vois(x) \subset X_{unaffected}$.*

4.1 Extensions de ConflictVar

Comme nous l'avons mentionné précédemment, `ConflictVar` sélectionne aléatoirement un sous-ensemble de variables en conflit sans tenir compte de la topologie du graphe de contraintes. Les variables à reconstruire peuvent alors avoir des degrés de liberté nuls. Dans ce cas de figure, l'algorithme de reconstruction a peu de chances de trouver une nouvelle affectation des variables qui minimise les inconsistances. Or, nous avons constaté que lors de la phase de reconstruction, plus le degré de liberté d'une variable est élevé, plus la méthode de reconstuction avait des chances de la réaffecter en minimisant les inconsistances. Nous proposons donc de nouvelles heuristiques exploitant la topologie du graphe de contraintes (i.e. permettant d'augmenter le degré de liberté des variables à reconstuire), et dédiées au cadre des WCSP.

Les figures 2 illustrent plusieurs choix de variables effectués par les heuristiques que nous proposons. Les contraintes en pointillés désignent l'ensemble des contraintes satisfaites par une affectation complète $\mathcal{A}$; les autres contraintes étant violées. La variable en noir dénote le premier choix effectué par chaque heuristique ; les choix suivants sont notés en gris. Pour chaque heuristique, la prochaine variable sera choisie aléatoirement parmi celles en pointillés.

1. `ConflictVar-Connected` : cette heuristique sélectionne la prochaine variable aléatoirement parmi celles en conflit et voisines des variables déjà sélectionnées (cf. figure 2a). La première variable est choisie aléatoirement parmi celles en conflit.

2. `ConflictVar-Star` : cette heuristique sélectionne une première variable x (en noir) dite "variable centre" (cf. figure 2b), puis choisit aléatoirement les prochaines variables parmi celles en conflit dans $vois(x)$ (i.e. en gris). Si toutes celles-ci sont sélectionnées, un centre est de nouveau déterminé parmi les variables en conflit voisines de celles déjà choisies (i.e. en pointillés).

3. `ConflictAndSatVar-Star` : cette heuristique est une extension de `ConflictVar-Star`. Elle élargit, lorsque toutes les variables en conflit voisines du centre ont été sélectionnées, le choix des variables à celles qui ne sont pas en conflit. Sur la figure 2c, la prochaine variable à être sélectionnée est celle en pointillés.

4. `ConflictVar-MaxDeg` : cette heuristique choisit en premier une variable parmi celles en conflit. La prochaine variable est sélectionnée parmi celles (en conflit ou non) ayant le plus de voisins parmi les variables déjà sélectionnées (les égalités sont départagées aléatoirement). Sur la figure 2c, chaque variable en pointillés a deux variables voisines sélectionnées, alors que les autres variables en ont au plus une.

5. `ConflictVar-H-Cost` : cette heuristique tient compte des coûts des contraintes lors de la sélection des variables et sera détaillée ultérieurement.

Remarque : pour simplifier l'écriture des pseudo-codes, nous supposerons que le graphe des contraintes est connexe. Cependant, les algorithmes peuvent facilement être adaptés au cas non connexe.

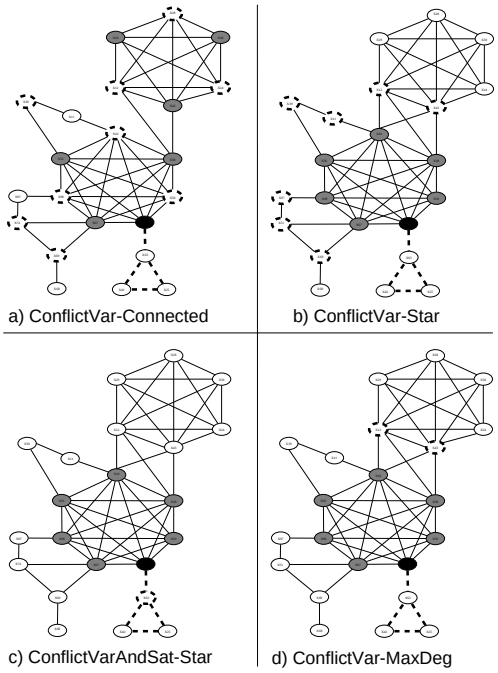

FIG. 2 – Heuristiques de choix de voisinage.

4.2 ConflictVar-Connected

Afin que toutes les variables à désaffecter aient au minimum un dégré de liberté de 1, nous proposons l'heuristique `ConflictVar-Connected`. Son pseudo-code est détaillé dans l'algorithme 4.

L'heuristique `ConflictVar-Connected` retourne un sous-ensemble $\mathcal{X}_{unaffected}$ de k variables à désaffecter. Chaque variable est sélectionnée aléatoirement dans le premier ensemble non vide ci-dessous :

1. l'ensemble $\mathcal{X}_{allowed}$ des variables en conflit et voisines des variables déjà sélectionnées (lignes 5 et 6),

2. l'ensemble des variables en conflit (lignes 7 et 8),

3. l'ensemble des variables du problème non encore choisies (ligne 9).

Lorsqu'une variable x est sélectionnée, elle est ajoutée à $\mathcal{X}_{unaffected}$ (ligne 10), retirée si nécessaire des variables en conflit (ligne 11), et les variables en conflit dans $vois(x)$ sont ajoutées aux variables autorisées à être sélectionnées par la suite ($\mathcal{X}_{allowed}$) (ligne 12).

Notons que le degré de liberté de chaque variable à désaffecter est au minimum 1 (car toutes les variables sont connexes) et au maximum égal à son degré. Cependant, nous avons constaté en pratique que ce maximum avait peu de chances d'être atteint. En effet, la taille de $\mathcal{X}_{allowed}$ grandissant rapidement, les chances de choisir tous les voisins d'une variable diminuent.

Algorithm 4: ConflictVar-Connected

```
   function ConflictVar-Connected (A, X, C, k)
   begin
1      X_unaffected ← ∅
2      X_allowed ← ∅
3      X_conflicted ← getConflict(A, C)
4      while #(X_unaffected) ≠ k do
5          if X_allowed ≠ ∅ then
6              x ← randomPick(X_allowed)
7          else if X_conflicted ≠ ∅ then
8              x ← randomPick(X_conflicted)
9          else x ← randomPick(X \ X_unaffected)
10         X_unaffected ← X_unaffected ∪ {x}
11         X_conflicted ← X_conflicted \ {x}
12         X_allowed ← (X_allowed \ {x}) ∪ (vois(x) ∩ X_conflicted)
13     return X_unaffected
   end
```

4.3 ConflictVar-Star

Nous proposons l'heuristique `ConflictVar-Star` qui est une amélioration de `ConflictVar-Connected`. Cette heuristique permet de maximiser le degré de liberté d'au moins une variable sélectionnée. Son pseudo-code est détaillé dans l'algorithme 5.

La fonction `GetCenter` sélectionne une nouvelle variable "centre" aléatoirement parmi celles en conflit et voisines des variables déjà sélectionnées.

L'heuristique `ConflictVar-Star` sélectionne une première variable x "centre", puis choisit aléatoirement les prochaines variables parmi celles en conflit dans $vois(x)$. Si toutes celles-ci sont sélectionnées, une nouvelle variable "centre" est choisie parmi le premier ensemble non vide :

1. l'ensemble $\mathcal{X}_{allowed}$ des variables en conflit et voisines des variables déjà sélectionnées (ligne 7),

2. l'ensemble $\mathcal{X}_{conflicted}$ des variables en conflit (ligne 9),

3. l'ensemble des variables du problème non encore choisies (ligne 10).

De la même manière que précédemment, les variables voisines du nouveau centre sont autorisées à être sélectionnées par la suite (ligne 11).

Notons que le degré de liberté de la première variable centre à désaffecter x_{c1} est égal à : $max(k-1, degConflit(x_{c1}))$. Celui des variables voisines est toujours compris entre 1 et son degré.

4.4 ConflictVarAndSat-Star

Dans cette section nous discutons de l'intérêt de sélectionner des variables qui ne sont pas en conflit. Soit $\mathcal{A}$ l'affectation complète courante, et supposons que dans le problème représenté sur la figure 2 : la contrainte c en pointillés portant sur x_1 (en noir) et sur

Algorithm 5: ConflictVar-Star

```
   function ConflictVar-Star (A, X, C, k)
   begin
1      X_unaffected ← ∅
2      X_allowed ← ∅
3      X_conflicted ← getConflict(A, C)
4      while #(X_unaffected) ≠ k do
5          if X_allowed ≠ ∅ then
6              x ← randomPick(X_allowed)
           else
7              x ← GetCenter(X_unaffected, X_conflicted)
8              if x = null then
9                  if X_conflicted ≠ ∅ then
                       x ← randomPick(X_conflicted)
10                 else x ← randomPick(X \ X_unaffected)
11             X_allowed ← (vois(x) ∩ X_conflicted)
12         X_unaffected ← X_unaffected ∪ {x}
13         X_conflicted ← X_conflicted \ {x}
14         X_allowed ← X_allowed \ {x}
15     return X_unaffected
   end
   function GetCenter (X_unaffected, X')
   begin
16     X'' ← ∪_{x∈X_unaffected} (vois(x) ∩ X')
17     if X'' ≠ ∅ then return randomPick(X'')
18     return null
   end
```

x_2 soit une contrainte dure d'égalité, et que $(x_1 = a)$ et $(x_2 = a)$. Nous rappelons que les contraintes en pointillés sont satisfaites par l'affectation courante. Une heuristique basée sur la notion de conflit pourrait choisir de désaffecter la variable x_1, mais pas x_2, car celle-ci n'est pas en conflit. Dans ce cas, l'algorithme de reconstruction ne pourra jamais réaffecter x_1 à une autre valeur (puisque c est une contrainte dure). Nous proposons donc une extension `ConflictVarAndSat-Star` de `ConflictVarStar` qui autorise également la sélection des variables qui ne sont pas en conflit.

L'heuristique `ConflictAndSatVarStar`, de la même manière que précédemment choisit aléatoirement une variable "centre" x parmi les variables en conflit, puis autorise les variables en conflit dans $vois(x)$ à être choisies. Si toutes celles-ci sont sélectionnées, l'heuristique sélectionne, avant de choisir un nouveau centre, les prochaines variables parmi celles qui ne sont pas en conflit dans $vois(x)$. A chaque fois qu'un nouveau centre doit être choisi, celui-ci est d'abord déterminé si possible parmi les variables en conflit voisines de celles déjà sélectionnées, et sinon parmi les variables pas en conflit voisines de celles déjà sélectionnées.

Notons que le degré de liberté de la première variables centre est égal : $max(k-1, deg(x_{c1}))$, avec $deg(x_{c1}) \geq degConflict(x_{c1})$. Le degré de liberté des autres variables voisines est toujours compris entre 1 et leur degré.

Algorithm 6: ConflictVar-MaxDeg

```
   function ConflictVar-MaxDeg (A, X, C, k)
   begin
1      Let card un tableau de n cases
2      x ← randomPick(getConflict(A, C))
3      X_unaffected ← {x}
4      foreach x' ∈ vois(x) do card[x'] ← card[x'] + 1
5      while #(X_unaffected) ≠ k do
6          x ← GetVarsMaxCard(X \ X_unaffected, card)
7          X_unaffected ← X_unaffected ∪ {x}
8          foreach x' ∈ vois(x) do
9              card[x'] ← card[x'] + 1
10     return X_unaffected
   end
```

4.5 ConflictVar-MaxDeg

Nous proposons également une heuristique notée `ConflictVar-MaxDeg` qui tente de maximiser le degré de liberté de toutes les variables sélectionnées. Le pseudo-code est détaillé dans l'algorithme 6.

La fonction `GetVarsMaxCard` choisit aléatoirement une variable parmi celles du problème, non encore choisies, ayant le plus de voisins déjà sélectionnés.

L'heuristique `ConflictVar-MaxDeg` sélectionne d'abord aléatoirement une variable x parmi celles en conflit. Puis, pour chaque variable voisine de x, son *card* (nombre de variables voisines déjà sélectionnés) est incrémenté. Les prochaines variables sont alors sélectionnées aléatoirement parmi celles (en conflit ou non) ayant la plus forte valeur de *card*.

Notons que le degré de liberté minimal des variables à reconstruire est toujours égal à 1 dans le pire cas. Cependant, en pratique, cette approche permet d'augmenter le degré de liberté des variables sélectionnées. Toutefois, pour cette heuristique, le biais aléatoire est moins important, car elle est souvent guidée par un critère plus déterministe.

4.6 ConflictVar-H-Cost

Dans le cadre des WCSP, certaines contraintes ayant un coût de violation plus élevé que d'autres, il est naturel de chercher plus intensément à les satisfaire. Néanmoins, il se peut que plusieurs contraintes ayant un poids de violation élevé soient violées dans toutes les solutions optimales. Une heuristique adaptée au cadre des WCSP doit alors tenir compte plus souvent des contraintes violées de fort poids (en désaffectant plus régulièrement ses variables), mais sans toutefois essayer de les satisfaire à tout prix.

Dans cette section, nous proposons un cadre générique permettant d'étendre les heuristiques définies précédemment aux WCSP. Une heuristique, par exemple `ConflictVar-Star` étendue aux WCSP sera

l0	c_1	c_2	c_3	c_4	c_5	c_6
l1	14	2	1	1	0	0
l2	$ec_1 = \{14, 2, 1\}$			$ec_2 = \{14, 2, 1, 1, 0, 0\}$		
l3	$\min(ec_1) = 1$			$\min(ec_2) = 0$		

TAB. 1 – Tableau sur les coûts.

Algorithm 7: ConflictVar-Cost

```
   function ConflictVar-Cost
   (A, X, C, k, k_init, k_max, nbSets)
   begin
1  |  Let bounds[] un tableau d'entiers de taille nbSets
2  |  bounds ← InitBounds(A, bounds, C, nbSets)
3  |  b ← getBound(k, k_init, k_max, nbSets)
4  |  X_unaffected ← ∅
5  |  X_conflicted ← getOptConflict(A, C, bounds[b])
6  |  while #(X_unaffected) ≠ k do
7  |  |  while X_conflicted = ∅ do
8  |  |  |  b ← b + 1
9  |  |  |  X_conflicted ←
   |  |  |  getOptConflict(A, C, bounds[b])\X_unaffected
10 |  |  x ← randomPick(X_conflicted)
11 |  |  X_unaffected ← X_unaffected ∪ {x}
12 |  |  X_conflicted ← X_conflicted\{x}
13 |  return X_unaffected
   end
   function getBound (k, k_init, k_max, nbSets)
   begin
14 |  return ⌊1 + (nbSets − 1) * (k − k_init)/(k_max − k_init)⌋
   end
```

notée `ConflictVar-Star-Cost`.

Dans un premier temps, les coûts des contraintes dans l'affectation complète courante $\mathcal{A}$ sont triés par ordre décroissant de violation, puis réparties en $nbSets$ sous-ensembles notés $ec_1, ec_2, ...ec_{nbSets}$, avec $nbSets$ un paramètre à fixer au début de la recherche. Chaque sous-ensemble ec_i contient les $(i*e)/nbSets$ plus fortes violations. Sur l'exemple du tableau 1, les violations des 6 contraintes (ligne l0) sont triées (ligne l1) et réparties en deux sous-ensembles (ligne l2). Au cours de la recherche, seules les contraintes ayant un coût de violation supérieur ou égal à $min(ec_b)$ (ligne l3) sont considérées comme étant en conflit ; la valeur de b augmentant proportionnellment suivant la valeur de k. Lorsque k est égal à k_{min} (resp. k_{max}), b est égal à 1 (resp. $nbSets$). Ainsi, au cours de la recherche (si le TimeOut n'est pas atteint), toutes les contraintes auront été considérées au moins 1 fois en conflit, et celles violées avec de forts coûts plus régulièrement (k étant réinitialisée à chaque nouvelle solution).

Nous détaillons ici uniquement l'extension au cadre des WCSP de `ConflictVar`. Cependant, l'extension de `ConflictVar-Star` a aussi été expérimentée.

La fonction `InitBounds` retourne un tableau noté *bounds* contenant les $min(ec_i)$ définis précédemment.

La fonction `getBound`, retourne la valeur de b, suivant la dimension courante du voisinage.

La fonction `getOptConflict` retourne l'ensemble des variables en conflit qui participe à au moins une contrainte violée de coût supérieur ou égal à $bounds[b]$.

L'heuristique `ConflictVar-Cost` sélectionne aléatoirement les variables parmi celles en conflits (10). Si toutes les variables en conflit ont été sélectionnées (ligne 7), on élargit la notion de conflit aux contraintes violées de plus faible poids dans l'ordre induit par le tableau *bounds* (ligne 8 et 9).

5 Expérimentations

Instances RLFAP : le Centre d'Electronique de l'Armement (CELAR) a fourni des instances réelles d'affectation de fréquences radio (Radio Link Frequency Assignment Problem [1]). L'objectif est d'affecter un nombre limité de fréquences à des liens radio en minimisant les interférences dues à la réutilisation des fréquences. Nous avons considéré les instances 6 à 10 prétraitées afin de diminuer le nombre de variables et de contraintes.

Instances aléatoires : GRAPH (Generating Radio link frequency Assignment Problems Heuristically, voir [13]) est un générateur aléatoire d'instances similaires à celles du CELAR. Les domaines des variables et le ratio entre le nombre de liens et d'interférénces sont proches de ceux utilisés dans le CELAR.

Chaque instance a été résolue par VNS/LDS+CP avec une discrepancy de 4, qui est la meilleure valeur trouvée sur les instances RLFAP (cf. [6]). k_{min}, k_{max} ont été respectivement fixés à 5 et 25 et le timeout à 6 minutes. L'implémentation a été écrite en Java en utilisant la librairie *choco* [4]. Chaque courbe représente la moyenne de 100 résolutions effectuées sur un Pentium 4 d'1.6 GHz. Pour les heuristiques `ConflictVar-H-Cost`, $nbSets$ a été fixé arbitrairement à 5.

Les figures 3 à 7 (resp. 8 à 13) détaillent les résultats obtenus sur les instances du CELAR (resp. GRAPH). A partir de ces figures, nous constatons que :

1. Sur l'ensemble des instances, les performances obtenues avec `ConflictVar` et avec `ConflictVar-Connected` sont fortement similaires. Ainsi, malgré un degré de liberté minimal (égal à 1), on constate que cela ne permet pas à LDS de trouver de meilleures solutions.

2. Les heuristiques basées sur un degré de liberté plus grand (i.e. permettant d'augmenter sensiblement celui-ci), sont nettement plus pertinentes. En effet, sur l'ensemble des instances considérées (excepté celles où les résultats sont similaires et la Scen10), `ConflictVar-MaxDeg` surclasse `ConflictVar-Star` et `ConflictVarAndSat-Star` (en particulier sur les

Scen06 et Scen07, qui sont parmi les plus difficiles).
3. L'élargissement des variables à réaffecter à l'ensemble des variables qui ne sont pas en conflit est également un critère déterminant. En effet, les résultats obtenus par `ConflictVarAndSat-Star` sont de loin meilleurs que ceux obtenus avec `ConflictVar-Star` (excepté sur les instances où les résultats sont similaires et sur la Scen10).
4. `ConflictVar-MaxDeg` en combinant les deux notions précédentes (en sélectionnant également les variables qui ne sont pas en conflit tout en essayant d'augmenter le degré de liberté des variables choisies), guide efficacement la recherche. En effet, sur la plupart des instances elle est parmi les meilleures (excepté sur la Scen10).
5. En comparant `ConflictVar` à `ConflictVar-Cost`, on constate que cette dernière est nettement plus pertinente. En outre, la combinaison de `ConflictVar-Star` avec les coûts (i.e. `ConflictVar-Star-Cost`) surclasse nettement `ConflictVar-MaxDeg` sur la Scen10 et obtient des résultats comparables sur les instances GRAPH. Cela confirme donc qu'il peut être important de considérer les poids des contraintes lors du choix des variables à désaffecter.

Le tableau 2 détaille pour chaque instance la meilleure solution trouvée selon l'heuristique utilisée. Dans un souci de place, nous avons numéroté les heuristiques, marqué en gras les meilleurs résultats de nos heuristiques, et en italique les meilleures solutions connues non prouvées optimales.

Comme nous pouvons le constater, `ConflictVar-Star-Cost` et `ConflictVar-MaxDeg` sont clairement les meilleures heuristiques, avec un léger avantage pour `ConflictVar-Star-Cost` qui permet de trouver les solutions optimales (ou meilleures connues) sur quatre instances GRAPH, deux du CELAR, et est très proche de l'optimum sur les autres. Il est à noter que ces résultats sont de qualité comparable à ceux obtenus par `ID Walk` [8] sur les instances du CELAR.

s* Solution optimale	**1** `ConflictVar`
2 `ConflictVar-Connected`	**3** `ConflictVar-MaxDeg`
4 `ConflictVar-Star`	**5** `ConflictVarAndSat-Star`
6 `ConflictVar-Cost`	**7** `ConflictVar-Star-Cost`

	g-05	g-06	g-07	g-11	g-12	g-13
s*	221	4123	4324	*3080*	11827	*10110*
1	332	12434	4368	25183	11846	38149
2	320	10563	4383	22911	11839	37944
3	**221**	**4123**	**4324**	3350	**11827**	**11599**
4	290	4250	**4324**	5380	**11827**	15384
5	**221**	4126	**4324**	3473	**11827**	12335
6	250	8277	**4324**	19112	11828	35606
7	**221**	**4123**	**4324**	**3235**	**11827**	12794

	s-06	s-07	s-08	s-09	s-10
s*	3389	*343592*	*262*	15571	31516
1	3434	2599275	490	15675	31518
2	3514	2394230	491	15821	**31516**
3	3401	354302	**286**	**15571**	**31516**
4	3528	364410	343	**15571**	**31516**
5	3412	364103	**286**	**15571**	**31516**
6	3463	586159	468	**15571**	**31516**
7	**3399**	**343800**	296	**15571**	**31516**

TAB. 2 – Meilleures solutions trouvées sur les instances

6 Conclusion

Dans cet article, nous avons proposé de nouvelles heuristiques de choix de voisinage pour les WCSP exploitant, outre la notion de conflit, à la fois la topologie du graphe de contraintes et leur poids associés. Les expérimentations réalisées avec VNS/LDS+CP ont montré que nos heuristiques guident plus efficacement la recherche que `ConflictVar`.

Dans nos travaux futurs, nous souhaitons dans un premier temps étudier l'influence de la valeur de *nbSets* sur les performances des heuristiques basées sur les coûts. Puis expérimenter nos heuristiques sur d'autres problèmes et étudier différents critères permettant de départager la première variable à désaffecter.

Références

[1] Bertrand Cabon, Simon de Givry, Lionel Lobjois, Thomas Schiex, and Joost P. Warners. Radio link frequency assignment. *Constraints*, 4(1) :79–89, 1999.

[2] F. Focacci, F. Laburthe, and A. Lodi. Local search and constraint programming. In *MIC*, pages 451–454, 2001.

[3] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. *IJCAI*, pages 607–615, 1995.

[4] N. Jussien and V. Barichard. The PALM system : explanation-based constraint programming. *Proceedings of TRICS a post-conference workshop of CP*, pages 118–133, 2000.

[5] Javier Larrosa and Thomas Schiex. In the quest of the best form of local consistency for Weighted CSP. In *IJCAI*, pages 239–244, 2003.

[6] S. Loudni and P. Boizumault. Combining VNS with constraint programming for solving anytime optimization problems. *to appear in EJOR*, pages 1–31, 2008.

[7] N. Mladenović and P. Hansen. Variable neighborhood search. *Comps. in Opns. Res.*, 24 :1097–1100, 1997.

[8] Bertrand Neveu, Gilles Trombettoni, and Fred Glover. Idwalk : A candidate list strategy with a simple diversification device, 2004.

[9] Laurent Perron, Paul Shaw, and Vincent Furnon. Propagation guided large neighborhood search. In *CP*, pages 468–481, 2004.

[10] Steven David Prestwich. A hybrid search architecture applied to hard random 3-SAT and low-autocorrelation binary sequences. In *CP*, pages 337–352, 2000.

[11] Thomas Schiex, Hélène Fargier, and Gérard Verfaillie. Valued constraint satisfaction problems : Hard and easy problems. In *IJCAI (1)*, pages 631–639, 1995.

[12] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. *CP*, 1520 :417–431, 1998.

[13] H. van Benthem. Graph : Generating radiolink frequency assignment problems heuristically, 1995.

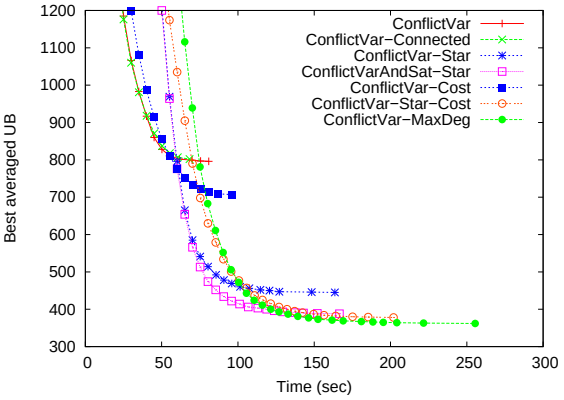

FIG. 5 – Scen08(458 vars,5286 contraintes).

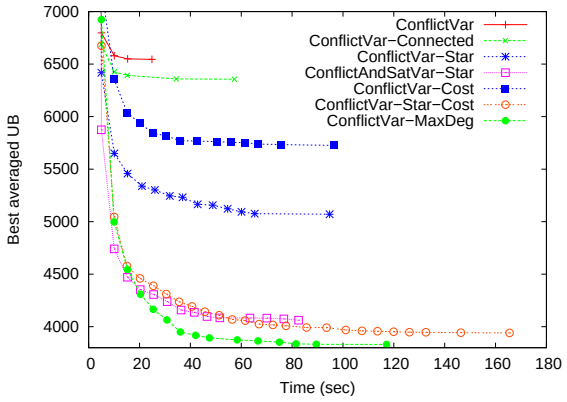

FIG. 3 – Scen06(100 vars,1222 contraintes).

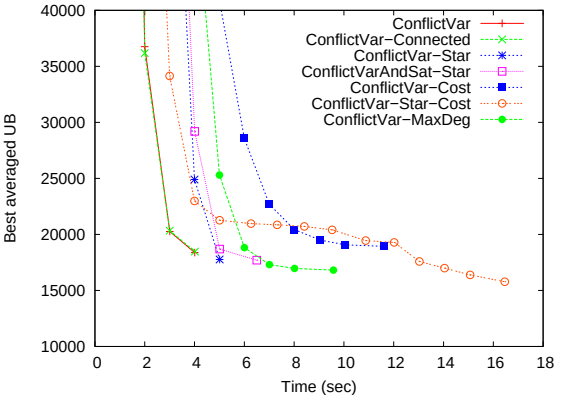

FIG. 6 – Scen09(200 vars,4209 contraintes).

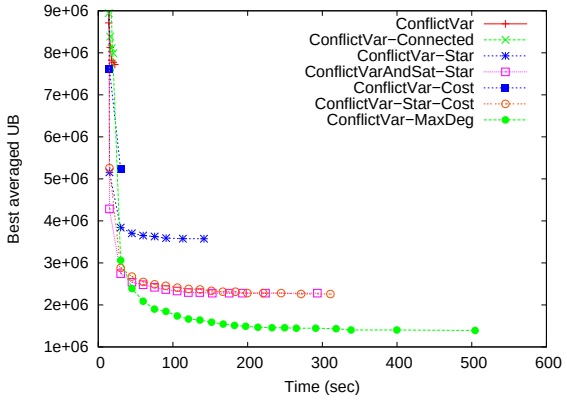

FIG. 4 – Scen07(200 vars,2665 contraintes).

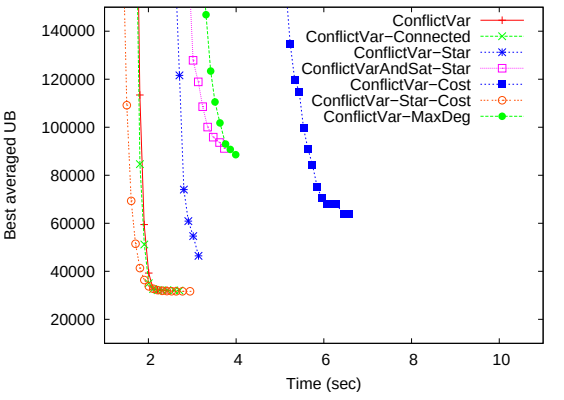

FIG. 7 – Scen10(200 vars,4209 contraintes).

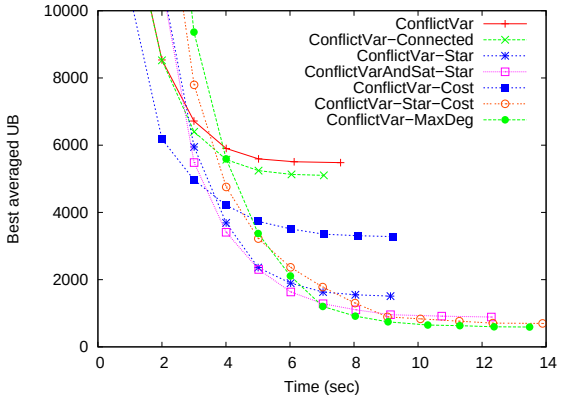

FIG. 8 – Graph05(100 vars,1034 contraintes).

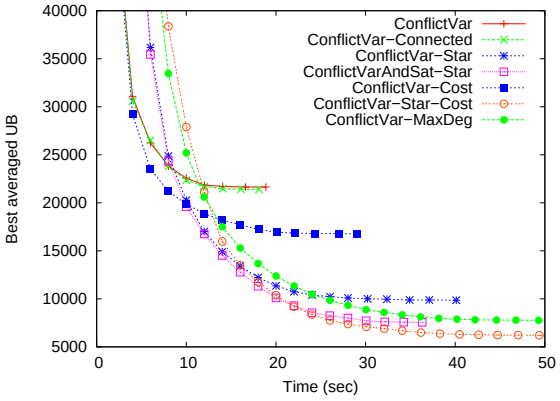

FIG. 9 – Graph06(200 vars,1970 contraintes).

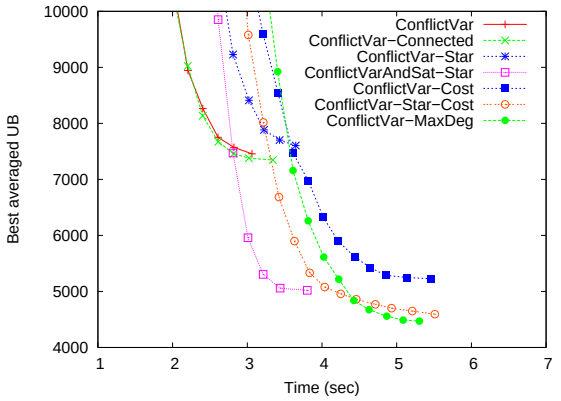

FIG. 10 – Graph07(141 vars,2213 contraintes).

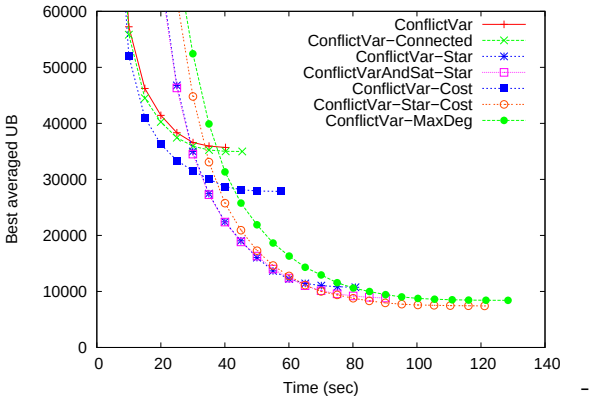

FIG. 11 – Graph11(340 vars,3417 contraintes).

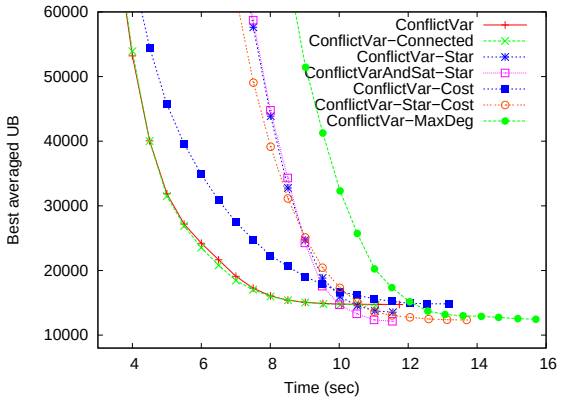

FIG. 12 – Graph12(252 vars,4099 contraintes).

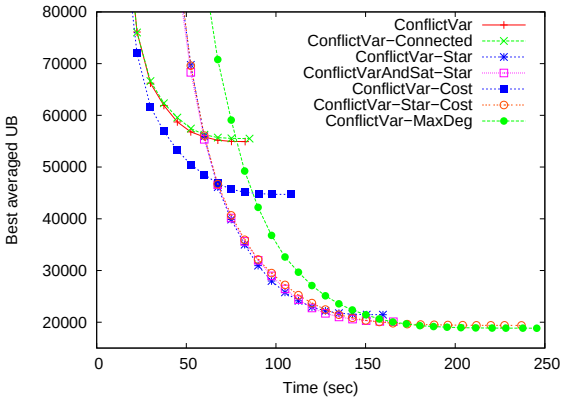

FIG. 13 – Graph13(458 vars,4815 contraintes).