HAL
open science

Heuristique de choix de valeur dirigée par HQ dans les WCSP

Nicolas Levasseur, Patrice Boizumault, Samir Loudni

To cite this version:

Nicolas Levasseur, Patrice Boizumault, Samir Loudni. Heuristique de choix de valeur dirigée par HQ dans les WCSP. JFPC 2008- Quatrièmes Journées Francophones de Programmation par Contraintes, LINA - Université de Nantes - Ecole des Mines de Nantes, Jun 2008, Nantes, France. pp.257-266. inria-00292657

HAL Id: inria-00292657

https://inria.hal.science/inria-00292657

Submitted on 2 Jul 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Heuristique de choix de valeur dirigée par HQ 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 WCSP [11, 14] (Weighted Constraint Satisfaction Problem) est un cadre général pour modéliser et résoudre des problèmes d'optimisation sous contraintes. Les WCSP sont très souvent résolus par des méthodes arborescentes qu'elles soient complètes (Depth First Branch and Bound) ou partielles (Limited Discrepancy Search) combinées avec des mécanismes de filtrage basés sur un calcul de minorants. Dans ces méthodes, les heuristiques de sélection dynamique de variable (MinDom/FutDeg) et de valeur (MinAC) sont parmi les plus efficaces. Dans cet article, nous proposons des nouvelles heuristiques de sélection de valeur basées sur la qualité des solutions (H-Quality). La H-Quality d'une affectation estime sa capacité à être présente dans des solutions de "bonne" qualité. Les expérimentations réalisées avec LDS et DFBB sur des instances réelles (CELAR) et aléatoires montrent que nos heuristiques guident plus efficacement la recherche que MinAC.

1 Introduction

En programmation par contraintes, les heuristiques de sélection de variable et de sélection de valeur jouent un rôle primordial dans la résolution d'un problème en permettant de définir des stratégies de recherche spécifiques et efficaces. Si de telles stratégies ont été étudiées intensivement dans le cadre des Problèmes de Satisfaction de Contraintes [6, 7, 2, 13], ce n'est pas encore le cas pour les WCSP.

Le formalisme des WCSP (Weighted Constraint Satisfaction Problem) [11, 14] est un cadre général pour modéliser et résoudre des problèmes d'optimisation sous contraintes. Les WCSP sont très souvent résolus par des méthodes arborescentes qu'elles soient complètes (Depth First Branch and Bound) ou partielles (Limited Discrepancy Search [8]) combinées avec des mécanismes de filtrage basés sur un calcul de minorants. Dans ces méthodes, les heuristiques de sélection de variable (MinDom/FutDeg [15]) et de valeur (MinAC) sont considérées parmi les plus efficaces.

Dans cet article, nous proposons une nouvelle heuristique de sélection de valeur dans le cadre des WCSP basée sur la qualité des solutions. La H-Quality d'une affectation estime sa capacité à être présente dans des solutions de "bonne" qualité. Notre heuristique `select-value-HQ` ordonne les valeurs des variables par ordre croissant des H-Quality. Les expérimentations réalisées avec LDS et DFBB sur des instances réelles (CELAR) et aléatoires montrent que sur la plupart de celles-ci, `select-value-HQ` guide plus efficacement la recherche que MinAC.

La section 2 présente le cadre WCSP. Les sections 3 et 4 rappellent et décrivent DFBB combiné avec les mécanismes de filtrage. La section 5 détaille la version de LDS que nous avons utilisée et la section 6 l'heuristique de choix de valeur MinAC. La notion de H-Quality ainsi que les nouvelles heuristiques de sélection de valeur qui en résultent sont définies dans la section 7. Dans la section 8, nous comparons la notion de H-Quality à celle utilisée par MinAC. La section 9 est consacrée aux expérimentations, puis nous concluons.

2 Problème de Satisfaction de Contraintes Pondérées

Un WCSP (Weighted Constraint Sastisfaction Problem) est défini par un quadruplet $(\mathcal{X}, \mathcal{D}, \mathcal{C}, k)$, avec $\mathcal{X} = \{x_1, ..., x_n\}$ l'ensemble des variables (de taille n),

Algorithm 1: Depth-First Branch and Bound.

```
   fonction DFBB (A, X_f)
   begin
1    if X_f = ∅ then
2      UB ← V(A)
3      return A
4    x_i ← select-variable (X_f)
5    while (d_i ≠ ∅) do
6      a ← select-value (d_i)
7      if Filtering (A ∪ {(x_i = a)}, X_f, UB) then
8        BestS ← DFBB(A ∪ {(x_i = a)}, X_f \ {x_i})
9      cancelFiltering (A, (x_i = a), X_f)
10     d_i ← d_i \ {a}
11   return BestS
   end
```

$\mathcal{D} = \{d_1, ..., d_n\}$ les domaines finis associés (la taille maximale des domaines est notée d) et $\mathcal{S}(k)$ une structure de valuation. $\mathcal{S}(k)$ est le triplet ([0,1,...,k], $\oplus, \geq$) où k est un entier naturel dans $[1, .., \infty]$, $\oplus$ est défini par : $a \oplus b = \min(k, a + b)$ 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 $f_c : \prod_{x_i \in \mathcal{X}_c} d_i \mapsto [0, k]$, qui retourne 0 quand c est satisfaite. Une affectation de x_i à la valeur a est notée : $(x_i = a)$. Une affectation *complète* $\mathcal{A}$ (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 Depth-First Branch and Bound

L'algorithme 1 décrit un parcours de type DFBB. Au début de la recherche, $\mathcal{UB}$ est initialisé au coût maximal $\top$. L'affectation partielle courante $\mathcal{A}$ est vide. $\mathcal{X}_f$, l'ensemble des variables futures (i. e. non encore affectées), est initialisé à $\mathcal{X}$.

Si toutes les variables sont affectées (ligne 1), on met à jour la borne supérieure (ligne 2) et la solution courante devient la meilleure solution connue (lignes 3 et 8). Sinon, on sélectionne la prochaine variable à affecter (heuristique de choix de variable, ligne 4). Puis, on sélectionne la prochaine valeur à lui affecter (heuristique de choix de valeur, ligne 6). On procède alors à une phase de filtrage des valeurs des variables futures (ligne 7) qui est détaillée dans la section 4. Si aucun domaine vide n'a été trouvé (ligne 7), on explore alors le sous-espace (ligne 8). 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 9). Puis le domaine de la variable est réduit (ligne 10). La variable est ensuite affectée à la valeur suivante de son domaine réduit (si celui-ci n'est pas vide) (ligne 5). Une fois l'exploration terminée, la fonction DFBB retourne la solution optimale (ligne 11).

4 Filtrage et Calcul de Minorant

Soit $\mathcal{A}$ une affectation partielle. A chaque nœud de l'arbre de recherche, plusieurs méthodes *forward checking* [5], *directed arc-consistency* [16]), *Full dac* [11], *Existential dac* [4] ont été proposées afin d'établir une propriété de consistance locale basée sur un calcul de minorant. Ce minorant, noté $\mathcal{LB}(\mathcal{A})$ représente l'agrégation des coûts des contraintes qui seront nécessairement violées par toutes les affectations complètes étendant $\mathcal{A}$. Dans ces méthodes, pour chaque affectation $(x_i = a)$, la somme des violations des contraintes qui seront nécessairement violées par les extensions de $\mathcal{A} \bigcup \{(x_i = a)\}$ est projetée dans le compteur d'une variable (déterminée par une heuristique d'ordonnancement des variables). Dans ce papier, nous nommerons ce compteur *compteur d'arc consistance* et le compteur associé à $(x_i = a)$ est noté ac_{ia}.

La fonction $Filtering(\mathcal{A}, \mathcal{X}_f, \mathcal{UB})$, utilisée dans les algorithmes 1 et 2, établit une consistance locale à chaque nœud et permet ainsi de :

1. propager les effets d'une affectation $(x_i = a)$ en mettant à jour les compteurs ac des valeurs des variables futures (i. e. non encore affectées) liées par une contrainte avec x_i.
2. mettre à jour les domaines de variables futures en supprimant les valeurs inconsistantes grâce aux compteurs d'arc consistance et au minorant $\mathcal{LB}(\mathcal{A})$.
3. propager les suppressions de valeur. La propagation peut entraîner la mise à jour de compteurs ac, et provoquer la suppression de nouvelles valeurs.

La fonction `Filtering` retourne `false` si toutes les valeurs d'un domaine sont supprimées, et `true` sinon. Selon, le niveau de consistance souhaité, le filtrage peut être effectué par différents algorithmes :

- Partial Forward Checking-Directed Arc Consistency (PFC-DAC) [10], il réalise les points (1) et (2),
- Maintaining Full Directional Arc Consistency (MFDAC*) [11], il réalise les points (1), (2) et (3). MFDAC* détecte plus d'inconsistances que PFC-DAC (voir [11] pour plus de détails).

5 Limited Discrepancy Search

LDS (Limited Discrepancy Search) est une méthode de recherche partielle introduite par Harvey et Ginsberg [8]. 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) de fois. On s'autorise donc δ écarts (*discrepancies*) à l'heuristique h. Pour un nombre maximal δ d'écarts autorisé, LDS explore l'arbre de recherche de manière itérative selon un nombre croissant d'écarts allant de $i = 0$ à $i = \delta$. A chaque itération, i est incrémenté de 1. Nous avons étendu LDS aux WCSP comme dans [12] en trois étapes :

D'un arbre binaire à un arbre n-aire. La comptabilisation des écarts dans un arbre n-aire est un sujet très rarement traité dans la littérature. Un écart de j est comptabilisé lorsque l'on sélectionne le (j+1)ème fils (selon l'heuristique h) à la place du premier. Ainsi, le choix du premier fils (selon h) provoquera un écart de 0, le choix du second un écart de 1, et ainsi de suite.

De la satisfaction à l'optimisation. Comme DFBB, LDS essaie d'étendre une affectation partielle en une affectation complète. Soit $\mathcal{A}$ une affectation partielle ; une phase de filtrage, basée sur le calcul d'un minorant $\mathcal{LB}(\mathcal{A})$ du coût de toutes les affectation complètes prolongeant $\mathcal{A}$, permet comme pour DFBB (algorithme 1), l'élagage de l'arbre de recherche.

N'effectuer qu'une seule itération. Le principal inconvénient de LDS est la redondance d'exploration de certains nœuds. En effet, l'itération i revisite les nœuds déjà visités aux itérations 0, 1, ..., et $i - 1$. Nous effectuons uniquement la dernière itération afin de ne pas revisiter plusieurs fois les mêmes sous arbres.

L'algorithme 2 détaille le pseudo-code de LDS n-aire, $\mathcal{A}$ représente l'affectation courante, $\mathcal{X}_f$ les variables futures et δ le nombre d'écarts à h. Au début $\mathcal{A} = \emptyset, \mathcal{X}_f = \mathcal{X}$ et δ =valeur maximale autorisée.

Comme pour DFBB (algorithme 1), si toutes les variables sont affectées, on met à jour $\mathcal{UB}$ et la solution courante devient la meilleure solution connue. Sinon, on sélectionne la prochaine variable à affecter (ligne 4) ; 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 est sélectionnée (ligne 7) et le nombre restant d'écarts est décrémenté à $(\delta - j)$. Si la phase de filtrage ne produit aucun domaine vide, la recherche se poursuit (ligne 9).

Algorithm 2: n-ary LDS for optimisation.

```
   function n-ary-LDS-PROBE-optim (A, X_f, δ)
   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 ← n-ary-LDS-PROBE-optim(A ∪ {(x_i =
         a)}, X_f \ {x_i}, δ − j)
10     cancelFiltering (A, (x_i = a), X_f)
11     d_i ← d_i \ {a}
12     j ← j + 1
13   return BestS
   end
```

6 Heuristique de choix de valeur générique pour les WCSP

Les compteurs d'arc consistance (ac) détaillés section 4 prennent en compte pour chaque affectation ($x_i = a$) les coûts des contraintes qui seront nécessairement violées lors de l'extension de l'affectation partielle $\mathcal{A}$ avec ($x_i = a$). En sélectionnant la valeur ayant le compteur d'inconsistances minimal ac, la recherche est guidée vers les sous-espaces de recherche les moins inconsistants. L'heuristique de choix de valeur MinAC décrite dans [11] est une heuristique *générique* (indépendante du problème) basée sur ces compteurs. Elle ordonne les valeurs des domaines des variables par ordre croissant des valuations comptabilisées par les compteurs d'arc consistance.

7 Heuristiques de choix de valeur basées sur les H-Quality pour WCSP

Les choix effectués par MinAC dépendent fortement de la capacité des mécanismes de filtrage à détecter les inconsistances futures. La plupart des algorithmes de filtrage sont basés sur un critère local (i. e. arc cohérence). De plus, nous avons constaté sur les nombreuses expérimentations effectuées que MinAC engendrait de nombreux candidats ex-aequo.

7.1 L'historique de la recherche

Au cours de la recherche, on effectue des observations basées sur la qualité des solutions obtenues. En effet, toutes les inconsistances étant connues dans les affectations complètes, on se base alors sur un critère global plus indépendant des mécanismes de filtrage pour dériver des heuristiques de choix de valeur.

Définition 1 *L'historique de recherche $\mathcal{H}$ est défini par l'ensemble des solutions obtenues depuis le début de la résolution.*

7.2 H-Quality d'une affectation

La notion de H-Quality d'une affectation, selon un historique, estime la capacité d'une valeur à être présente dans les solutions de "bonne" qualité, et donc sa capacité à guider la recherche vers les sous-espaces où les inconsistances (somme des coûts des contraintes violées) seront plus faibles.

Définition 2 *Pour $x_i \in \mathcal{X}, a \in d_i$ et un historique $\mathcal{H}$.*

$$HQ_{val}(x_i, a, \mathcal{H}) = \min_{\mathcal{A} \in \mathcal{H} \wedge (x_i = a) \in \mathcal{A}} \mathcal{V}(\mathcal{A})$$

$HQ_{val}(x_i, a, \mathcal{H})$ représente le coût de la meilleure solution de l'historique $\mathcal{H}$ contenant l'affectation $(x_i = a)$. Plus les solutions contenues dans $\mathcal{H}$ sont de bonne qualité, plus la notion de H-Quality devient pertinente car approchant, pour chaque affectation (x_i, a), la valeur théorique : $min_{\mathcal{A} \in d_1 \times \ldots \times (x_i = a) \times \ldots \times d_n} \mathcal{V}(\mathcal{A})$, qui représente le coût minimal d'une affectation complète contenant $(x_i = a)$. Cette valeur qui ne peut être obtenue qu'en énumérant toutes les affectations complètes d'un problème, permettrait de prédire *en haut* de l'arbre de recherche que l'affectation $(x_i = a)$ ne pourrait être étendue en une solution de qualité meilleure que $HQ_{val}(x_i, a, \mathcal{H})$. Par conséquent, nous avons choisi d'utiliser l'opérateur min dans la définition des H-Quality, car il permet de s'approcher au cours de la recherche de cette valeur, et également de guider vers les zones les moins inconsistantes.

L'opérateur max nous a semblé inadapté, car le rôle de l'heuristique est de guider la recherche vers les solutions de "bonne" qualité et non de "mauvaise". Enfin, l'opérateur *average* nous a semblé inadapté également, car de nombreuses solutions étant générées par des descentes gloutonnes sans filtrage (voir section 7.4 pour plus de détails), elles sont régulièrement de moins bonne qualité que la meilleure solution connue. La moyenne reflète alors plus les moins bonnes qualités que les bonnes, et ne guide pas la recherche vers les meilleurs sous-espaces.

Complexité. Pour une affectation (x_i, a), on stocke en mémoire uniquement la meilleure (plus petite) $HQ_{val}(x_i, a, \mathcal{H})$; l'espace mémoire nécessaire pour représenter $\mathcal{H}$ est donc en $0(n \times d)$. Mettre à jour l'historique $\mathcal{H}$, en ajoutant une solution $\mathcal{A}$, se fait en $O(n)$ (cf. la procédure `update-HQ` de l'algorithme 3).

7.3 LDS guidée par des heuristiques basées sur les H-Quality

Nous proposons l'heuristique `select-value-HQ` (cf. Algo. 3) qui trie les valeurs du domaine d'une variable

Algorithm 3: H-Quality heuristics

```
   function select-value-HQ (d_i) begin
1     Select a ∈ d_i having the smallest H-Quality value
2     Break the ties using MinAC
3     return a
   end
   procedure update-HQ(A, H)
   begin
4     foreach (x_i = a) ∈ A do
5         if HQ_val(x_i, a, H) > V(A) then
              HQ_val(x_i, a, H) ← V(A)
   end
   procedure compute-HQOnce (A, d_i, X_f)
   begin
6     for a ∈ d_i do
7         if HQ_val(x_i, a, H) is unknown then
8             A ← greedy(A ∪ (x_i = a), X_f \ {x_i})
9             update-HQ(A, H)
   end
   procedure compute-HQAll (A, d_i, X_f)
   begin
10    for a ∈ d_i do
11        A ← greedy(A ∪ (x_i = a), X_f \ {x_i})
12        update-HQ(A, H)
   end
```

selon l'ordre croissant des valeurs d'H-Quality. A égalité, les ex-aequo seront départagés par MinAC. Cependant, lors de l'appel à cette fonction, certaines valeurs d'H-Quality peuvent être inconnues : pour une valeur $a \in d_i$, $\mathcal{H}$ peut très bien ne contenir aucune solution contenant $(x_i = a)$. Dans ce cas, nous choisissons d'ajouter $(x_i = a)$ à l'affectation partielle et de l'étendre en une affectation complète à l'aide d'une descente gloutonne. Pour combiner `select-value-HQ` avec LDS (cf. Algo. 4), on doit :

1. mettre à jour les H-Quality des affectations lorsqu'une nouvelle solution est trouvée (ligne 2) en appelant la procédure `update-HQ` ;

2. appeler la procédure `compute-HQOnce` juste après la sélection d'une variable (ligne 4) ;

3. remplacer l'heuristique `select-value` par `select-value-HQ` (ligne 7).

La pertinence de notre heuristique est directement liée à la diversité et au nombre de solutions présentes dans l'historique. Nous avons donc aussi étudié une combinaison de l'algorithme de recherche avec une variante de `select-value-HQ` notée `HQAll`, la précédente étant notée `HQOnce`, où l'on effectue à *chaque* nœud, une descente gloutonne pour *chaque* valeur du domaine (et non plus uniquement pour les valeurs d'H-Quality inconnues). Pour cela, il suffit de substituer l'appel à la procédure `compute-HQOnce` par un appel à la procédure `compute-HQAll` dans l'algorithme 4. Le sur-coût est négligeable car chaque descente gloutonne a une faible complexité, comme nous allons le voir.

Algorithm 4: LDS+`select-value-HQ`

```
   function n-ary-LDS-PROBE-optimization (A, X_f, δ)
   begin
1    if X_f = ∅ then
2      update-HQ(A)
3      UB ← V(A)
4      return A
5    x_i ← select-variable (X_f)
6    compute-HQOnce(A, d_i, X_f)
7    j ← 0
8    while (d_i ≠ ∅) ∧ (j ≤ δ) do
9      a ← select-value-HQ(d_i)
10     if Filtering (A ∪ {(x_i = a)}, X_f, UB) then
11       BestS ← n-ary-LDS-PROBE-optimization(A∪
         {(x_i = a)}, X_f \ {x_i}, δ − j)
12     cancelFiltering (A, (x_i = a), X_f)
13     d_i ← d_i \ {a},
14     j ← j + 1
15   return BestS
   end
```

Algorithm 5: Greedy search.

```
   function greedy(A, X_f)
   begin
1    if X_f = ∅ then return A
     else
2      x_i ← select-variable (X_f)
3      a ← select-value-MinAC-Tie (d_i)
4      forwardChecking (x_i, A ∪ {(x_i = a)}, X_f \ {x_i})
5      return greedy(A ∪ {(x_i = a)}, X_f \ {x_i})
   end
   procedure forwardChecking(x_i, A, X_f)
   begin
6    foreach c ∈ C_quasi−assigned(A) ∧ x_i ∈ X_↓c do
7      let x_j be the single element of X_↓c ∩ X_f
8      foreach b ∈ d_j do
9        affect(x_j = b)
10       if addInconsistency((A ∪ (x_j = b))_↓X_c) then
11         ac_jb ← ac_jb + c(A_↓X_c)
   end
```

7.4 Descente gloutonne

Un des rôles des descentes gloutonnes, est de diversifier *rapidement* la recherche. En effet, si celles-ci trouvent une solution de "bonne" qualité, les H-Quality sont mis à jour et l'heuristique tient compte de ces nouvelles informations pour guider plus finement la suite de la recherche. Cela est d'autant plus le cas, que le calcul des H-Quality se fait grâce au minimum des coûts des solutions contenues dans l'historique et non avec la moyenne. Dans le cas contraire, si les descentes gloutonnes ne génèrent pas de solutions de "bonne" qualité, l'heuristique de choix de valeur n'est pas affectée car les H-Quality restent basés sur les meilleures solutions trouvées par l'algorithme de recherche.

Chaque descente gloutonne est guidée par une heuristique de choix de variable `select-variable` et par une heuristique de choix de valeur `select-value-MinAC-Tie`. Comme pour l'heuristique de choix de valeur, nous utilisons MinAC, et en cas d'égalité nous départageons en choisissant la valeur la moins souvent affectée depuis le début de la recherche [1]. Lors de la descente gloutonne, la phase de filtrage est désactivée afin de trouver rapidement une solution. Le pseudocode du forward checking et de la descente gloutonne est détaillé dans l'algorithme 5.

La fonction `greedy` retourne une affectation dès qu'elle est complète (ligne 1). Sinon, on sélectionne par `select-variable` la variable future à affecter (ligne 2), ainsi que sa valeur par `select-value-MinAC-Tie` (ligne 3). Puis, pour la nouvelle affectation partielle, les compteurs *ac* sont mis à jour par *forward checking* (ligne 4) afin de renseigner MinAC sur les inconsistances futures avant de continuer la descente (ligne 5).

La procédure `forwardChecking` considère, pour chaque contrainte quasiment affectée (i.e. ayant une seule variable non affectée) et portant sur x_i (ligne 6), la variable x_j non affectée (ligne 7). Pour chaque valeur b de son domaine (ligne 8), si l'affectation $(x_j = b)$ (ligne 9) génère une nouvelle violation (ligne 10), celle-ci est ajoutée au compteur ac_{jb} (ligne 11).

complexité. Lors d'une descente gloutonne, la procédure `forwardChecking` ne traite que les contraintes quasi-instanciées portant sur la dernière variable affectée. Ainsi, une contrainte ne pourra être traitée qu'une fois et une seule (coût de e) sur l'ensemble de la descente. La complexité d'une descente gloutonne est par conséquent en $O(e \times d)$ [2].

N.B. Notre heuristique pourrait améliorer l'efficacité de la recherche en mettant à jour la borne supérieure $\mathcal{UB}$ lorsqu'une recherche gloutonne trouve une solution de meilleure qualité. Nous avons décidé d'étudier les heuristiques de choix de valeur sans changer le comportement de l'algorithme de recherche, et par conséquent nous n'avons pas effectué cette mise à jour.

8 Critère local Vs. global

Comme nous l'avons dit précédemment, les choix effectués par MinAC dépendent fortement de la capacité des mécanismes de filtrage à détecter les inconsistances futures. La plupart des algorithmes de filtrage étant basés sur un critère local (i. e. arc cohérence), les choix de l'heuristique sont réalisés à partir d'informations locales (au niveau de la contrainte), sans tenir

[1] Pour chaque valeur, nous maintenons un compteur mémorisant le nombre de fois que celle-ci a été affectée.

[2] Cette complexité ne tient pas compte des n appels aux heuristiques de choix de valeur et de variables lors de la descente.

compte des répercussions au niveau de l'ensemble du graphe de contraintes.

Notre heuristique de choix de valeur se base elle sur un critère global où toutes les inconsistances sont connues. Cependant, contrairement à MinAC, elle ne tient pas compte *directement* de l'affectation partielle courante pour affecter les variables futures, mais *indirectement*, car :

1. les compteurs d'inconsistances sont utilisés par l'algorithme de filtrage, et ainsi les valeurs générant "beaucoup" d'inconsistances avec l'affectation partielle courante seront filtrées, même si elles ont des "bonnes" valeurs d'H-Quality.

2. les compteurs d'inconsistances guident les descentes gloutonnes qui permettent de mettre à jour également les H-Quality. Ces descentes, peuvent alors apporter des informations qui sont prises en compte dans l'exploration du sous-espace de recherche.

3. les H-Quality ont été "apprises" sur des affectations partielles différentes de l'affectation partielle courante, mais fortement similaires. En effet, hormis à la racine de l'arbre de recherche, LDS et DFBB ne modifient jamais complètement l'affectation partielle courante, seul le dernier choix est remis en cause, le reste de l'affectation étant conservée.

4. les H-Quality permettent de mémoriser les meilleures extensions d'affectations partielles similaires à la courante $\mathcal{A}$. En effet, lors de la remise en cause de la dernière affectation ($x_i = a$), cela a peu de chances d'influencer l'ensemble des affectations des variables futures. Ainsi, l'affectation pour des variables futures éloignées (au sens du graphe des contraintes) de x_i avec leur meilleure trouvée précédemment générera "peu" d'inconsistances. Cela est d'autant plus pertinent si le graphe de contraintes est structuré.

9 Expérimentations

9.1 Benchmarks

Instances WCSP générées aléatoirement : chaque instance est caractérisée par un quadruplet $< n, d, p_1, p_2 >$ où n est le nombre de variables, d le nombre de valeurs par domaine, p_1 la connectivité du graphe de contraintes (définissant le nombre de contraintes du graphe), et p_2 la dureté des contraintes (le pourcentage de n-uplets interdits par contrainte) et par une distribution des coûts ; 20% des contraintes ont comme valuation pour les n-uplets interdits $\top$, 20% 1000, 20% 100, 20% 10 et 20% 1. Les tuples autorisés ont pour valuation 0. Chaque courbe représente une moyenne de 25 résolutions de l'évolution de la qualité meilleure de la solution au cours du temps.

Instances binaires RLFAP : le CELAR (Centre d'Electronique de l'Armement) a fourni des instances réelles d'affectation de fréquences radio (Radio Link Frequency Assignment Problem [1]). Dans ces instances, l'objectif est d'affecter un nombre limité de fréquences à un ensemble de liens radio définis entre des sites, dans le but de minimiser les interférences dues à la réutilisation des fréquences. Dans nos expériences, nous avons considéré les sous-instances (Sub0 à Sub4) de l'instance 6. Un pré-traitement est réalisé avant le début de la recherche sur chaque sous-instance dans le but de diminuer le nombre de variables et de contraintes.

Chaque instance a été résolue par soit LDS, soit DFBB en utilisant soit MFDAC*, soit PFC-DAC. Pour LDS, l'écart (discrepancy) avait pour valeur 4, qui est la valeur optimale trouvée par [12] sur les instances RLFAP ou 5. Toutes les stratégies de recherche ont été implémentées en Java en utilisant la bibliothèque *choco* [9]. Toutes les expérimentations ont été effectuée sur un Pentium 4 d'1.6 GHz.

Nous avons comparé MinAc à deux heuristiques : `HQOnce` qui effectue à chaque nœud, une recherche gloutonne pour initialiser les valeurs ayant des H-Quality inconnues et `HQAll` qui effectue à chaque nœud, une recherche gloutonne pour chaque valeur du domaine. Pour l'heuristique de choix de variable, nous avons utilisé MinDom/FutDeg.

9.2 DFBB

Les figures 1 à 3 montrent les résultats obtenus avec MFDAC* sur différents problèmes aléatoires. Sur la classe des problèmes "faciles" $< 50, 10, 20, 60 >$ (fig. 2), les résultats obtenus avec MinAC, `HQAll` et `HQOnce` sont très similaires. Sur ces problèmes l'optimum a été prouvé 24 fois sur les 25 résolutions par les trois heuristiques. Lorsque la connectivité augmente (fig. 3), nos heuristiques sont nettement plus efficaces que MinAC, toutefois l'optimum n'est jamais prouvé (le timeout été fixé à 2 heures). Même sur des instances plus petites ayant la même connectivité $< 40, 8, 48, 60 >$ (fig. 1), le timeout a été atteint plus de 20 fois sur les 25 résolutions pour les 3 heuristiques.

En raison de son caractère local (arc-consistency), on observe que les éventuels mauvais choix de MinAC ont moins de conséquences sur les problèmes "faciles", ce qui n'est pas le cas sur les problèmes "difficiles" où les conséquences peuvent être très importantes.

Sur les sous-instances de la scène 6 du Celar, nos heuristiques ont de meilleurs résultats sur l'instance Sub2 (fig. 5), et sont comparables sur l'instance Sub3 (fig. 6). Notons toutefois que sur l'instance Sub0 (fig. 4) MinAC est nettement meilleur. Cela peut s'expliquer par le sur-coût induit par les descentes les gloutonnes et du fait que cette instance est facile à résoudre

(l'optimum est obtenu en moins d'une seconde).

9.3 LDS

Nous avons également comparé les trois heuristiques combinées avec LDS sur les classes de problèmes aléatoires $< 50, 10, p_1, p2 >$, avec p_1 égal à 20 ou 48 et différentes valeurs de p_2 (figures 10-14). De nouveau, `HQAll` et `HQOnce` surpassent clairement MinAC sur la plupart des instances. Par ailleurs, on observe que l'écart de performances entre les heuristiques s'accroît avec l'augmentation de la dureté et/ou de la connectivité du graphe. En effet, en augmentant un de ces paramètres (ou les deux), on accroît les conséquences des "mauvais" choix, car le nombre d'inconsistances dans le graphe croît. En effet, MinAc guidée par l'arc consistance ne permet pas d'anticiper ces conséquences, alors que nos heuristiques basées sur les H-Quality, calculées à partir d'affectations complètes où toutes les inconsistances sont connues, sont plus efficaces. En outre, les choix effectués par nos heuristiques assurent qu'au moins une affectation $(x_i = a)$ est présente dans une solution de qualité $HQ_{val}(x_i, a, \mathcal{H})$.

Notons que les performances de `HQAll` et `HQOnce` sont très comparables au début de la recherche. Mais, dès que la qualité de l'historique est suffisamment pertinente, `HQAll` devient très efficace et surpasse `HQOnce`. En effet, les mises à jour régulières des H-Quality avec les descentes gloutonnes diversifient l'historique et augmentent sa qualité, guidant la recherche vers des sous-espaces plus prometteurs.

Sur les sous-instances de la scène 6 du Celar (figures 7-9), `HQAll` obtient de meilleurs performances que MinAC, excepté sur l'instance Sub3 où les résultats sont comparables.

Nous avons étudié l'influence de la valeur de la discrepancy sur nos heuristiques en augmentant celle-ci à 5 (fig. 15 et 16). Les résultats reportés montrent que `HQOnce` et `HQAll` restent toujours plus pertinentes que MinAC. Nous avons également étudié l'influence des algorithmes de filtrage sur notre heuristique en substituant MFDAC* par PFC-DAC (fig. 17 et 18), et il apparaît que `HQOnce` et `HQAll` sont moins sensibles aux mécanismes de filtrage que MinAC et restent bien meilleurs.

10 Conclusion

Dans ce papier, nous avons proposé de nouvelles heuristiques de choix de valeur pour les WCSP basées sur la qualité des solutions. Les expérimentations sur des instances aléatoires et sur des problèmes réels (CELAR) montrent que nos heuristiques de choix de valeur guident plus efficacement la recherche sur la plupart des instances. Nous souhaitons également confirmer les résultats obtenus par nos heuristiques en initialisant la borne supérieure par une méthode de recherche locale, et en utilisant EDAC comme algorithme de filtrage combiné avec une phase de pré-traitement [3].

Références

[1] B. Cabon, S. De Givry, L. Lobjois, T. Schiex, and al. Radio link frequency assignment. *Constraints*, 4(1) :79–89, 1999.

[2] H. Cambazard and N. Jussien. Identifying and exploiting problem structures using explanation-based constraint programming. *Constraints*, 11(4), 2006.

[3] Martin C. Cooper, Simon de Givry, and Thomas Schiex. Optimal soft arc consistency. In *IJCAI*, pages 68–73, 2007.

[4] S. de Givry, F. Heras, M. Zytnicki, and J. Larrosa. Existential arc consistency : Getting closer to full arc consistency in weighted CSPs. *IJCAI*, pages 84–89, 2005.

[5] E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *AI*, 58 :21–70, 1992.

[6] P. A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. *ECAI*, pages 31–35, 1992.

[7] I. P. Gent, E. MacIntyre, P. Prosser, B. Smith, and T. Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. *CP*, pages 179–193, 1996.

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

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

[10] J. Larrosa and P. Meseguer. Exploiting the use of DAC in MAX-CSP. *Proceedings of CP*, 1996.

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

[12] S. Loudni and P. Boizumault. Combining VNS with constraint programming for solving anytime optimization problems. *to appear in EJOR*, pages 1–31, 2007.

[13] P. Refalo. Impact-based search strategies for constraint programming. *CP*, pages 557–571, 2004.

[14] T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems : Hard and easy problems. *IJCAI*, pages 631–639, 1995.

[15] B. M. Smith and S. A. Grant. Trying harder to fail first. *ECAI*, pages 249–253, 1998.

[16] R. J. Wallace. Directed arc consistency preprocessing. *ECAI, Workshop on constraint Processing*, pages 121–137, 1994.

Fig. 1 – DFBB-<40,8,48,60>

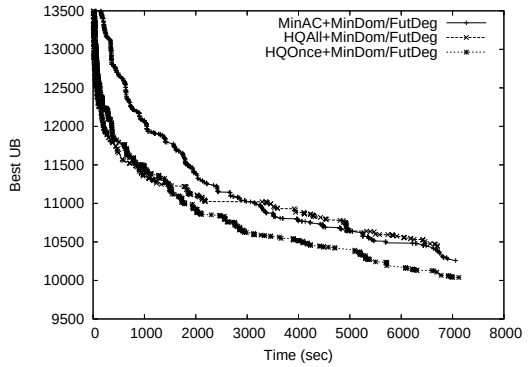

Fig. 2 – DFBB-<50,10,20,60>

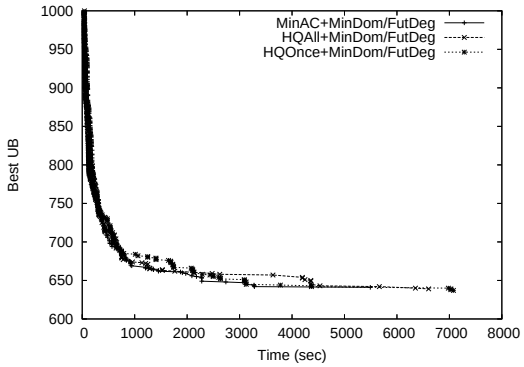

Fig. 3 – DFBB-<50,10,48,60>

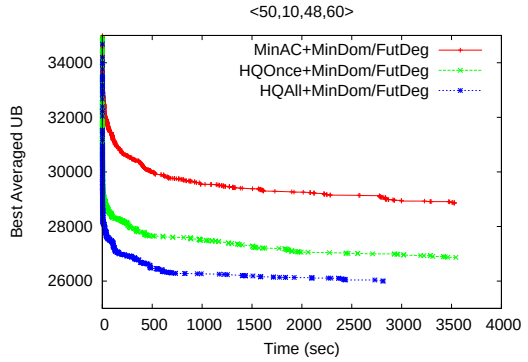

Fig. 4 – DFBB-Sub0 (32 vars, 223 contraintes)

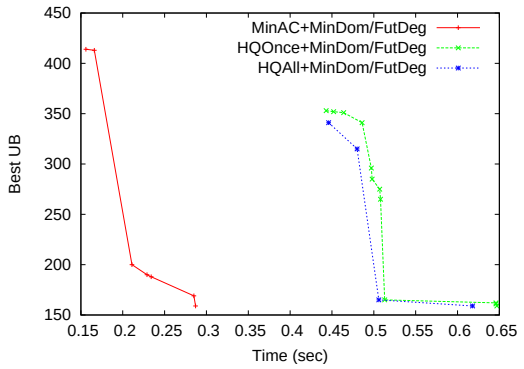

Fig. 5 – DFBB-Sub2(32 vars,369 constraints)

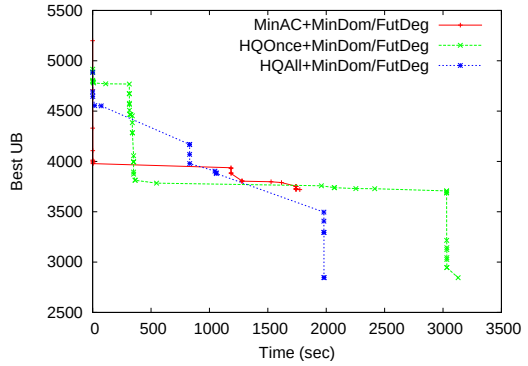

Fig. 6 – DFBB-Sub3(36 vars,439 constraints)

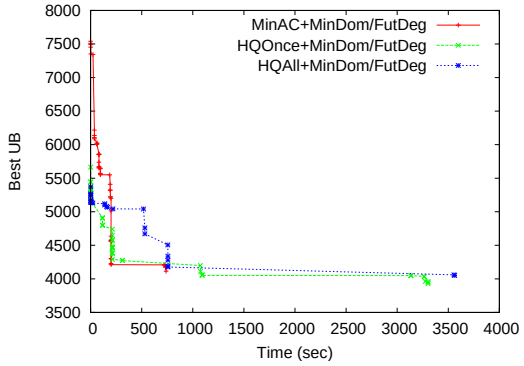

FIG. 7 – LDS(4)-Sub2(32 vars,369 constraints)

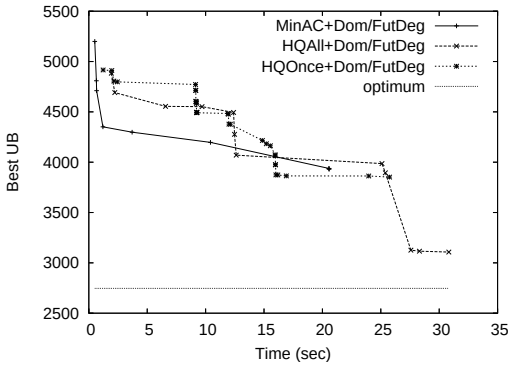

FIG. 8 – LDS(4)-Sub3(36 vars,439 constraints)

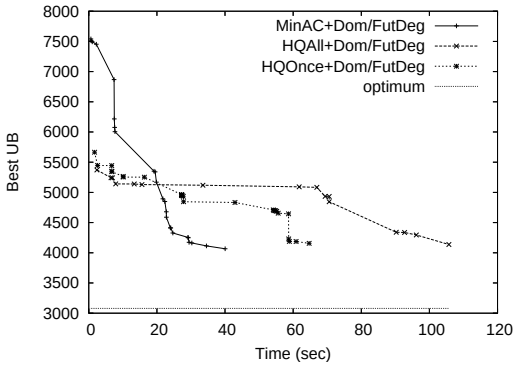

FIG. 9 – LDS(4)-Sub4(44 vars,499 constraints)

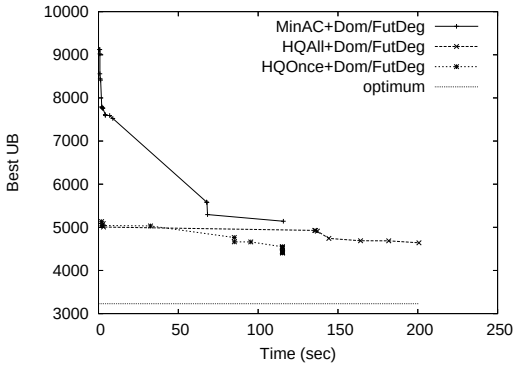

FIG. 10 – LDS(4)<50,10,48,40>

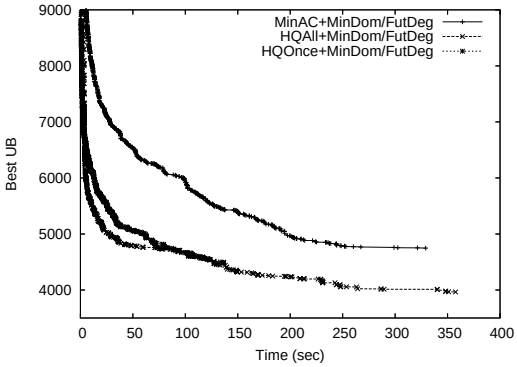

FIG. 11 – LDS(4)<50,10,48,60>

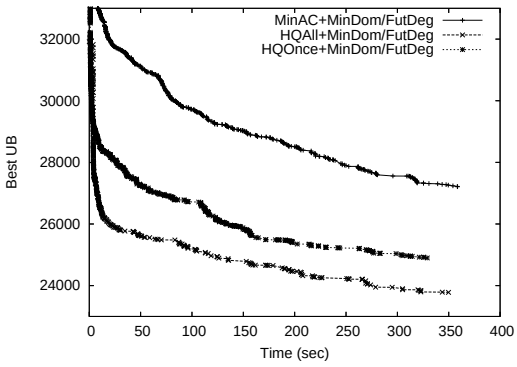

FIG. 12 – LDS(4)<50,10,48,80>

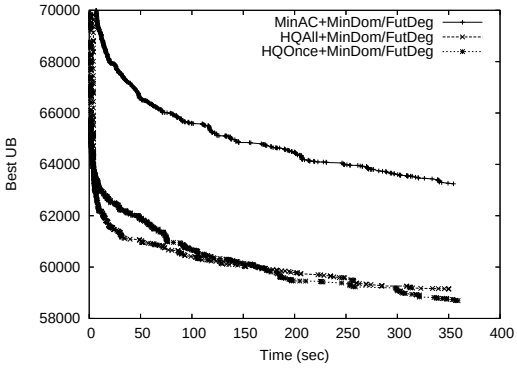

Fig. 13 – LDS(4)<50,10,20,40>

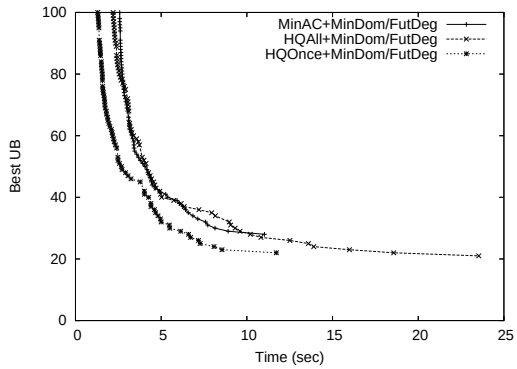

Fig. 14 – LDS(4)<50,10,20,80>

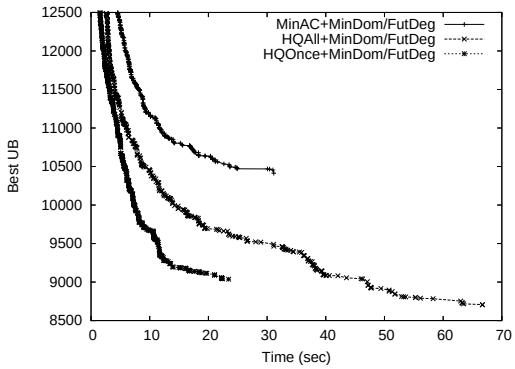

Fig. 15 – LDS(5)<50,10,20,60>

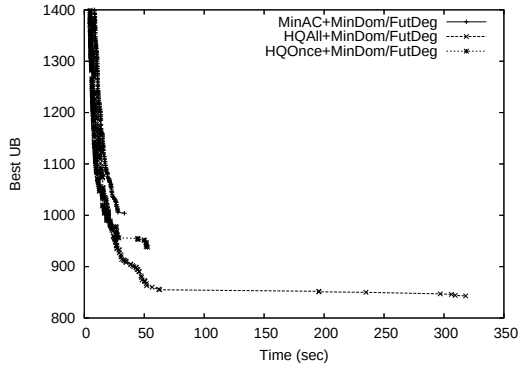

Fig. 16 – LDS(5)<50,10,48,60>

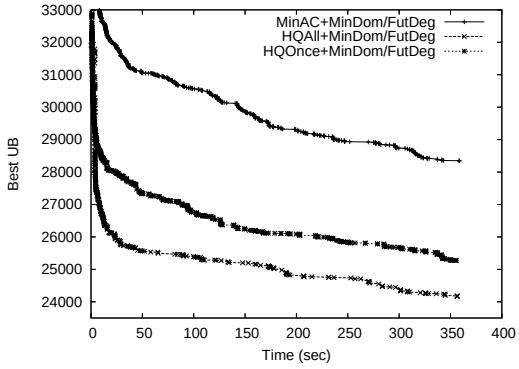

Fig. 17 – LDS(4)<50,10,20,60> PFC-DAC

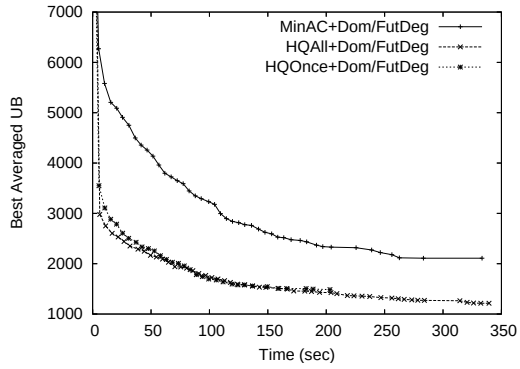
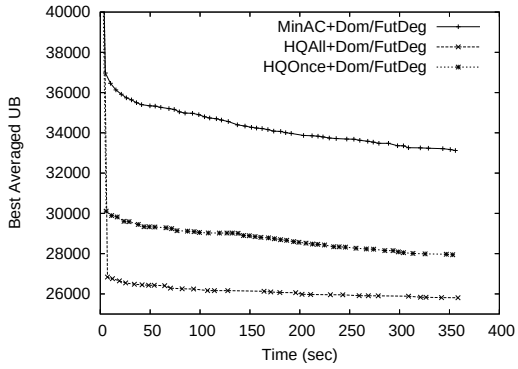
Fig. 18 – LDS(4)<50,10,48,60> PFC-DAC