Une approche parallèle coopérative exploitant la décomposition arborescente dans VNS

Abdelkader Ouali, Samir Loudni, Lakhdar Loukil, Yahia Lebbah

▶ **To cite this version:**

Abdelkader Ouali, Samir Loudni, Lakhdar Loukil, Yahia Lebbah. Une approche parallèle coopérative exploitant la décomposition arborescente dans VNS. 11ème Colloque sur l'Optimisation et les Systèmes d'Information et les Systèmes d'Information (COSI 2014), Jun 2014, Béjaia, Algeria. pp.P1-12. hal-01026276

HAL Id: hal-01026276
https://hal.science/hal-01026276

Submitted on 21 Jul 2014

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.

Une approche parallèle coopérative exploitant la décomposition arborescente dans VNS

Abdelkader Ouali[1], Samir Loudni[2], Lakhdar Loukil[1], Yahia Lebbah[1]

[1] Université d'Oran, Laboratoire LITIO, BP 1524, El-M'Naouer, 31000 Oran, Algeria
[2] Université de Caen Basse-Normandie, CNRS, UMR 6072 GREYC, 14032 Caen, France.

Résumé La décomposition arborescente introduite par Robertson et Seymour permet de découper un problème en un ensembles de sous-problèmes (clusters) constituant un graph acyclique. Récemment, Fontaine et al. [8] ont introduit `DGVNS` (Decomposition Guided `VNS`) qui utilise le *graphe des clusters* fourni par une décomposition arborescente afin de gérer l'exploration de grands voisinages. Cependant, sur des problèmes de grande taille, les performances de `DGVNS` décroient significativement en raison du nombre important de clusters à considérer séquentiellement. Pour y remédier, nous proposons `CPDGVNS` (Cooperative Parallel `DGVNS`) où les clusters sont explorés en parallèle dans une architecture maitre-esclaves. Les expérimentations effectuées sur des problèmes combinatoires difficiles montrent l'adéquation et l'efficacité de notre approche.

1 Introduction

La notion de décomposition arborescente, proposée par Robertson et Seymour [16], vise à découper un problème en sous-problèmes (*clusters*) constituant un graphe acyclique. Chaque cluster correspond à un sous-ensemble de variables fortement connectées. Chaque sous-problème, étant plus petit que le problème original, devient plus facile à résoudre. L'intérêt d'exploiter les propriétés structurelles d'un problème a été attesté dans divers domaines : pour vérifier la satisfiabilité dans SAT [15], pour résoudre les CSP (`CTE` [6]), dans les réseaux bayésiens (`AND/OR graph search` [14]), en bases de données relationnelles [9,10], sur des problèmes d'optimisation sous contraintes (`BTD` [19], `Lc-BTD`$^+$ [5], `RDS-BTD` [17]). Toutes ces propositions exploitent la décomposition arborescente dans des méthodes de recherche complète.

Plus récemment, Fontaine et al. [8] ont proposé la méthode `DGVNS` (Decomposition Guided `VNS`) permettant d'exploiter les clusters issus de la décomposition arborescente du graphe de contraintes, pour guider l'exploration des voisinages dans les méthodes de type `VNS`. Cependant, sur des problèmes de grande taille, les performances de `DGVNS` décroient significativement en raison du nombre important de clusters à considérer séquentiellement.

⋆⋆. Ce travail a été soutenu par l'Agence nationale de la Recherche, reference ANR-10-BLA-0214.

Dans ce papier, nous proposons une première stratégie de parallélisation de `DGVNS` appelée `CPDGVNS` (Cooperative Parallel `DGVNS`) qui consiste simplement à explorer tous les clusters en parallèle. `CPDGVNS` suit l'architecture maître-esclave, où le processus maître mémorise, met à jour, et communique la meilleure solution courante, alors que les processus esclaves gèrent l'exploration des clusters individuels. Les processus individuels coopèrent d'une façon asynchrone en échangeant des informations sur la meilleure solution courante. Ceci garantit l'indépendance des processus esclaves individuels et permet de démarrer à partir de plusieurs solutions initiales différentes, favorisant ainsi une meilleure diversification.

Les expérimentations effectuées sur des instances réelles (`RLFAP` et `tagSNP`) montrent que `CPDGVNS` produit un gain significatif en termes de temps d'exécution par rapport à `DGVNS`. A notre connaissance, notre proposition est la première tentative utilisant la décomposition arborescente pour paralléliser efficacement l'exploration de grands voisinnages dans `VNS`.

2 Définitions et notations

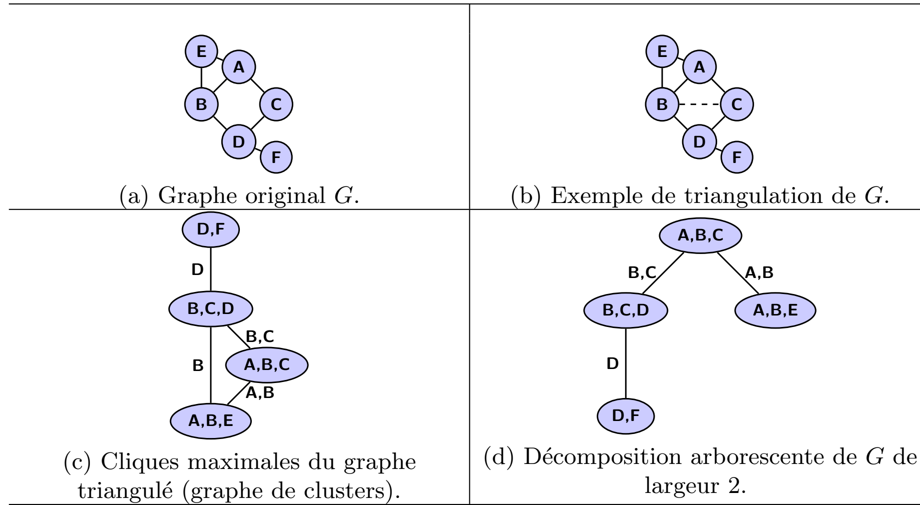
2.1 Réseau de fonctions de coût

Un réseau de fonctions de coût (CFN) est un couple (X, W) où $X=\{x_1, \ldots, x_n\}$ est un ensemble de n variables et W est un ensemble de e fonctions de coût. Chaque variable $x_i \in X$ a un domaine fini D_i de valeurs qui peuvent lui être affectées. La taille du plus grand domaine est notée d. Une affectation de x_i à la valeur $a \in D_i$ est notée (x_i, a). Pour un sous-ensemble de variables $S \subseteq X$, on note D^S le produit cartésien des domaines des variables de S. Pour un n-uplet donné t, $t[S]$ représente la projection du n-uplet t sur l'ensemble de variables S. Une affectation *complète* $t=(a_1, ..., a_n)$ est une affectation de toutes les variables ; dans le cas contraire, on l'appelle affectation *partielle*. Une fonction de coût $w_S \in W$, de portée $S \subseteq X$, est une fonction $w_S : D^S \mapsto [0, k_\top]$ où $k_\top$ est un coût entier maximum (fini ou non) utilisé pour représenter les affectations interdites (exprimant des contraintes dures). Pour capturer fidèlement les contraintes dures, les coûts sont combinés par l'addition bornée $\oplus$, définie par $\alpha \oplus \beta = \min(k_\top, \alpha + \beta)$. Le problème consiste à trouver une affectation complète t de l'ensemble des variables minimisant la combinaison des fonctions de coût $\bigoplus_{w_S \in W} w_S(t[S])$.

2.2 Décomposition arborescente

Le graphe de contraintes d'un CFN est un graphe G=(X,E) composé d'un sommet par variable et il existe une arête $\{u, v\} \in E$ si, et seulement si, $\exists\ w_S \in W, u, v \in S$.

Définition 1. *Une décomposition arborescente [16] de G=(X, E) est un couple (C_T, T) où : $T = (I, A)$ est un arbre avec pour ensemble de nœuds I et pour ensemble d'arêtes A, $C_T = \{C_i \mid i \in I\}$ est une famille de sous-ensembles de $\mathcal{X}$ (appelés clusters) telle que : (i) $\cup_{i \in I} C_i = \mathcal{X}$, (ii) $\forall (u, v) \in E$, $\exists C_i \in C_T$*



(a) Graphe original G.

(b) Exemple de triangulation de G.

(c) Cliques maximales du graphe triangulé (graphe de clusters).

(d) Décomposition arborescente de G de largeur 2.

Figure 1. Étapes de calcul d'une décomposition arborescente d'un graphe G.

t.q. $u,\ v \in C_i$, *(iii)* $\forall\, i, j, k \in I$, *si* j *est sur le chemin de* i *à* k *dans* T*, alors* $C_i \cap C_k \subseteq C_j$.

L'intersection entre deux *clusters* est appelée *séparateur*, et est notée $sep(C_i, C_j)$. Deux *clusters* sont *adjacents* s'ils partagent au moins une variable. Le *voisinage d'un cluster* C_i dans T est $voisinage(C_i)$=$\{C_j \mid j \in I, sep(C_i, C_j) \neq \emptyset\}$.

Définition 2. *Un graphe de clusters, pour une décomposition arborescente* (C_T, T)*, est un graphe non-orienté* $G_T = (C_T, E_T)$ *dont les sommets sont les éléments de* C_T *et il existe une arête* $(C_i, C_j) \in E_T$ *entre les sommets* C_i *et* C_j *ssi* $sep(C_i, C_j) \neq \emptyset$.

Il existe beaucoup de travaux sur les décompositions arborescentes. Habituellement, le problème considéré est de produire une décomposition avec une largeur arborescente minimale, un problème NP-difficile [1]. Des décompositions approchées, obtenues par *triangulation*, sont souvent exploitées.

Nous avons utilisé l'heuristique *Maximum Cardinality Search* (MCS) [18] permettant de produire des décompositions arborescentes avec des largeurs de décomposition plus petites.

La Fig. 1 illustre les trois étapes nécessaires au calcul d'une décomposition arborescente d'un graphe G (a). Tout d'abord, le graphe G est triangulé par l'ajout de l'arête BC (b). Ensuite, on calcule les cliques maximales afin de constituer le graphe de clusters (c). Enfin, on obtient la décomposition arborescente de G (d).

2.3 `VNS` guidée par la décomposition arborescente (`DGVNS`)

`DGVNS` (Decomposition Guided `VNS`) [8] étend le principe de `VNDS` [11] (Variable Neighborhood Decomposition Search), en exploitant le graphe de clusters

Algorithme 1 DGVNS

```
Require: The constraint graph (X,W), initial number of variables to unassign k_init, maximum
    number of variables to unassign k_max, discrepancy δ_max for LDS.
 1: let G be the constraints graph of (X, W)
 2: let (C_T, T) be a tree decomposition of G
 3: let C_T = {C_1, C_2, ..., C_p}
 4: S ← genRandomSol()
 5: k ← k_init, i ← 1
 6: while (k < k_max)∧(not TimeOut) do
 7:     Cand ← CompleteCluster(C_i, k)
 8:     X_un ← Hneighborhood(Cand, k, S)
 9:     A ← S\{(x_i, a) | x_i ∈ X_un}
10:     S' ← LDS+CP(A, X_un, δ_max, f(S), S)
11:     neighbourhoodChange(S, S', k, i)
12: end while
13: return S
14: procedure neighbourhoodChange(S, S', k, i)
15:     if f(S') < f(S) then
16:         S ← S', k ← k_init, i ← succ(i)
17:     else
18:         k ← k + 1, i ← succ(i)
19:     end if
20: end procedure
```

pour guider l'exploration de grands voisinages. Les voisinages sont obtenus en désaffectant une sous-partie de la solution courante selon une heuristique de choix de variables. La reconstruction de la solution sur les variables désinstanciées est effectuée par une recherche arborescente partielle, LDS (*Limited discrepancy search*, [12]), aidée par la propagation des contraintes (CP) basée sur un calcul de minorants.

Définition 3. *Soit* G_T=($\mathcal{C}_T$,E_T) *le graphe de clusters associé à* G. *Soient* $C_i \in \mathcal{C}_T$ *un cluster de* G_T *et* $k \in [1 \dots n]$ *la dimension du voisinage. La structure de voisinage* $N_{k,i}$ *désigne l'ensemble des combinaisons de* k *variables parmi* C_i.

L'algorithme 1 présente le pseudo-code de DGVNS. Tout d'abord, il construit une décomposition arborescente de G (ligne 2), puis génère aléatoirement une solution initiale S (fonction genInitSol, ligne 4)). Afin de favoriser les mouvements dans des régions fortement liées, DGVNS se base sur les structures de voisinages $N_{k,i}$ (cf. Définition 3). En effet, le concept de cluster permet d'exhiber ces régions, de part sa taille (plus petite que le problème initial), et de part la forte connexion entre les variables qu'il contient. Ainsi, l'ensemble de variables candidates $Cand$ à désaffecter sont sélectionnées à partir du cluster C_i. Si ($k > |C_i|$), $Cand$ est étendu aux variables des clusters C_j voisins de C_i afin de prendre en compte la topologie du graphe de clusters. Ce traitement est réalisé par la fonction CompleteCluster(C_i, k) (ligne 7). Un sous ensemble X_{un} de k variables est sélectionné aléatoirement dans $Cand$ parmi les variables en conflit par l'heuristique de voisinage Hneighborhood (line 8). Une affectation partielle $\mathcal{A}$ est générée à partir de la solution courante S en désaffectant les k variables de X_{un} (ligne 9). Ensuite, ces variables sont reconstruites (ligne 10) par une recherche arborescente partielle LDS [12], aidée par la propagation de contraintes (CP) (voir [13] pour plus de détails). La recherche s'arrête dès que la dimension maximale de voisinage k_{max} ou le *TimeOut* est atteint (ligne 6).

La procédure `NeighborhoodChange` contrôle les mécanismes d'intensification et de diversification de `DGVNS` (cf. Algorithme 1). Soit p le nombre total de clusters, *succ* une fonction de succession [3], et $N_{k,i}$ la structure de voisinage courante. Si `LDS+CP` ne trouve par de meilleure solution S' dans le voisinage de S, `DGVNS` cherche des améliorations dans $N_{(k+1),succ(i)}$ (la structure de voisinage où $(k+1)$ variables de $Cand$ seront désaffectées) (ligne 18)). Quand une solution de meilleure qualité S' est trouvée par `LDS+CP` dans le voisinage $N_{k,i}$, S' devient la solution courante (ligne 15), k est réinitialisé à k_{init} et le prochain cluster est considéré (ligne 16).

3 La méthode `CPDGVNS` (Cooperative Parallel `DGVNS`)

3.1 Architecture maître-esclave asynchrone de `CPDGVNS`

La motivation principale de la parallélisation est l'amélioration des performances des algorithmes liées au temps d'exécution et à la qualité des solutions. Plusieurs travaux sur la parallélisation des métaheuristiques confirment cette tendance. Les divers travaux sur les métaheuristiques parallèles indiquent également que, bien que les méthodes basées sur des stratégies de recherche indépendantes donnent de bons résultats, elles sont généralement surclassées par les stratégies de recherche *asynchrones coopératives* (voir [4] pour plus de détails). C'est cette approche que nous avons adoptée dans le présent papier.

Notre méthode, nommée `CPDGVNS` (pour *Cooperative Parallel* `DGVNS`), consiste simplement à explorer en parallèle tous les clusters fournis par une décomposition arborescente. Elle suit une architecture *maître-esclave* où le processus maître maintient, met à jour et communique la meilleure solution courante ; les processus esclaves explorent les clusters et coopèrent en échangeant des informations sur les meilleures solutions trouvées lors de la recherche. Ils communiquent exclusivement avec le processus maître. Les mises à jour des solutions et les communications s'effectuent de manière *asynchrone*. Ainsi, cette approche est plus avantageuse que l'approche synchrone car elle permet aux processus esclaves de démarrer à partir de solutions initiales différentes au cours des résolutions intermédiaires des clusters, permettant ainsi une plus grande diversification.

3.2 Algorithme du processus maître

Soient $C_T = \{C_1, ..., C_p\}$ l'ensemble des clusters et n_{sl} le nombre de processus esclaves utilisés. Dans notre approche, pour profiter pleinement de la parallélisation, le nombre n_{sl} [4] de processus esclaves est égal au nombre de clusters. Les processus esclaves sont numérotés de 1 à n_{sl}, alors que le processus maître sera désigné par la valeur zéro.

3. si $i < p$ alors $succ(i) = i + 1$ sinon $succ(p) = 1$.

4. Conceptuellement, le nombre de processus esclaves est égal au nombre de clusters. En pratique, si le nombre de coeurs est inférieur au nombre de clusters, un même coeur sera utilisé pour traiter différents clusters.

Algorithme 2 Processus `maître`

```
1: function CPDGVNS(X, W, k_init, δ_max, n_sl)
2:     let G be the constraints graph of (X, W)
3:     let (C_T, T) be a tree decomposition of G
4:     let C_T = {C_1, C_2, ..., C_p}
5:     S ← genRandomSol()
6:     i ← 1
7:     for each slave r = 1, ..., n_sl, in parallel do
8:         k_max ← |C_i|
9:         Send(r, i, k_init, k_max, δ_max, S)
10:        i ← succ(i)
11:    end for
12:    Finished ← 0, adj ← 0
13:    while (Finished < n_sl) do
14:        Receive(r, S'_r)
15:        if (f(S'_r) < f(S)) then
16:            S ← S'_r, adj ← 0
17:            i ← succ(i), k_max ← |C_i|
18:        else
19:            i ← succ(i), k_max ← |C_i|
20:            adj ← adj + 1
21:            for j = 1, ..., adj and adj ≤ |neighbor(C_i)| do
22:                Select the jth cluster C_j from neighbor(C_i)
23:                k_max ← k_max + |C_j|
24:            end for
25:        end if
26:        if (not global_TimeOut) then
27:            Send(r, i, k_init, k_max, δ_max, S)
28:        else Finished + +
29:        end if
30:    end while
31:    return S
32: end function
```

L'algorithme 2 décrit le pseudo-code du processus maître. Il démarre d'une décomposition arborescente de G (ligne 3) et d'une solution initiale S générée aléatoirement (ligne 5). Le maître initie la recherche en lançant l'exécution en parallèle de n_{sl} processus esclaves (ligne 7). Ceci est réalisé par l'envoie à chaque processus esclave r de la même solution initiale, du cluster associé C_i de l'ensemble C_T et des valeurs des paramètres k_{init}, k_{max} et δ_{max} (ligne 9). La liste C_T des clusters est gérée en FIFO pour assurer que tout cluster soit traité par un seul processus esclave (ligne 10). La valeur de k_{max} est initialisée à la taille du cluster affecté au processus esclave (ligne 8). Ceci restreint le choix du nombre de variables à désinstancier uniquement aux variables du cluster. Après la phase d'initialisation, le maître attend de recevoir la meilleure solutions trouvée par chaque processus esclave (lignes 13-28). La réception de ce message est traitée dans la ligne 14. Soit S'_r la meilleure solution communiquée par le processus esclave r au maître. Si S'_r est de meilleure qualité que S (ligne 15), S'_r devient la meilleure solution globale (ligne 16), le cluster suivant C_i est considéré et k_{max} est réinitialisé à $|C_i|$ (ligne 17). Sinon, on cherche à améliorer la solution dans le cluster suivant C_i (ligne 19) et on élargie l'ensemble des variables candidates à désinstancier en ajoutant les clusters C_j adjacents à C_i. Ce traitement est réalisé en augmentant le nombre de clusters adjacents adj à considérer (ligne 20) et la valeur de k_{max} en conséquence (lignes 21-23). Ceci est fait chaque fois que le processus esclave ne réussit pas à améliorer la meilleure solution globale.

Algorithme 3 Processus `esclave` r

Require: Tree decomposition (C_T, T)
1: RECEIVE$(r, i, k_{init}, k_{max}, \delta_{max}, S)$
2: $S_r \leftarrow S$
3: $k \leftarrow k_{init}$
4: **while** $(k < k_{max}) \wedge$(not local_TimeOut) **do**
5: $Cand \leftarrow$ COMPLETECLUSTER(C_i, k)
6: $X_{un} \leftarrow$ HNEIGHBORHOOD$(Cand, k, S_r)$
7: $\mathcal{A} \leftarrow S_r \backslash \{(x_i, a) \,|\, x_i \in \mathcal{X}_{un}\}$
8: $S'_r \leftarrow$ LDS+CP$(\mathcal{A}, X_{un}, \delta_{max}, f(S_r), S_r)$
9: NEIGHBOURHOODCHANGE(S_r, S'_r, k)
10: **end while**
11: SEND$(0, S_r)$
12: **procedure** NEIGHBOURHOODCHANGE(S, S', k)
13: **if** $f(S') < f(S)$ **then**
14: $S \leftarrow S'$
15: $k \leftarrow k_{init}$
16: **else**
17: $k \leftarrow k + 1$
18: **end if**
19: **end procedure**

Tout d'abord, la diversification effectuée en passant du cluster C_i au cluster $C_{succ(i)}$ est nécessaire. En effet, les expérimentations que nous avons menées ont montré que le fait de rester sur le même cluster conduit à de faibles améliorations : le choix d'un nouveau cluster permet d'améliorer la qualité de la solution en visitant de nouvelles zones de l'espace de recherche. De plus, quand un minimum local est trouvé dans le voisinage courant, l'augmentation de la valeur de k_{max} permettra également une certaine diversification en agrandissant la taille du voisinage. Si le délai *global_TimeOut* n'est pas atteint, on poursuit la recherche en relançant le processus esclave r à partir de la meilleure solution globale disponible (ligne 27). Sinon, elle est arrêtée (ligne 31). L'ensemble du processus de résolution se termine lorsque tous les processus esclaves se terminent (ligne 13).

3.3 Algorithme du processus esclave

L'algorithme 3 décrit le pseudo-code du processus esclave r. Il requiert la décomposition arborescente (C_T, T) de G. Il reçoit du maître l'index du cluster à traiter, les valeurs des paramètres k_{init}, k_{max}, la valeur du paramètre δ_{max} pour `LDS+CP` et la solution initiale S (ligne 1). Comme pour `DGVNS`, l'ensemble $Cand$ des variables candidates à désinstancier est choisi parmi les variables du cluster C_i. Si $(k > |C_i|)$ et $(k_{max} > |C_i|)$ (voir traitement réalisé par les lignes 21-23, Algorithme 2), alors on complète $Cand$ en considérant les clusters C_j adjacents à C_i. Ce traitement est réalisé par la fonction COMPLETECLUSTER(C_i, k) (ligne 5). Un sous-ensemble de k variables X_{un} est aléatoirement sélectionné dans $Cand$ parmi les variables en conflit par l'heuristique de voisinage HNEIGHBORHOOD

(ligne 6). Une affectation partielle $\mathcal{A}$ est générée à partir de la solution courante S_r en désinstanciant les k variables sélectionnées ; les $(n-k)$ variables non sélectionnées conservent leurs valeurs actuelles dans S (ligne 7). Les variables non affectées sont ensuite construites par `LDS+CP` (ligne 8). Si `LDS+CP` trouve une solution de mailleure qualité S'_r dans le voisinage de S_r (ligne 13) alors S'_r devient la solution courante (ligne 14) et k is est réinitialisé à k_{init} (ligne 15). Sinon, contrairement à `DGVNS`, le processus esclave cherche à améliorer la solution dans la structure de voisinage où $(k+1)$ variables de X seront désinstanciées (ligne 17). Ce traitement est réalisé par la procedure NEIGHBOURHOODCHANGE(S_r, S'_r, k) (ligne 9). La recherche s'arrête quand elle atteint le nombre maximal k_{max} de variables à désinstancier ou le *local_TimeOut* (ligne 4).

4 Jeux de test

Les expérimentations ont été réalisées sur les instances de deux problèmes modélisés sous forme de CFN (voir la Section 2.1).

Instances `RLFAP` : Le `CELAR` (Centre d'Electronique de l'Armement) a rendu publique un ensemble d'instances du problème d'affectation de fréquences radio (`RLFAP`) [3]. Le problème consiste à affecter un nombre limité de fréquences à un ensemble de liens radio définis entre des paires de sites, dans l'objectif de minimiser les interférences dues à la réutilisation des fréquences. Nous détaillons les expérimentations sur les instances les plus difficiles : Scen06, Scen07 et Scen08.

Instances `SPOT5` : La gestion quotidienne d'un satellite d'observation terrestre comme `SPOT5` nécessite la sélection d'un sous-ensemble de photos candidates conformément à des limitations d'ordre physique, en maximisant l'importance des photos sélectionnées [2]. Nous détaillons les expérimentations de sept instances parmi celles qui ne comportent pas des contraintes dures de capacité.

Instances `tagSNP` : Un polymorphisme nucléotidique simple "A Single Nucleotide Polymorphism" (SNP) est une variation d'une séquence DNA qui apparait quand un nucléotide unique - `A`, `T`, `C` ou `G` - dans le génome diffère entre des membres de l'espère biologique ou des paires de chromosomes dans un individu [7]. Les SNPs apparaissent comme des marques biologiques qui peuvent aider pour prédire le risque de développement de certaines maladies. Le problème tagSNP consiste en la sélection d'un petit sous-ensemble de SNPs, appelé tagSNPs, qui capture une grande part de l'information génétique. Ce problème est réputé difficile à résoudre, en raison de sa relation étroite au problème de couverture d'ensembles *(set covering problem)* qui est NP-dur [17]. Nous présentons des expérimentations sur neuf instances difficiles dérivées du chromosome humain 1-data[5] avec r_0=0.5 (jusqu'à n=1550 variables où la taille maximale du domaine d est comprise entre 30 et 266, ainsi que e=250, 000 fonctions de coût). Trois instances sont de taille moyenne, et six autres instances sont de grande taille.

5. `http://www.costfunction.org/benchmark`

Instance	Method	Succ.	Time	Avg.
Scen06 $S^* = 3,389$	CPDGVNS($n_{sl} = 12$)	**50/50**	**5.28**	**3,389**
$n = 100$, $d = 44$, $e = 1,222$	DGVNS	45/50	49.70	3,390.80
	VNS/LDS+CP	30/50	85.33	3,396
Scen07 $S^* = 343,592$	CPDGVNS($n_{sl} = 19$)	**50/50**	**221.07**	**343,592**
$n = 200$, $d = 44$, $e = 2,665$	DGVNS	50/50	344.48	343,592
	VNS/LDS+CP	22/50	1,980.93	346,535.36
Scen08 $S^* = 262$	CPDGVNS($n_{sl} = 46$)	**50/50**	**371.57**	**262**
$n = 458$, $d = 44$, $e = 5,286$	DGVNS	15/50	826.26	273
	VNS/LDS+CP	0/50	-	308 (284)

Table 1. Comparaison de CPDGVNS, DGVNS et VNS/LDS+CP sur les instances RLFAP.

5 Expérimentations

5.1 Le protocole expérimental

Pour comparer CPDGVNS à DGVNS et VNS/LDS+CP [6], nous avons adopté les mêmes paramètres que ceux décrits dans [8]. La valeur de déviation de LDS est fixée à 3. Les paramètres k_{min} et k_{max} sont respectivement fixés à 4 et à n (le nombre total de variables) de telle sorte que toutes les variables du problèmes soient couvertes, et la valeur *global_TimeOut* est fixée à 3600 secondes. En ce qui concerne CPDGVNS, le nombre de processus n_{sl} est fixé à $|\mathcal{C}_T|$ qui est le nombre de clusters de la décomposition arborescente. Tous les processus utilisent la même décomposition arborescente au niveau de chaque instance.

Les expérimentations ont été effectuées à l'unité de calcul intensif de l'université d'Oran en Algérie [7]. Chaque instance est exécutée 50 fois. Toutes les stratégies de recherche sont implantées en C++ en utilisant la librairie toulbar2 [8]. La parallélisation est mise en œuvre dans l'environnement MPI (Message Passing Interface) [9]. Par exemple, les routines de transmission *send* et *receive* correspondent respectivement aux procédures *MPI_Send* et *MPI_Recv* de la librairie MPI.

Pour évaluer l'impact de la stratégie de parallélisation, nous comparons la qualité des solutions obtenues avec VNS/LDS+CP, DGVNS et CPDGVNS en considérant le temps d'exécution. Nous avons fixé le temps d'exécution limite alloué à chaque processus esclave (i.e. *local_TimeOut*) à *global_TimeOut*/n_{sl}. Au niveau de chaque instance et chaque méthode, nous indiquons le nombre de processus esclaves utilisés par CPDGVNS, le nombre d'exécutions avec succès pour atteindre l'optimum, "succ. exécutions/exécutions totales", le temps CPU moyen (en secondes) des exécutions avec succès, le coût moyen sur les 50 exécutions, et le meilleur coût (entre crochets) des exécutions sans succès.

6. Pour DGVNS et VNS/LDS+CP, l'étape de reconstruction est effectuée en utilisant LDS+CP, et VNS/LDS+CP utilise les structures de voisinage N_k de dimension k.
7. http ://www.univ-oran.dz/uci/index.html
8. http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/SoftCSP
9. http ://www.mcs.anl.gov/research/projects/mpi/

Instance	Method	Succ.	Time	Avg.
#3792, $n = 528$, $d = 59$, $e = 12,084$, $S^* = 6,359,805$	CPDGVNS($n_{sl} = 70$)	**50/50**	**19.57**	**6,359,805**
	DGVNS	50/50	90.59	6,359,805
	VNS/LDS+CP	16/50	570.20	6,359,842.90
#4449, $n = 464$, $d = 64$, $e = 12,540$, $S^* = 5,094,256$	CPDGVNS($n_{sl} = 56$)	**50/50**	**16.61**	**5,094,256**
	DGVNS	50/50	36.81	5,094,256
	VNS/LDS+CP	38/50	155.35	5,094,258.18
#8956, $n = 486$, $d = 106$, $e = 20,832$, $S^* = 6,660,308$	CPDGVNS($n_{sl} = 54$)	**50/50**	**19.24**	**6,660,308**
	DGVNS	48/50	85.99	6,666,309.76
	VNS/LDS+CP	8/50	221.28	6,660,356.80
#6858, $n = 992$, $d = 260$, $e = 103,056$, $S^* = 20,162,249$	CPDGVNS($n_{sl} = 105$)	**40/50**	**398.93**	**20,833,413.20**
	DGVNS	33/50	2,788.92	20,565,101.90
	VNS/LDS+CP	34/50	2,840.54	20,296,702.96
#9150, $n = 1,352$, $d = 121$, $e = 44,217$, $S^* = 43,301,891$	CPDGVNS($n_{sl} = 120$)	**50/50**	**260.07**	**43,301,891**
	DGVNS	27/50	2,660	43,497,252.54
	VNS/LDS+CP	0/50	-	46,343,898.24 (44,696,532)
#14007, $n = 1,554$, $d = 195$, $e = 54,753$, $S^* = 50,290,563$	CPDGVNS($n_{sl} = 31$)	**50/50**	**665.54**	**50,290,563**
	DGVNS	19/50	2,523.38	50,913,924.36
	VNS/LDS+CP	0/50	-	59,193,679.32 (57,218,493)
#10442, $n = 908$, $d = 76$, $e = 28,554$, $S^* = 21,591,913$	CPDGVNS($n_{sl} = 25$)	**50/50**	**168.59**	**21,591,913**
	DGVNS	50/50	228.50	21,591,913
	VNS/LDS+CP	19/50	2,551.72	21,645,921.24
#14226, $n = 1,058$, $d = 95$, $e = 36,801$, $S^* = 25,665,437$	CPDGVNS($n_{sl} = 94$)	**50/50**	**159.18**	**25,665,437**
	DGVNS	50/50	295.78	25,665,437
	VNS/LDS+CP	8/50	2,958.70	25,665,620.22
#17034, $n = 1,142$, $d = 123$, $e = 47,967$, $S^* = 38,318,224$	CPDGVNS($n_{sl} = 120$)	**50/50**	**166.65**	**38,318,224**
	DGVNS	50/50	565.06	38,318,224.00
	VNS/LDS+CP	0/50	-	38,380,035.48 (38,318,309)

Table 2. Comparaison de CPDGVNS, DGVNS et VNS/LDS+CP sur les instances tagSNP.

5.2 La contribution de la Parallélisation

Pour quantifier la contribution de la parallélisation de DGVNS, nous commençons par comparer CPDGVNS, DGVNS et VNS/LDS+CP sur plusieurs instances de RLFAP et tagSNP décrites dans la section 4. Par la suite, nous synthétisons les résultats obtenus.

Instances RLFAP. En premier, CPDGVNS est clairement meilleur que DGVNS et VNS/LDS+CP sur les instances RLFAP (voir Table 1). CPDGVNS atteint l'optimum avec un taux de succès de 100% sur toutes les instances RLFAP. DGVNS a un taux de succès de 90%, 100% et 30% sur Scen06, Scen07 et Scen08 respectivement. VNS/LDS+CP est moins performante : il obtient un taux de succès de 60% et 44% sur Scen06, Scen07 respectivement, et n'atteint pas l'optimum au niveau de l'instance Scen08. Deuxièmement, le temps moyen d'exécution de CPDGVNS est nettement meilleur que le temps obtenu par DGVNS et VNS/LDS+CP, tout particulièrement sur l'instance Scen08, où CPDGVNS est 2.22 plus rapide que DGVNS.

Instances tagSNP. La table 2 compare CPDGVNS, DGVNS et VNS/LDS+CP sur les instances tagSNP. La table est décomposée en deux parties. La première partie contient des instances de taille moyenne, et la seconde partie contient des instances de grande taille. Au niveau des instances de taille moyenne, CPDGVNS est nettement plus performant que DGVNS et VNS/LDS+CP. L'optimum est atteint au niveau de toutes les 50 exécutions. CPDGVNS améliore le taux de succès d'environ

4% sur l'instance #8956, et obtient le même taux de succès sur le reste des instances par rapport à DGVNS. En plus, CPDGVNS est approximativement 3.77 fois plus rapide que DGVNS. VNS/LDS+CP montre un faible taux de succès sur toutes les instances par rapport à CPDGVNS et DGVNS, et prend nettement plus de temps pour atteindre l'optimum.

Sur les grandes instances, CPDGVNS domine VNS/LDS+CP et DGVNS en termes de taux de succès et de temps d'exécution, particulièrement sur les trois instances #6858, #9150 et #14007. CPDGVNS améliore le taux de succès d'environ 14% sur l'instance #6858, 50% sur l'instance #9150, et 38% sur l'instance #14007 comparativement à DGVNS. Nous observons que VNS/LDS+CP a un meilleur taux de succès que DGVNS sur l'instance #6858. Ceci peut être expliqué par la décomposition arborescente utilisée par DGVNS. En ce qui concerne CPDGVNS et DGVNS, les résultats de la recherche dépendent essentiellement de la qualité de la décomposition arborescente. Si une décomposition arborescente de moins bonne qualité est utilisée, alors DGVNS aura le même comportement que VNS/LDS+CP (pour plus de détails, voir [8]), menant souvent au même résultat. Toutefois, CPDGVNS améliore le taux de succès de VNS/LDS+CP d'environ 12% sur l'instance #6858. Nous notons aussi que CPDGVNS est approximativement 4.60 fois plus rapide que DGVNS.

Synthèse. Les expérimentations montrent clairement l'efficacité deCPDGVNS par rapport à DGVNS et VNS/LDS+CP sur les problèmes structurés comme RLFAP et tagSNP. Sur les instances RLFAP et tagSNP, CPDGVNS est plus performant que les deux stratégies DGVNS et VNS/LDS+CP au niveau du taux de succès et du temps d'exécution.

6 Conclusion

Dans ce papier, nous avons introduit une première approche parallèle coopérative pour DGVNS qui explore de manière parallèle les clusters issus de la décomposition arborescente du graphe de contraintes. CPDGVNS affecte à de nombreux processus esclaves des parties différentes de l'espace de recherche, permettant ainsi une meilleure diversification. Notre approche parallèle intensifie la recherche, via les processus esclaves, autour de la meilleure solution courante. Les résultats expérimentaux montrent que la version parallèle de DGVNS est beaucoup plus efficace que la version séquentielle, à la fois en taux de succès et en temps d'exécution. Nous sommes entrain de prospecter une meilleure coopération entre les processus esclaves sur des processeurs graphiques GPU.

Remerciements. Nous remercions l'Unité de Calcul Intensif de l'Université d'Oran et du CERIST pour avoir mis à notre disposition des ressources de calcul pour mener à bien nos expérimentations.

Références

1. S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM. J. on Algebraic and Discrete Methods*, 8 :277–284, 1987.

2. E. Bensana, M. Lemaître, and G. Verfaillie. Earth observation satellite management. *Constraints*, 4(3) :293–299, 1999.
3. B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio link frequency assignment. *Constraints*, 4(1) :79–89, 1999.
4. Teodor Gabriel Crainic, Michel Gendreau, Pierre Hansen, and Nenad Mladenovic. Cooperative parallel variable neighborhood search for the p-median. *J. Heuristics*, 10(3) :293–314, 2004.
5. S. de Givry, T. Schiex, and G. Verfaillie. Exploiting tree decomposition and soft local consistency in weighted csp. In *AAAI*, pages 22–27. AAAI Press, 2006.
6. R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artif. Intell.*, 38(3) :353–366, 1989.
7. C.S. Carlson et al. Selecting a maximally informative set of single-nucleotide polymorphisms for association analyses using linkage disequilibrium. *Am. J. of Hum. Genetics*, 74(1) :106–120, 2004.
8. M. Fontaine, S. Loudni, and P. Boizumault. Exploiting tree decomposition for guiding neighborhoods exploration for VNS. *RAIRO Oper. Res.*, 47(2) :91–123, 2013.
9. G. Gottlob, S. T. Lee, and G. Valiant. Size and treewidth bounds for conjunctive queries. In Jan Paredaens and Jianwen Su, editors, *PODS*, pages 45–54. ACM, 2009.
10. G. Gottlob, R. Pichler, and F. Wei. Tractable database design through bounded treewidth. In Stijn Vansummeren, editor, *PODS*, pages 124–133. ACM, 2006.
11. P. Hansen, N. Mladenovic, and D. Perez-Brito. Variable neighborhood decomposition search. *Journal of Heuristics*, 7(4) :335–350, 2001.
12. W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *Proc. of the 14th IJCAI*, pages 607–615, 1995.
13. S. Loudni and P. Boizumault. Combining VNS with constraint programming for solving anytime optimization problems. *EJOR*, 191 :705–735, 2008.
14. R. Marinescu and R. Dechter. AND/OR branch-and-bound search for combinatorial optimization in graphical models. *Artif. Intell.*, 173(16-17) :1457–1491, 2009.
15. I. Rish and R. Dechter. Resolution versus search : Two strategies for SAT. *J. Autom. Reasoning*, 24(1/2) :225–275, 2000.
16. N. Robertson and P.D. Seymour. Graph minors. ii. algorithmic aspects of treewidth. *J. Algorithms*, 7(3) :309–322, 1986.
17. M. Sánchez, D. Allouche, S. de Givry, and T. Schiex. Russian doll search with tree decomposition. In *IJCAI*, pages 603–608, 2009.
18. R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3) :566–579, 1984.
19. C. Terrioux and P. Jégou. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146(1) :43–75, 2003.