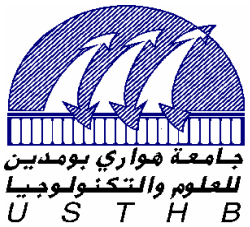
N° d'ordre : 07/2006-M/IN

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE HOUARI BOUMEDIENE (U.S.T.H.B)

Faculté d'Electronique et d'Informatique



MEMOIRE

Présenté pour l'obtention du diplôme de MAGISTER

En **: INFORMATIQUE**

Spécialité : Intelligence Artificielle et Bases de Données Avancées

Par: **M^{elle} SADEG Souhila**

Une approche par assemblage de performances pour la parallélisation de la métaheuristique 'Optimisation par Essaim d'Abeilles' adaptée au problème MAX-W-SAT

Soutenu publiquement le 19/07/2006 devant le jury composé de

M. AZZOUNE Hamid	Maître de conférence	U.S.T.H.B	Président
M^{me}. DRIAS Habiba	Professeur	U.S.T.H.B/I.N.I	Directrice de Mémoire
M. BABA-ALI Riadh	Maître de conférence	U.S.T.H.B	Examinateur
M. BALLA Amar	Maître de conférence	U.S.T.H.B	Examinateur

Remerciements

Je remercie en premier lieu, Mme Drias Habiba, ma directrice de recherche, d'avoir accepté de diriger ce travail malgré ses nombreuses occupations. Je la remercie de m'avoir accueillie au sein de l'Institut National de formation en Informatique (INI), en mettant à ma disposition des moyens qui m'ont énormément facilité le travail. Je lui suis extrêmement reconnaissante d'avoir guidé mes premiers pas dans le monde de la recherche, et de m'avoir fait bénéficier de sa riche expérience qui m'a été d'un grand apport, aussi bien sur le plan scientifique que sur le plan humain.

Je remercie vivement M.AZZOUNE Hamid, qui m'a fait l'honneur de présider le jury de ma soutenance. J'apprécie l'effort qu'il a fait en se penchant sur mon travail pour le juger.

Mes remerciements vont aussi à M.BABA-ALI Ryad, et M.BALLA Amar d'avoir accepté de faire partie du jury.

Dédicaces

Je dédie ce mémoire de magister :

A mon très cher père, qui a toujours cru en moi plus que moi-même.

A la plus tendre et la plus merveilleuse des mères, qui m'a soutenue et supportée durant tout ce temps. Merci maman.

A ma chère sœur et confidente Lamia, qui est toujours là pour moi.

A ma petite sœur Ines pour qui j'exprime toute ma tendresse.

A mon rayon de soleil, mon petit frère Lyes, qui sait toujours trouver les mots pour me redonner le sourire.

Enfin, A Yacine Hemdane, que je remercie sincèrement pour ses conseils et ses encouragements.

A toutes ces personnes qui me sont si chères, je souhaite une longue vie, et encore une fois, merci.

Table des matières

Chapitre III Calcul parallèle et stratégies de parallélisation des métaheuristiques

Chapitre IV. Parallélisation de la métaheuristique BSO

Chapitre V : Implémentation et résultats expérimentaux

Introduction générale

La résolution des problèmes combinatoires a toujours suscité l'intérêt des chercheurs. Certains de ces problèmes, appelée problèmes NP-Complets, occupent une place privilégiée vu le nombre important de leurs applications et la difficulté de leur résolution avec des méthodes classiques. C'est le cas du problème de satisfiabilité maximale pondérée ou MAX-W-SAT en abrégé.

Ces dernières années, l'avènement de nouvelles méthodes de résolution, appelées métaheuristiques, a permis de réaliser des progrès significatifs en terme de qualité des solutions obtenues et de temps de calcul.

En effet, les métaheuristiques sont largement reconnues pour être des outils essentiels pour aborder des problèmes difficiles dans de nombreux domaines. En fait, les métaheuristiques offrent souvent la seule approche pratique pour résoudre des problèmes complexes à l'échelle réelle.

Cependant, même en utilisant les métaheuristiques, les limites de ce qui pourrait être résolu en un temps raisonnable sont vite atteintes; du moins, trop rapidement pour les besoins croissants de la recherche et de l'industrie. Le calcul parallèle, offre la perspective de pouvoir réduire le temps de calcul.

Parmi la multitude de métaheuristiques existantes, certaines s'inspirent du comportement fascinant des insectes sociaux, caractérisés par leur organisation et leur coopération. Ces métaheuristiques émergent d'un nouvel axe de recherche en intelligence artificielle, appelé 'Intelligence en essaim'. Ant Colony Optimzation, qui s'inspire du comportement des fourmis, a été proposée il y a une dizaine d'années et a fait ses preuves dans de nombreuses applications industrielles.

Bees Swarm Optimization, une nouvelle métaheuristique proposée dans le cadre de notre PFE, s'inspire du comportement des abeilles réelles. Celui-ci est impressionnant à plus d'un titre. En effet, ces petits insectes jouissent d'une capacité de coopération étonnante et d'un mécanisme de communication impressionnant, dont la découverte a valu un prix Nobel à l'ethnologue autrichien Karl Von Frisch.

Le travail mené dans le cadre de ce magistère, vise à proposer une parallélisation de la métaheuristique BSO, dans le but d'améliorer ses performances. Pour cela, nous avons adopté une démarche constructive qui consiste à considérer, un à un, les différents aspects et

caractéristiques des l'algorithmes parallèles. Vu que pour chacun, plusieurs alternatives sont possibles, au lieu de concevoir directement une méthode de résolution, nous avons choisi de procéder par étapes, en concevant un algorithme pour chaque alternative. La comparaison des résultats obtenus par les différentes solutions, a permis de sélectionner la meilleure d'entre elles. Enfin, nous avons assemblé les meilleures alternatives dans un algorithme final.

Notre travail est réparti en 5 chapitres organisés comme suit:

Dans le premier chapitre, nous présentons le problème de satisfiabilité, SAT, ainsi que certaines de ses variantes, telles que MAX-W-SAT. Nous donnons aussi quelques exemples de ses nombreuses applications. Nous consacrons une bonne partie du chapitre à la présentation des principales méthodes de résolution, notamment, les métaheuristiques.

Le second chapitre, est consacré à la présentation de l'intelligence en essaim et de la métaheuristique optimisation par essaim d'abeilles (BSO pour Bees Swarm Optimization). Nous détaillons les différentes étapes de son algorithme, et son adaptation à la résolution du problème MAX-W-SAT.

Le chapitre trois, est consacré à l'étude du calcul parallèle, et des stratégies de parallélisation des métaheuristiques. Nous présentons deux classifications de ces stratégies, publiées dans la littérature ainsi que leur application sur quelques métaheuristiques telles que: Le recuit simulé, les algorithmes génétique, la recherche taboue, la procédure GRASP et le système de colonie de fourmis.

Dans le quatrième chapitre, nous présentons la proposition avancée. Nous décrivons les algorithmes conçus pour les différentes solutions envisagées.

Le dernier chapitre, expose les résultats des nombreux tests effectués. L'analyse des performances des différentes méthodes de résolution, nous a permis de sélectionner les meilleures solutions, et de les assembler dans un algorithme final. Les performances de ce dernier, sont comparées à celles de la BSO séquentielle et aux résultats d'autres métaheuristiques.

Une conclusion générale clos le mémoire, en présentant un résumé du travail effectué et des perspectives pour les travaux futurs.

Chapitre I

Le problème MAX-W-SAT

1. Introduction

2. Le problème de satisfiabilité (SAT)

3. Le problème de satisfiabilité maximum (MAX-SAT)

4. La notion de NP-Complétude

5. Applications du problème MAX-W-SAT

6. Les méthodes de résolution

7. Conclusion

1. Introduction

L'objet de ce chapitre introductif est de : définir le problème SAT ainsi que ses variantes, entre autres MAX-W-SAT, donner la notion de NP-Complétude, esquisser ses domaines d'application, et enfin, présenter les principales méthodes utilisées pour sa résolution, à savoir: Les méthodes exactes et les méthodes approximatives.

2. Le problème de satisfiabilité (SAT)

2.1. Définition [PET00]

Le problème de satisfiabilité, SAT en abrégé, est défini comme suit :

- Soit $X = \{ x_1 , x_2 , \ldots , x_n \}$, un ensemble de n variables booléennes.
- Soit $C = \{ c_1 , c_2 , \ldots , c_m \}$, un ensemble de m clauses où :
  - chaque clause est une disjonction de littéraux.
  - chaque littéral est une variable (un littéral positif) ou sa négation (un littéral négatif).
- Soit D - la donnée SAT - une conjonction des clauses de C.

Le problème SAT consiste à déterminer, s'il existe une assignation (affectation) aux variables x_i de X , telle que la donnée SAT D soit satisfaite (évaluée à VRAI).

Exemples :

Dans ce qui suit, le *ou logique* est dénoté par '+', la *négation* d'une variable par '¬', la valeur *VRAI* par '1' et la valeur *FAUX* par '0'.

* Soit *D1 :*
$$\begin{cases} c_1 = x_1 + !x_2 \\ c_2 = \neg x_1 + x_2 + x_3 \\ c_3 = \neg x_1 + x_4 \end{cases}$$

$X = \{x_1, x_2, x_3, x_4\}$, $C = \{c_1, c_2, c_3\}$.

Si $(x_1 , x_2 , x_3, x_4) = (1 , 1 , 0 , 1)$ alors la donnée est satisfaite .

* Soit *D2* :
$$\begin{cases} c_1 = x_1 + x_2 \\ c_2 = x_1 + \neg x_2 \\ c_3 = \neg x_1 + x_2 \\ c_4 = \neg x_1 + \neg x_2 \end{cases}$$

$X = \{x1, x_2\}$. $C = \{c_1, c_2, c_3, c_4\}$.

Quelque soit l'assignation des variables, cette donnée est insatisfaite.

2.2. Les variantes du problème SAT [PET00]

La restriction du problème SAT aux cas où toutes les clauses ont la même longueur *k* (la longueur d'une clause étant le nombre de ses littéraux), est appelée *k*-SAT.

Un intérêt particulier est porté aux valeurs 2 et 3 de *k*. En effet, 3 est la plus petite valeur pour laquelle k-SAT est NP-Complet, et, 2-SAT peut être résolu en un temps linéaire.

Horn-SAT représente les problèmes SAT, où, chaque clause a au plus une variable sans négation, résolus également en temps linéaire.

3. Le problème de satisfiabilité maximum (MAX-SAT)

3.1. Définition :

Le problème MAX-SAT se définit comme suit : Etant donnés les ensembles *X* et *C* définis précédemment, résoudre le problème MAX-SAT revient à chercher une assignation aux variables x_i de *X* satisfaisant le plus grand nombre possible de clauses de *C* [HAJ90].

Exemple :

On reprend la donnée *D2* :
$$\begin{cases} c_1 = x_1 + x_2 \\ c_2 = x_1 + \neg x_2 \\ c_3 = \neg x_1 + x_2 \\ c_4 = \neg x_1 + \neg x_2 \end{cases}$$

Quelle que soit l'assignation des variables, le nombre maximum de clauses satisfaites pour cette donnée est de 3.

3.2. La variante MAX-W-SAT :

Si en outre, on associe à chaque clause c_i un poids *w(ci)*, l'objectif revient ainsi à trouver une assignation aux variables x_i de *X,* qui maximise la somme des poids des clauses satisfaites simultanément, d'où la définition du problème MAX-SAT pondéré, ou MAX-W-SAT pour WEIGHTED-MAX-SAT.

Il importe de dire que le problème MAX-SAT, est un cas particulier du problème MAX-W-SAT, et ceci en associant à toutes les clauses le même poids (1 par exemple).

Exemple :

Soit la donnée SAT *D* :

$c_1 = \neg x_1 + x_2$ $\quad w(c_1)=10$.

$c_2 = \neg x_1 + \neg x_3 + x_4$ $\quad w(c_2)=20$.

$c_3 = x_4$ $\quad w(c_3)=25$.

Si $(x_1, x_2, x_3, x_4) = (0, 0, 0, 1)$, alors la somme des poids des clauses satisfaites simultanément atteint son maximum.

3.3. Le concept Partial MAX-SAT : [PET00]

Très récemment, le concept Partial MAX-SAT a été introduit. Il est voué, dans le cas de deux données SAT *D1* et *D2*, à satisfaire toutes les clauses de *D1,* et le maximum de clauses de *D2* : c'est le principe de l'hybridation des problèmes SAT et MAX-SAT.

4. La Notion de NP-Complétude :

Le problème de satisfiabilité SAT est le premier problème démontré NP-Complet. [COO71].

La NP-Complétude est une notion informatique prépondérante. Sa définition est inhérente à d'autres notions aussi importantes, telles que : la complexité, le non déterminisme et la réductibilité.

4.1. Complexité algorithmique : [PAR98]

La complexité algorithmique est un critère permettant l'analyse, et la comparaison des performances algorithmiques.

Pour un algorithme donné, on cherche à évaluer :

- Sa *complexité temporelle* : le nombre d'opérations élémentaires qu'il effectue.
- Sa *complexité spatiale* : la taille de l'espace mémoire qu'il requiert pour son exécution.

La complexité d'un algorithme est fonction de la (des) donnée(s) caractéristique(s) de l'instance du problème qu'il résout. Ainsi, pour un problème mettant en œuvre un ensemble d'objets, elle pourrait être sa cardinalité. Pour la manipulation d'un graphe défini explicitement, ce sera par exemple, la somme des sommets et des arcs. Pour un problème de satisfiabilité, ce sera le nombre de variables, et le nombre de clauses.

Dire qu'un algorithme est en *O(f(n)),* signifie que sa complexité temporelle est bornée par $c * f(n)$ lorsque n tend vers l'infini, tel que : c est une constante réelle et f est fonction de la donnée caractéristique n .

Exemples :

- ✓ Soit l'algorithme de tri connu sous le nom de «Tri-Bulle », la donnée caractéristique est le nombre d'éléments à trier n, l'opération élémentaire est la permutation. Sa complexité temporelle C_t est calculée comme suit :

$$C_t = n\text{-}1+n\text{-}2+\ldots+3+2+1 = n(n\text{-}1)/2 \rightarrow n^2, \quad lorsque\ n \rightarrow infini$$

- ✓ Soit l'algorithme qui fait la multiplication de deux matrices carrées d'ordre n, on prend comme donnée caractéristique l'ordre de la matrice, et comme opérations élémentaires : l'addition et la multiplication. La complexité temporelle C_t de cet algorithme est donnée par :

$$C_t = n^2 (n (multi) + (n\text{-}1) (add)) \rightarrow n^3, \quad lorsque\ n \rightarrow infini .$$

- ✓ Soit l'algorithme permettant de déterminer si une donnée SAT est satisfiable. On prend comme opération élémentaire l'interprétation d'une combinaison de ses variables. Puisqu'il existe 2^n combinaisons possibles, il en existe autant d'interprétions. Donc $Ct = 2^n$.

Asymptotiquement, un algorithme en *O (n log n)* est plus performant qu'un algorithme en $O(n^2)$, lequel est beaucoup plus performant qu'un algorithme en $O(2^n)$.

Les algorithmes sont classés, selon leur complexité, en deux classes : Les algorithmes *polynomiaux* et les algorithmes *exponentiels*.

- Un algorithme est dit polynomial, s'il est en $O(P(n))$, P étant un polynôme dont la variable est la donnée caractéristique n.
- Un algorithme est dit exponentiel, par convention, s'il n'est pas polynomial.

4.2. Réductibilité d'algorithmes : [GAJ79]

Un problème Q est réductible en un problème P, s'il existe une fonction g de complexité polynomiale, telle que pour toute solution x de P, $y=g(x)$ soit une solution de Q. On dit que le problème P est plus général que le problème Q.

4.3. Déterminisme et non déterminisme : [PAR98]

Le concept de non déterminisme permet d'introduire une notion purement théorique d'algorithmes très puissants.

Contrairement à un algorithme déterministe défini par une suite d'instructions bien déterminées, un algorithme non déterministe, a la remarquable capacité, lors d'une étape du choix d'une alternative parmi plusieurs, d'effectuer toujours le bon choix, en adoptant l'alternative qui mène à une solution, si celle-ci existe.

Ce concept abstrait n'est pas dénué d'intérêt : il constitue la base de toute la théorie de la NP-Complétude.

4.4. Classes des problèmes : [PAR98]

4.4.1. La classe P :

La classe **P** contient tous les problèmes relativement faciles, c'est-à-dire, ceux pour lesquels on connaît des algorithmes efficaces. Plus formellement, ce sont les problèmes pouvant être résolus par un algorithme déterministe de complexité **P**olynomiale.

4.4.2. La classe NP :

La classe NP, est l'ensemble des problèmes dont la résolution pourrait être faite par un algorithme **N**on déterministe, de complexité **P**olynomiale. En quelque sorte, c'est la classe des

problèmes qui auraient été bien résolus, si on disposait de la puissance considérable des machines non déterministes.

Pour savoir si un problème donné appartient ou non à la classe NP, il suffit de pouvoir vérifier efficacement (en un temps polynomial), si une solution donnée est correcte. Par exemple, le problème consistant à trouver un cycle hamiltonien dans un graphe appartient à **NP,** puisque étant donné un cycle, il est facile de vérifier en un temps linéaire, qu'il contient bien une et une seule fois chaque sommet.

Il importe de remarquer que $P \subset NP$. On ne sait toujours pas si P=NP.

4.4.3. La classe NP-Complet :

La classe NP-Complet est un sous ensemble des problèmes de la classe NP : les problèmes les plus difficiles.

Un problème de cette classe, possède la propriété que tout problème NP lui est réductible. Ainsi, si un problème NP-Complet admet un algorithme déterministe polynomial, alors tout problème NP en admet un.

Il existe aujourd'hui un long répertoire de problèmes NP-Complets ayant trait aux systèmes d'exploitation, aux systèmes de base de données, à la recherche opérationnelle, à la logique et à la théorie des graphes.

Pour prouver qu'un problème *P* est NP-Complet, il faut partir d'un autre problème *Q* que l'on sait NP-Complet, et tenter de le réduire en *P*. Cependant, une question pertinente se pose : Comment a-t-on bien pu démontrer le premier problème NP-Complet ?

En 1971, Cook a démontré, en utilisant directement la définition de la NP-Complétude., que tout problème de NP peut être réduit en un problème dit de satisfiabilité : il s'agit du problème SAT que nous avons défini précédemment.

5. Applications du problème MAX-W-SAT :

Bien que le problème MAX-W-SAT semble théorique et sans grand intérêt, il possède plusieurs applications importantes.

Il suscite un intérêt particulier dans divers domaines notamment: L'intelligence artificielle, la logique mathématique, la théorie des graphes, la recherche opérationnelle, les réseaux, les bases de données évolutives, la vision par ordinateur, les architectures VLSI, le raisonnement automatique, les jeux de puzzles,… etc.

La base de connaissance d'un système expert correspond à un ensemble de règles exprimées par des implications et pouvant être transformées en un ensembles de clauses.

Après la mise à jour de cette base par une série d'ajouts et de suppressions de certaines règles, se pose le problème de la vérification de sa consistance. Si la base est inconsistante, on préfère trouver le plus petit ensemble de règles devant être supprimées pour éliminer l'incohérence. Un tel ensemble peut être obtenu en résolvant un problème MAX-W-SAT [DRK01].

Un graphe à niveaux, est défini comme étant un graphe muni d'une fonction, qui associe à chacun de ses nœuds une valeur, leur permettant d'être classés par plans horizontaux (niveaux).

La condition pour obtenir une bonne lisibilité dans la présentation des graphes à niveaux (par exemple les hiérarchies), est de produire des diagrammes dont le nombre de croisements entre les arcs est minimum.

Ce problème possède diverses applications, à titre d'exemple : la visualisation des diagrammes de flux de données, des diagrammes entité-association et des hiérarchies des classes orientées objets.

Récemment, *Damien Petit* a utilisé le concept Partial MAX-SAT pour sa résolution [PET00].

6. Les méthodes de résolution :

Il existe deux alternatives pour résoudre une instance du problème MAX-W-SAT : Soit on cherche l'optimum en risquant l'explosion combinatoire, ou bien, on se contente d'une solution proche de l'optimale, pouvant être trouvée en un temps raisonnable.

6.1. Les algorithmes exactes (ou complets) :

Dans ces algorithmes, on utilise les méthodes arborescentes, la programmation linéaire ou la programmation dynamique.

Davis et *Putnam* ont introduit une méthode énumérative, dite « Generate & test », ou « Brute Force ». Dans cette ùéthode, les variables sont instanciées une par une. Après chaque instanciation d'une variable, on simplifie la formule en éliminant les clauses qui contiennent le littéral évalué à VRAI par cette affectation, ainsi que toutes les occurrences des littéraux évalués à FAUX. Ensuite, on effectue les tests suivants :

- Si la formule ne contient aucune clause ,alors, retourner "satisfiable".
- Si la formule contient une clause vide (ne contenant aucun littéral) alors, retourner "non satisfiable".
- Sinon continuer l'instanciation des autres variables.

Après le test, si la valeur retournée est « non satisfiable » alors, on réaffecte à la dernière variable instanciée la négation de la valeur assignée auparavant puis, on refait le test. [PET00]

6.2. Les algorithmes approximatifs :

Pour contourner l'explosion combinatoire des algorithmes exacts, les chercheurs se sont dirigés vers le développement des algorithmes approximatifs, pour la résolution du problème MAX-W-SAT, et ceci, en utilisant des heuristiques et des métaheuristiques.

Alors qu'une heuristique est une méthode approchée, conçue pour un problème d'optimisation donné, une métaheuristique est plutôt générale, adaptable et applicable à une large classe de problèmes. Sommairement, on peut distinguer des métaheuristiques qui se basent sur une solution unique, et des métaheuristiques qui utilisent une population de solutions (voir figure1). [RIA]

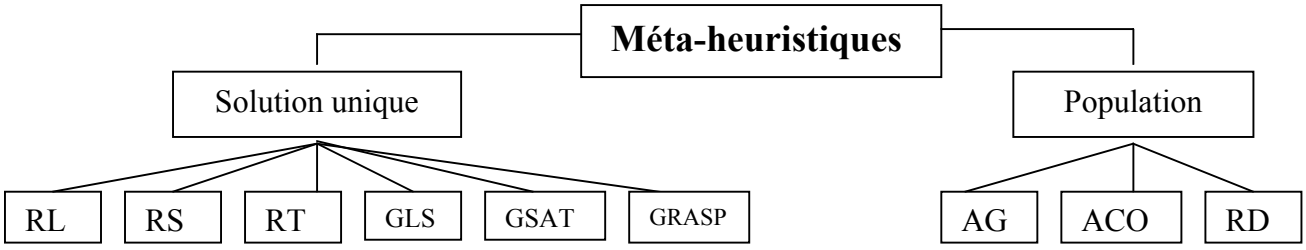


Figure 1. Classement des principales métaheuristiques.

RL: Recherche Locale. RS: recuit Simulé

RT: Recherche Taboue. GLS: Guided Local Search (Recherche Locale Guidée)

AG: Algorithmes génétiques. (Genetic Algorithms)

RD: Recherche Dispersée (Scatter Search).

GRASP: Greedy Randomized Adaptative Search Procedure.

En plus des métaheuristiques, certaines heuristiques ont été développées pour la résolution du problème MAX-W-SAT. Les Algorithmes JOHNSON en sont le meilleur exemple.

6.2.1. Les algorithmes de David JOHNSON [JOH74]

6.2.1.1 Algorithme JOHN 1 :

Cette heuristique, consiste à choisir à chaque étape, le littéral occurrent dans le plus grand nombre de clauses. Si le littéral sélectionné est positif, la variable correspondant est fixée à VRAI, sinon elle est fixée à FAUX. Cet algorithme a pour but de satisfaire le plus grand nombre de clauses, c'est-à-dire, toutes celles qui contiennent le littéral choisi. Celles-ci sont alors supprimées de l'ensemble des clauses de la formule.

JOHN1

Début

S ← Φ;

LEFT ← C;

X←{x / x variable dans C} ;

Répéter

Chercher *l*, avec x (*l*) ∈ X, qui apparaît dans le maximumde clauses de LEFT ;

Soit $\{c_{l1}, c_{l2},\ldots, c_{lk}\}$

S ← S ∪ $\{c_{l1},\ldots,c_{lk}\}$;

LEFT ←LEFT \ $\{c_{l1},\ldots,c_{lk}\}$;

Si *l* est positif alors x (*l*) ←Vrai **sinon** x (*l*) ←Faux **fsi** ;

X ← X \ {x (*l*)} ;

Jusqu'à ce qu'aucun littéral *l* avec x (*l*) ∈X ne soit contenu dans une clause LEFT.

Si X ≠ ∅ **Alors** pour tout x∈ X **faire** x←Vrai **fsi** ;

Retourner la solution obtenue ;

Fin

Tel que :

C : l'ensemble de toutes les clauses.

LEFT : l'ensemble des clauses non encore satisfaites (LEFT = C - S)

S : l'ensemble des clauses satisfaites.

X : l'ensemble des variables

x (*l*) : la variable associée au littéral *l*.

6.2.1.2. Algorithme JOHN2 :

Johnson a introduit un second algorithme, qui améliore la performance du premier en affectant une masse $W(C_i)=2^{-|Ci|}$ à chaque clause ($|C_i|$ dénote la longueur de C_i). Les clauses ayant moins de littéraux, ont un poids plus élevé car les grandes clauses sont plus faciles à satisfaire. En plus, à chaque itération de l'algorithme, ce poids augmente pour chaque clause où on vient de fixer un littéral à FAUX (en fixant son opposé à VRAI).

JOHN2

Début
S ← ∅;
LEFT ← C ;
X←{x / x variable dans C} ;
Assigner à chaque clause c_i un poids $w(c_i)= 1/2^{|ci|}$
Répéter
Chercher x ∈ X, qui apparaît dans au moins une clause de LEFT
Soient CT les clauses de LEFT contenant x, et CF celles contenant !x.
Si $\sum_{ci \in CT} w(c_i) \geq \sum_{ci \in CF} w(c_i)$ **alors**
x (*l*) ←Vrai;
S←S ∪ CT ;
LEFT ← LEFT\CT
Pour toute $c_i \in$ CF **faire** w(ci) ← 2.w(ci);
Sinon
x (*l*) ←Faux;
S←S U CF ;
LEFT ← LEFT\CF
Pour toute $c_i \in$ CT **faire** w (ci) ← 2.w (ci) ;
Fsi
Jusqu'à ce qu'aucun littéral *l* avec x (*l*) ∈X ne soit contenu dans une clause de LEFT ;
Si X ≠ ∅ alors pour tout x ∈X **faire** x ← vrai ;
Retourner la solution obtenue ;
Fin

6.2.2. Algorithme aléatoire

L'intérêt de cet algorithme, réside dans la diversité des solutions qu'il propose, vu qu'il les construit d'une manière aveugle. Il est sous forme d'une procédure appelée Random qui se présente comme suit :

RANDOM

Début

Fixer un entier k ; /*le nombre de solution à générer.*/

s← assignation aléatoire ;

/* f(x), la fonction objectif de x, se définit comme étant le nombre de clauses satisfaites pour Max-Sat*/

Répéter

s^*← Solution générée aléatoirement ;

Si $f(s^*) > f(s)$ **alors** $s \leftarrow s^*$; **fsi**

Jusqu'à k ;

Fin

Les performances de l'algorithme ne sont intéressantes, que dans le cas où il est exécuté à plusieurs reprises.

6.2.3. La recherche locale (local search)

La recherche locale, appelée aussi la descente ou l'amélioration itérative, représente une classe de méthodes heuristiques très anciennes, elle est souvent utilisée dans les métaheuristiques. Traditionnellement, la recherche locale constitue une arme redoutable pour attaquer des problèmes réputés très difficiles, tels que le voyageur de commerce. [RIA]

Le fonctionnement général de la RL est de :

- Générer un point initial dans l'ensemble des solutions possibles.
- Essayer d'améliorer la fonction objectif f (f étant la somme des poids des clauses satisfaites pour MAX-W-SAT), en faisant des transitions dans le voisinage de la solution courante.

L'efficacité le la recherche locale, dépend essentiellement du voisinage choisi. Pour le problème MAX-W-SAT, l'espace de recherche est constitué de toutes les assignations possibles ($U= \{o, 1\}^n$ pour un problème de taille n), et une transition est l'inversion de la valeur d'une variable. Le voisinage d'une solution X se définit par l'ensemble de toutes les solutions obtenues par une transition à partir de X :

Voisinage(X) ={ $(x_1, \ldots, 1\text{-}x_i, \ldots, x_n)$ pour i :=1..n }.[BAP97]

Cette procédure fait intervenir à chaque itération, le choix d'un voisin qui améliore la configuration courante. Plusieurs possibilités peuvent être envisagées pour effectuer ce choix: Il est possible d'énumérer les voisins jusqu'à ce qu'on en découvre un qui l'améliore strictement (première amélioration). On peut également rechercher le meilleur voisin (meilleure amélioration). Cette dernière solution peut sembler plus coûteuse, mais le voisin découvert sera en général de meilleure qualité. On peut obtenir de bons résultats avec la RL, et ce, avec de multiples exécutions de solutions différentes à chaque fois pour obtenir un optimum local, qui peut s'approcher de l'optimum global [RIA]. L'algorithme de la recherche locale la plus simple est le suivant :

La recherche locale

```
Début
    s ← une solution aléatoire ;
    sortir ← faux ;
    Tant que non sortir Faire
        s' ← meilleur voisin de s ;
        Si f(s') > f(s) alors
        s ← s'
        Sinon sortir ← vrai ; fsi
    Fait
Fin
```

Comme l'espace des solutions est fini, cette procédure de descente s'arrête toujours, et, la dernière configuration trouvée ne possède pas de voisin strictement meilleur qu'elle même. Autrement dit, la recherche locale retourne toujours un optimum local.

Une amélioration de la RL a été définie, où un nombre spécifique d'itérations est exécuté, et le meilleur voisin est accepté même s'il n'apporte pas d'amélioration à f .Son algorithme est donc le suivant :

La recherche locale améliorée

```
Début
    s ← une solution aléatoire ;
   Répéter
      s← son meilleur voisin  ;
  Jusqu'à  Max-Iter
Fin
```

6.2.4. L'algorithme GSAT : [BAP96]

La procédure GSAT permet d'obtenir une assignation, pour laquelle on a le maximum de clauses satisfaites. Elle consiste en de multiples exécutions de la recherche locale modifiée. C'est une heuristique puissante, qui donne de bons résultats pour le problème MAX-W-SAT.

6.2.4.1. Algorithme GSAT simple : [JEW93]

Il démarre d'une solution générée aléatoirement, et change de façon répétée l'assignement d'une variable qui conduit à la plus grande augmentation dans le nombre des clauses satisfaites. L'algorithme se présente comme ci-dessous :

GSAT

```
Début
   i←1 ;
    Tant que  ((i ≤ nb-ass) et (B= Faux))  Faire
                s ← une assignation générée aléatoirement ;
                Si    s  satisfait  D alors  B← Vrai ;
               Sinon
                   j ← 1 ;
                   Tant que  ((j ≤ nb-var) et (B=Faux))    Faire
                           x ← variable quelconque vérifiant    Δf_x = MAX_{1≤k≤n} Δf_k
                           x←1-x ;
                          Si  s satisfait D  alors   B ←Vrai ;
                                                     j ← j+1 ;
                    Fait
                Fsi
                i←i+1 ;
      Fait
Fin
```

Tel que :

- D: la donnée SAT
- nb-var : le nombre de changements effectués sur une assignation générée aléatoirement (nombre d'itérations pour chaque essai)
- nb-ass : le nombre d'assignation générées aléatoirement puis explorées nb-var fois pour chacune d'elles (nombre d'essais).
- $\Delta f_k = f(\mu_k s^{(t)}) - f(s^{(t)})$ ou $s^{(t)}$ est l'assignation courante et $f(s^{(t)})$ est la valeur de la fonction objectif lui correspondant.
- $\mu_k s^{(t)} = (x_1, \ldots, x_{k-1}, 1-x_k, x_{k+1}, \ldots, x_n)$ est l'assignation obtenue en inversant la k^{ieme} variable de $S^{(t)}$.

6.2.4.2. Algorithme GSAT avec bruit aléatoire (GSAT with walk) : [BAP96]

Une stratégie « bruit » a été ajoutée à l'algorithme GSAT afin de perturber la règle de choix du voisin. Cet algorithme se présente comme suit :

GSAT-with-walk

```
Début
   Pour  i←1 à Max-Tries  Faire

         s← assignation aléatoire ;

         Pour j← 1 à max-flips

         Faire
               Si    RANDOM < p   alors

                  x  ←  variable quelconque appartenant à une clause non satisfaite ;

               Sinon
                     x ← la variable dont le Δ f est le plus grand ;

               Fsi
                     x ←1-x ;
          Fait
   Fait
Fin
```

Tel que :

- RANDOM : génère un nombre aléatoire dans l'intervalle [0 , 1].
- Max-Tries : le nombre d'essais.
- Max-Flips : le nombre d'itérations pour chaque essai.

6.2.5. Le recuit simulé (Simulated Annealing) : [HAJ90]

La méthode du recuit simulé [KIR 83, CER 85], s'inspire du processus de recuit physique. Ce processus utilisé en métallurgie pour améliorer la qualité d'un solide, cherche un état d'énergie minimale, qui correspond à une structure stable du solide. En partant d'une haute température à laquelle le solide est devenu liquide, la phase de refroidissement conduit la matière liquide à retrouver sa forme solide par une diminution progressive de la température. Chaque température, est maintenue jusqu'à ce que la matière trouve un équilibre thermodynamique. Quand la température tend vers zéro, seules les transitions d'un état à un état d'énergie plus faible sont possibles. Le principe est donc le suivant :

Soit un système d'atomes en mouvement avec une énergie E, et une température t, un atome est choisi aléatoirement, sur lequel on applique un déplacement aléatoire, dE est la variation en énergie obtenue du système. Si dE est négatif, le déplacement est accepté, et E se réduit à E+dE. Par contre, si dE est positif, on calcule la probabilité donnée par la règle de Boltzman :

$$P(dE) = exp\ [dE/kt]$$

- *K* la constante de Boltzman.
- *t* la température du système évaluée à l'échelle Kelvin.

En pratique, l'algorithme s'arrête et retourne la meilleure configuration trouvée, lorsqu'aucune configuration voisine n'a été acceptée pendant un certain nombre d'itérations à une température, ou lorsque la température atteint la valeur zéro.

Le recuit simulé

Début

t ← grande valeur;

s ← solution initiale ;

change ← vrai ;

Tant que t > 0 et change = vrai **faire**

Répéter

Change ← faux ;

Générer une autre solution s^* à partir de s en inversant une variable quelconque

Si $f(s^*) > f(s)$ **Alors** s ← s^* ; change ← vrai ;

Sinon

Générer un nombre aléatoire r ∈ [0 , 1]

P ← exp [(f(s^*)-f(s)) / t] ;

Si (r < p) **alors** s ← s^* ; change ← vrai ;**Fsi**

Fsi

Jusqu'à nb-iters

Diminuer la température t;

Fait

Fin

6.2.6. La procédure GRASP: (Greedy randomised Adaptative search Procedure) [PR97]

C'est un processus itératif, où chaque itération consiste en deux phases: une phase de construction, et une phase de recherche locale. La meilleure solution pendant toute l'exécution est gardée comme résultat.

Dans la phase de construction, une solution est construite itérativement, un élément à la fois. A chaque itération de la construction, le choix de l'élément suivant à ajouter est déterminé en ordonnant tous les éléments dans une liste de candidats (RCL), en respectant une certaine fonction Greedy. Cette fonction mesure l'avantage de sélectionner chaque élément, mais sa valeur est mise à jour à chaque itération. L'élément sélectionné de la liste n'est pas nécessairement le premier de la liste, mais un des meilleurs. Ainsi, on obtient une solution différente à chaque itération de GRASP.

La solution donnée par la première phase n'est pas garantie comme optimum local, pour cette raison, on fait appel à une recherche locale à partir de la solution obtenue afin de l'améliorer.

GRASP

Procedure GRASP()
1. initialisation : $\alpha \leftarrow 0$;
2. **Tant que** (critère d'arrêt non satisfait) **faire**
   Construire une solution aléatoire greedy x ;
   Trouver l'optimum local $x^{\sim}$ dans le voisinage N(x) de x ;
   (En appliquant la recherche locale)
   Si $f(x^{\sim}) > \alpha$ **alors** $x^{*} \leftarrow x^{\sim}$; $\alpha \leftarrow f(x^{*})$;
   Fait
3. **Return** la meilleure solution trouvée x^{*} ;

Procedure Construction ()
1. Initialisation de la solution $S \leftarrow \varnothing$;
2. **Tant que** (construction non terminée) **faire**
   En utilisant une fonction greedy établir la liste RCL ;
   Choisir un élément s' de RCL aléatoirement ;
   Mettre s' dans la solution i.e $S \leftarrow S \cup \{s\}$;
   Changer la fonction greedy pour prendre en compte S après la m.à.j ;
   Fait
3. **Return** la solution x qui correspond à S ;

6.2.7. Les algorithmes évolutifs :

Ces algorithmes sont basés sur le principe du processus d'évolution naturell e[DEJ 93, BAC 93a, SCH 97b]. Les algorithmes évolutifs doivent leur nom à l'analogie avec les mécanismes d'évolution des espèces vivantes. Un algorithme évolutif typique est composé de trois éléments essentiels :

1) une *population* constituée de plusieurs individus représentant des solutions potentielles (configurations) du problème donné.

2) un *mécanisme d'évaluation* de l'adaptation de chaque individu de la population à l'égard de son environnement extérieur.

3) un *mécanisme d'évolution* composé d'opérateurs, permettant d'éliminer certains individus et de produire de nouveaux individus à partir des individus sélectionnés.

Du point de vue opérationnel, un algorithme évolutif typique débute avec une population initiale, souvent générée aléatoirement, et répète ensuite un cycle d'évolution composé de 3 étapes séquentielles : 1) mesurer l'adaptation (la qualité) de chaque individu de la population par le mécanisme d'évaluation, 2) sélectionner une partie des individus, et 3) produire de nouveaux individus par des recombinaisons d'individus sélectionnés. Ce processus se termine quand la condition d'arrêt est vérifiée, par exemple, quand un nombre maximum de cycles (générations), ou un nombre maximum d'évaluations est atteint. Selon l'analogie de l'évolution naturelle, la qualité des individus de la population devrait tendre à s'améliorer au fur et à mesure du processus.

D'une manière générale, on peut distinguer trois grandes familles d'algorithmes évolutifs : les algorithmes génétiques, la programmation évolutive, et les stratégies d'évolution. Ces méthodes se différencient par leur manière de représenter les données et par leur façon de faire évoluer la population d'une génération à l'autre.

Dans le paragraphe suivant nous nous intéresserons de plus prés aux algorithmes génétiques et à leur application au problème MAX-SAT.

- ***Les algorithmes génétiques :***

Les algorithmes génétiques sont inspirés de la génétique, et de la théorie de sélection naturelle citée par *Charles Darwin* au 19ème siècle, et développée par *John Holland,* ses collègues, et ses étudiants de l'université du *Michigan.*

Les algorithmes génétiques classiques introduits par *Holland,* s'appuient fortement sur un *codage universel* sous forme de chaînes 0/1 de longueur fixe. Un individu sous ce codage, appelé *chromosome*, représente une configuration du problème.

Des opérateurs « génétiques » sont définis de manière à opérer aléatoirement sur un ou deux individus.

La résolution du problème MAX-W-SAT par les AGs, nécessite la résolution des problèmes suivants :

- le codage des solutions du problème.
- La définition d'une méthode pour générer la population initiale.
- La définition d'une fonction d'évaluation mesurant la qualité des individus.
- La définition des opérateurs génétiques.

- ***Le codage des solutions :***

Le codage doit être valide, dans le sens où, les solutions obtenues doivent correspondre aux contraintes du problème. Pour le problème MAX-W-SAT, l'individu représente une assignation du problème, donc il est représenté sous forme d'une chaîne binaire.

- ***La population initiale :***

La population initiale est un ensemble d'individus de taille *N*, chaque individu est une solution potentielle. Elle est générée aléatoirement, ou via une heuristique, pour garantir de démarrer avec des solutions assez bonnes, par exemple JHON1, JHON2, GSAT.

- ***La fonction d'évaluation :***

Chaque individu doit être évalué. Soit une donnée SAT à *m* clauses et *n* variables booléennes. On définit pour chaque clause C_i ($1 \leq i \leq m$), une variable entière y_i pouvant prendre la valeur « 0 » ou « 1 ».

Soit une instanciation (individu) $x = (x_1, x_2, \ldots\ldots, x_n)$.

$$y_i(x) = \begin{cases} 1 & \text{si x satisfait la clause Ci ;} \\ 0 & \text{sinon.} \end{cases}$$

La valeur de la fonction d'évaluation de l'individu x est donné par :

$$f(x)=\sum_{i=1}^{m} w(c_i) * y_i(x).$$

Pour MAX-W-SAT $f(x) = w(C_i) * y_i(x)$. ($w(C_i)$ étant le poids associé à la clause C_i.)

- ***Les opérateurs génétiques :***

  - ***La sélection :*** La *sélection* a pour objectif, de choisir les individus qui vont pouvoir survivre, et/ou se reproduire pour transmettre leurs caractéristiques à la génération suivante. La sélection se base, généralement, sur le principe de conservation des individus les mieux adaptés, et d'élimination des moins adaptés.

  Soit *f* (ind i) la valeur de la fonction d'évaluation de l'individu *i*. La probabilité P_i de sélection d'un individu i est :

$$P_i = f\ (ind\ i) / \sum_{i=1}^{taille\text{-}pop} f(ind\ i)$$

  Plus la valeur de la fonction d'évaluation d'un individu est grande, plus sa probabilité de sélection augmente.

- ***Le croisement :*** Le *croisement* permet de produire deux nouveaux individus (enfants) ,à partir de deux individus existants (parents). Par exemple, le croisement bipoints consiste à choisir aléatoirement deux points de croisement, et à échanger les segments des deux parents déterminés par ces deux points. Le croisement réalise donc uniquement, des recombinaisons de valeurs (gènes) existantes entre deux parents et ne permet pas d'introduire de nouvelles valeurs dans les individus enfants. Pour cela, on applique la mutation.

- ***La mutation :*** L'opérateur de *mutation* consiste à changer aléatoirement la valeur de certaines variables dans un individu. Pour le problème MAX-W-SAT, cela revient tout simplement à changer un « 1 » par « 0 » ou vice versa. Dans les algorithmes génétiques, la mutation est considérée comme un opérateur secondaire par rapport au croisement. Un cycle d'évolution complet d'un algorithme génétique, est formé par l'application des opérateurs de sélection, croisement et mutation, sur une population d'individus.

Après l'application des opérateurs génétiques (croisement, mutation), une nouvelle population, appelée population d'enfants est engendrée. Les meilleurs individus de celle-ci, vont remplacer les plus mauvais de la population des parents.

L'algorithme génétique pour le problème MAX-W-SAT est le suivant :

Algorithme génétique

Début
Générer la population initiale ;
Tant que (le nombre de génération total n'est pas atteint)
et (solution optimale non trouvée) **Faire**
Pour i ←1 à Taille-Pop **Faire**
Sélectionner deux individus ;
Choisir un nombre aléatoire entier R_c dans l'intervalle [0,100] ;
Si Rc < taux de croisement alors appliquer le croisement **Fsi**
Choisir un nombre aléatoire entier r dans l'intervalle [0,100]
Tant que r < taux de mutation **Faire**
Choisir un gène aléatoire sur l'individu obtenu
après croisement et l'inverser ;
Choisir un nombre aléatoire r dans [0.100] ;
Fait
Evaluer l'individu produit ;
Fait
Remplacer les individus produits en favorisant les meilleurs ;
Fait
Fin

6.2.8. La recherche Tabou : (Tabu search) [RIA]

La méthode Tabou a été développée par *Glover* [GLO 86], et indépendamment par *Hansen* [HAN 86]. Cette méthode fait appel à un ensemble de règles et de mécanismes généraux, pour guider la recherche de manière intelligente, au travers de l'espace des solutions.

A l'inverse du recuit simulé, qui génère de manière aléatoire une seule solution voisine à chaque itération, Tabou examine un échantillonnage de solutions voisines, et retient la meilleure même si elle est plus mauvaise que la solution actuelle. La recherche Tabou ne s'arrête donc pas au premier optimum trouvé. Cependant cette stratégie peut entraîner des cycles. Pour empêcher ce type de cycle, on mémorise les *k* dernières configurations visitées

dans une mémoire, et on interdit (d'où le nom de la méthode) tout mouvement qui conduit à une de ces configurations. Cette mémoire est appelée la *liste tabou*, une des composantes essentielles de cette méthode. Elle permet d'éviter tous les cycles de longueur inférieure ou égale à *k*. La valeur de *k* dépend du problème à résoudre, et peut éventuellement, évoluer au cours de la recherche. A chaque fois que l'on veut passer d'un état à un autre, on vérifie si ce dernier n'est pas dans la liste tabou.

Il existe d'autres techniques intéressantes pour améliorer la puissance de la méthode Tabou, en particulier, *l'intensification* (qui utilise une mémoire à moyen terme) et *la diversification* (qui utilise une mémoire à long terme). Elles se différencient selon la façon d'exploiter les informations de cette mémoire.

Voici à présent, l'algorithme général de cette métaheuristique :

La recherche tabou

Début

Déterminer s_0 ;

Tant que

Faire

Sélectionner la meilleure solution s_m ;

$s \leftarrow s_m$

Mise à jour de la liste tabou ;

Fait

Fin

(s_m est la meilleure solution des s' dans le voisinage de s
telle que s' n'est pas dans la liste tabou)

6.2.9. La recherche locale guidée GLS (Guided Local Search) [DAD00]

La recherche locale est une métaheuristique basée pénalité, dont le principe est de guider les algorithmes de la recherche locale en dehors de l'optimum local [VDR97], et ce, en pénalisant les mauvaises solutions rencontrées.

La recherche locale guidée, repose essentiellement sur l'algorithme de la recherche locale. Lorsque ce dernier se stabilise sur l'optimum local, GLS modifie sa fonction objectif, afin de le rendre le plus coûteux de son environnement.

Ensuite, la recherche locale est appelée, en utilisant la fonction objectif modifiée, qui est désignée pour échapper à l'optimum local.

Pour le problème MAX-W-SAT, GLS définit pour chaque clause :

- Un coût :

$\boldsymbol{C}_{\text{clause}} = \boldsymbol{W}_{\text{clause}}$ W_{clause} étant le poids de la clause.

- Une fonction indicatrice :

$$\boldsymbol{I}_{\text{clause}} = \begin{cases} 1 & \text{si la clause est non satisfaite} \\ 0 & \text{sinon.} \end{cases}$$

- Une pénalité :

P_{clause} qui enregistre le nombre d'occurrences de la non satisfiabilité de la clause dans s optimums locaux trouvés.

- Une utilité :

$$\boldsymbol{Util}\,(s,\text{clause}) = \boldsymbol{I}_{\text{clause}} * (\boldsymbol{W}_{\text{clause}} / (1+\boldsymbol{p}_{\text{clause}}))$$

La fonction coût augmentée qu'utilise GLS est calculée par:

$$\boldsymbol{h}(s) = \boldsymbol{g}(s) + \lambda \sum_{i=1..n} \boldsymbol{I}_i(s) * \boldsymbol{P}_i$$

Tel que:

- $\boldsymbol{g}(s)$: est la fonction objectif de minimisation (la fonction objectif de GLS minimise le poids des clauses non satisfaites).

- λ un paramètre de diversification /intensification.

L'algorithme de la recherche locale guidée est le suivant :

La recherche locale guidée

Début

iter ← 0 ;

s← solution initiale (générée aléatoirement ou via une heuristique) ;

g ←la somme des poids des clauses non satisfaites par la solution s

Faire

h ← g augmentée par le terme de pénalité

s ← Recherche Locale (s, h) ;

Calculer l'utilité des clauses pour la solution s ;

Util (s, c_i) ; i=1,….n ;

Pour chaque clause non satisfaite c_i avec une utilité maximale

faire pi← pi+1 ;

Iter← iter + 1 ;

Tant que (h(s) > 0) et iter ←iter-max

Fin

6.2.10. La recherche dispersée (Scatter search) [DRK01]

La recherche dispersée est une métaheuristique basée population. Elle consiste à chaque itération, à générer une population diversifiée, à partir d'une solution initiale appelée *semence*. De cette population, un ensemble *RefSet* des meilleures solutions en terme de qualité et de diversité est choisi.

L'ensemble *RefSet* est ensuite partitionné en sous ensembles, auxquels des combinaisons linéaires seront appliquées pour créer de nouvelles solutions ; contrairement aux AGs, les combinaisons dans la recherche dispersée peuvent se faire entre 2,3 ou plusieurs éléments.

Les nouvelles solutions générées, sont améliorées par des processus heuristiques ; une solution améliorée peut remplacer la mauvaise solution dans *RefSet* si elle est meilleure en qualité sinon en diversité.

La génération de nouvelles solutions s'arrête lorsque l'ensemble ne change pas, dans ce cas, une nouvelle itération est commencée en utilisant la meilleure solution de *RefSet* comme semence.

La recherche dispersée

Début

Semence ← solution aléatoire ;

Pour iter ← 1 à Max-Iter

Générer un ensemble initial des solutions P ;

Améliorer les solutions de cet ensemble ;

Désigner un ensemble de références ***RefSet.***

NouvElements = Vrai ;

Tant que NouvElements

NouvElements = faux

Générer des sous ensembles de RefSet ;

Supprimer tous les sous ensembles qui ne contiennent pas au moins un nouvel élément

Appliquer une méthode de combinaison sur l'ensemble ***RefSet*** pour produire de nouvelles solutions.

Pour chaque élément produit s **Faire**

$s^* \leftarrow$ améliorer(s)

Si s^* n'est pas dans RefSet et $\exists s \in RefSet_1$ tel que $f(x^*)>f(x)$ **Alors**

$s \leftarrow s^*$ /* meilleure solution en qualité*/

NouvElements=Vrai ;

Sinon

Si s^* n'est pas dans $RefSet_2$ et $\exists s \in RefSet_2$ tel que

$d_{min}(s^*) > d_{min}(s)$ **Alors**s ← s* ; /*meilleure en diversité*/

NouvElements = Vrai ;

fsi

fsi

Fin Pour

Si (iter <MaxIter) **alors** Semence ← le meilleur élément de Refset1 **Fsi**

Fait

Fin

7. Conclusion:

Vu la complexité du problème MAX-W-SAT, les métaheuristiques constituent les principales méthodes de résolutions qui lui sont appliquées. Beaucoup d'entre elles, présentées dans ce chapitre, se sont avérées très efficaces. Cependant, les recherches dans ce domaine ne cessent d'avancer dans le but d'améliorer les résultats obtenus.

Récemment, de nouvelles métaheuristiques (telles que l'Optimisation par colonie de fourmis ou ACO) sont apparues dans le domaine de l'intelligence artificielle. Elles émergent d'un nouvel axe de recherche en intelligence artificielle, appelé *Intelligence en essaim*. L'application des métaheuristiques développées sur des problèmes d'optimisation combinatoire a donné de très bons résultats

Chapitre II

La métaheuristique Bees Swarm Optimization et son adaptation à MAX-W-SAT

1. Introduction

L'intelligence en essaim est un axe de recherche apparu récemment en intelligence artificielle. Il s'inspire du comportement des insectes sociaux, qui malgré la simplicité de leurs comportements individuels, réalisent grâce à leur coopération, des tâches très complexes, qu'un individu doté d'une intelligence sophistiquée serait incapable d'accomplir seul.

Cette manière de concevoir les solutions aux problèmes, comme étant le résultat de la coopération de plusieurs individus, a donné naissance à des métaheuristiques puissantes et très efficaces, comme l'optimisation par colonie de fourmis.

2. Domaines d'application de l'intelligence en essaim :

2.1. Programmation des équipements d'une usine :[BDT99]

Dans une colonie d'abeilles, les individus ont une spécialité qui dépend de leur âge. Par exemple, les abeilles les plus vieilles sont plutôt des butineuses, tandis que les jeunes sont des nourrices. L'affectation des tâches n'est pas rigide. Elle s'adapte perpétuellement aux besoins de la ruche. Par exemple, lorsque la nourriture manque, les nourrices butinent également.

Cette répartition souple et flexible de tâches a inspiré *Michael Campos* et ses collègues de l'Université *Northwestern* une technique de programmation des cabines de peinture, dans une entreprise de construction automobile. Dans l'usine, les cabines peignent les véhicules assemblés, et chaque cabine, à l'instar d'une abeille réelle, est spécialisée dans une couleur. Le changement de couleur d'une cabine est long et coûteux.

En outre, ils ont supposé qu'un individu abandonne une fonction, s'il perçoit un besoin important pour une autre. Ainsi, une cabine à peinture rouge utilisera cette couleur jusqu'à ce qu'il soit urgent de peindre un véhicule en blanc, alors que les cabines spécialisées en blanc sont saturées.

En dépit de la simplicité des règles, le système des abeilles permet aux cabines de peinture d'être programmées plus efficacement, notamment, avec moins de changement de couleurs qu'avec un ordinateur centralisé. D'autre part, la méthode s'adapte aux souhaits des consommateurs : quand la demande de véhicules blancs augmente de façon inattendue, des cabines renoncent rapidement à leur couleur de spécialisation, et reçoivent les véhicules non

affectés. En plus, le système traite facilement les contretemps (tels que les pannes de cabines), que d'autres stations compensent rapidement, en partageant le travail supplémentaire.

2.2. La robotique : [BDT99]

D'innombrables comportements inhérents aux insectes sociaux sont copiés par les roboticiens. Par exemple, ils étudient le transport coopératif chez les fourmis pour concevoir des techniques de commande d'un groupe de robots.

Chez certaines espèces, une fourmi qui ne peut récupérer seule une proie, recrute parfois des congénères pour l'aider : Pendant quelques minutes, les fourmis changent de positions et d'alignements autour de l'objet, jusqu'à ce qu'elles soient capables de transporter la proie vers le nid.

Ce comportement a été reproduit avec des robots mécaniques. Ces derniers devaient pousser une boite vers un emplacement donné. La boite ne pouvait être poussée par un seul robot et, de surcroît, ceux-ci étaient dotés d'instructions simples telles que : Trouver la boite, établir un contact avec elle, se positionner de manière à ce que la boite se trouve entre le robot et le but, puis pousser la boite en direction du but.

Malgré la simplicité des programmes, la similitude entre le comportement des robots et celui d'une colonie de fourmis est frappant : Les robots se déplacent d'abord au hasard, à la recherche de la boite. Puis quand ils la localisent, et s'ils sont en nombre suffisant, ils la poussent. Ils se repositionnent en permanence quand la boite reste immobile, quand ils perdent le contact avec elle, quand ils se bloquent mutuellement ou lorsque la boite tourne .Enfin, en dépit de leur capacités limitées, les robots amènent la boite au but.

Confrontés au même problème, des êtres humains imaginent des collaborations plus efficaces. Toutefois, l'application de l'intelligence en essaim est propice à la miniaturisation et à la réduction des coûts. Les ingénieurs concevraient ainsi, des robots simples et peu onéreux qui coopéreraient pour des tâches de plus en plus complexes et difficiles à réaliser même par un robot doté d'une intelligence artificielle sophistiquée.

2.3. L'analyse des données : [BDT99]

Des méthodes d'analyse des données financières, reproduisent les stratégies de regroupement des cadavres, et de tri des larves, mis en œuvre par les fourmis.

Chez les fourmis *Messor sancta*, les ouvrières nettoient les nids en entassant à l'extérieur les cadavres. De la même manière, les ouvrières de l'espèce, *Leptothorax,* trient

systématiquement les larves et les œufs. Les petites larves sont regroupées avec les œufs, les pupes et prépupes (des stades intermédiaires dans le développement des insectes) sont autour, elles même entourées par les larves les plus grandes.

Selon *Deneubourg*, de petits amas de cadavres s'agrandissent, car ils attirent les ouvrières qui y déposent plus d'éléments : cela entraîne la formation de tas de plus en plus grands. Pour les couvées, les ouvrières ramassent et déposent les éléments, en fonction d'objets similaires environnants. Par exemple, quand une fourmi trouve une grande larve entourée d'œufs, elle prend de préférence la larve qui constitue 'l'intrus' et la dépose dans une région qui contient déjà de grandes larves.

En se fondant sur ce tri du couvain, une méthode d'exploration d'une grande banque de données a été mise au point. Supposons, par exemple, qu'une banque veuille identifier ses clients les plus susceptibles de rembourser un prêt. La banque dispose de nombreux renseignements sur ses clients, tels que l'âge, le sexe, la situation familiale, le type de logement, les services bancaires favoris…etc.

En étudiant des groupes de personne de caractéristiques similaires, les responsables des prêts détermineraient la fiabilité des divers clients, et ils pourraient exiger des garanties solides, de la part des demandeurs de prêt appartenant aux groupes dominés par des débiteurs défaillants.

Les spécialistes d'analyse de données, aiment visualiser les groupes sur un plan (au-delà, l'interprétation des données est difficile),où chaque client est représenté par un point. Ces clients sont donc, dans la situation des œufs et des larves, que des fourmis logicielles peuvent déplacer, ramasser, et déposer selon les éléments environnants. La distance entre deux clients, indique leur degré de similitude. Les fourmis artificielles, prennent leur décision de tri en fonction de toutes les caractéristiques des clients ,et, selon les objectifs de la banque, le logiciel peut avantager mathématiquement certains attributs .

Les banques et les compagnies d'assurance, utilisent déjà ce type d'analyse de profils. Grâce au modèle des fourmis , la visualisation des données est faciles et, surtout, le nombre de groupes ressort automatiquement des données, alors que les méthodes classiques requièrent un nombre prédéfini de groupes, où les données sont réparties. Ainsi, le tri des fourmis artificielles met au jour des communautés, qui autrement demeurent dissimulées.

2.4. Les réseaux de télécommunication : [BDT99]

Les techniques de l'intelligence en essaim démontrent également toute leur puissance, dans le cas de problèmes dont l'énoncé, les données ou les paramètres varient en permanence sur des échelles de temps très courtes. C'est le cas du routage dans les réseaux de communications.

Schématiquement, le problème est le suivant : lorsqu'une communication est établie entre deux ordinateurs, le message initial est découpé en paquets de données qui circulent le long d'un réseau constitué de lignes de transmission dont les capacités de débit peuvent être très diverses et variables au cours du temps, et de routeurs qui constituent les nœuds de ce réseau. La fonction des routeurs est de diriger les paquets de données vers l'un des autres routeurs du réseau et ce, jusqu'à ce que les paquets de données arrivent à leur destination finale. Mais le routeur doit aussi tenir compte de l'importance du trafic sur les voies de communication auxquelles il est relié de manière à éviter l'engorgement de ces voies.

Pour résoudre ce problème, et en s'inspirant du comportement de forage des fourmis, on fait circuler en parallèle avec les paquets de données, des agents de routage, une sorte de fourmis virtuelles, qui analysent en temps réel l'état d'encombrement des différentes voies du réseau, et qui indiquent ensuite cet état à chacun des routeurs. Les fourmis calculent le temps qu'elles mettent pour aller d'un nœud du réseau à un autre et marquent ensuite à l'aide de phéromone virtuelle la voie qu'elles viennent d'emprunter. Plus le délai est court, plus l'intensité du marquage est importante. Ainsi, lorsqu'un paquet de données arrive au niveau d'un routeur donné, il aura d'autant plus de chance d'emprunter une voie que la densité de phéromone virtuelle sur cette voie sera importante.

De cette manière, le réseau s'adapte en permanence et de manière totalement décentralisée à l'activité du trafic.

Aussi bien pour la maximisation des débits que pour la minimisation des délais, les premiers résultats des simulations indiquent que ce système de gestion de trafic (AntNet) est plus efficace que ceux qui sont actuellement utilisés.

2.5. L'optimisation combinatoire :

A Bruxelles, *Jean Louis Deneubourg* et ses collègues, des pionniers de l'intelligence en essaim, ont montré que les processions de fourmis résultent de la sécrétion de la phéromone, une molécule qui attire d'autres fourmis. La création d'une telle piste marquée,

est une bonne stratégie pour trouver le chemin le plus court entre un nid et une source de nourriture.

Dans une expérience, des fourmis d'Argentine étaient séparées d'une source de nourriture par deux voies, l'une d'une longueur double de l'autre. En quelques minutes, elles choisissaient le chemin le plus court . Comment cela se fait-il?

Les fourmis suivent des pistes marquées par une phéromone et elles en déposent derrière elles. Les premières fourmis qui retournent au nid à partir de la source de nourriture ont emprunté le chemin le plus court dans les deux sens : ce chemin, marqué deux fois par la phéromone, attire plus les autres fourmis que le long chemin, marqué une seule fois.

Toutefois, quand le chemin le plus court n'est trouvé qu'après le chemin déjà marqué à la phéromone, les fourmis continuent de parcourir le chemin plus long (trouvé initialement). Dans les systèmes artificiels, les informaticiens évitent cet écueil en employant des phéromones volatiles : Ainsi, les pistes de phéromone subsistent difficilement sur les chemins les plus longs. Les fourmis virtuelles empruntent alors, les voies les plus courtes, même quand elles ont été découvertes plus tardivement. Grâce à cette propriété, le système ne se stabilise pas sur des solutions médiocres. Notons que, chez certaines espèce de fourmis, les concentrations en phéromone décroissent effectivement, mais très lentement.

L'étude et la modélisation de ce comportement ont donné naissance à une nouvelle métaheuristique dite «Ant Colony Optimization » en anglais, ou ACO en abrégé. Cette dernière peut être résumée comme suit :

Une colonie de fourmis cherche d'une manière collective et asynchrone, des solutions qui présentent de bonnes caractéristiques, et ce, en effectuant des transitions inter nœuds dans le graphe de recherche. Ces transitions, sont régies par des règles de décisions probabilistes. Les fourmis construisent leurs solutions d'une manière progressive. Une fois que la solution est construite, ou durant sa construction, les fourmis procèdent à l'évaluation de la solution (partielle), et déposent des informations (une sorte de phéromone virtuelle) en fonction de sa qualité sur les connections utilisées. Ces informations vont guider des recherches ultérieures.

Cette métaheuristique a fait ses preuves dans la résolution de nombreux problèmes d'optimisation combinatoire : Le problème du voyageur de commerce, le problème d'affectation quadratique, le problème de scheduling, le problème MAX-SAT...etc.

❖ **ACO pour la résolution du problème MAX-W-SAT :**

Une adaptation de la métaheuristique ACO au problème MAX-W-SAT a été réalisée par [DTZ03].

Avant de présenter l'algorithme correspondant, on donne les indications suivantes :

- La fonction objectif *f* est une fonction de minimisation : minimiser la somme des poids des clauses non satisfaites simultanément.
- La phéromone est représentée par une table *Phero* à deux dimensions :
  - Les lignes désignent les variables.
  - Les colonnes désignent les valeurs logiques qu'elle peuvent prendre : '0' ou '1'.
- La probabilité pour que x_i prenne la valeur '0' est calculée comme suit :

$P(x_i = 0) = Phero[x_i , 0] / (Phero[x_i , 0] + Phero[x_i , 1])$.

La probabilité pour que x_i prenne la valeur '1' : $P(x_i = 1) = 1- P(x_i = 0)$.

ACO pour Max-W-Sat

Début

Engendrer une population de *m* solution aléatoires s^k $(1 \le k \le m)$;

Initialiser la table *Phero* ;

Répéter

Pour chaque fourmi *k* Faire

$s'_k \leftarrow s_k$ transformée en fonction de la phéromone ;

$s'_k \leftarrow$ Recherche Locale(s'_k) ;

Si $f(s'_k) < f(Best)$ alors $Best \leftarrow s'_k$; /* best : la meilleure solution trouvée*/

Fait

Mettre à jour la phéromone ;

Jusqu'à MaxIter

Fin

L'initialisation de *Phero* se fait comme suit :

Chaque solution initiale subit une amélioration à l'aide d'une recherche locale. Soit *BestInit* la meilleure solution trouvée.

$Phero[x_i , j] = 1/ (100 * f(BestInit))$ $i=1..n , j \in \{0,1\}$.

La transformation de la solution, s'effectue selon l'algorithme suivant:

Début

Pour j ← 1 à MaxChanges **Faire**

Choisir une variable à positionner aléatoirement, soit x_i ;

Calculer P (x_i=0) ;

R ← générer un réel aléatoire ∈[0,1];

Si R ≤ P(x_i =0) **alors** $s_k(x_i)$ ← 0

sinon $s_k(x_i)$ ← 1 ;

Fait

Fin

La mise à jour de la phéromone est sollicitée pour simuler le processus d'acquisition de nouvelles connaissances et l'oubli du passé. Elle se présente comme suit :

Début

Pour i← 1 à n **Faire** /* simuler l'évanescence*/

Phéro $[xi,0] \leftarrow (1-\alpha) * Phero[x_i,0]$;

Phero $[xi,1] \leftarrow (1-\alpha) * Phero[x_i,1]$;

Fait

Pour chaque solution s_k ($1 \le k \le m$) **Faire** /*ajouter des informations*/

Pour i ← 1 à *n* **Faire**

$Phéro\ [x_i,s^k(x_i)] = Phéro\ [x_i,s^k(x_i)] +$

$\alpha * (1 / (1+f(s^k))) * ((f(worst)-f(s^k)) / (f(Worst) - f(Best)))$;

Fait

Fait

Fin

Tel que :

- α, est un facteur d'évaporation, $0 \le \alpha \le 1$.
- *Worst,* est la mauvaise solution rencontrée jusqu'à l'itération courante.
- *Best,* est la meilleure solution trouvée.

3. Bees Swarm Optimization (BSO):

Les recherches menées dans le cadre de notre projet de fin d'étude [SAY04] et dont les résultats ont été publiés [DSY05], ont donné naissance à une nouvelle métaheuristique : « Optimisation par Essaim d'Abeilles », « Bees Swarm Optimization » en anglais ou BSO en abrégé. Cette méthode de résolution de problèmes d'optimisation combinatoire est inspirée du comportement des abeilles réelles.

Avant de présenter BSO, il convient d'illustrer succinctement ce comportement fascinant.

3.1. Les abeilles réelles : [BOT94]

En 1946, l'éthologue autrichien *KARL VON FRIS* avait présenté à Zurich, devant la Société suisse des sciences naturelles, l'essentiel de ses conclusions : par l'orientation et la vitesse des mouvements qu'elle effectue sur les rayons de la ruche, la butineuse (l'ouvrière chargée de la récolte du nectar et du pollen) indique à ses congénères la direction, la distance, et la richesse de la source de nourriture qu'elle a découverte.

Un demi-siècle plus tard, on en sait certes, un peu plus. Mais le langage de la butineuse, malgré des décennies d'exploration scientifique et de progrès analytiques, n'a pas encore révélé tous ses secrets. Ce sommet de la communication animale, continue d'attiser la curiosité des chercheurs . Comme, d'ailleurs, la plupart des facettes biologiques de ce fabuleux insecte social.

3.1.1. L'exploitation sélective de la meilleure source de nourriture :

Il n'est pas rare qu'au cours d'une même journée, les butineuses d'une même colonie visitent plus d'une douzaine de zones d'exploitation potentielles, mais on observe, que la colonie concentre ses efforts de récolte sur un petit nombre d'entre elles, les plus riches et les plus faciles d'accès. En outre, de nombreuses observations font apparaître qu'une colonie peut rapidement déplacer son exploitation d'une source de nourriture à une autre.

Les abeilles sont capables d'ajuster très finement plusieurs composantes de leur comportement de récolte. Ainsi, en changeant périodiquement la richesse relative de deux sources de nourriture, vers lesquelles on a préalablement entraîné les récolteuses d'une ruche à venir s'approvisionner, on a pu mettre en découvrir, que lors de leur retour au nid, le comportement de recrutement des abeilles était profondément affecté.

L'expérience de Seely , Cazmazine et Sneyd,1991 :

Lorsque l'on donne le choix à un essaim d'abeilles, entre l'exploitation de deux sources de nourriture dont la concentration en sucre est très inégale (1M et 2.5M respectivement), et situées de manière diamétralement opposées par rapport à la ruche, l'une au Nord, et l'autre au Sud, la colonie va concentrer son effort de récolte sur la plus riche d'entre elles. Le nombre total des récolteuses présentes au niveau de chacune des deux sources, est directement proportionnel aux changements ayant affecté les taux de recrutement et d'abandon vers chacune des deux sources. C'est grâce à une modulation très précise de ces deux taux, que peut se produire, au niveau de la colonie, une exploitation sélective de la source la plus riche . (Voir figure1)

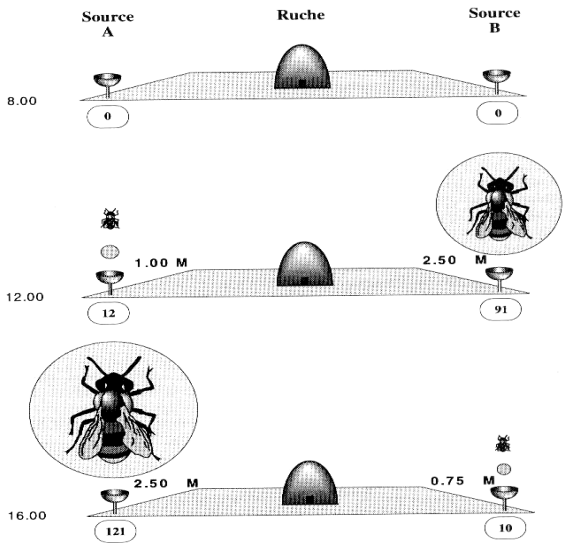


Figure1: L'expérience de Seely, Camazine et Sneyd, 1991

3.1.2. Mécanisme de communication :

Chaque récolteuse, après s'être déchargée de sa récolte de nectar, entame une danse qui indique à ses congénères qui la suivent: La direction, la distance et la richesse de la source de nourriture qu'elle vient de visiter.

Elle va de cette manière, inciter d'autres individus à se rendre dans cette zone pour y récolter à leur tour. Plus la source est riche, plus la vigueur avec laquelle l'abeille effectue cette danse augmente, de même que la cadence de ses visites. Parallèlement, l'intervalle de temps qui sépare son arrivée au nid du déchargement de sa récolte, diminue. Par la suite, lorsque l'on change expérimentalement la qualité de la source de nourriture, en offrant une

source beaucoup moins riche, les récolteuses, de retour au nid, n'effectuent presque plus de danse, et lorsque celles ci se produisent, leur rythme s'affaiblit notablement.

Comment une abeille peut- elle estimer la richesse relative d'une source de nourriture qu'elle vient de visiter, par rapport à d'autres sources simultanément présentes ?

On a montré que cette estimation s'effectuait sans que l'abeille n'ait besoin de visiter plusieurs sources pour les comparer, ni qu'elles reçoivent cette information des abeilles chargées du stockage de la nourriture rapportée au nid. Expérimentalement, il est vérifié que les récolteuses ajustent l'intensité de leur danse, en intégrant de multiples facteurs, comme l'abondance de la source, la distance de la source au nid, et la difficulté à puiser le nectar.

Néanmoins, on ignore toujours comment le traitement de l'information par le système nerveux central de l'abeille permet cette estimation !

3.2 La métaheuristique *"Optimisation par essaim d'abeilles" :*

BSO peut être appliquée aux problèmes d'optimisation combinatoire à espace d'état discret présentés comme suit : Un problème d'optimisation combinatoire est défini par un ensemble d'instances.

A chaque instance du problème, sont associés : un ensemble discret de solutions S, un sous-ensemble X de S représentant les solutions admissibles (réalisables), et une fonction objectif f (ou fonction de coût) qui assigne à chaque solution $s \in X$ le nombre réel (ou entier) $f(s)$

Résoudre un tel problème (plus précisément une telle instance du problème), consiste à trouver une solution $s^* \in X$,optimisant la valeur de la fonction objectif f. Une telle solution s^* s'appelle *olution optimale* ou *optimum global*. Ainsi, et selon la fonction objectif f, on peut distinguer deux cas: [RIA]

- Si f est une fonction de maximisation, s^* doit vérifier : $f(s^*) \geq f(s)$ pour tout élément $s \in X$.
- Si f est une fonction de minimisation, s^* doit vérifier: $f(s^*) \leq f(s)$ pour tout élément $s \in X$.

Etant donné un problème d'optimisation combinatoire discret, La métaheuristique BSO est basée sur un essaim d'abeilles artificielles qui coopèrent pour le résoudre :

- D'abord, une des abeilles appelée *BeeInit,* se charge de trouver un sommet présentant de bonnes caractéristiques, qu'on appellera *Sref.* Ce sommet, désigne la région d'exploitation qu'on appellera *SearchArea, et* dans laquelle, les abeilles intensifieront leurs recherches.

- La région d'exploitation, est constituée d'un ensemble de sommets. Chacun d'eux est obtenu à partir du sommet *Sref,* et ce, en utilisant une certaine stratégie.
- Après la définition de cette région, chaque abeille artificielle qu'on appellera *Bee,* se voit assignée à un sommet, depuis lequel,elle démarre une recherche, qui consiste à transiter d'un sommet à un autre sommet voisin jugé meilleur.
- A l'issue de sa recherche, chaque abeille *k* mémorise le meilleur sommet qu'elle a visité à la $k^{ème}$ entrée qui lui est réservée dans une table appelée *Danse.* A l'instar de la fameuse danse des abeilles réelles, cette table permet aux abeilles virtuelles de communiquer leurs résultats à l'essaim.
- A la fin de l'exploitation (terminaison de la recherche de toutes les abeilles), chaque abeille de l'essaim peut prendre connaissance des résultats de ses congénères à travers la table *Danse*.
- Un des sommets de la table *Danse* deviendra le de référence *Sref de l'itération suivante.* Ainsi, l'effort d'exploitation de l'essaim sera dirigé vers une nouvelle zone prometteuse de l'espace de recherche. L'essaim choisit ce sommet en fonction de sa *qualité*. Cependant, si au bout d'un certain nombre de fois, l'essaim constate que cette qualité ne s'améliore plus, il intègre un second critère, appelé critère de *diversité,* lui permettant de s'échapper de la région actuelle où il est éventuellement piégé. Un sommet est jugé meilleur en diversité, s'il est le plus éloigné des régions déjà exploitées.
- Le choix du sommet *Sref* effectué par l'essaim, n'implique aucune forme de communication entre les abeilles. La table *Danse* est le seul moyen pour les abeilles artificielles de communiquer les résultats de leur recherche, et c'est évident qu'elles se dirigent toutes vers la même région après consultation de la table, c'est le principe même des abeilles réelles, qui consiste à se diriger toujours vers la région la plus riche.
- Le même sommet peut être désigné comme sommet de référence une seconde fois, ce qui conduit à l'apparition d'un cycle. Pour affronter cet écueil, une liste *Tabou* dans laquelle l'essaim mémorise à chaque étape le sommet *Sref* courant a été prévue. Ainsi, si celui-ci figure dans la liste tabou, il sera remplacé en adoptant une certaine stratégie.

L'algorithme général de BSO :

L'algorithme général de la métaheuristique Optimisation par Essaim d'Abeilles se présente comme suit :

Bees Swarm Optimization

Début

Sref ← Sommet retourné par *BeeInit* ;

Tant que non critère d'arrêt **faire**

Insérer *Sref* dans *Tabou ;*

Déterminer la région *SearchArea* référencée par *Sref ;*

Affecter chaque abeille à un sommet de *SearchArea ;*

Pour chaque abeille *k* **faire**

Recherche depuis le sommet S_k qui lui est associé ;

Stocker le résultat dans la $k^{ème}$ entrée de la table *Danse* ;

Fait

Choisir le nouveau sommet de référence *Sref* ;

Fait

Fin

4. BSO pour MAX-W-SAT:

Dans ce paragraphe, nous adaptons la métaheuristique BSO au problème MAX-W-SAT.

4.1. L'espace de recherche :

L'espace de recherche (le monde artificiel dans lequel opèrent les abeilles), est un graphe *G(X,U)* hypercubique d'ordre *n* (*n* étant le nombre de variables de la donnée SAT) tel que :

- $X= \{0,1\}^n$ est un ensemble fini de sommets. Chaque sommet correspond à une assignation aux variables, donc chaque sommet peut être une solution.
- $U= \{u_{x,y} / x \text{ et } y \in X$ et différent d'un seul bit$\}$ est un ensemble fini de connexions (arêtes) entre les sommets de X.

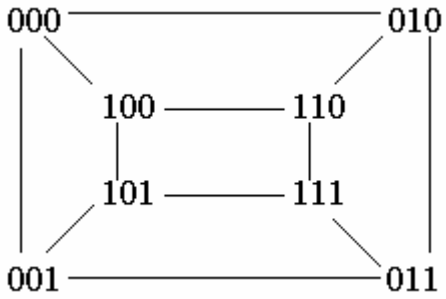
Remarque : [LEI92]

Un graphe G=(X,U) est dit hypercubique d'ordre 'n', ou n_hypercubique, si et seulement si :

- $|X| = 2^n$.
- $|U|=n.2^{n-1}$.
- U(x,y) ∈ U si et seulement si la représentation binaire de x ne diffère de celle de y que d'un seul bit .

Exemple :

Soit une donnée SAT à 3 variables, l'espace de recherche associé est le suivant :



4.2. La qualité d'une solution :

La qualité d'une solution *S,* correspond à la valeur de sa fonction objectif *f(S)*. On rappelle que pour le problème MAX-W-SAT, l'objectif est de maximiser la somme des poids des clauses satisfaites simultanément.

4.3. La diversité d'une solution :

Le degré de diversité qu'offre une solution *S,* est mesuré par le minimum des distances entre *S* et les éléments de la liste *Tabou*. Elle est donnée par :

$$diversité(S) = \text{Min } \{ d(S,T) / T \in Tabou \} .$$

Soient deux solutions *S* et *T*, la distance entre elles, *d(S,T)*, est donnée par la distance de *Hamming* définie comme suit :

$$d(S,T) = \sum_{i=1}^{n} S_i \oplus T_i$$

$$\text{Avec : } a \oplus b = \begin{cases} 0 \text{ si } a=b \\ 1 \text{ sinon} \end{cases}$$

Exemple :

Soient les deux solutions *S* et *T* suivantes:

S = (0,1,1,1,0) *T* = (0,1,0,0,1)

d(S,T) = 0+0+1+1+1 =3 est la distance entre elles.

4.4. L'algorithme de BSO appliqué à MAX-W-SAT :

L'algorithme de BSO appliqué à MAX-W-SAT se présente comme suit :

BSO pour MAX-W-SAT

Début

Sref ← solution construite par *BeeInit* ;

*S** ← *Sref* ;

Tant que non critère d'arrêt **faire**

Générer *SearchArea* à partir de *Sref* ;

Mettre *Sref* dans la liste *Tabou* ;

Pour chaque *Bee* k **faire**

Lui affecter une solution S_k de *SearchArea ;*

$S_{k'}$ ← le résultat de sa recherche à partir de S_k *;*

Stocker $S_{k'}$ dans la table *Danse* ;

Fait

Trier *Danse* dans l'ordre décroissant selon *f* ;

Si *f* (premier (*Danse*)) > *f*(*S**) **alors** *S** ← premier (*Danse*) ;

Sref ← le nouveau sommet de référence ;

Fait

Retourner *S** comme solution au problème ;

Fin

4.4.1. La solution initiale :

Initialement, l'abeille *BeeInit* construit la solution de référence *Sref,* via une heuristique ou via une recherche locale. Les heuristiques que nous avons utilisées sont : John1, John2. Leurs algorithmes ont été présentés au 2ème chapitre.

Remarque *:* [NOD]

Deux versions améliorées : John2_a et John2_b, ont été dérivées à partir l'algorithme John2 présenté au chapitre 2. Et ce, dans le but de guider la sélection de la variable *x,* qui se faisait de manière aveugle dans l'instruction suivante :

« Chercher $x \in$ X, qui apparaît dans au moins une clause de LEFT ; ».

Pour ce faire, deux stratégies ont été adoptées:

- ✓ *La première stratégie* :

Cette stratégie consiste à choisir, à chaque itération de l'algorithme, la variable maximisant la différence : |ST - SF| tel que :

- ST(X) est la somme des poids des clauses de LEFT, où le littéral correspondant à X apparaît positif.
- SF(X) est la somme des poids des clauses de LEFT, où le littéral correspondant à X apparaît négatif.

- ✓ *La deuxième stratégie :*

Cette stratégie, consiste à affecter à chaque variable *X,* un poids *w(X)* calculé par la formule suivante:

$$w(X) := Abs\left(\sum_{X \in Ci} \frac{1}{|ci|} - \sum_{X \in Cj} \frac{1}{|Cj|}\right)$$

On choisira la variable ayant le poids maximum. Ceci revient à choisir, à chaque itération de l'algorithme, la variable correspondant au littéral occurrent dans le plus grand nombre de clauses, et ce, en favorisant toujours les clauses courtes.

4.4.2. La génération de la région d'exploitation :

La région d'exploitation *SearchArea,* est représentée par un ensemble de *N* solutions (*N* étant le nombre d'abeilles constituant l'essaim). Chacune de ces solutions est calculée à partir de *Sref,* en inversant *1/Flip* de ses variables.

Le choix du paramètre *Flip* est délicat, puisqu'il détermine le nombre de variables à inverser à partir de *Sref*. En effet, une valeur trop petite de ce nombre, implique que *Sref* est fort probablement l'optimum local de la nouvelle région d'exploitation, donc la probabilité d'une amélioration est très faible. En revanche, si cette valeur est trop importante, l'essaim

s'éloignera de la région contenant *Sref* au risque de perdre de bonnes solutions. Pour procéder aux inversions, nous proposons deux stratégies assurant que les solutions obtenues soient aussi distinctes que possible. Si le nombre de solutions générées s'avère insuffisant, nous aurons recours à une approche aléatoire.

Première stratégie

Début
h ← 0 ;
Tant que taille de *SearchArea* non atteinte et h < *Flip* **faire**
S ← Sref ;
k ← 0 ;
Répéter
inverser S[*Flip**k+h] ;
k ← k+1 ;
Jusqu'à *Flip**k+h ≥ n
SerachArea ← *SerachArea* ∪ {S} ;
h ← h+1 ;
Fait
Fin

Deuxième Stratégie

Début
h ← 0 ;
Tant que taille *SerachArea* non atteinte et h < *Flip* **faire**
S ← *Sref* ;
k ← 0 ;
Répéter
Inverser S[(n/*Flip*) * h +k] ;
k := k+1 ;
jusqu'à k ≥ n/*Flip*
SerachArea ← *SerachArea* ∪ {S} ;
h := h+1 ;
Fait
Fin

Chacune de ces deux stratégies génère des solutions, dont les ensembles des variables inversées à partir de *Sref* sont deux à deux disjoints.

Exemple :

Supposons que n=20, *Flip* = 5. (Les variables sont indicées de 0 à 19)

La stratégie 1 consiste à inverser les variables aux positions suivantes :

(0,5,10,15) , (1,6,11,16),(2,7,12,17),(3,8,13,18) et (4,9,14,19).

La stratégie 2 consiste à inverser les variables aux positions suivantes :

(0,1,2,3) , (4,5,6,7) , (8,9,10,11) , (12,13,14,15) et (16,17,18,19).

L'algorithme de la stratégie aléatoire se présente comme suit:

Stratégie aléatoire

Début

Tant que taille *SerachArea* non atteinte **faire**

S ← *Sref* ;

Inverser aléatoirement n/*Flip* variables de S ;

SerachArea ← *SerachArea* ∪ {S} ;

Fait

Fin

4.4.3. La recherche d'une abeille :

La recherche d'une abeille se résume en une recherche locale, dite *recherche locale par étapes* : c'est un processus itératif à deux phases :

- Une phase de recherche locale simple.
- Une phase qui consiste à inverser le maximum de variables de la solution trouvée (dans la première phase), de telle sorte que la fonction objectif de la nouvelle solution lui soit supérieure ou égale.

L'algorithme exécuté par chaque abeille est le suivant:

Début

S une solution donnée ;

Répéter

Sortir ← faux ;

Tant que (non *sortir*) **faire**

Soit *N(S)* le voisinage de *S* ;

Chercher *V* tel que $f(V) \geq f(Y) \ \forall \ Y \in N(S)$;

Si $f(V) > f(S)$ **alors** $S=V$;

Sinon *sortir* = vrai ;

Fait

Best = *f*(*S*) ;

Pour i ← 1 à n **faire**

$S_i \leftarrow 1- S_i$; /*Si étant la $i^{ème}$ variable de *S**/

Si f(*S*) < *Best* **alors** $S_i \leftarrow 1- S_i$;

Fait

Jusqu'à *MaxIter*

Fin

4.4.4. Le choix de *Sref* :

Soit *Sref(t)* la solution de référence choisie à l'itération *t*. Le choix de *Sref(t+1)* dépend de la quantité Δf :

$\Delta f = f(Sbest) - f(Sref(t))$; *Sbest* étant la meilleure solution à l'itération *t+1* ;

L'algorithme est comme suit :

Début

Si $\Delta f > 0$ **alors**

Début

Sref ← la meilleure solution en qualité ;

Si *NbChances* < *MaxChances* **alors**

NbChances ← *MaxChances* ;

Fin Si

Fin

Sinon

Début

NbChances ← *NbChances* –1 ;

Si *NbChances* > 0 **alors** *Sref* := la meilleure solution en qualité ;

sinon

Début

Sref := la meilleure solution en diversité ;

NbChances ←*MaxChances ;*

Fin

Fin Si

Fin

Fin Si

Fin

Remarques :

- *Sref* est meilleure en qualité ⇔ *f(Sref)* = Max *f*(*S*) où : $S \in Danse$ et $S \notin Tabou$.
- *Sref* est meilleure en diversité ⇔ *diversité(Sref)* = Max *diversité*(*S*)

  où : $S \in Danse$.
- Si deux solutions *S1* et *S2* sont de même qualité (elles ont la même valeur de la fonction objectif), c'est celle dont le degré de diversité est le plus élevé qui sera choisie.
- De même, si deux solutions *S1* et *S2* présentent le même degré de diversité, c'est la celle qui améliore la fonction objectif qui l'emporte.

- Il peut arriver, bien que très rarement, que toutes les solutions de *Danse* existent dans la liste *Tabou*. Pour palier à ce problème, la solution de référence sera générée aléatoirement.

5. Conclusion :

Dans ce chapitre, nous avons fourni une description sommaire du méta- heuristique Optimisation par Essaims d'Abeilles (BSO), puis nous avons présenté une adaptation de BSO aux caractéristiques du problème MAX-W-SAT. Les éléments essentiels de cette adaptation, et dont dépendent la puissance, et l'efficacité de l'algorithme sont : la définition du choix de *Sref,* Les stratégies de génération des solutions de *SearchArea*, ainsi que la recherche effectuée par les abeilles.

Chapitre III

Calcul parallèle et stratégies de parallélisation des métaheuristiques

1. Introduction

Durant ces dernières années, nous avons assisté à une demande croissante et variée, d'applications informatiques nécessitant une grande puissance de calcul. Ceci se justifie par des besoins plus accrus en précision de calcul, le traitement des données de très grandes tailles, ou tout simplement, le passage de la théorie à la pratique de certains concepts des mathématiques discrètes et combinatoires. Dans cette dernière gamme, on note l'utilisation des techniques heuristiques, qui sont une preuve que les performances nécessaires, sont encore, et très souvent, au-delà du possible. Les machines monoprocesseurs n'étant pas capables de résoudre de tels problèmes, les chercheurs ont eu recours au *parallélisme*.

Le principe de base qui est à l'origine du calcul parallèle, est à la fois simple et ancien: Le travail avance plus vite en étant à plusieurs que seul. C'est une idée qui est déjà exploitée dans la plupart des activités humaines, par des entreprises regroupant de plus en plus d'hommes et de moyens. Le concept du parallélisme, rompt avec l'approche classique qui consiste à diminuer le temps de calcul en effectuant plus vite une opération. En calcul parallèle, le gain de temps provient de la réalisation simultanée de plusieurs opérations.

Face aux demandes toujours croissantes en puissance de calcul, le domaine du parallélisme s'est considérablement développé depuis 1985. La plupart des ordinateurs et des logiciels utilisent ce concept d'une manière ou d'une autre.

Dans ce chapitre, nous passons en revue les différentes architectures parallèles. Nous définissons ensuite, les mesures de performance des algorithmes parallèles.

2. Classification des architectures parallèles

Le processus essentiel dans un ordinateur, est l'exécution d'une suite d'instructions sur un ensemble de données. En général, les ordinateurs peuvent être classifiés selon la multiplicité des flots d'instructions, et des données disponibles matériellement.

Dans la classification de Flynn [FLY66], on distingue essentiellement quatre types de machines :

- **SISD:** *Single Instruction Single Data.*
- **SIMD:** *Single Instruction Multiple Data*.
- **MISD:** *Multiple Instruction Single Data.*
- **MIMD:** *Multiple Instruction Multiple Data*.

Un flot d'instructions est une suite d'instructions issues d'une partie contrôle, en direction d'un ou plusieurs processeurs. Un flot de données, est une suite de données venant d'une zone mémoire, en direction d'un processeur ; ou venant d'un processeur, en direction d'une zone mémoire.

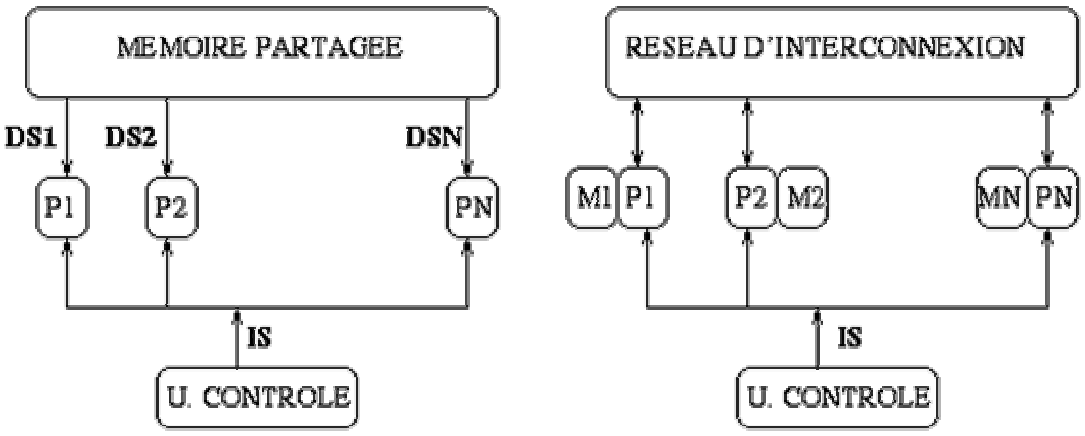
2.1. Machine SISD

Une machine *SIS,* est ce que l'on appelle d'habitude une machine de Von Neumann. Une seule instruction est exécutée, et une seule donnée est traitée à tout instant. Les algorithmes pour les calculateurs SISD ne contiennent aucun parallélisme. [POL]

2.2. Machine SIMD

Plusieurs unités de traitement sont supervisées par la même unité de contrôle. Toutes les unités de traitement reçoivent la même instruction diffusée par l'unité de contrôle, mais opèrent sur des données différentes. [COS93]. Ces données peuvent être soit resserrés, on parle alors de *Mémoire Partagée* (SM pour Share Memory),soit disjointes, ce qui correspond au modèle dit à *Mémoire Distribuée* (DM pour Distributed Memory).[TAD01]

Fonctionnellement, on peut avoir les deux schémas suivants (mémoire partagée ou distribuée) :



Les machines SIMD sont particulièrement utiles pour traiter les problèmes à structure régulière, où la même instruction, s'applique à des sous-ensembles de données.

***Exemple*:** Addition de deux matrices A + B = C.

Soient deux matrices A et B d'ordre 2, et 4 processeurs.

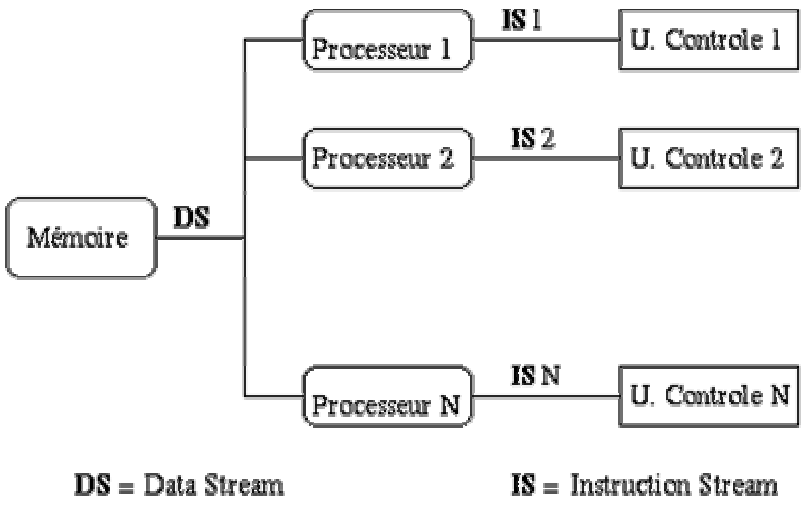
A11 + B11 = C11 ... A12 + B12 = C12

A21 + B21 = C21 ... A22 + B22 = C22

La même instruction est envoyée aux 4 processeurs (ajouter les 2 nombres), et tous les processeurs exécutent cette instruction simultanément. Un pas de temps suffit, contre quatre sur une machine séquentielle. [POL]

2.3. Machine MISD

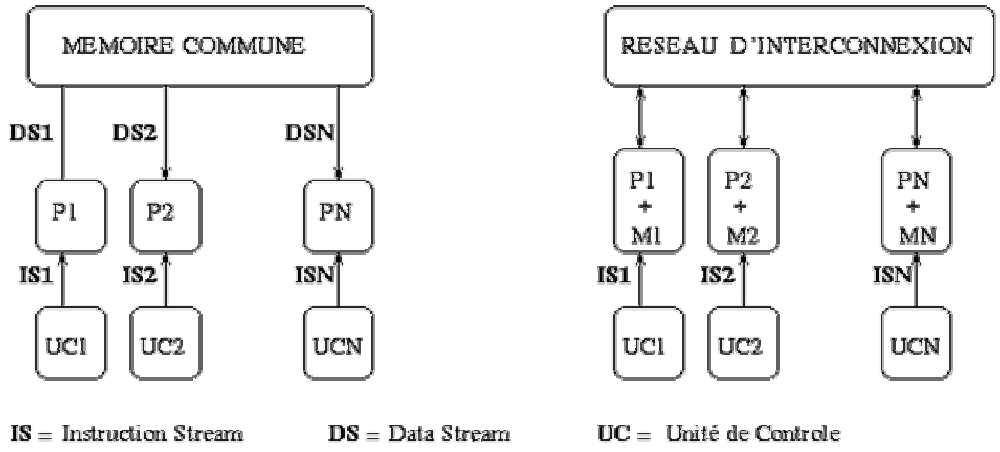
Plusieurs instructions s'exécutent sur une même donnée: *N* processeurs, chacun avec sa propre unité de contrôle, se partagent une mémoire commune. Il y a *N* flots d'instructions (Algorithmes / programmes), et un seul flot de données. Chaque processeur effectue une opération différente, au même moment, sur la même donnée. Actuellement, il n'existe aucune machine construite sur ce modèle. [classification.htm]. Toutefois, on peut théoriquement lui faire correspondre le mode d'exécution en *pipeline* [TAD01]



2.4. Machine MIMD

C'est la classe la plus générale et la plus puissante de toute cette classification.

Il y a *N* processeurs, *N* flots d'instructions et *N* flots de données. Chaque processeur exécute un flot d'instructions généré par sa propre unité de contrôle. (i.e. chaque processeur est capable d'exécuter son propre programme sur des données différentes). Donc, chaque processeur opère de façon *asynchrone***.** Et on peut aussi avoir fonctionnellement, les deux schémas suivants (mémoire partagée ou distribuée) :



Dans le premier modèle (calculateurs MIMD à mémoire commune ou mémoire partagée), tout le trafic entre les processeurs et la mémoire commune passe par un bus. Chaque processeur lit en mémoire les données dont il a besoin, effectue son traitement, puis écrit les résultats en mémoire. Le trafic augmentant avec le nombre de processeurs, ce bus devient rapidement un goulet d'étranglement. Souvent, pour résoudre partiellement ce problème, une mémoire cache est associée à chaque processeur. L'objectif est de diminuer le trafic sur le bus, et de rendre les données accessibles plus rapidement au processeur, puisqu'il est plus rapide pour lui de les lire dans un cache, que dans une grande mémoire globale. Typiquement, le nombre de processeurs ne dépasse pas quelques dizaines.

Dans la seconde classe de calculateurs (Machines MIMD avec un réseau de connexion ou à mémoire distribuée), les processeurs sont souvent appelés "noeuds" et ne partagent rien d'autre que le réseau. Ce réseau est comparativement peu utilisé par rapport au trafic sur le bus d'une mémoire commune. Il sert uniquement à échanger des données entre les processeurs, en utilisant un protocole de communication de type "passage de messages". Chaque processeur possède sa propre zone mémoire, lit sur un ou plusieurs canaux de communication les données dont il a besoin, en provenance d'autres processeurs, effectue son traitement, puis transfère les résultats en direction des processeurs qui les demandent. Le nombre de noeuds dans ce type d'architectures peut atteindre plusieurs centaines.

Duncan [DUN90] a étendu cette classification selon le mode de synchronisation des noeuds :

– Les machines *synchrones* : Une horloge globale synchronise les différents noeuds du système sur une base de temps commune.

– Les machines asynchrones : Chaque machine est autonome et possède une horloge locale qui lui est propre.

Les machines SIMD sont synchrones par définition. La majorité des machines MIMD sont asynchrones.

3. Performances d'un algorithme parallèle [COS93] [BOU98]

L'évaluation des performances d'un algorithme parallèle est importante, pour connaître le "gain" obtenu par la parallélisation, par rapport à l'algorithme séquentiel.

3.1 Le facteur d'accélération

Considérons un algorithme qui s'exécute sur un ordinateur parallèle comportant p processeurs (identiques) en un temps $t_{p,,}$ *et soit* t_s son temps d'exécution séquentiel (c'est-à-dire sur un ordinateur avec un seul processeur). On définit le facteur d'accélération (*Speed up*) par le rapport:

$$s_p = \frac{t_s}{t_p}$$

3.2 Loi d'Amdhal

Divers auteurs ont essayé de préciser les bornes du facteur d'accélération. Amdhal [AMD67] propose de considérer la quantité inhérente de programme séquentiel, qui ne peut donc être exécutée que par un seul processeur.

Si $t_s = t_{seq} + t_{par}$ on en déduit que $t_p = t_{seq} + \frac{t_{par}}{p}$. Par conséquent :

$$S_p = \frac{t_{seq} + t_{par}}{t_{seq} + \frac{t_{par}}{p}} \leq \frac{t_s}{t_{seq}}$$

Donc, le facteur d'accélération est, quelque soit p, inférieur à la proportion de code séquentiel. Par exemple, si 10% du programme est séquentiel, le facteur d'accélération sera inférieur à 10, quelque soit p. Ce résultat est connu sous le nom de *Loi d'Amdhal* (1967). Elle exprime donc, que le facteur d'accélération, est borné par une borne indépendante du nombre de processeurs et de la structure de la machine.

3.3 Notion d'efficacité

Le facteur d'efficacité est une mesure globale de la qualité d'un algorithme parallèle. On introduit l'efficacité d'un algorithme parallèle, comme la rapport de l'accélération sur le nombre de processeurs.

$$e_p = \frac{s_p}{p}$$

Idéalement, l'accélération devrait atteindre *p,* le nombre de processeurs. En conséquence, l'efficacité serait égale à 1. En réalité, ces chiffres ne sont jamais atteints, à cause des temps de communication inter processeurs.

4. Stratégies de parallélisation des métaheuristiques

Bien que les métaheuristiques fournissent des stratégies tout à fait efficaces pour trouver des solutions approximatives aux problèmes d'optimisation combinatoire, le temps de calcul nécessaire à l'exploration de l'espace des solutions peut être très grand. Avec la prolifération des ordinateurs parallèles, et la communication rapide dans les réseaux, des implémentations parallèles de métaheuristiques, apparaissent tout à fait naturellement, comme étant une bonne alternative pour accélérer la recherche des solutions approximatives.

Plusieurs stratégies ont été proposées et appliquées à différents problèmes. D'ailleurs, les implémentations parallèles, permettent également de résoudre des instances de problèmes plus réalistes, donc plus grandes, et de trouver des solutions améliorées, grâce à la division de l'espace de recherche et aux possibilités d'intensification et de diversification de la recherche.

Les versions parallèles des métaheuristiques sont proposées avec une fréquence croissante. Les métaheuristiques les plus souvent utilisées et parallélisées sont : *les approches évolutionnaires - algorithmes génétiques* (GA: Holland 1975; Goldberg, 1989; Whitley, 1994; Fogel,1994; Michalewicz, 1992; etc) *Le recuit simulé* (SA; Metropolis et al, 1953; Kirkptrick, gelatt et vecchi, 1983; Laarhoven et Aarts, 1989; Aarts et Korst, 1989,etc), *La recherche Tabou* (TS:Glover 1986,1989,1990,1996; Glover et laguna,1993; Osmn et kelly, 1996, etc), *GRASP* (Feo et resende, 1995) et *Système de colonie de fourmis* (Dorigo, 1992; Colorni, Dorigo et Manniezzo, 1991,1992).

L'augmentation rapide et continue du nombre de développements de métaheuristiques parallèles, et leurs applications jusqu'en 1997, est illustrée dans la table 1 [CRC98], et cet intérêt pour la parallélisation des métaheuristiques ne cesse d'augmenter.

GA		SA		TS	
<1990	≥ 1990	<1990	≥ 1990	<1990	≥ 1990
22	88	33	42	1	35

Table1: nombre de publications dans le domaine des métaheuristiques parallèles

La table synthétise les rapports, thèses et articles publiés, séparés en deux colonnes, avant 1990 et après 1990 jusqu'en 1997. Ces deux colonnes représentent approximativement, le même intervalle de temps, puisque la plupart des contributions avant 1990 ont été rapportées dans la seconde moitié des années 1980.

En fait, le nombre de recherches et d'études dédiées aux métaheuristiques et à leur parallélisation, a atteint le niveau où des surveys, des taxonomies et des synthèses sont proposés.

4.1 Stratégies de parallélisation [CRT03][CRC98][CUN02]

Le but central du calcul parallèle, est d'accélérer le calcul, en partageant la charge du travail sur plusieurs processeurs. Du point de vue conception algorithmique, les stratégies de calcul parallèle "pures", exploitent l'ordonnancement partiel des algorithmes, c'est à dire, les ensembles d'opérations qui pourraient s'exécuter de manière concurrente sans modifier la méthode de résolution et la solution finale obtenue, et cela correspond au parallélisme "naturel" présent dans l'algorithme. L'ordonnancement partiel des algorithmes fournit deux sources principales de parallélisme, le parallélisme de *données* et le parallélisme *fonctionnel.*

Le parallélisme de données, est la première source de parallélisme. Il consiste à effectuer la même opération en parallèle, sur un ensemble de données réparties sur les processeurs. Le programme parallèle se résume en une succession de phases de calcul et de phases de communication. La phase de communication permet de mettre à jour des données distantes, ou à redistribuer des données pour améliorer l'équilibrage de charge entre les processeurs.

Pour illustrer, considérons la multiplication de deux matrices. Pour effectuer cette opération, on doit exécuter plusieurs opérations identiques, en effectuant des sommes et des produits de nombres. Il est possible de chevaucher l'exécution de ces opérations identiques, sur différentes données d'entrée. Parmi les structures d'ordinateurs avec plusieurs unités arithmétiques et logiques (UAL), les ordinateurs SIMD (*single flow of instructions, multiple flow of data),* conviennent particulièrement pour ce type de parallélisme, car ils peuvent charger la même opération sur toutes les UALs, et les exécuter sur différentes données d'entrée (*multiple flow of data*).Le nombre total des opérations requises pour le produit matriciel n'est pas réduit, mais étant donnée la parallélisation de l'exécution des opérations, le temps de calcul est réduit.

Le parallélisme de contrôle ou fonctionnel, consiste à découper une application en plusieurs tâches qui peuvent s'exécuter en parallèle. L'algorithme parallèle, peut être décrit sous la forme d'un graphe orienté, sans cycle de dépendances entre les tâches : une tâche ne peut être effectuée qu'après la fin de l'exécution de toutes les tâches qui la précèdent dans le graphe de dépendance. Dans ce paradigme de parallélisme, ce sont les traitements qui sont placés sur un processeur. Les données sont accédées à distance, ou déplacées en tant que paramètre du traitement.

Le calcul parallèle basé sur le parallélisme fonctionnel ou de données, est particulièrement efficace, lorsque les algorithmes manipulent des structures de données fortement régulières, comme les matrices dans le produit matriciel. Les algorithmes opérant sur des structures de données irrégulières, comme les graphes, ou sur des données avec de fortes dépendances entre différentes opérations, restent difficiles à paralléliser. Cependant, comme nous allons le voir, la parallélisation des métaheuristiques, offre des opportunités pour trouver de nouvelles manières d'utiliser les machines parallèles, et de concevoir des algorithmes parallèles.

D'un point de vue calcul, les métaheuristiques ne sont que des algorithmes desquels on peut tirer un parallélisme fonctionnel ou de données. Malheureusement, ces types de parallélismes sont difficiles à obtenir dans beaucoup de métaheuristiques. C'est le cas lorsque les itérations successives de l'algorithme présentent une forte dépendance de données (ex: la recherche locale), ou lorsque certaines étapes de l'algorithme doivent être exécutées de manière séquentielle (ex: le passage d'une génération à une autre dans un algorithme génétique est un processus essentiellement séquentiel).

Plusieurs classifications des stratégies de parallélisation des métaheuristiques ont été proposées dans la littérature. Dans cette étude, nous avons retenu deux d'entre elles qui semblent être les plus générales. La première a été donnée par cung et al [CUN02], dont l'étude s'est basée sur celle de verhoeven et aarts [VEA95], qui a présenté une classification des méthodes de parallélisation des algorithmes de la recherche locale. Cette classification se base sur le nombre de chemins (trajectoires) examinés dans le graphe du voisinage. La seconde classification a été présentée par crainic et Toulouse (1997, 2003) [CRT98][CRT03], qui classent les stratégies de parallélisations en trois types, selon la source de parallélisme utilisée, et selon le niveau de l'impact qu'a la stratégie de parallélisation sur la conception algorithmique de la métaheuristique.

4.1.1. Classification des stratégies de parallélisation selon le nombre de chemins

Les métaheuristiques basées sur la recherche locale, peuvent être vues comme étant une exploration du graphe du voisinage (l'espace de recherche) associé à une instance de problème, dans lequel, les nœuds correspondent à des solutions et les arcs connectent les solutions voisines. Chaque itération, consiste en l'évaluation des solutions dans le voisinage de la solution courante, suivie d'un déplacement vers l'une d'entre elle, jugée meilleure, en évitant le plus possible de s'arrêter à un optimum local, et jusqu'à ce qu'aucune amélioration supplémentaire de la meilleure solution trouvée ne peut être obtenue. Les solutions visitées durant la recherche, définissent un chemin (ou une trajectoire) dans le graphe du voisinage.

Les implémentations parallèles des métaheuristiques, utilisent plusieurs processeurs pour générer ou explorer le graphe du voisinage de manière concurrente. Puisque ce graphe n'est pas connu à priori, les parallélisations des métaheuristiques sont des *applications irrégulières,* dont l'efficacité dépend fortement de la granularité de l'algorithme, et de l'utilisation des techniques de l'équilibration des charges.

Dans cette classification, nous distinguons entre deux approches de parallélisation de la recherche locale, selon le nombre de trajectoires examinées dans le graphe du voisinage.

1. Chemin unique.
2. Chemins multiples.
    a- processus de recherche indépendants.
    b- Processus de recherche coopératifs.

Dans le cas de la *parallélisation à chemin unique*, un seul chemin ou trajectoire est traversé dans le graphe du voisinage. La recherche du meilleur voisin à chaque itération, est effectuée en parallèle, soit par la parallélisation de l'évaluation de la fonction coût, ou par la décomposition du domaine (l'évaluation du voisinage ou l'instance du problème elle-même sont décomposées et distribuées sur différents processeurs). Une *parallélisation à chemins multiples,* est caractérisée par l'exploration en parallèle de plusieurs trajectoires, chacune par un processeur différent. Les processus de recherche (les processus s'exécutant sur chaque processeur, et traversant un chemin du graphe du voisinage), peuvent être indépendants (ne s'échangent aucune information) ,ou coopératifs (l'information collectée au long de chaque trajectoire est disséminée et utilisée par d'autres processus).

4.1.1.1. Parallélisation à chemin unique

Le but de cette stratégie, est simplement d'accélérer l'exploration séquentielle du graphe du voisinage. La tâche dont l'exécution est distribuée sur les différents processeurs, peut être l'évaluation de la fonction coût pour chaque voisin de la solution courante, ou la construction du voisinage lui-même. Dans le premier cas, les accélérations peuvent être obtenues, sans aucune modification dans la trajectoire suivie par l'implémentation séquentielle. Dans le second cas, la décomposition du voisinage, et sa distribution sur plusieurs processeurs, permettent souvent une exploration plus profonde d'une portion du voisinage, plus grande que celle explorée par l'implémentation séquentielle qui, souvent, utilise des techniques de réduction du voisinage. Donc, la trajectoire suivie dans le graphe du voisinage, peut être mieux guidée dans le mode parallèle que dans le mode séquentiel, menant ainsi, probablement, à de meilleures solutions.

L'idée des voisinages distribués, peut être utilisée pour formaliser des stratégies de parallélisations basées sur la décomposition du domaine. Une stratégie basée sur les voisinages distribués, consiste en deux parties. Chaque itération de la recherche locale, commence par la diffusion de la solution courante à tous les processeurs, avec une décomposition du domaine, qui définit les voisinages locaux (ie, la portion du voisinage qui sera explorée par chaque processeur). Cette décomposition est temporaire, et peut changer d'une itération à l'autre. Dans la seconde partie, chaque processeur trouve, et propose un déplacement dans son voisinage local. Ou bien ces déplacements sont combinés, ou alors le meilleur déplacement trouvé dans les voisinages locaux est sélectionné, ainsi, une nouvelle solution est élue. Ce processus, est répété jusqu'à ce que la solution courante ne puisse plus être améliorée.

Les parallélismes à chemin unique, ont une granularité fine ou moyenne, et ont besoin de synchronisations fréquentes. Elles sont extrêmement dépendantes de l'application, en terme de définition de la fonction coût, et de la structure du voisinage. Leurs premières applications, sont apparues dans le contexte du recuit simulé, et des algorithmes génétiques. Même si les stratégies de décomposition du domaine ne réduisent pas toujours les temps de calculs, elles sont souvent utilisées pour explorer de grands voisinages.

4.1.1.2 Parallélisation à chemins multiples

La plupart des implémentations parallèles des métaheuristiques, suivent des stratégies à chemins multiples. Les tâches parallèles ont une plus grande granularité. En plus de la recherche de l'accélération, l'amélioration de la qualité de la solution est aussi visée. Les processus peuvent être indépendants (pas de partage d'informations), ou bien coopératifs (communication par échange ou partage d'informations).

a- Processus de recherche indépendants

Il existe deux approchent fondamentales:

- L'exploration en parallèle de plusieurs trajectoires, originaires de nœuds différents du graphe du voisinage: Chaque chemin démarre d'une solution différente (ou d'une population différente dans le cas des méthodes basées population). Les processus peuvent utiliser le même algorithme de recherche, ou des algorithmes différents, avec les mêmes valeurs de paramètres, ou avec des valeurs différentes. Les chemins peuvent se croiser à un, ou plusieurs nœuds du graphe du voisinage. Si p processeurs sont utilisés, cette stratégie correspond à l'exécution de p recherches séquentielles indépendantes.
- L'exploration en parallèle des sous graphes du graphe de voisinage, obtenus par la décomposition de l'instance du problème: Plusieurs sous graphes du graphe du voisinage, sont explorés en parallèle, sans intersection des trajectoires correspondantes.

Les stratégies à plusieurs chemins basées sur de processus indépendants, peuvent être implémentées très facilement. Elles conduisent à de bonnes accélérations. Des implémentations très robustes peuvent être obtenues, en utilisant différentes valeurs de paramètres à chaque processeur. Du reste, un bon partitionnement de l'espace de recherche, permet une bonne couverture de ce dernier et évite un travail redondant. D'autre part, ce modèle est plutôt pauvre et peut, très facilement, être simulé dans le mode séquentiel, par

plusieurs exécutions successives avec différentes initialisations. Le manque de coopération entre les processus, ne permet pas l'utilisation des informations collectées par les différents processeurs durant leur recherche. Comme les trajectoires peuvent être longues, les problèmes de déséquilibre des charges, sont susceptibles d'apparaître. Un travail redondant peut être fait, si l'espace de recherche n'est pas bien partitionné.

b- Processus de recherche coopératifs

C'est le type de stratégies le plus général et le plus prometteur, demandant plus d'efforts de programmation. Les processus échangent et partagent des informations, collectées au long des trajectoires qu'ils explorent. Ces informations, sont implémentées comme étant des variables globales, ou bien comme étant une zone dans la mémoire locale du processeur central, qui peut être accédée par tous les processeurs.

Dans ce modèle, dans lequel les processus coopèrent, et l'information collectée tout au long de chaque trajectoire est utilisée pour améliorer les autres trajectoires, on escompte non seulement, à accélérer la convergence vers la meilleure solution, mais aussi à trouver de meilleures solutions que celles obtenues par les stratégies de processus indépendants, en moins de temps. L'aspect le plus difficile à appréhender, est la détermination de la nature des informations à échanger et à partager, pour améliorer la recherche, sans prendre beaucoup de temps ou d'espace mémoire additionnels. Ces informations peuvent être: Les solutions élues et leurs coûts, les meilleures solutions trouvées, les fréquences de déplacement, les listes tabous, les tailles des populations…etc. Ces données peuvent offrir une vue globale sur l'espace de recherche, et peuvent être utilisées pour guider les procédures d'intensification et de diversification.

4.1.2. Classification des stratégies de parallélisation selon la source du parallélisme

Selon la source de parallélisme, et le niveau de l'impact que peut avoir la stratégie de parallélisation sur la conception algorithmique de la métaheuristique, on peut classifier les méthodes parallèles selon les trois stratégies suivantes:

Type 1: *Parallélisation d'opérations dans une itération de la méthode de résolution*. Cette stratégie nommée aussi *parallélisme de bas-niveau,* vise uniquement à accélérer les calculs, sans aucune tentative d'améliorer l'exploration de l'espace de recherche (sauf si on alloue au processus parallèle le même temps alloué au séquentiel), ou la qualité de la solution. Cette stratégie retourne la même solution que l'algorithme séquentiel seulement, plus rapidement.

Type 2: *Décomposition de l'espace de recherche.* Cette stratégie, est basée sur le fait que la puissance de calcul, peut être exploitée pour résoudre une multitude de problèmes plus petits. Une solution globale, pourrait ensuite être extraite ou construite. La trajectoire de la recherche de l'approche parallèle résultante, est cependant, différente de celle correspondant à la méthode séquentielle.

Type 3: *Processus de recherche avec différents degrés de synchronisation et de coopération.* Cette stratégie, tente de faire une exploration plus minutieuse de l'espace de recherche, en initialisant plusieurs processus de recherche, qui procèdent simultanément, à la recherche à travers l'espace de recherche.

4.1.2.1 Type 1: *La parallélisation de bas-niveau.*

Les parallélisations de type 1, correspondent au calcul parallèle de certaines, ou toutes les opérations, qui composent une itération de la méthode séquentielle correspondante. L'évaluation des individus et les déplacements, sont typiquement, les opérations sujettes à la parallélisation de type 1.

Les stratégies de type 1, visent directement à réduire le temps d'exécution d'une méthode de résolution donnée. En fait, lorsque le même nombre d'itérations est accordé aux deux versions, séquentielle et parallèle de la méthode, et que les mêmes opérations sont exécutées à chaque itération (exemple: le même ensemble de déplacements est évalué et le même critère de sélection est utilisé), l'implémentation parallèle suit le même chemin que la méthode séquentielle, et retourne la même solution.

Certaines implémentations, modifient la version parallèle, pour profiter de l'extra puissance de calcul disponible, sans pour autant modifier la méthode de recherche de base. Par exemple, on peut évaluer tous les déplacements dans le voisinage de la solution courante, au lieu de se contenter de ceux d'un sous ensemble donné. Les modèles de recherche résultant des implémentations séquentielles et parallèles, sont alors différents dans la plupart des cas. Cependant, comme la conception algorithmique n'est pas modifiée, ces approches restent qualifiées de parallélisme de bas niveau.

4.1.2.2 **Type2**: *Parallélisation par décomposition du domaine*

Les méthodes de parallélisation de métaheuristiques de type 2, sont généralement basées sur la décomposition du vecteur des variables de décision en ensemble disjoints. La procédure heuristique, est alors appliquée à chaque sous-ensemble, les variables en dehors du

sous ensemble étant considérées fixées. De telles stratégies sont généralement implémentées en utilisant une certaine architecture maître esclaves:

- Un processus maître, effectue le premier partitionnement de l'ensemble des variables de décisions en sous ensembles, qu'il distribue sur les processus esclaves. Durant la recherche, il modifie régulièrement les partitions. Les modifications, peuvent être effectuées à des intervalles déterminés au départ, ou durant l'exécution, ou encore plus souvent, lorsqu'on relance la recherche.
- Les processus esclaves explorent indépendamment, et de manière concurrente, les partitions qui leur sont affectées. La recherche peut procéder exclusivement à l'intérieur de la partition, les autres variables étant considérées fixées et non affectées par les déplacements qui sont effectués, ou, peut avoir accès à tout le vecteur de solutions.
- Quand les esclaves ont accès à tout le voisinage, le maître doit combiner les solutions partielles, obtenues à partir de chaque sous-ensemble, en une solution complète du problème.

Notons qu'une décomposition basée sur le partitionnement du vecteur des variables de décision, peut laisser de grandes partitions de l'espace de recherche non exploré. C'est la raison pour laquelle, le partitionnement est répété, pour créer différents segments du vecteur des variables de décision, et la recherche est recommencée.

4.1.2.3 Type3: *Plusieurs processus de recherche*

Les méthodes parallèles qui consistent en plusieurs recherches concurrentes dans l'espace de recherche, sont classifiées comme étant des stratégies de type 3. Ces méthodes, peuvent utiliser, ou pas, la même approche métaheuristique. Elles peuvent démarrer de la même solution initiale, ou de solutions différentes, et peuvent communiquer durant la recherche, ou uniquement à la fin, pour identifier la meilleure solution globale. Les communications peuvent s'effectuer de manière synchrone, ou asynchrone, et peuvent être guidées par les événements, ou exécutées à des moments prédéterminés, ou décidés dynamiquement. Ces stratégies, appartiennent à la classe des stratégies de parallélisation à chemin multiples, dans la classification de Verhoeven et Aarts [VEA95].

Les stratégies de recherche multiple,s peuvent de plus, être divisées en deux classes: Approches *indépendantes,* et approches *coopératives.* Dans la première, plusieurs recherches sont initiées, et le meilleur résultat est sélectionné à la fin, parmi les résultats des processus

individuels. Les approches de recherches indépendantes, sont donc équivalentes à une stratégie de recommencement accélérée. Pour définir une recherche multiple coopérative, un nombre de paramètres importants doivent de plus être déterminés:

- La topologie de connexion définissant la manière dont les processeurs sont liés.
- la méthode de communication (diffusion, propagation, utilisation d'une mémoire centrale…etc.).
- les processus à exécuter entre les échanges d'informations.
- le type de communication: synchrone ou asynchrone.
- le moment de l'échange d'informations.
- les informations à échanger.

La détermination de ces paramètres, peut avoir un impact significatif sur le comportement de la procédure parallèle, et la performance de la recherche.

4.2. Métaheuristiques parallèles

Dans cette partie, nous allons passer en revue quelques parallélisations, qui ont été faites des métaheuristiques les plus utilisées: La recherche tabou, le recuit simulé, les algorithmes génétiques, GRASP, et colonie de fourmis.

4.2.1. Recherche Tabou parallèle [DPR02]

La recherche Tabou (RT) introduite pour la première fois par Glover [GLO89] [GLO90], est une métaheuristique pour trouver de bonnes solutions à des problèmes d'optimisation combinatoire. Une liste taboue, est un ensemble de solutions déterminées à partir des t dernières itérations de l'algorithme, où, *t* est fixé ou variable selon l'état de la recherche, ou selon le problème. A chaque itération, étant donnée la solution courante x et ses voisins correspondants *N(x),* la procédure se déplace vers la solution dans le voisinage *N(x,)* qui améliore le plus la fonction objectif. Cependant, les déplacements qui mènent à des solutions figurant dans la liste taboue, sont interdits. S'il n'y a aucun déplacement améliorant la fonction objectif, la RT choisit le dernier déplacement qui modifie la fonction objectif. La liste taboue évite de retourner à un optimum local, auquel la procédure a récemment échappé. Un critère d'arrêt pour la RT, est la limite du nombre de déplacements consécutifs n'apportant aucune amélioration.

La stratégie de parallélisation de bas niveau a été appliquée à la RT (Crainic, Toulouse et Gendreau, 1997). Ces implémentations, correspondent à un processus maître qui exécute une procédure taboue séquentielle, mais les déplacements possibles dans le voisinages, sont évalués en parallèle, par des processus esclaves, à chaque itération. Les processus esclaves, peuvent évaluer uniquement les déplacements dans l'ensemble qu'ils reçoivent du processus maître, ou peuvent sonder au-delà de chaque déplacement dans l'ensemble. Le processus maître, reçoit et traite les informations résultant des opérations esclaves, puis sélectionne et implémente le prochain déplacement. Le maître, rassemble aussi toutes les informations générées durant l'exploration taboue, met à jour les mémoires, décide quand activer les différentes stratégies de recherche – diversification par exemple- ,et quand stopper la recherche.

Les succès des parallélisations de type1 de la RT, sont plus significatifs que pour les algorithmes génétiques (AG) et le recuit simulé (RS). En effet, beaucoup de résultats intéressants, ont été obtenus lorsque les voisinages sont très grands et que le temps de l'évaluation et de l'exécution d'un déplacement donné est relativement petit. De plus historiquement (la première moitié des années 1990) les stratégies de parallélisation de type 1 de la RT ont permis d'obtenir des améliorations des meilleures solutions de problème connues dans la littérature.

Les implémentations typiques des stratégies de parallélisations de la RT de type 2, partitionnent le vecteur des variables de décision, et effectuent la recherche dans chaque sous ensemble [CTG95a]. Comme avec les implémentations de type 1, les méthodes parallèles de type 2, ont été plus réussies pour des problèmes, pour lesquels, plusieurs itérations peuvent être faites en un temps relativement court.

Les parallélisations de type 3 de la RT, suivent le même modèle: p processus de recherche à travers le même espace de solutions, démarrant éventuellement, par différentes solutions initiales, et utilisant éventuellement, différentes stratégies de la RT. Historiquement, les méthodes multi processus indépendants et synchrones coopératifs, ont été proposées en premier. Cependant, actuellement, les procédures asynchrones sont développées d'une manière croissante.

Une autre stratégie de type 3, basée sur l'approche maître esclaves peut être implémentée. Chaque esclave effectue une RT séquentielle complète. Le maître rassemble les solutions trouvées, sélectionne la meilleure solution globale, et réinitialise les processus pour

une nouvelle recherche. Un parallélisme de type 1 accélère les évaluations des déplacements des recherches individuelles.

Crainic, Toulouse et Gendreau (1997), ont aussi présenté une comparaison profonde de différentes stratégies de parallélisation basée sur cette taxonomie (classification selon la source du parallélisme). Les auteurs, ont implémenté plusieurs stratégies de type 1 et 2, une approche multi processus indépendants, et un nombre de méthodes multi processus coopératifs, synchrones et asynchrones. Ils rapportent que les versions parallèles, sont parvenues à des solutions meilleures que celles obtenues par la RT séquentielle, et qu'en général, les méthodes asynchrones étaient plus performantes que les méthodes synchrones. Les processus indépendants, et les approches coopératives ont offert les meilleures performances.

4.2.2. Les algorithmes génétiques parallèles

Un algorithme génétique (AG), est une procédure de recherche proposée et développée par Holland [HOL75] [HOL92]. Elle est basée sur le processus naturel de l'évolution, suivant les principes de la sélection naturelle et de la survie du meilleur.

D'abord, une population d'individus (solutions) est engendrée. Chaque individu est évalué selon la fonction d'évaluation. Les meilleurs individus sont sélectionnés, pour générer une nouvelle génération de cette population. L'opérateur de croisement, est un opérateur génétique qui combine des parties des individus, pour en générer d'autres nouveaux. L'opérateur de mutation, est une transformation génétique unaire, qui crée un nouvel individu, par de petits changements appliqués à un individu existant. De nouveaux individus créés par croisement et/ ou mutation, remplacent toute ou une partie de la population initiale. Le processus d'évaluation et de création d'une nouvelle génération de population, est répété jusqu'à ce que le critère d'arrêt soit satisfait.

Dans la littérature, la parallélisation de bas niveau des AG est appelée parallélisation globale [CAN98]. Dans cette approche, une seule population est considérée. La parallélisation est axée sur l'évaluation des individu,s et à un degré moindre, sur l'application des opérateurs génétiques. L'évaluation parallèle des individus, est obtenue par une approche maître esclaves synchrone, où, le maître alloue des individus à des processeurs esclaves pour être évalués. Il n'y a pas de communication entre les processeurs esclaves, seul le maître communique avec les esclaves au début, et à la fin du processus d'évaluation. Pour obtenir une nouvelle génération, la population est divisée, et distribuée sur les processeurs, et les opérateurs génétiques usuels sont appliqués à chaque partition.

Si l'on considère une représentation binaire de l'individu de longueur n. En fixant un ensemble quelconque de n-p bits (gènes dans le vocabulaire des AG), et en considérant que les p bits restant sont libres, et peuvent prendre n'importe quelle valeur, on crée un schéma qui représente tous les individus possibles obtenus, en affectant des valeurs spécifiques aux p gènes. L'ensemble des schémas, représente un partitionnement implicite de l'espace des solutions. Une implémentation parallèle des AG basée sur les principes de type 2, peut être construite en se basant sur ces schémas, selon deux mécanismes : Le premier correspond aux opérateurs des AG séquentiels appliqués exclusivement aux p gènes libres de chaque schéma.

Le second, adapte itérativement les règles de partitionnement et sélectionne les individus qui vont produire le schéma.

Nous n'avons pas trouvé dans la littérature des implémentations des AGP qui suivent ces mécanismes, ou aucune autre idée pour le partitionnement de la représentation des individus. La plupart des AGP qui incorporent les principes de partitionnement, prennent avantage du parallélisme naturel, inhérent aux techniques évolutionnaires, et travaillent sur le partitionnement des populations, et sont donc considérés comme étant de type 3.

Nous trouvons dans la catégorie de recherches multiples, la forme la plus utilisée des AGP. Deux principales stratégies de parallélisation sont utilisées:

1. Prallélisme "coarse-grained", où des sous populations sont allouées à chaque processeur de la machine parallèle. Les étapes d'application des opérateurs génétiques, sont exécutées à l'intérieur de la population.
2. Parallélisme "fine-grained", où un seul individu, ou un petit nombre d'individus, sont alloués à chaque processeur.

Pour les AGP "coarse-grained", le parallélisme est obtenu en appliquant l'approche utilisée dans l'AGP global sur plusieurs processeurs, à des sous populations. Un opérateur de migration est ajouté à la liste des opérateurs génétiques. Il permet l'échange d'informations entre sous populations.

Les parallélisations "fine-grained", sont des méthodes synchrones, qui divisent la population en un grand nombre de sous-ensembles. Chaque sous ensemble, est alors, connécté à plusieurs autres dans son voisinage, et les opérateurs génétiques sont appliqués à travers des échanges asynchrones dans le même voisinage seulement.

4.2.3. Recuit simulé parallèle

La méthode du RS, a été proposée en 1983 par Kirkpatrick et al [KGV83]. Depuis, beaucoup de recherches ont été accomplies en ce qui concerne son implémentation (séquentielle ou parallèle), pour résoudre une variété de problèmes d'optimisation combinatoire.

Une itération du RS, consiste en quatre étapes principales: Sélectionner un déplacement, évaluer la fonction coût, accepter ou rejeter le déplacement, mettre à jour (remplacer) la solution actuelle si le déplacement est accepté. Deux approches sont utilisées pour obtenir des algorithmes de RSP : Parallélisme à essai unique, où, un seul mouvement est calculé en parallèle, et stratégies à essais multiples, où, plusieurs mouvements sont évalués simultanément.

L'évaluation de la fonction coût dans certaines applications, peut être assez intensive du point de vue calculs. C'est pour cela, que le parallélisme fonctionnel est suggéré. Les stratégies à essai unique, exploitent le parallélisme fonctionnel, en décomposant l'évaluation de la fonction coût en problèmes plus petits, qui sont affectés à différents processeurs. Le degré de parallélisme résultant, est cependant, très limité.

Les stratégies de parallélisation à essais multiples, distribuent les itérations de la recherche sur les différents processeurs. Chaque processeur, exécute pleinement les quatre étapes de chaque itération. Ces stratégies appartiennent au parallélisme de type ,1 où, les essais parallèles démarrent de la même solution et ont accès à des déplacements dans le voisinage entier.

La plupart des stratégies de parallélisation à essais multiples pour le RS, suivent une approche de type 2, et partitionnent les variables en sous ensembles. Une approche maître esclave, est généralement utilisée. Pour commencer la recherche, le processus maître, partitionne les variables de décision en p ensembles initiaux. L'ensemble approprié, et les valeurs initiales, sont envoyés à chaque processeur. Dans une seconde étape, chaque processeur esclave, exécute la recherche RS, et envoie sa configuration partielle de la solution entière au processus maître. Une fois que l'information aura été reçue de la part de tous les esclaves, le processus maître, fusionne les solutions partielles en une solution complète, et vérifie le critère d'arrêt. Si la recherche continue, il génère un nouveau partitionnement des variables.

La performance relativement pauvre des approches de type 1 et 2 a été observée, et par conséquent, il y a peu d'applications de ces stratégies. Les stratégies de parallélisation les plus utilisées sont de type 3. Un développement intéressant, consiste à inclure les principes des opérateurs des AG dans les procédures multi processus du RS. Laursen (1994) [LAU94] propose que chaque processus, exécute parallèlement *k* procédures de recuit simulé, pour un nombre donné d'itérations. Les processus sont ensuite appariés, et chaque processeur migre (copie) ses solutions chez le processeur avec lequel il a été apparié. Ainsi, après la phase de migration, chaque processeur a *2k* solutions initiales, et ce nombre est réduit à *k* par sélection. Ces k nouvelles solutions, deviennent les configurations initiales des k processus de RS parallèles, et la recherche redémarre sur chaque processeur. Cette approche hybride, tend à donner de très bonnes performances. Les méthodes de type 3 semblent très prometteuses, et continuent de donner les meilleures performances pour le recuit simulé.

4.2.4. GRASP parallèle

GRASP (Greedy Randomized Adaptative Search Procedure) [CEF88] [FEO95], est un processus itératif pour trouver des solutions approximatives à des problèmes d'optimisation combinatoire. Les itérations de GRASP terminent, lorsqu'un critère d'arrêt (comme le nombre maximum d'itérations) est satisfait. Chaque itération, consiste en une phase de construction, dans laquelle une solution est construite, un élément à la fois, et une phase de recherche, qui consiste à trouver l'optimum local dans le voisinage de la solution construite auparavant.

Deux stratégies ont été proposées pour la parallélisation de GRASP. Dans la décomposition de l'espace de recherche, celui-ci est partitionné en plusieurs régions, et GRASP est appliquée à chacune d'elle en parallèle. Dans la parallélisation de l'itération, les itérations de GRASP sont partitionnées, et chaque partition est affectée à un processeur.

4.2.5. Système de colonies de fourmis parallèle

Le système de colonies de fourmis introduit par Dorigo [DMC96], est présenté comme étant une méthode évolutionnaire, inspirée du comportement intéressant des fourmis réelles. En effet, celles-ci sont capables de trouver les plus courts chemins entre la source de leur nourriture et leur fourmilière, malgré le fait qu'elles soient aveugles.

Dans ses allers et retours entre la source de nourriture et la fourmilière, la fourmi dépose une substance nommée "la phéromone". Ses congénères peuvent la sentir, et ont tendance à choisir les chemins les plus marqués par cette substance, donc les plus courts.

Bolondi et Bondanza [BOL93] et al [BKS98], ont proposé deux systèmes de fourmis parallèles très similaires, et fortement synchronisés, appliqués au problème TSP. A chaque itération parallèle, un ensemble de fourmis est généré par le processus " reine", et chaque fourmi explore un chemin en parallèle. Après que chaque fourmi ait fini son exploration, elles sont synchronisées, et l'information phéromone est mise à jour par le processus reine, avec le meilleur chemin trouvé. Comme la quantité de calculs effectués par chaque fourmi est assez petite, les fourmis passent la plupart du temps à se synchroniser avec le processus reine. Pour cette raison, les auteurs ont proposés aussi, dans [BKS98], un système de fourmis parallèle moins synchronisé, pour réduire le temps de synchronisation.

L'idée consiste à implémenter une approche hiérarchique à deux niveaux. Chacun des maîtres du premier niveau, s'occupe en parallèle, mais localement, de sa propre colonie et information phéromone, en utilisant un algorithme séquentiel. Après un certain nombre d'itérations séquentielles, les maîtres du premier niveau sont synchronisés par un processus maître de second niveau, pour mettre à jour globalement l'information phéromone. Puisque les synchronisations sont principalement faites au second niveau de maîtres, cette stratégie, a montré moins de synchronisation générale que la première.

Stuzle [STU98] a proposé une implémentation parallèle très simple, mais l'intérêt principal de son travail, était de comparer la qualité des solutions obtenues par plusieurs exécutions courtes et indépendantes, avec la qualité d'une seule longue exécution. Il a conclu que sous certaines conditions, les courtes exécutions donnent de meilleurs résultats. Les exécutions courtes indépendantes, ont aussi l'avantage d'être facilement faites en parallèle, il est également possible d'utiliser différents ensembles de paramètres pour chacune d'elles.

Dans leur système ANTabu, Talbi et al [TAL99][TAL01] ont appliqué l'approche maître esclave. Le maître gère la matrice de phéromone et la meilleure solution trouvée. A chaque itération, le maître diffuse la matrice de phéromone à toutes les fourmis esclaves. Chaque esclave construit une solution complète, l'améliore en utilisant la RT, et renvoie la solution améliorée au maître.

5. Conclusion

Les implémentations parallèles des métaheuristiques, sont une alternative tout à fait efficace, pour la résolution approximative des problèmes d'optimisation combinatoire. Elles sont habituellement appliquées, dans le contexte des applications complexes, permettant souvent, des réductions en temps de calculs. De bonnes accélérations peuvent être obtenues avec des stratégies parallèles non coopératives. Cependant, les métaheuristiques parallèles, permettent non seulement de résoudre de plus grands problèmes, et de trouver de meilleures solutions par rapport à leurs versions séquentielles, mais en plus, les parallélisations basées sur plusieurs processus de recherche coopératifs, mènent également à des implémentations plus robustes, qui sont susceptibles d'être la contribution la plus importante du parallélisme dans le domaine des métaheuristiques.

L'utilisation de plusieurs processeurs dans un espace de recherche plus grand, en utilisant différentes stratégies et valeurs de paramètres, permet une plus grande diversité, et une recherche plus profonde dans l'espace de solutions. Par conséquent, des solutions améliorées peuvent être trouvées, très souvent, plus rapidement qu'en mode séquentiel.

En résumé, le parallélisme mène souvent à des accélérations pour des applications complexes, et permet de concevoir des algorithmes plus robustes, grâce à des mécanismes d'intensification et de diversification améliorés.

Chapitre IV

Parallélisation de la métaheuristique BSO

1. Introduction

2. Solution initiale

3. Mode de communication

4. Le moment de l'échange d'informations

5. Les valeurs des paramètres empiriques

6. Conclusion

1. Introduction

Dans ce chapitre, nous proposons une parallélisation de la métaheuristique "Bees Swarm Optimization". Pour cela, nous adoptons une démarche originale, qui procède de manière constructive et aboutit à la solution finale par assemblage de performances.

Cette approche rompt avec l'approche classique, qui consiste à proposer une solution, avant de l'évaluer en la soumettant à des tests. Pour concevoir notre algorithme parallèle, nous avons jugé préférable de traiter chaque paramètre de l'algorithme séparément. Comme il existe plusieurs options pour chacun d'eux, celles-ci seront toutes étudiées, et leurs résultats comparés, avant de sélectionner les meilleures d'entre elles.

1.1. Type de stratégies de parallélisation adopté

L'étude des types de stratégies de parallélisation des métaheuristiques, qui a fait l'objet du chapitre précédent ,a montré que: Selon la métaheuristique et le problème auquel celle-ci est appliquée, un type de stratégies de parallélisation, peut lui être plus adapté qu'un autre. Néanmoins, les stratégies de parallélisation de métaheuristiques basées sur plusieurs processus coopératifs, semblent être les plus prometteuses. Par conséquent, nous avons choisi d'adopter ce type de stratégies, pour paralléliser notre algorithme BSO séquentiel.

1.2 Présentation de la démarche de parallélisation par assemblage de performances

Pour concevoir une recherche multiple coopérative, certains paramètres, d'une très grande importance doivent être déterminés. La détermination de ces paramètres, a un impact certain sur le comportement de la procédure parallèle, et la performance de la recherche. Ces paramètres, sont les suivants:

- La solution initiale.
- Le type de communication inter processus.
- Le moment où l'information est échangée.
- Les valeurs des paramètres empiriques.

Dans la suite de ce chapitre, nous consacrons chaque paragraphe, à l'étude de l'un des paramètres cités.

Pour chaque paramètre, une, ou plusieurs alternatives peuvent être envisagées. Nous devons choisir laquelle des alternatives sélectionner. Un tel choix ne peut pas être fait à la légère, et doit surtout, être justifié. Pour cela, nous avons adopté la démarche suivante: Pour

chaque paramètre, nous citons d'abord les alternatives (options) possibles. Nous concevons ensuite, autant d'algorithmes que d'alternatives. Ces algorithmes, doivent être similaires, et leur différence doit résider dans le choix de l'option, pour le paramètre en question.

Une fois cette étape effectuée, nous nous retrouvons avec plusieurs algorithmes, dont nous ignorons encore, les performances en termes de qualité, et de temps de calcul. De ce fait, nous ne serons pas en mesure de concevoir une solution finale. Cela ne sera possible, qu'après la comparaison des performances des différents algorithmes conçus pour chaque paramètre. Ces analyses nous aiderons à sélectionner la meilleure alternative, pour chaque paramètre de l'algorithme. Il sera enfin possible, de définir l'algorithme final de notre métaheuristique BSO parallèle.

2. Solution initiale

Dans les méthodes parallèles qui consistent en plusieurs recherches concurrentes dans l'espace de recherche; les différents processus peuvent démarrer de la même solution initiale, ou de solutions différentes.

Dans l'algorithme BSO séquentiel, une abeille nommée *BeeInit* est chargée de la recherche d'une solution initiale, qui représente le premier sommet de référence. A partir de ce sommet, la zone de recherche est déterminée, en utilisant essentiellement deux stratégies décrites dans le chapitre II. Ceci a pour effet, d'affecter aux abeilles, des sommets appartenant à la même zone de recherche.

Pour une meilleure couverture de l'espace de recherche, et dans le souci de se rapprocher du comportement des abeilles réelles, nous avons envisagé, de faire démarrer les abeilles (processus de recherche) à partir de solutions différentes, générées aléatoirement. Il est clair à ce moment, que nous avons de faibles chances d'affecter la même solution initiale, à deux processus différents. Ainsi, les recherches des abeilles, pourront se faire dans des zones différentes.

Les deux algorithmes suivants, se conforment à une architecture maître-esclaves. Leur parallélisme réside, dans le fait de lancer toutes les abeilles à la fois pour effectuer leurs recherches de manière concurrente. Dans l'algorithme séquentiel, ceci se fait dans une boucle. Une abeille n'effectue sa recherche qu'après le retour de la précédente.

2.1. Solution initiale unique

Dans ce premier algorithme, le maître détermine la zone de recherche à partir d'un sommet de référence, retourné par l'abeille initiale *BeeInit.*

Algorithme exécuté par le processus maître

Début
Sref ← Sommet retourné par *BeeInit* ;
Tant que non critère d'arrêt **faire**
Insérer *Sref* dans *Tabou ;*
Déterminer la région *SearchArea* référencée par *Sref ;*
Affecter chaque abeille à un sommet de *SearchArea ;*
Lancer les abeilles;
Attendre la fin de leurs recherches;
Choisir le nouveau sommet de référence *Sref* ;
Fait
Retourner la solution finale;
Fin

Algorithme exécuté par les processus esclaves (les abeilles)

Début
Effectuer une recherche à partir de son sommet ;
Retourner le résultat au maître;
Fin

2.1. Différentes solutions initiales

Cet algorithme est identique au précédent, à la seule différence, qu'à la première itération le maître générera pour chaque abeille, une solution initiale. La génération de ces solutions, peut se faire de manière aléatoire, comme elle peut être améliorée en utilisant une recherche locale.

Algorithme exécuté par le processus maître

Début

Générer pour chaque abeille une solution initiale;

Lancer les abeilles

Attendre la fin de leurs recherches;

Sref ← la meilleure solution retournée par les abeilles;

Tant que non critère d'arrêt **faire**

Insérer *Sref* dans *Tabou ;*

Déterminer la région *SearchArea* référencée par *Sref ;*

Affecter chaque abeille à un sommet de *SearchArea ;*

Lancer les abeilles;

Attendre la fin de leurs recherches;

Choisir le nouveau sommet de référence *Sref* ;

Fait

Retourner la solution finale;

Fin

Algorithme exécuté par les processus esclaves (les abeilles)

Début

Effectuer une recherche à partir de son sommet ;

Retourner le résultat au maître;

Fin

3. Mode de communication

Dans une recherche parallèle multi processus coopératifs, les processus s'échangent des informations. Ces échanges peuvent se faire de manière synchrone ou asynchrone. Pour déterminer lequel des ces deux modes de communication est le plus performant, deux algorithmes ont été conçus:

3.1. Synchrone

Cet algorithme est celui énoncé dans la section 2.a. A la fin de chaque itération, les abeilles retournent leurs solutions au processus maître. Celui-ci choisit la meilleure solution trouvée, détermine la zone de recherche référencée par cette solution, puis relance les abeilles pour effectuer une nouvelle recherche. Dans cet algorithme, c'est au maître que revient la

charge de la gestion de la liste *tabou* et de la table *Dance,* dans laquelle les abeilles stockent leurs solutions, à la fin de chaque itération.

3.2. Asynchrone

Dans la solution asynchrone, nous avons plusieurs abeilles (processus) possédant chacune une table *Dance,* dans laquelle elle sauvegarde les résultats de ses congénères. Chaque abeille possède également une liste *Tabou,* dans laquelle elle mémorise les sommets qu'elle a visités. Dans cet algorithme, il n'y a aucune forme de synchronisation. A la fin de sa recherche, chaque processus diffuse sa solution, puis détermine la meilleure solution parmi les celles retournées par les processus ayant terminés leur recherche à ce moment là, y compris la sienne. A aucun moment, les processus ne se mettent en attente, ils n'ont pas de moments d'inactivité. Cette méthode se rapproche davantage du comportement des abeilles réelles.

Dans cette architecture, il n'y a pas de maître. Chaque abeille détermine elle-même son sommet dans la zone de recherche, à partir du sommet de référence qu'elle a elle-même choisi précédemment. Selon son numéro, elle peut déterminer son sommet dans la zone de recherche de la manière suivante:

```
Début
    Si  Num < Flip
        h ← Num                          //Numéro de l'abeille;
        S ← Sref ;
        k ← 0 ;
        Répéter
          Inverser   S[Flip*k+h] ;
          k ← k+1 ;
        Jusqu'à  Flip*k+h  ≥ n
    Fin Si
    Si   Num >Flip-1 et  Num < 2*Flip
        h← Num [Flip];
        S ← Sref;
        k  ← 0 ;
        Répéter
          Inverser S[(n/Flip) * h +k];
          k := k+1 ;
        Jusqu'à  k ≥ n/Flip
    Fin Si
    Si  Num>2*Flip et Num< Nombre d'abeilles
        S ← Sref ;
        Inverser aléatoirement n/Flip variables de S ;
    Fin Si
Fin
```

L'avantage de cet algorithme, est la décentralisation de la détermination de la zone de recherche qui, désormais, n'est plus à la charge du maître. Les abeilles déterminent leur sommet, à chaque itération, de manière autonome. Il convient de noter que, pour un même sommet de référence, cet algorithme donne exactement les mêmes résultats que la procédure exécutée par le maître, dans les algorithmes précédents. Autrement dit, à partir d'un même sommet de référence, les deux algorithmes déterminent la même zone de recherche.

Algorithme exécuté par chaque processus (abeille)

Début

Sref← Solution retournée par BeeInit

Tant que non critère d'arrêt **faire**

Insérer *Sref* dans *Tabou ;*

Déterminer le sommet dans *SearchArea* référencée par *Sref ;*

Effectuer une recherche à partir de ce sommet;

Envoyer la solution trouvée aux autres processus;

Choisir le nouveau sommet de référence *Sref* ;

Fait

Fin

4. Le moment de l'échange d'informations

Dans une méthode parallèle basée sur plusieurs processus coopératifs, les processus communiquent en s'échangeant des informations. Le(s) moment(s) où, ces échanges ont lieu peut avoir un impact très important, aussi bien sur le résultat final de la recherche, que sur le temps de réponse. Pour notre algorithme, nous avons envisagé deux alternatives:

a) A chaque itération, chaque abeille communique sa solution à ses congénères. C'est l'approche qui a été adoptée jusqu'à présent.

b) A chaque itération, L'abeille ne communique sa solution que si celle-ci améliore la solution globale courante, c'est-à-dire, la meilleure solution trouvée jusqu'alors.

Comparée à la deuxième alternative, la première implique une communication relativement intensive entre les abeilles. Ceci nécessite plus de temps pour l'échange des solutions d'une part, et pour le choix de la meilleure solution d'autre part; vu que le nombre de solutions à traiter par chaque abeille est plus grand.

4.1. Première alternative

L'algorithme utilisé pour cette première alternative, n'est autre que celui énoncé dans la section (3.2). A la fin de chaque itération, chaque abeille communique à ses congénères, la solution trouvée.

4.2. Deuxième alternative

L'algorithme exécuté par chaque processus (abeille) est le suivant:

Début

Sref← Solution retournée par BeeInit

Tant que non critère d'arrêt **faire**

Insérer *Sref* dans *Tabou ;*

Déterminer le sommet dans *SearchArea* référencée par *Sref ;*

Effectuer une recherche à partir de ce sommet;

Si solution améliorée

Envoyer la solution trouvée aux autres processus;

Fin Si

Choisir le nouveau sommet de référence *Sref* ;

Fait

Fin

5. les valeurs des paramètres empiriques

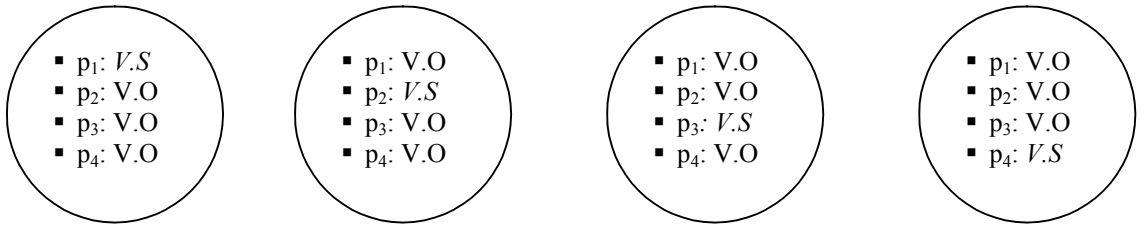
Dans la métaheuristique BSO, il y a des paramètres empiriques, qui ont une importance capitale, et un impact significatif sur les performances de l'algorithme. A savoir:

- L'heuristique utilisée dans la génération de la solution initiale.
- Le paramètre Flip.
- Le nombre de chances accordées à une zone de recherche.
- le nombre d'itérations dans la recherche locale effectuée par les abeilles.

Les processus qui participent à la recherche de la solution, peuvent assigner les mêmes valeurs à ces paramètres, comme ils peuvent leur assigner des valeurs différentes. Pour déterminer les valeurs optimales des paramètres empiriques, Il est nécessaire de faire des tests sur des instances de problèmes en utilisant plusieurs valeurs. C'est celle qui satisfait le plus grand nombre d'instances, qui est retenue comme valeur optimale. Cependant, les instances

non satisfaites par cette valeur optimale, peuvent l'être, en utilisant une des valeurs non retenues.

A partir de cette observation, nous avons pensé, qu'il serait judicieux de subdiviser l'essaim d'abeilles, en un nombre de sous essaims, égal au nombre de paramètres empiriques qui est de quatre (4). Pour chaque sous essaim E_i, nous affecterons à tous les paramètres leurs valeurs optimales (V.O), à l'exception d'un seul paramètre p_i. A ce paramètre, on affectera une valeur substitutive (V.S). Cette valeur peut être, par exemple, la deuxième meilleure valeur, ou alors, une valeur qui satisfait des instances insatisfaites, par la valeur optimale (voir schéma ci-dessous).



Essaim E_1 **Essaim E_2** **Essaim E_3** **Essaim E_4**

p_i: Paramètre i.
V.O: Valeur Optimale
V.S: Valeur substitutive.

Exemple: Si pour le paramètre Flip, la valeur optimale trouvée est cinq (V.O=5), et la seconde meilleure valeur est quatre (V.S=4), alors, trois essaims sur quatre assigneront la valeur 5 au paramètre Flip, et un seul essaim affectera à ce paramètre, la valeur quatre (04). De cette manière, nous pouvons espérer satisfaire un plus grand nombre d'instances du problème.

Chaque essaim appliquera la métaheuristique BSO parallèle synchrone (algorithme 2.1) et à la fin, la meilleure solution trouvée est retournée.

6. Conclusion

Après la conception des différents algorithmes, il est nécessaire de les exécuter sur des instances de problèmes, et d'analyser les résultats obtenus. Ces résultats, nous permettrons de choisir quelle alternative adopter, pour chacune des caractéristiques de l'algorithme. Les meilleures alternatives, seront regroupées dans un algorithme final.

Chapitre V

Implémentation et résultats expérimentaux

1. Introduction

Dans ce chapitre, nous présentons le résumé des nombreux tests effectués sur des instances du problème MAX-W-SAT. L'analyse des résultats obtenus, nous informera sur les performances des différents algorithmes. Ceci nous permettra, par conséquent, de déterminer pour chaque paramètre, quelle option permet d'atteindre les meilleurs résultats. Nous assemblerons ensuite les options sélectionnées, dans un algorithme final, qui sera la métaheuristique BSO parallèle.

L'absence d'une machine parallèle, nous a contraint à effectuer les tests sur une machine monoprocesseur (2 GHZ, 256 MO de RAM). L'implantation des algorithmes, a été faite en langage Java; qui permet d'implémenter des processus concurrents, en utilisant la classe Thread.

Les tests ont été effectués sur des instances MAX-W-SAT de la classe « Johnson », qui est très utilisée pour tester les performances des algorithmes. Les données MAX-W-SAT, sont contenues dans des benchmarks. Un benchmark est un fichier texte, qui commence par une ligne qui contient le nombre de variables, et le nombre de clauses de la donnée MAX-W-SAT. Chaque clause, est représentée sur une, ou plusieurs lignes. Elle donne comme informations, dans l'ordre suivant: Le nombre de variables dans la clause, le poids de la clause, et pour chaque variable, son index (littéral négatif, s'il est précédé d'un signe moins, et positif dans le cas contraire). Par exemple, l'instance suivante:

$$\begin{cases} x_1+x_3 & w(c1)=15. \\ \neg x_2+x_3+x_4 & w(c2)=10. \\ \neg x_1 & w(c3)=25. \end{cases}$$

est représentée comme suit :

```
4   3
2   15   1   3
3   10  -2   3   4
1   25  -1
```

La classe de benchmarks Johnson, comporte trois sous classes ayant toutes, le même nombre de variables, qui est de 100. Ces classes sont les suivantes :

- La *classe 1: {Jnh1,……., Jnh19},* dont le nombre de clauses est de 850, et la somme des poids de toutes les clauses est de 420925.
- La *classe 2: {Jnh201,……., Jnh220},* dont le nombre de clauses est de 800, la somme des poids de toutes les clauses est de 394238.
- La *classe 3: {Jnh301,…….Jnh310},* dont le nombre de clauses est de 900, et la somme des poids de toutes les clauses est de 444854.

Dans les sections suivantes, nous présentons les résultats des tests effectués, en utilisant les différents algorithmes. Mais avant cela, il faudra ajuster les valeurs des paramètres empiriques de BSO.

2. Détermination des valeurs des paramètres

Les paramètres empiriques ont une grande influence sur les performances des algorithmes. C'est la raison pour laquelle, nous avons consacré un temps, et des efforts considérables, pour déterminer les valeurs permettant d'obtenir les meilleurs résultats. Une valeur de paramètre est jugée bonne, si elle donne les meilleurs résultats en terme de qualité comme premier critère, et en terme de temps de calcul comme second critère. Les paramètres de métaheuristique BSO sont :

- L'heuristique par laquelle la solution initiale est générée (Heuristic)
- Le paramètre Flip.
- Le nombre d'itérations de la recherche locale effectuée par chaque abeille (IterLs).
- Le nombre d'abeilles (Nbees).
- Le nombre de chances accordées à une zone d'exploitation (MaxChances).
- Le nombre d'itérations de l'algorithme BSO (MaxBsoIter).

En fait, le même travail a été réalisé pour l'algorithme BSO séquentiel dans le cadre de notre projet de fin d'études. Les valeurs optimales obtenues étaient:

- Heuristic: John2a
- Flip : 5.
- Nbees : 10.
- MaxChances = 3
- IterLs: 15
- MaxBsoIter : 35.

De nouveaux tests ont été faits, dans le but de vérifier, si les valeurs optimales pour l'algorithme parallèle, sont les mêmes que pour l'algorithme séquentiel, cités ci-dessus.

Il convient de préciser, que tous les tests ont été effectués, en exécutant le premier algorithme (2.1 du chapitre IV).

Hormis le paramètre sur lequel portent les tests, et auquel nous attribuons différentes valeurs, nous affectons aux paramètres, les valeurs optimales de l'algorithme séquentiel.

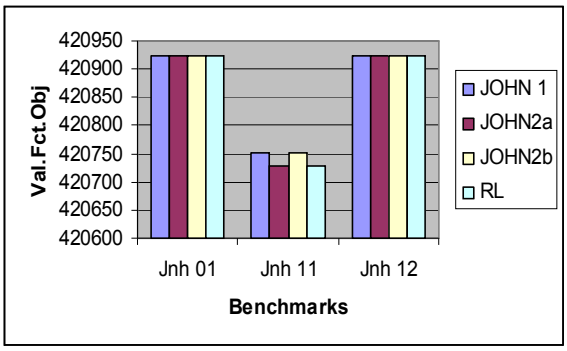
2.1. L'heuristique

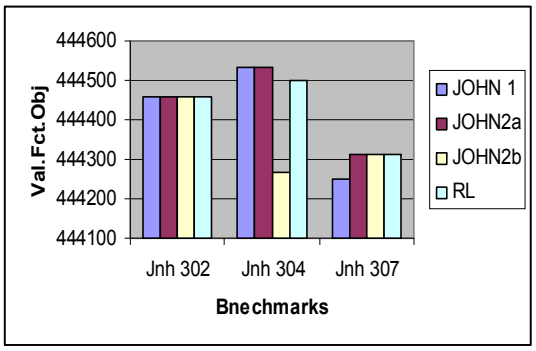
Nous avons comparé les résultats obtenus, en utilisant les différentes méthodes par lesquelles la solution initiale peut être générée. À savoir: John1, John2a, John2b et recherche locale (RL).

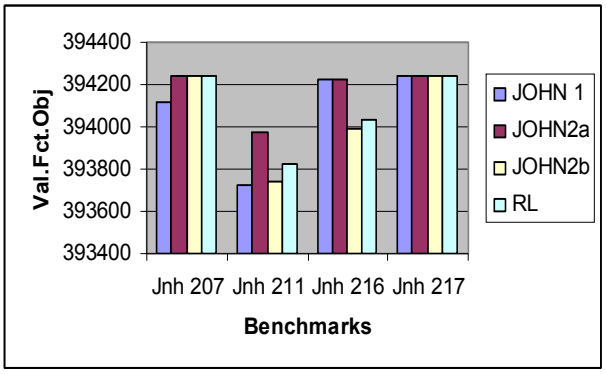
Les meilleurs résultats (valeurs de la fonction objectif), sont indiqués en gras dans le tableau ci-dessous. Un résultat est jugé meilleur qu'un autre, s'il lui est supérieur. Si deux résultats sont égaux, le meilleur est c'est celui qui a été obtenu en un temps plus court.

Bench	Val.opt	JOHN 1	Temps (s)	JOHN2 a	Temps (s)	JOHN2 b	Temps (s)	RL	Temps (s)
Jnh 01	420925	420925	18.126	420925	40.988	**420925**	12.829	420925	24.707
Jnh 11	420753	**420753**	35.632	420728	107.704	420753	81.597	420728	109.337
Jnh 12	420925	420925	33.237	**420925**	9.323	420925	9.484	420925	72.754
Jnh 207	394238	394118	99.343	**394238**	14.58	394238	66.725	394238	46.677
Jnh 211	393979	393723	101.486	**393979**	91.01	393742	98.631	393825	98.772
Jnh 216	394226	394226	32.347	**394226**	11.436	393993	103.078	394037	100.424
Jnh 217	394238	394238	9.323	394238	3.258	394238	3.125	**394238**	2.884
Jnh 302	444459	**444459**	33.448	444459	48.931	444459	40.198	444459	43.422
Jnh 304	444533	**444533**	33.859	444533	116.818	444266	112.011	444502	113.904
Jnh 307	444314	444249	139.13	444314	23.123	444314	105.692	**444314**	10.415

Table1. Comparaison des résultats des différentes méthodes.







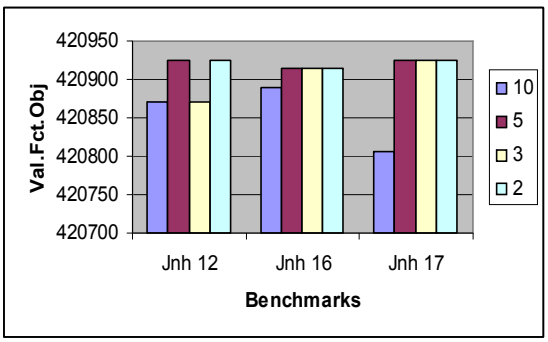
Nous remarquons que les 4 méthodes donnent de bons résultats, avec un léger avantage pour l'heuristique John2a. Nous pouvons constater aussi, que toutes les instances ont été satisfaites par au moins une méthode, et que même si John2a donne globalement les meilleurs résultats, elle ne satisfait pas certaines instances qui ont pu être satisfaites en utilisant une autre méthode. Cette observation, confirme l'hypothèse selon laquelle, la génération de différentes solutions initiales, pourrait donner de meilleurs résultats.

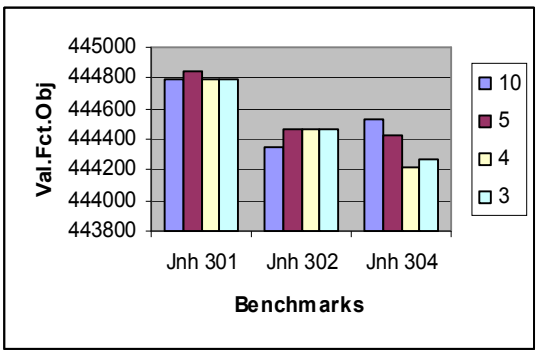
2.2. Le Flip

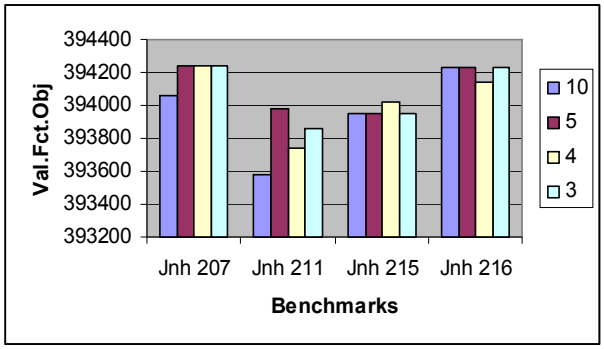
La valeur du paramètre Flip est cruciale dans la métaheuristique BSO. En effet, c'est à partir de cette valeur, que les sommets de la zone de recherche sont déterminés. Rappelons que Flip indique le nombre de variables à inverser (nombre de variables* 1/FLIP), à partir du sommet de référence. Des tests ont été effectués, en utilisant les valeurs 10, 5, 4, et 3.

Bench	Val.Opt	10	T(s)	5	T(s)	4	T(s)	3	T(s)
Jnh 12	420925	420871	83.049	**420925**	9.354	420871	114.535	420925	68.158
Jnh 16	420919	420889	88.858	**420914**	110.018	420914	120.173	420914	130.948
Jnh 17	420925	420806	81.677	**420925**	6.75	420925	70.051	420925	22.382
Jnh 207	394238	394065	79.825	**394238**	14.621	394238	23.214	394238	71.813
Jnh 211	393979	393580	78.072	**393979**	91.471	393742	107.294	393863	120.203
Jnh 215	394150	393951	79.304	393951	99.824	**394019**	108.216	393955	118.28
Jnh 216	394226	394226	25.137	**394226**	11.627	394144	110.669	394226	24.375
Jnh 301	444854	444790	92.022	**444842**	116.468	444790	125.36	444790	137.458
Jnh 302	444459	444351	94.085	444459	48.94	**444459**	11.406	444459	61.819
Jnh 304	444533	**444533**	10.616	444427	112.732	444211	123.428	444266	135.084

Table2: Comparaison des résultats obtenus avec les différentes valeurs de Flip.







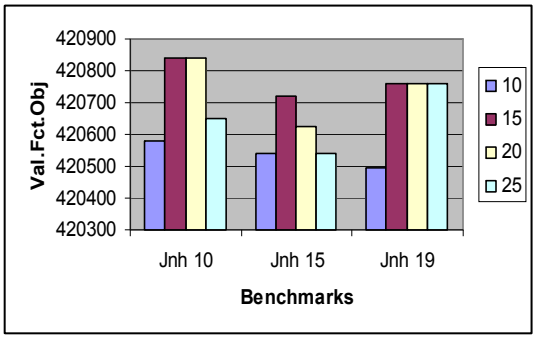
L'analyse des résultats montre bien que les résultats obtenus avec la valeur de Flip=5 dépassent ceux obtenus avec les autres valeurs, c'est donc la valeur optimale. Cependant, comme pour l'heuristique, certaines instances comme John 215 et John304, n'ont pu être satisfaites qu'avec d'autres valeurs.

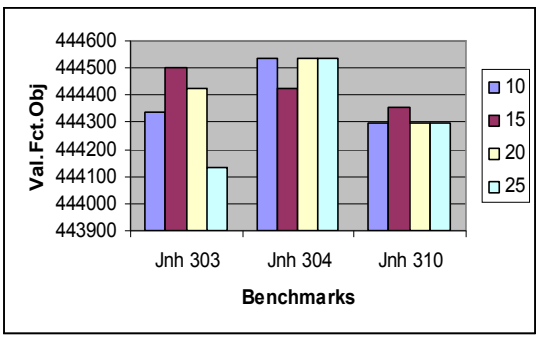
2.3. IterLs

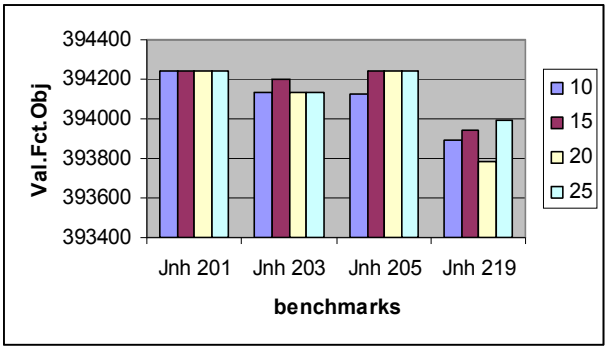
Les abeilles effectuent une recherche locale à partir d'un sommet de la zone de recherche. Le nombre d'itérations de cette recherche (IterLs), est un paramètre important. Dans nos tests, nous avons comparés les valeurs 10, 15, 20 et 25.

Bench	Val Opt	10	T(s)	15	T(s)	20	T(s)	25	T(s)
Jnh 10	420840	420581	87.086	**420840**	33.948	420840	70.221	420649	147.182
Jnh 15	420719	420538	87.716	**420719**	77.091	420625	129.416	420538	150.366
Jnh 19	420759	420494	85.693	**420759**	97.78	420759	119.102	420759	148.053
Jnh 201	394238	394238	8.963	**394238**	3.255	394238	3.665	394238	6.539
Jnh 203	394199	394135	81.488	**394199**	53.117	394135	123.387	394135	141.794
Jnh 205	394238	394127	80.025	394238	5.968	**394238**	3.665	394238	4.386
Jnh 219	394156	393893	105.171	**393942**	103.599	393782	122.526	393993	145.44
Jnh 303	444503	444338	92.073	**444503**	63.471	444423	136.576	444135	159.299
Jnh 304	444533	**444533**	11.346	444427	113.627	444533	32.096	444533	33.117
Jnh 310	444391	444294	93.515	**444353**	114.585	444294	136.547	444294	160.731

Table3: Comparaison des résultats obtenus pour les différentes valeurs de Flip







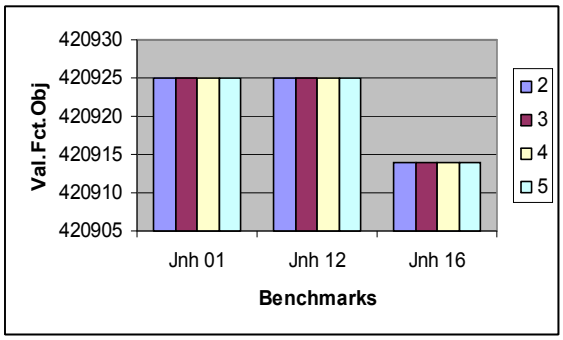
Les résultats des tests montrent que la valeur qui offre le meilleure compromis résultat/ temps de calcul, est la valeur 15, qui est donc, la valeur optimale.

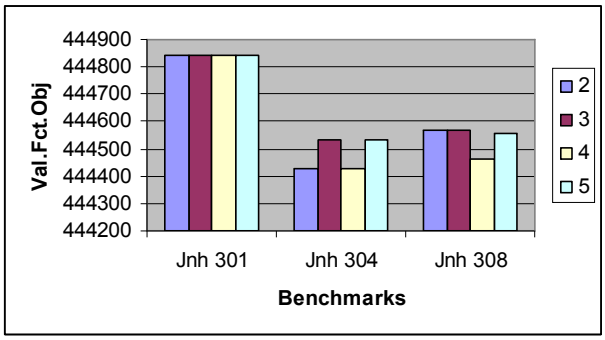
2.4. MaxChances

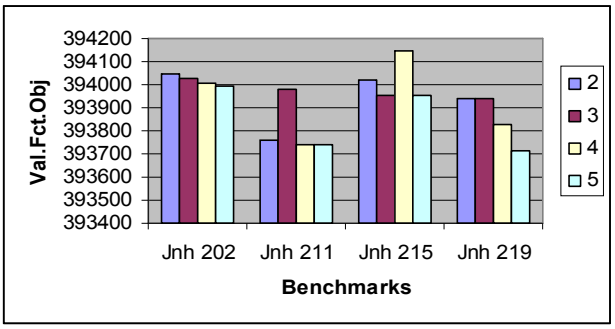
Le nombre maximal de chances accordé à une zone de recherche, peut avoir un impact très important sur les performances de l'algorithme. En effet, une valeur trop petite, pourrait nous détourner trop vite d'une zone prometteuse; alors qu'une valeur trop grande, pourrait nous piéger dans un optimum local. La valeur optimale a été trouvée, en effectuant des tests avec les valeurs 2, 3, 4 et 5.

Bench	Val.Opt	2	T(s)	3	T(s)	4	T(s)	5	T(s)
Jnh 01	420925	**420925**	22.002.	420925	40.739	420925	30.825	420925	24.705
Jnh 12	420925	420925	9.484	420925	9.323	**420925**	9.183	420925	9.264
Jnh 16	420919	420914	109.567	**420914**	109.408	420914	109.698	420914	128.765
Jnh 202	394170	**394044**	100.985	394029	101.927	394009	100.385	393995	100.945
Jnh 211	393979	393757	99.964	**393979**	91.311	393742	122.727	393742	98.582
Jnh 215	394150	394019	99.984	393951	99.724	**394150**	45.445	393951	98.411
Jnh 219	394156	393942	103.349	**393942**	103.299	393828	102.367	393713	102.438
Jnh 301	444854	444842	117.769	444842	116.257	**444842**	116.027	444842	117.229
Jnh 304	444533	444427	113.954	444533	112.872	444427	112.762	**444533**	64.133
Jnh 308	444724	444568	116.818	**444568**	116.407	444460	117.218	444556	117.8

Table4. Comparaison des résultats obtenus avec les valeurs 2, 3, 4, et 5.







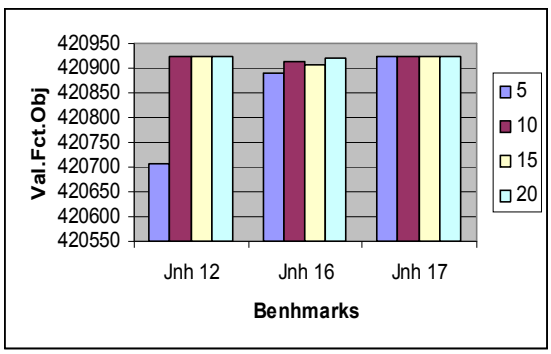
Les résultats montrent que la valeur 3 est la plus convenable. Il se trouve, en effet, que cette valeur n'est pas trop petite, ce qui permet une bonne intensification de la recherche dans la zone de recherche. Elle n'est pas trop grande non plus, ce qui permet de s'éloigner d'une zone, lorsque celle-ci ne présente plus de bons résultats, ce qui permet une bonne diversification de la recherche.

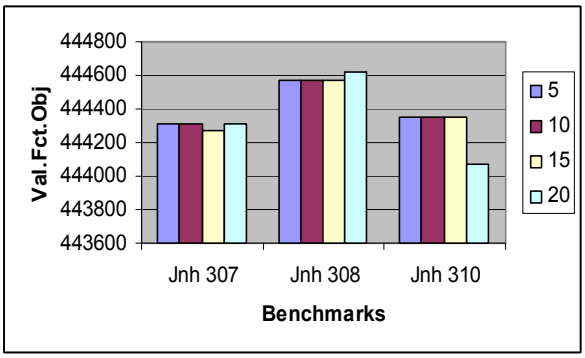
2.5. Nbees

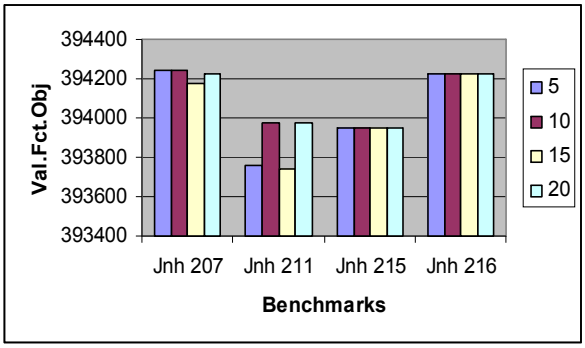
Le nombre d'abeilles chargées de la recherche de la solution optimale, dans une zone de recherche donnée, est égal au nombre de sommets représentants cette zone. Il est donc évident, que plus ce nombre est grand, mieux cette zone est couverte. Mais cela peut se faire au détriment du temps de calcul, qui peut devenir prohibitif. Pour déterminer la valeur optimale de ce paramètre, nous avons comparé les valeurs 5, 10, 15, et 20.

Benchmark	Val.Opt	5	T(s)	10	T(s)	15	T(s)	20	T(s)
Jnh 12	420925	420706	53.757	**420925**	9.323	420925	13.75	420925	128.315
Jnh 16	420919	420889	55.951	420914	109.868	420906	164.597	**420919**	193.999
Jnh 17	420925	420925	10.735	**420925**	6.72	420925	9.834	420925	12.107
Jnh 207	394238	394238	47.418	**394238**	14.801	394178	152.439	394229	202.481
Jnh 211	393979	393757	51.815	**393979**	91.311	393742	147.412	393979	202.481
Jnh 215	394150	**393951**	49.891	393951	99.724	393951	150.777	393951	200.298
Jnh 216	394226	394226	36.883	**394226**	11.436	394226	17.044	394226	17.665
Jnh 307	444314	444314	46.427	**444314**	22.873	444274	167.551	444314	25.927
Jnh 308	444724	444568	59.345	444568	116.407	444568	174.22	**444621**	233.486
Jnh 310	444391	**444353**	59.436	444353	114.585	444353	173.249	444068	229.139

Table5. Comparaison des résultats obtenus avec les valeurs 5, 10,15 et 20







En observant les résultats obtenus, nous pouvons constater que, certaines instances ne sont satisfaites, qu'en utilisant un grand nombre d'abeilles comme John11, John16 ou John202. Or, le temps de réponse est relativement grand.

Pour un bon compromis entre la qualité de la solution obtenue, et le temps de réponse, nous choisissons la valeur 10. Celle-ci satisfait un bon nombre de Benchmarks en un temps appréciable. Nous ne négligeons pas pour autant, les instances non satisfaites avec cette valeur, car celles-ci pourraient l'être en variant un autre paramètre, tel que Flip, MaxChances ou IterLs.

2.6. Conclusion

Les tests effectués,ont permis de confirmer les valeurs optimales obtenues pour la métaheuristique séquentielle à savoir: Heuristic: John2a, Flip : 5,Nbees : 10, MaxChances :3, IterLs: 15.Ce sont donc, ces valeurs là qui seront appliquées dans la suite de nos tests. Ceux-ci, concerneront les différents algorithmes conçus dans le but de déterminer les meilleures alternatives pour les différents paramètres, et les assembler dans l'algorithme final.

3. Comparaison des résultats et sélection des meilleures alternatives pour et l'élaboration de la solution finale

Après avoir fixé les valeurs optimales pour les différents paramètres de la métaheuristique, nous passons à la comparaison des performances des algorithmes conçus, dans le but de déterminer la meilleure option à choisir concernant chacune des caractéristiques de la métaheuristique parallèle. A savoir: la solution initiale, le mode de communication entre les processus, le moment de l'échange des informations et enfin la variation des valeurs des paramètres.

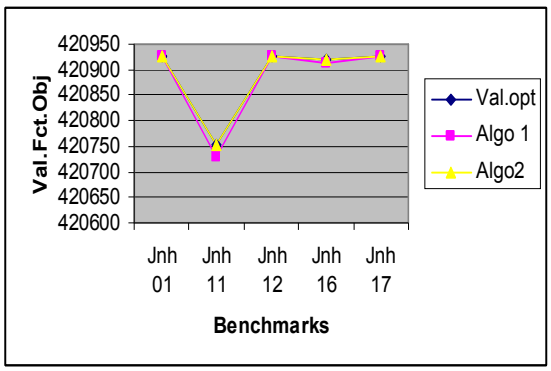
Pour chaque caractéristique, nous avons effectué des tests sur la totalité des benchmarks, puis nous avons choisi une vingtaine d'entre eux pour les représenter dans un tableau. Les meilleurs résultats obtenus sont indiqués en gras.

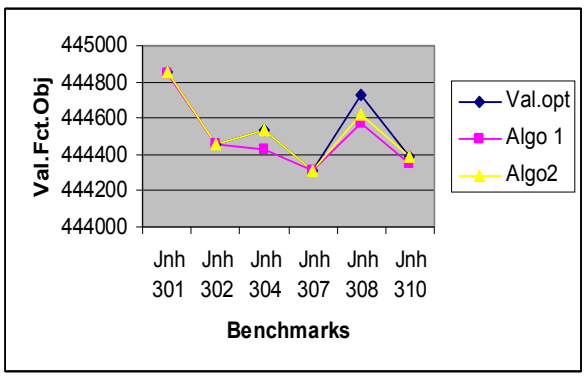
3.1. Solution initiale

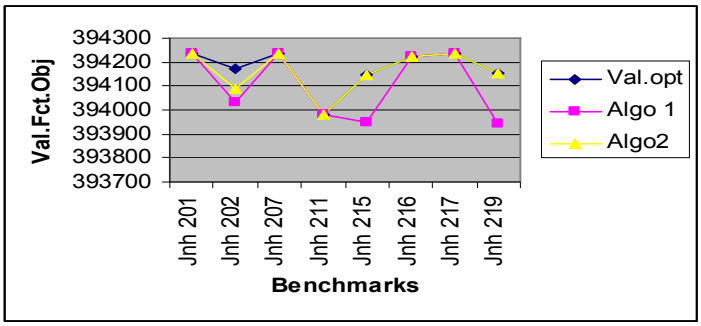
La solution initiale peut être unique, et donc la même pour tous les processus, ou alors différente, et dans ce cas, chaque processus démarre d'une solution différente. L'algorithme1 génère une seule solution via l'heuristique john2a, alors que l'algorithme2 génère pour chaque processus une solution initiale différente via une recherche locale.

Benchmark	Val.opt	Algo 1	T(s)	Algo2	T(s)
Jnh 01	420925	420925	42.0	**420925**	6.91
Jnh 11	420753	420728	108.045	**420753**	13.53
Jnh 12	420925	**420925**	9.193	420925	10.104
Jnh 16	420919	420914	109.126	**420919**	7.27
Jnh 17	420925	420925	6.259	**420925**	5.949
Jnh 201	394238	394238	3.074	**394238**	2.944
Jnh 202	394170	394029	100.956	**394091**	103.009
Jnh 207	394238	394238	14.51	**394238**	10.024
Jnh 211	393979	393979	90.68	**393979**	9.764
Jnh 215	394150	393951	99.352	**394150**	11.897
Jnh 216	394226	394226	11.377	**394226**	6.69
Jnh 217	394238	**394238**	3.204	394238	3.435
Jnh 219	394156	393942	102.648	**394156**	47.137
Jnh 220	394238	394155	99.013	**394238**	6.61
Jnh 301	444854	444842	115.446	**444854**	14.15
Jnh 302	444459	444459	48.57	**444459**	7.521
Jnh 304	444533	444427	112.562	**444533**	10.916
Jnh 307	444314	444314	22.603	**444314**	13.389
Jnh 308	444724	444568	115.667	444621	120.153
Jnh 310	444391	444353	113.954	**444391**	19.789

Table6. Comparaison entre les performances de l'algo1 (solution initiale unique) et de l'algo2 (Différentes solutions initiales)







Analyse

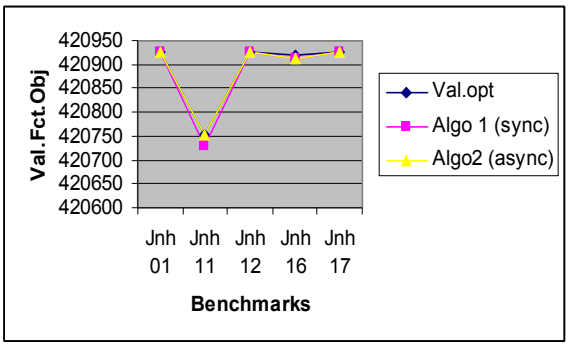
Les tests montrent clairement l'avantage qu'offre le deuxième algorithme par rapport au premier, en satisfaisant un plus grand nombre de benchmarks en un temps plus court. Ceci s'explique par le fait qu'une seule solution initiale, référence une seule zone de recherche dont les sommets sont affectés aux abeilles. Tandis que la génération de plusieurs solutions initiales offre une plus grande diversité des zones de recherche explorées par les abeilles, ce qui assure une meilleure couverture de l'espace de recherche.

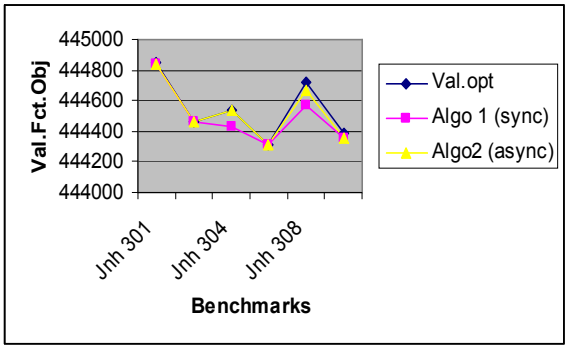
3.2. Mode de communication (synchrone/asynchrone)

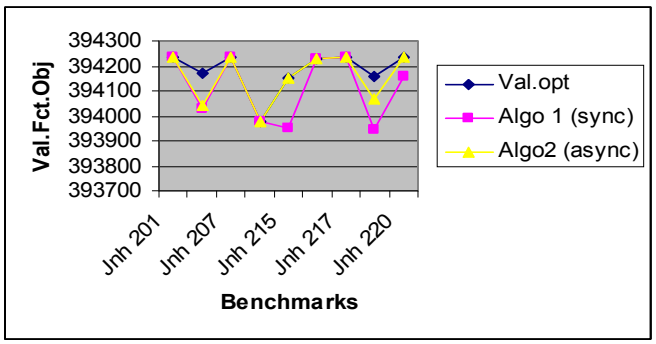
Pour le mode de communication entre les processus, nous avons le choix entre le mode synchrone et le mode asynchrone. Dans le tableau ci-dessous, nous présentons les résultats des deux algorithmes.

Benchmark	Val.opt	Algo 1 (sync)	T(s)	Algo2 (async)	T(s)
Jnh 01	420925	420925	42.0	**420925**	8.702
Jnh 11	420753	420728	108.045	**420753**	60.648
Jnh 12	420925	**420925**	9.193	420925	9.683
Jnh 16	420919	**420914**	109.126	420914	112.792
Jnh 17	420925	**420925**	6.259	420925	9.243
Jnh 201	394238	**394238**	3.074	394238	5.338
Jnh 202	394170	394029	100.956	**394044**	106.132
Jnh 207	394238	**394238**	14.51	394238	17.325
Jnh 211	393979	393979	90.68	**393979**	32.306
Jnh 215	394150	393951	99.352	**394150**	24.335
Jnh 216	394226	394226	11.377	**394226**	10.836
Jnh 217	394238	**394238**	3.204	394238	6.449
Jnh 219	394156	393942	102.648	**394070**	107.885
Jnh 220	394238	394155	99.013	**394238**	68.018
Jnh 301	444854	**444842**	115.446	444842	121.444
Jnh 302	444459	444459	48.57	**444459**	23.714
Jnh 304	444533	444427	112.562	**444533**	40.237
Jnh 307	444314	444314	22.603	**444314**	15.872
Jnh 308	444724	444568	115.667	**444664**	123.537
Jnh 310	444391	**444353**	113.954	444353	117.409

Table7: Comparaison des résultats des algorithmes synchrone, et asynchrone







Analyse

A priori, on pourrait penser que l'algorithme synchrone est le plus performant, car il présente l'avantage d'offrir à chaque itération, et à chaque processus, la meilleure solution trouvée au cours l'itération précédente; alors que dans l'algorithme asynchrone, à la fin de sa recherche, chaque processus choisit la meilleure solution trouvée par les processus qui ont terminé leurs recherches à cet instant, donc, pas forcément tous. Or, les résultats des tests, ont montré que l'algorithme asynchrone donne les meilleurs résultats. Ceci s'explique, par le fait que celui-ci, assure une meilleure diversité de la recherche. En effet, à chaque itération, les processus ne choisissent pas tous le même sommet de référence, et par conséquent, n'effectuent pas tous leur recherche dans la même zone à l'itération suivante.

L'algorithme qui donne les meilleurs résultats, est finalement, celui qui se rapproche le plus du comportement des abeilles réelles. En effet, celles-ci, une fois leur récolte effectuée, retournent à la ruche, et suivent l'abeille qui effectue la danse la plus vigoureuse, parmi celles qui sont dans la ruche, sans attendre le retour de toutes les abeilles.

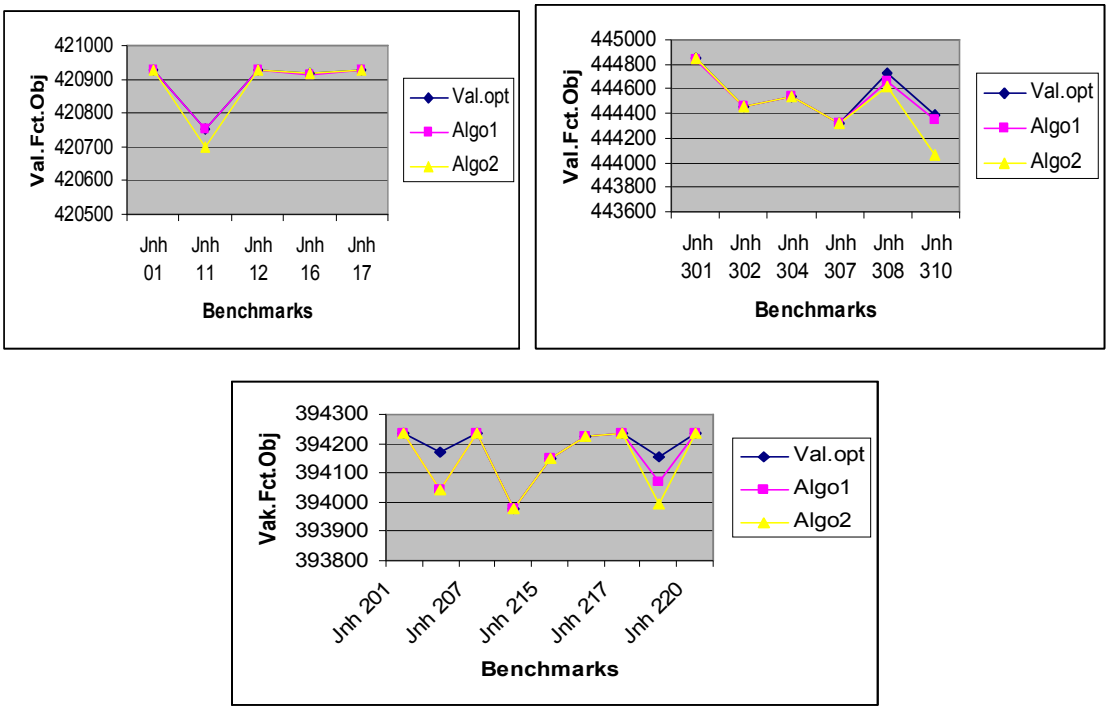
3.3. Moment de l'échange d'informations

Pour choisir à quel moment les processus doivent s'échanger leurs résultats, nous avons conçus deux algorithmes similaires décrits dans le chapitre précédent. Dans le premier algorithme, les processus communiquent entre eux à chaque fin d'itération, alors que dans le

deuxième, un processus ne diffuse sa solution que ci celle-ci est meilleure que la solution globale actuelle.

Benchmark	Val.opt	Algo31	T(s)	Algo4	T(s)
Jnh 01	420925	**420925**	8.702	420925	8.783
Jnh 11	420753	**420753**	60.648	420701	109.007
Jnh 12	420925	**420925**	9.683	420925	18.597
Jnh 16	420919	420914	112.792	**420919**	24.626
Jnh 17	420925	**420925**	9.243	420925	9.433
Jnh 201	394238	**394238**	5.338	394238	5.548
Jnh 202	394170	394044	106.132	**394044**	100.825
Jnh 207	394238	**394238**	17.325	394238	42.431
Jnh 211	393979	**393979**	32.306	393979	62.179
Jnh 215	394150	**394150**	24.335	394150	92.603
Jnh 216	394226	**394226**	10.836	394226	13.699
Jnh 217	394238	**394238**	6.449	394238	6.51
Jnh 219	394156	**394070**	107.885	393993	102.337
Jnh 220	394238	**394238**	68.018	394238	96.048
Jnh 301	444854	444842	121.444	**444854**	104.0
Jnh 302	444459	**444459**	23.714	444459	25.478
Jnh 304	444533	444533	40.237	**444533**	18.817
Jnh 307	444314	**444314**	15.872	444314	25.617
Jnh 308	444724	**444664**	123.537	444621	121.144
Jnh 310	444391	444353	117.409	**444060**	116.497

Table8: Comparaison des résultats des deus algorithmes



Analyse

Les résultats obtenus, favorisent le premier algorithme par rapport au second. Même si ce dernier a réussi à atteindre l'optimum pour certains benchmarks qui n'ont pas été satisfaits avec le premier algorithme, comme john16 et john301, globalement, les meilleurs résultats

et/ou les temps les plus courts, sont obtenus par le premier algorithme. Il semble donc, qu'un échange des résultats à chaque itération, aide les abeilles à atteindre l'optimum plus rapidement.

3.4 Les valeurs des paramètres empiriques:

Au lieu de donner des valeurs fixes aux paramètres de la métaheuristique, nous avons créés 4 sous essaims d'abeilles, adoptant chacun, l'approche maître esclave, et appliquant l'algorithme (2.1 du chapitre IV):

- *SwarmHeuristic.*
- *SwarmFlip.*
- *SwarmChance.*
- *SwarmIterLs*.

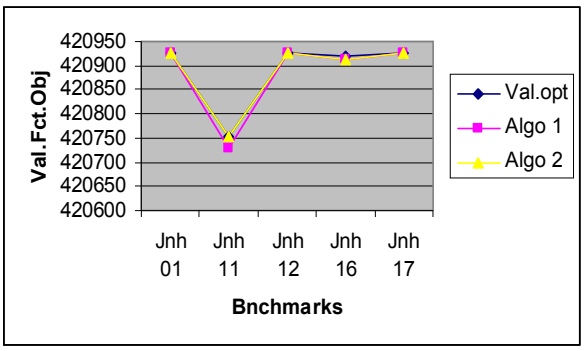
Dans chaque essaim *SwarmX*, nous avons affecté à chaque paramètre la valeur optimale, à l'excéption du paramètre *X*, auquel nous avons assigné une valeur substitutive.

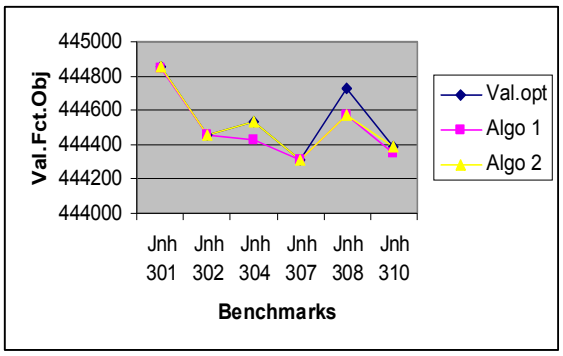
- Dans SwarmHeuristic nous avons utilisé la Recherche locale pour la génération de la solution initiale au lieu de l'heuristique John2a.
- Dans SwarmFlip, la valeur attribuée au flip est 4.
- Dans SwarmChance, nous accordons 4 chances à chaque zone de recherche au lieu de 3.
- Enfin, dans SwarmIterLs, chaque abeille effectue 20 itérations dans sa recherche locale au lieu de 15 seulement.

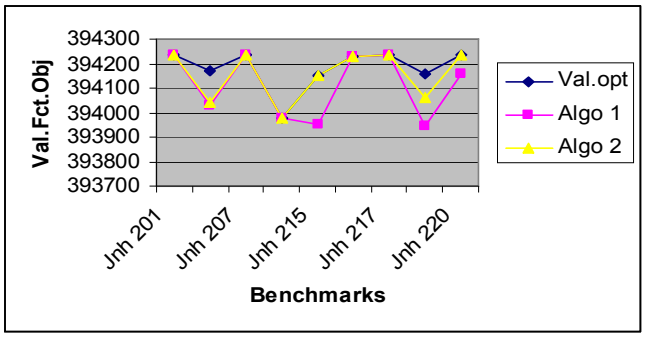
Nous avons comparé les résultats de cet algorithme avec ceux de l'algorithme (IV.2.1)

Benchmark	Val.opt	Algo1 (IV.2.1)	T(s)	Algo2	T(s)
Jnh 01	420925	420925	42.0	**420925**	12.738
Jnh 11	420753	420728	108.045	**420753**	90.861
Jnh 12	420925	**420925**	9.193	420925	19.518
Jnh 16	420919	**420914**	109.126	420914	114.425
Jnh 17	420925	**420925**	6.259	420925	6.439
Jnh 201	394238	394238	3.074	**394238**	2.754
Jnh 202	394170	394029	100.956	**394044**	104.971
Jnh 207	394238	**394238**	14.51	394238	88.106
Jnh 211	393979	393979	90.68	**393979**	20.35
Jnh 215	394150	393951	99.352	**394150**	14.04
Jnh 216	394226	**394226**	11.377	394226	36.833
Jnh 217	394238	394238	3.204	**394238**	3.044
Jnh 219	394156	393942	102.648	**394059**	105.292
Jnh 220	394238	394155	99.013	**394238**	57.503
Jnh 301	444854	444842	115.446	**444854**	15.492
Jnh 302	444459	444459	48.57	**444459**	3.926
Jnh 304	444533	444427	112.562	**444533**	13.533
Jnh 307	444314	**444314**	22.603	444314	23.284
Jnh 308	444724	**444568**	115.667	444568	120.203
Jnh 310	444391	444353	113.954	**444391**	48.56

Table9: Comparaison des résultats des deux algorithmes.







Analyse

Nous constatons qu'en variant les valeurs des paramètres empiriques, selon la technique décrite précédemment, les résultats de l'algorithme, sont nettement supérieurs à ceux de l'algorithme 1. Ceci était prévisible, vu qu'au cours des tests pour la détermination des valeurs des paramètres, nous avons constaté que certaines valeurs, dites non optimales, pouvaient donner de très bon résultats pour certaines instances, en atteignant ,souvent, la valeur optimale pour la fonction objectif. En variant les valeurs des paramètres, nous avons plus de chance de satisfaire un plus grand nombre de benchmarks, et cela a été confirmé par les différents tests.

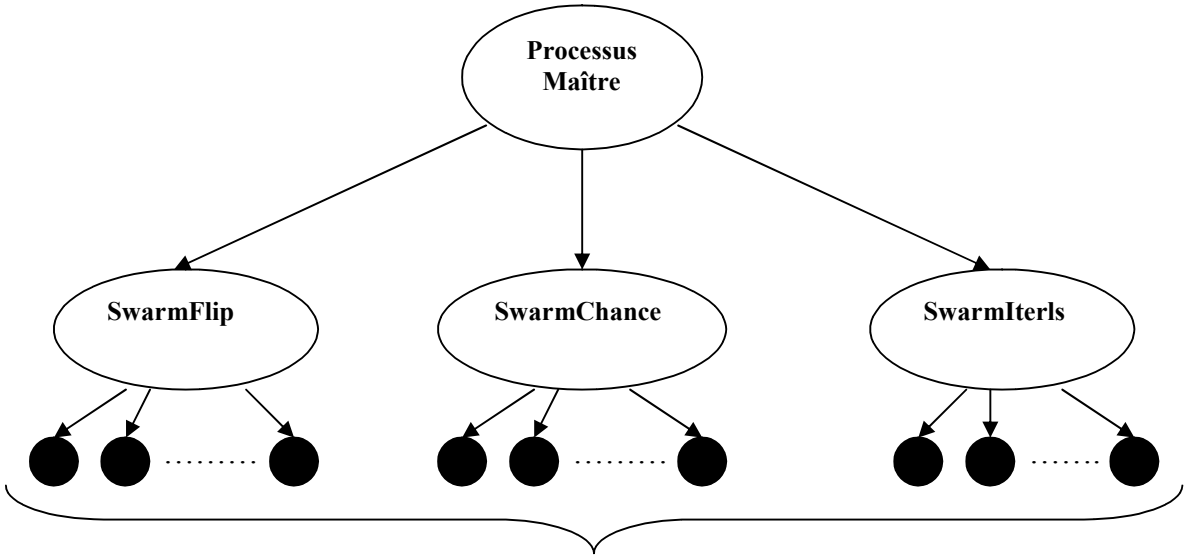
4. BSO Parallèle

Après l'analyse des tests effectués en appliquant les différents algorithmes aux benchmarks Johnson, nous avons abouti à la conception de notre métaheuristique BSO parallèle, dans laquelle:

- L'essaim est subdivisé en 3 sous essaims : Swarmflip, SwarmChance et SwarmIterLs.
- Une hiérarchie à deux niveaux est adoptée. Un processus maître s'occupe de l'affectation des paramètres aux différents processus Swarm.
- Chaque processus esclave génère sa propre solution initiale, gère sa propre table *Dance* et sa propre liste *Tabou*.

- Les processus des trois essaims communiquent entre eux, de manière asynchrone, sans passer par leurs processus Swarm.
- A chaque itération, chaque processus diffuse la meilleure solution qu'il a obtenue.*

Le schéma suivant, illustre l'architecture du système conçu.



L'algorithme de la métaheuristique BSO parallèle

Les algorithmes exécutés par le processus maître, les processus Swarm et les processus esclaves sont les suivants.

L'algorithme exécuté par le processus Maître

Début

Affecter à chaque essaim ses valeurs des paramètres empiriques;
Lancer les processus SwarmFlip, SwarmChance, SwarmIterls;
Attendre la fin des recherches;
Retourner la solution;

Fin

L'algorithme exécuté par les processus Swarm

Début

Lancer les abeilles;

Tant que non critère d'arrêt **Faire**

Si une abeille améliore la meilleure solution globale courante **alors**

Envoyer la solution au maître pour mettre à jour la solution globale;

Fin Si

Fait

Fin

L'algorithme exécuté par les processus esclaves

Début

Sref← Solution retournée par BeeInit

Tant que non critère d'arrêt **faire**

Insérer *Sref* dans *Tabou ;*

Déterminer le sommet dans *SearchArea* référencée par *Sref ;*

Effectuer une recherche à partir de ce sommet;

Envoyer la solution trouvée au maître et aux autres processus;

Choisir le nouveau sommet de référence *Sref* ;

Fait

Fin

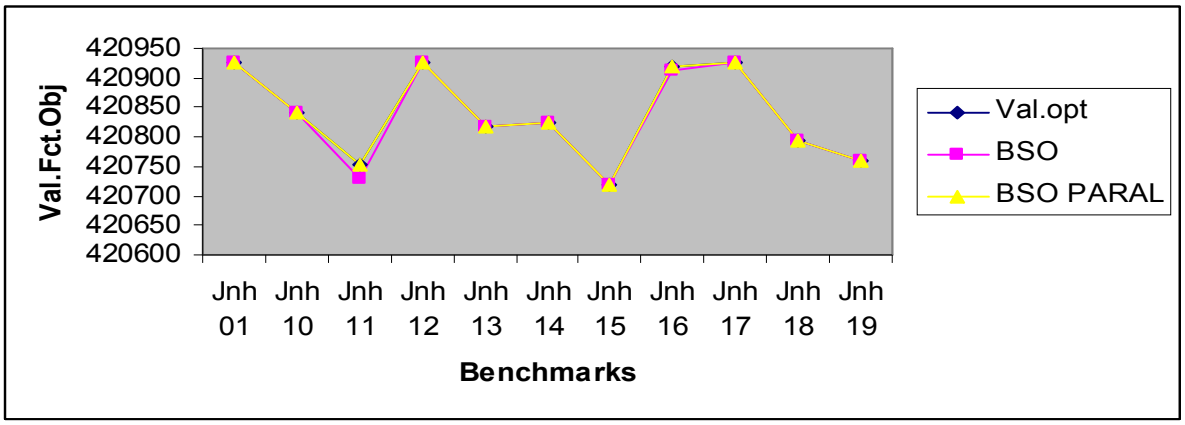
5. Résultats empiriques

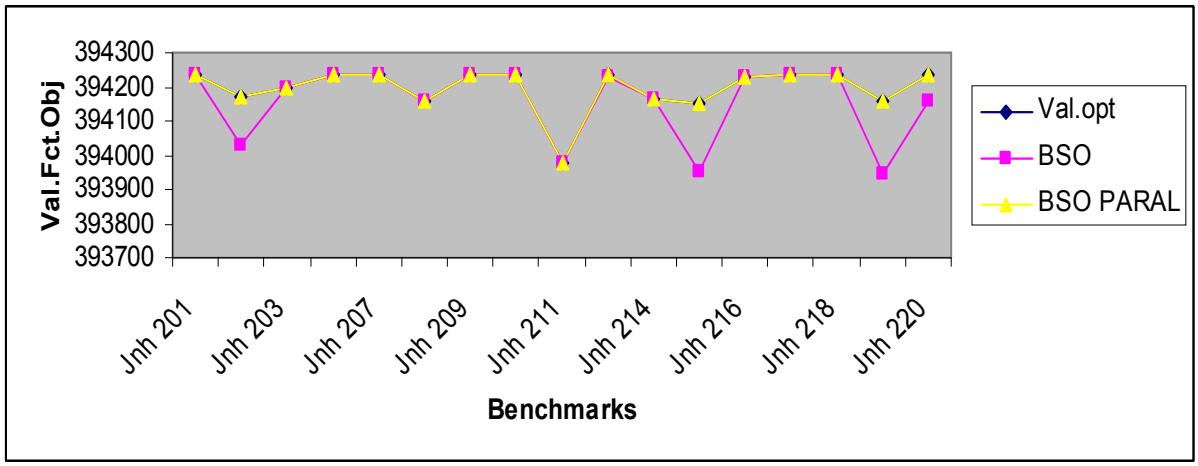
5.1 Comparaison entre BSO séquentielle et BSO parallèle

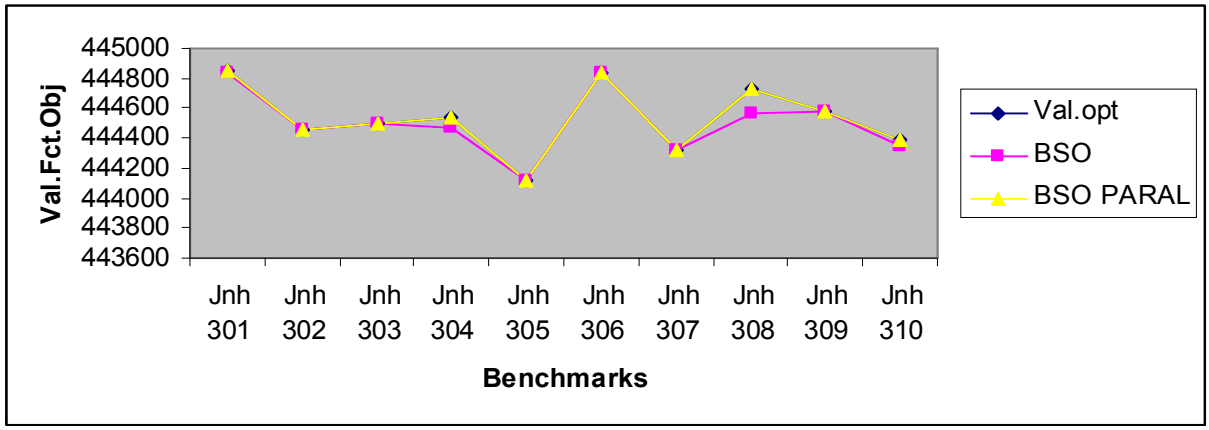
Les résultats obtenus par l'algorithme séquentiel et l'algorithme parallèle, montrent que les performances de la BSO Parallèle en terme de valeur de la fonction objective et en terme de temps dépassent celles de l'algorithme séquentiel.

- ***Comparaison en terme de qualité de la solution***

Les graphiques ci-dessous, montrent une comparaison entre les résultats de BSO séquentielle et ceux de BSO parallèle. Les benchmarks sont regroupés par classes.

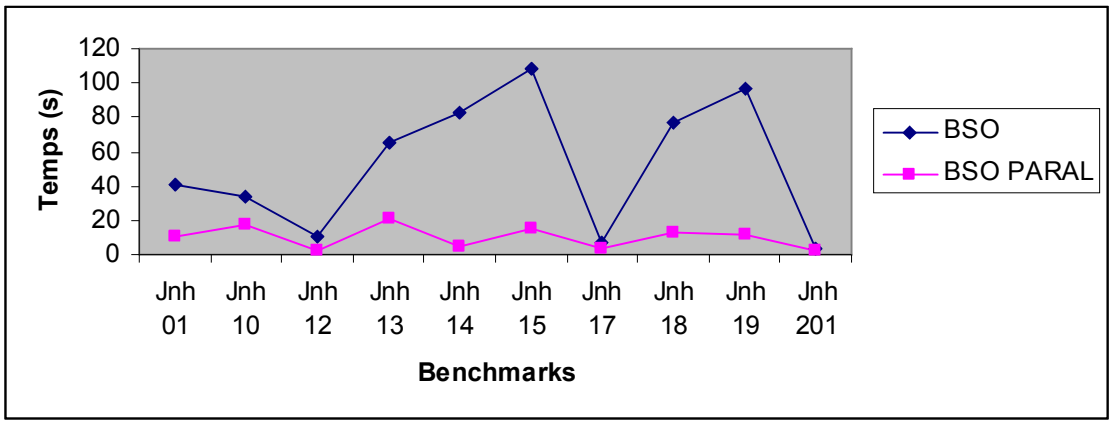


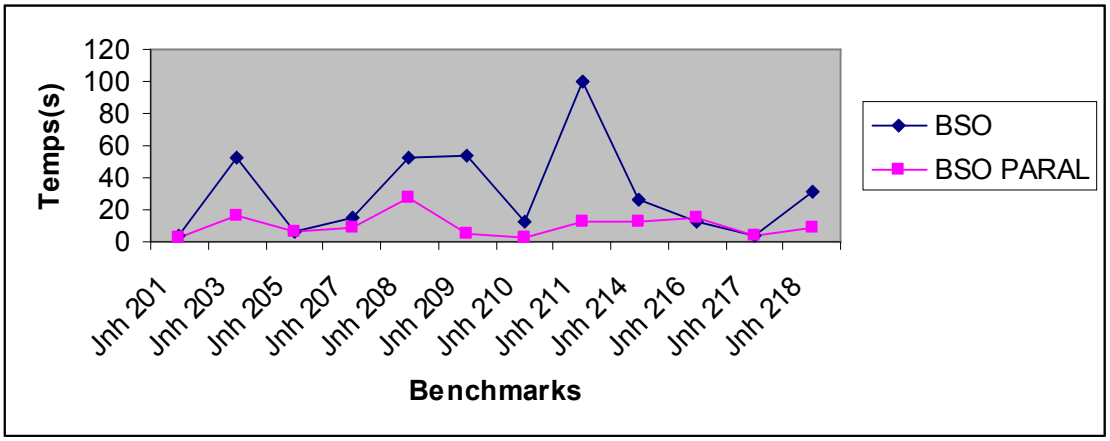


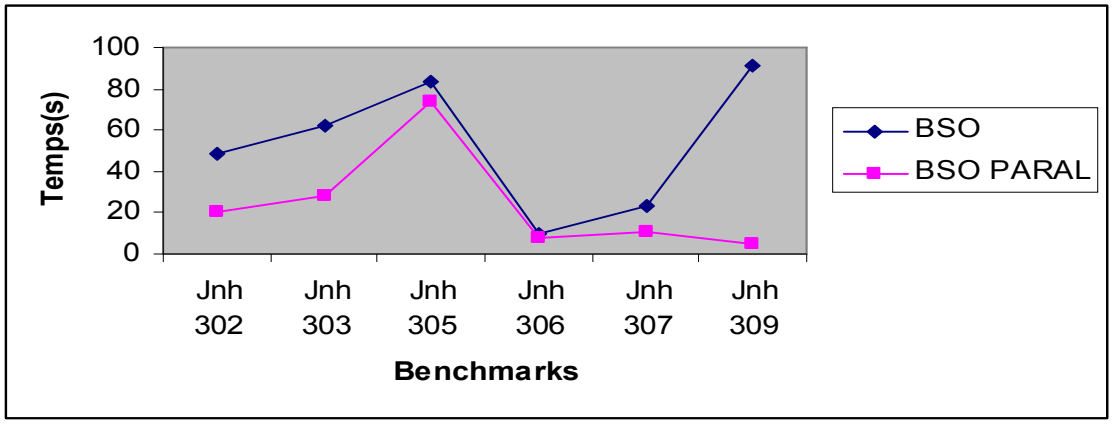


- ***Comparaison en terme de temps de calcul***

Pour comparer les performances des deux méthodes en terme de temps de calcul, nous avons considéré les instances pour lesquelles les deux méthodes ont obtenu la même valeur de la fonction objectif, autrement dite, nous avons considéré uniquement les instances satisfaites par les deux algorithmes. Les résultats ont montré que pour toutes les instances, La métaheuristique parallèle atteint la solution optimale on un temps inférieur ou égal à celui de la métaheuristiques séquentielle.







5.2 Comparaison avec d'autres métaheuristiques

Dans la table 10, nous comparons les performances de BSO à ceux d'autres métaheuristiques, à savoir: Ant Colony Optimisation (ACO), Scatter Search (SS) et La procédure GRASP. Nous ne faisons pas de comparaison en terme de temps de calcul, vu que ces algorithmes n'ont pas été exécutés sur des machines équivalentes.

Benchmark	Optimum	BSO	ACO	SS	GRASP
Jnh 01	420925	**420925**	**420925**	420892	420737
Jnh 10	420840	**420840**	**420840**	420479	420565
Jnh 11	420753	**420753**	420674	420141	420642
Jnh 12	420925	**420925**	**420925**	420701	420737
Jnh 13	420816	**420816**	**420816**	420716	420783
Jnh 14	420824	**420824**	**420824**	420616	420592
Jnh 15	420719	**420719**	**420719**	420632	420429
Jnh 16	420919	**420919**	420914	420889	420851
Jnh 17	420925	**420925**	**420925**	420794	420807
Jnh 18	420795	**420795**	**420795**	420404	420372
Jnh 19	420759	**420759**	420680	420330	420323
Jnh 201	394238	**394238**	**394238**	**394238**	**394238**
Jnh 202	394170	**394170**	**394170**	393752	393983
Jnh 203	394199	**394199**	394135	393876	393889
Jnh 205	394238	**394238**	**394238**	394060	394224
Jnh 207	394238	**394238**	394237	394107	394101
Jnh 208	394159	**394159**	**394159**	393560	393987
Jnh 209	394238	**394238**	**394238**	**394238**	394031
Jnh 210	394238	**394238**	**394238**	394067	**394238**
Jnh 211	393979	**393979**	**393979**	393742	393739
Jnh 212	394238	**394238**	**394227**	394082	394043
Jnh 214	394163	**394163**	**394163**	394152	393737
Jnh 215	394150	**394150**	**394150**	393942	393858
Jnh 216	394226	**394226**	**394226**	393933	394042
Jnh 217	394238	**394238**	**394238**	**394238**	394232
Jnh 218	394238	**394238**	**394238**	**394238**	394099
Jnh 219	394156	**393156**	394070	393942	393792
Jnh 220	394238	**394238**	**394238**	393985	394053
Jnh 301	444854	**444854**	444807	444842	444670
Jnh 302	444459	**444459**	**444459**	443895	444248
Jnh 303	444503	**444503**	444457	444223	444244
Jnh 304	444533	**444533**	**444533**	**444533**	444310
Jnh 305	444112	**444112**	443970	443594	**444112**
Jnh 306	444838	**444838**	**444838**	444515	444658
Jnh 307	444314	**444314**	**444314**	443662	444159
Jnh 308	444724	**444724**	**444621**	444250	444222
Jnh 309	444578	**444578**	**444578**	444483	444349
Jnh 310	444391	**444391**	444353	444313	444282
Nombre d'instances satisfaites		**38**	26	5	4

Table10: Comparaison avec d'autres métaheuristiques.

6. Conclusion

D'après les résultats numériques obtenus à l'issue des séries de tests effectués, nous avons constaté la puissance et l'efficacité de la métaheuristique BSO parallèle. La démarche suivie a porté ses fruits, et les résultats obtenus surpassent ceux des autres méthodes. Il va sans dire, qu'avec l'utilisation d'une machine parallèle, les temps de réponses, déjà très compétitifs, seront réduits.

Conclusion générale

Les problèmes d'optimisation combinatoire, sont l'intersection de plusieurs domaines de recherche: L'informatique, la recherche opérationnelle et les mathématiques discrètes. Un grand interêt leur est porté de par leurs diverses applications théoriques et industrielles. Leur résolution par des méthodes exactes s'avère excessivement onéreuse, si bien qu'on fait appel à des méthodes dites approximatives, les métaheuristiques.

Ces dernières années, d'innombrables travaux de recherche, ont été réalisés dans le but de mettre au point des métaheuristiques. Leurs sources d'inspiration sont diverses et variées : les processus physiques, les phénomènes biologiques et, plus récemment, l'intelligence en essaim.

Après la conception de la métaheuristique Bees Swarm Optimization dans le cadre de notre projet de fin d'études. Nous avons voulu pousser nos recherches, en proposant une parallélisation de cette métaheuristique.

Nous avons, dans un premier temps, dressé l'état de l'art des différentes stratégies de parallélisation des métaheuristiques: cette étude nous a permis de connaître les différentes stratégies de Parallélisation et les différents paramètres influents sur les performances des métaheuristiques parallèles.

Compte tenu des nombreuses alternatives qui s'offrent pour chacun des paramètres, nous avons adopté une démarche constructive et progressive pour la conception de l'algorithme final de BSO parallèle. Ainsi, plusieurs solutions ont été envisagées et plusieurs algorithmes ont été conçus, avant de proposer une solution finale.

Les résultats des tests qui ont demandé un temps et des efforts considérables, ont été très satisfaisants, puisque La métaheuristique BSO Parallèle a réussi à satisfaire la totalité des benchmarks de la classe Johnson. Elle dépasse donc et la métaheuristique BSO Séquentielle et les autres métaheuristiques dont les résultats ont été publiés tels que ACO, GRASP ou Scatter Search.

Perspectives

Comme perspectives, nous envisageons de :

- Traduire le code de la métaheuristiques en un langage compilé comme le C++ afin de réduire les temps de réponse qui sont déjà très compétitifs.
- Exécuter l'algorithme sur une machine parallèle virtuelle (réseaux de machines).
- Appliquer cette métaheuristique à d'autres problèmes d'optimisation combinatoire afin de mieux l'évaluer.

Bibliographie

[AMD67] G.M. Amdahl *Validity of the single processor approach of schieving large scale computing capabilities,* AFIPS Conf Proc 30 (1967) 483-485.

[BAC 93a] T. Back .*Applications of evolutionary algorithms*. Report of the System Analysis Research Group.University of Dortmund , 1993.

[BAP96] R. Battiti et M. Protasi. Solving MAX-SAT with Non_Oblivious Function and History based heuristics. Presented at Dimacs workshop on satisfiability problem : theory and application. Rulgers University, March 1996.

[BAP97]: R. Battiti et M. *Protasi.Reactive Search, a history-based Heuristics for MAX-SAT.* Journal of expériemntal algorithmics, 1997.

[BAT92] Battiti, R.and Tecchiolli, G. *Parallel Based Search for Combinatorial Optimization: Genetic Algorithms and TABU*. Microprocessors and Microsystems,16(7):351–367, 1992.

[BAT94]. Battiti, R. and Tecchiolli, G. *The Reactive Tabu Search*. ORSA Journal on Computing, 6(2):126–140, 1994

[BKS98] bulnheimer, G. Kotsis, and C. Strauss*, Parallelization strategies for the ant system,* Applied Optimization 24 (1998), 87–100.

[BOB93] Bolondi and M. Bondanza, *Parallelizzazione di un algoritmo per la risoluzione del problema del commesso viaggiatore*, M.Sc. Dissertation, Dipartimento di Elettronica eInformazione, Politecnico di Milano, 1993.

[BOL92]Boissin, N. and Lutton, J.L. *A Parallel Simulated Annealing Algorithm*. Parallel Computing, 19(8):859–872, 1993.

[BOT94] : E. Bonabeau et G. Théraulaz. *Intelligence collective*. Editions Hermès, 1994.

[BOU98] M.E BOUZGARROU *Parallélisation de la méthode du "Branch and Cut" pour résoudre le problème du voyageur de commerce*, Thèse de doctorat 1998.

[CAN98] Cant´u-Paz, E. (1998). *A Survey of Parallel Genetic Algorithms* . Calculateurs Parallèles, Réseaux et Systèmes répartis, 10(2):141–170.

[CEF88] R.D. Chamberlain, M.N. Edelman, M.A. Franklin, E.E. Witte, *Simulated Annealing on a Multiprocessor,* Proceedings of 1988 IEEE International Conference on Computer design, pp. 540{544.

[CHS92]. Chakrapani, J. and SkorinKapov, J. *A Connectionist Approach to the Quadratic Assignment Problem*. Computers & Operations Research, 19(3/4):287–295, 1992.

[CER 85] : V. Cerny. *A thermodynamical approach to the travelling salesman problem : an efficient simulated annealing algorithm*. Journal of Optimization Theory and Applications, 1985.

[CHS93]. Chakrapani, J. and SkorinKapov,J. *Massively Parallel Tabu Search for the Quadratic Assignment Problem*. Annals of Operations Research, 41:327–341,1993.

[CHS95]. Chakrapani, J. and SkorinKapov, J. *Mapping Tasks to Processors to Minimize Communication Time in a Multiprocessor System.* In The Impact of EmergingTechnologies of Computer Science and Operations Research, pages 45–64. Kluwer Academic Publishers, Norwell, MA, 1995.

[COO71] S.A.Cook. *The complexity of Theorem-proving* Procedures.Association of computing Machinery, New York, 1971.

[COS93] Cosnard. M et Trystram.D, *Algorithmes et architectures parallèles*, InterEditions, 1993.

[CRT98] Crainic, T.G. and Toulouse,M. (1998). *Parallel Metaheuristics*. In T.G. Crainic and G. Laporte, editors, Fleet Management and Logistics, pages 205–251. Kluwer Academic Publishers, Norwell, MA.

[CRT03] Crainic, T.G. and Toulouse, M. *Parallel Strategies for Metaheuristics*. In F. Glover and G. Kochenberger, editors, Handbook in Metaheuristics, pages 475– 513. Kluwer Academic Publishers, Norwell, MA, 2003.

[CTG95a]Crainic, T.G., Toulouse,M., and Gendreau,M. (1995a). *Parallel Asynchronous Tabu Search for Multicommodity Location-Allocation with Balancing Requirements*. Annals of Operations Research, 63:277–299.

[CRH05] Crainic, T.G, Hail,N. *Parallel metaheuristics applications*. Edition CRT (Centre de Recherche sur les Transports). Montréal Canada. Mai 2005.

[CUN97] Cung, V.D.,Mautor, T., Michelon, P., and Tavares, A. *A Scatter Search Based Approach for the Quadratic Assignment Problem on Evolutionary Computation and Evolutionary Programming.* In T. Baeck, Z. Michalewicz, and X. Yao, SUMMARY 39 editors, Proceedings of the IEEE InternationalConference on Evolutionary Computation, pages 165–170. IEEE Press, 1997.

[CUN02] Cung, V.-D., Martins, S.L., Ribeiro, C.C., and Roucairol, C. (2002). *Strategies for the Parallel Implementations of Metaheuristics*. In Ribeiro, C. andHansen, P., editors, Essays and Surveys in Metaheuristics, pages 263–308.Kluwer Academic Publishers, Norwell, MA.

[DAD00] : H. Drias, N. Alliche et S.Dahmani.*Contribution à la résolution du problème MAX-SAT par la méta-heuristique GLS*. Memoire de PFE, USTHB, 2000.

[DEF94] De Falco, I., Del Balio, R., Tarantino, E., and Vaccaro, R. *Improving Search by Incorporating Evolution Principles in Parallel Tabu Search.* In Proceedings International Confonference on Machine Learning, pages 823–828, 1994.

[DEJ 93] : K.A. De Jong. *On the state of evolutionary computation*. Proc of International Conference on Genetic Algorithms, 1993.

[DMC96] Dorigo, M., Maniezzo, V., and Colorni, A. (1996). *The Ant System: Optimization by a Colony of Cooperating Agents*. IEEE Transactions on Systems, Man, and Cybernetics - Part B, 26(1):29–41.

[DPR02] S. Duni Ekisoglu, P.M. Pardalos, and M.G.C. Resende, Parallel metaheuristics for combinatorial optimization, Models for Parallel and Distributed Computation - Theory, Algorithmic Techniques and Applications, R. Correa et al., Eds., Kluwer Academic Publishers, pp. 179-206, 2002

[DRI 03]Drias, H. and Ibri , *A. Parallel ACS for Weighted MAXSAT*.In Mira, J. and ´Alvarez, J., editors, Artificial Neural Nets Problem Solving Methods Proceedings of the 7th International WorkConference on Artificial and Natural Neural Networks, volume 2686 of Lecture Notes in Computer Science, pages 414–421.SpringerVerlag,Heidelberg, 2003.

[DRK 01]Drias, H. and Khabzaoui,M. *Scatter Search with RandomWalk Strategy for SAT and MaxSAT Problems.* In L. Monostori, V´ancza, J., and A. Moonis, editors, Proceedings of the 14th International Conference on Industrial and Engineering Applications of Artificial intelligence and Expert Systems, IEA/AIE 2001I, pages 35–44. SpringerVerlag, 2001.

[DSY05] Drias.H, Sadeg.S, Yahi.S, Cooperative Bees swarm optimisation for solving the maximum Weighted Maximum satisfiability Problem. IWANN 2005: 318-325.

[DUN90] R. Duncan, *A survey of parallel computer architectures*, Computer (1990) 5-16.

[FER95] T.A. Feo and M.G.C. Resende*, Greedy Randomized Adaptive Search Procedures*, Journal of Global Optimization, 6 (1995), pp. 109{133.
[FLY66] M.j. Flynn, *Very high-speed computing systems*, Proc. IEEE 54, (1966) 1901-1909.

[FPS98a]Folino,G., Pizzuti, C., and Spezzano,G. *Combining CellularGeneticAlgorithms and Local Search for Solving Satisfiability Problems*. In Proceedings of the Tenth

IEEE International Conference on Tools with Artificial Intelligence, pages 192–198. IEEE Computer Society Press, 1998.

[FPS98b]. Folino, G., Pizzuti, C., and Spezzano, G. *Solving the Satisfiability Problem by a Parallel Cellular Genetic Algorithm.* In Proceedings of the 24th EUROMICRO Conference, pages 715–722. IEEE Computer Society Press, 1998.

[GAJ79] **:** Garey et Johnson. *Computers and intractability: a guide of the theory of NP-Computeness.* Willey and Son 1979.

[GLO86] : F. Glover. Future paths for integer programming and links to artificial intelligence. University of colorado boulde, 1986.

[GLO89] F.Glover . *Tabu Search. partI*. ORSA, Journal of Computing 1(3): 190-206,1989.

[GLO90] F.Glover . *Tabu Search. partII*. ORSA, Journal of Computing 2(1): 4-32,1990

[GLO94] Glover, F. (1994). *Genetic Algorithms and Scatter Search: Unsuspected Potentials*. Statistics and Computing, 4:131–140.

[HAJ90] P. Hansan et B. Jaumar. *Algorithms for the maximum satisfiability problems.* Computing 44, Springer Verlag, 1990.

[HAN86] P. Hansen. The steepest ascent mildest descent heuristic for combinatorial programming. Congress on Numerical Methods in Combinatorial Optimization, Capri, Italie, 1986.

[HGH99] J. Hao, P. Galinier, M. Habib. *Méta heuristiques pour l'optimisation combinatoire et l'affectation sous contraintes*. Revue d'Intelligence Artificielle, 1999.

[HOL75]J.H. Holland, *Adaptation in natural and artificial systems: An introductory analysis withapplications to biology, control, and artificial intelligence*, University of Michigan Press,1975.

[HOL92] J.H. Holland, Genetic algorithms, Scientific American 267 (1992), 44–50.

[JEW93] : I.P.Jean et T.Walsh.*An ampirical Analysis Problem Using Guided Local Search*. Journal of artificial intelligence research 1.1993,P47-59.

[JOH74] : Johnoson, D.S *.Approximate Algorithmics for combinatrial Problems.* Journal of computer and System Sciences, P256-278, 1974.

[KGV83]Kirkpatrick, S., Gelatt, C.D., and Vecchi, M.P. (1983). *Optimization by Simulated Annealing.* Science, 220:671–680.

[KIR83] : S. Kirkpatrick. *Optimization by simulated annealing*. Science 220 ; 671-680, 1983.

[KKK04] Kokosinski, Z., Kolodziej, M., and Kwarciany, K. *Parallel Genetic Algorithm for Graph Coloring Problem.* In Bubak, M., van Albada, G.D., and Sloot, P.M.A., editors, International Conference on Computational Science, volume 3036 of Lecture Notes in Computer Science, pages 215–222. SpringerVerlag,Heidelberg, 2004.

[LAU94] Laursen, P.S. (1994). *Problem-Independent Parallel Simulated Annealing Using Selection and Migration.* In Davidor, Y., Schwefel, H.-P., and M¨anner, R., editors, Parallel ProblemSolving fromNature III, Lecture Notes in ComputerScience 866, pages 408–417. Springer-Verlag, Berlin.

[LPR94]Li, Y., Pardalos, P.M., and Resende, M.G.C. *A Greedy Randomized Adaptive Search Procedure for Quadratic Assignment Problem.* In DIMACS ImplementationChallenge, DIMACS Series on Discrete Mathematics and Theoretical ComputerScience, volume 16, pages 237–261. American Mathematical Society,1994.

[MUH89] H. M¨uhlenbein. *Parallel Genetic Algorithms, Population Genetics and Combinatorial Optimization*. In J.D. Schaffer, editor, Proceedings of the Third International Conference on Genetic Algorithms, pages 416–421. Morgan Kaufmann,San Mateo, CA, 1989.

[PAR98] : M. PARIZEAU. *Introduction à la NP-complétude* Université Laval, Hiver 98.

[PET00] : D. Petit. *Méthodes SAT pour un problème d'optimisation dans les graphes.* (Laboratoire d'Informatique Théorique et Appliquée de l'université de Metz).Octobre 2000.

[POL] : http://www.enseignement.polytechnique.fr/profs/informatique/Eric.Goubault/poly/

[PPR95] Pardalos, P.M., Pitsoulis, L., and Resende, M.G.C. *A Parallel GRASP implementation for the Quadratic Assignment Problem*. In A. Ferreira and J. Rolim, editors, Solving Irregular Problems in Parallel: State of the Art, pages 115–130. Kluwer Academic Publishers, Norwell, MA, 1995.

[PPR96]L. Pitsoulis, Pardalos, P.M., and Resende,M.G.C. *A Parallel GRASP forMAXSAT*. In Wasniewski J., Dongarra, J., Madsen, K., and Olesen, D., editors, Third International Workshop on Applied Parallel Computing, Industrial Computation and Optimization, volume 1180 of Lecture Notes in Computer Science, pages575–585. SpringerVerlag,Berlin, 1996.

[PPR97] : L.Pitsoulis, Pardalos et M.G.C.Resnede.1997. *Approximate Solution of Weighted MAX-SAT problems Using GRASP*. AT & T Research, Florham Park, NJ 07932 USA.

[REF97] Resende, M.G.C. and Feo, T.A. *Approximative Solution of Weighted MAXSAT Problems Using GRASP*. Discrete Mathematics and Theoretical Computer Science, 35:393–405, 1997.

[RIA99] **:** J. Hao, P. Galinier, M. Habib. Méta-heuristiques pour l'optimisation combinatoire et l'affectation sous contraintes.Revue d'Intelligence Artificielle, 1999.

[SAY04] Sadeg.S, Yahi.S. *Conception de la métaheuristique 'Optimisation par essaim d'abeilles et son adaptation au problèle MAX-W-SAT*. Mémoire du projet de fin d'études. sujet N° 22.USTHB.2004.

[SCH97b] : M. Schoenauer *Evolutionary computation: an introduction.* Control and Cybernetics 26(3) : 307-338, 1997

[SOB96b]. Sohn, A. and Biswas, R. *Satisfiability Tests with Synchronous Simulated annealing on the Fujitsu AP1000 Massivelyparallel Multiprocessor*. In Proceedings of the International Conference on Supercomputing, pages 213–220, 1996.

[SOH96a] Sohn, A. *Parallel Satisfiability Test with Synchronous Simulated Annealing on Distributed Memory Multiprocessor*. Journal of Parallel and Distributed Computing, 36:195–204, 1996.

[STU98]T. Stutzle, *Parallelization strategies for ant colony optimization*, Lecture Notes in Computer Science 1498 (1998), 722–731.

[TAD01] C.M Tadonki, *Contribution à l'algorithmique parallèle,* Thèse de doctorat, 2001.

[TAI91] Taillard, ´E.D. *Robust Taboo Search for the Quadratic Assignment Problem.* Parallel Computing, 17:443–455, 1991.

[TAI93]. Taillard, ´E.D. Recherches itératives dirigées parallèles. PhD thesis, ´Ecole Polytechnique Fédérale de Lausanne, 1993.

[TAL99] E.-G. Talbi, O. Roux, C. Fonlupt, and D. Robillard, *Parallel ant colonies forr combinatorial optimization problems*, Lecture Notes in Computer Science 1586 (1999), 239–247.

[TAL01]] E.-G. Talbi, O. Roux, C. Fonlupt, and D. Robillard, *Parallel ant coloniesfor the quadratic assignment problem,* Future Generation Computer Systems 17 (2001), 441–449.

[THG98]Talbi, E.G.,Hafidi, Z., and Geib, J.M. *ParallelAdaptive Tabu SearchApproach.* Parallel Computing, 24:2003–2019, 1998.

[VDR97] : C.Voudouris. *Guided Local Search and its application to the Travelling Salesman Problem*. PhD, Thesis, Department of Computer Science, University of Essex, Colchester, UK, July 1997.

[VEA95] M.G.A. Verhoeven and E.H.L. Aarts, *Parallel Local Search*, Journal of Heuristics 1(1995), 43–65.

[WIN98] Wilkerson, R. and Nemer Preece, N. *Parallel Genetic Algorithm to Solve the Satisfiability Problem*. In Proceedings of the 1998 ACM symposium on Applied Computing, pages 23–28. ACM Press, 1998.

Résumé

Les métaheuristiques sont largement reconnues pour être des outils essentiels pour aborder des problèmes difficiles dans de nombreux domaines. En fait, les métaheuristiques offrent une approche pratique pour résoudre des problèmes complexes.

Même en utilisant les métaheuristiques, les limites de ce qui pourrait être résolu en un temps raisonnable sont vite atteintes. Du moins, trop rapidement, pour les besoins croissants de la recherche et de l'industrie. Le calcul parallèle offre la perspective de pouvoir réduire le temps de calcul.

L'objectif de ce travail, est de paralléliser la métaheuristique "optimisation par essaim d'abeilles" que nous avons proposée dans le cadre notre projet de fin d'études, et dont la version séquentielle a donné de résultats très satisfaisants, publiés dans les LNCS. Pour évaluer notre algorithme parallèle, nous l'avons testé sur des instances du problème MAX-W-SAT.

Mots clés : Optimisation combinatoire, métaheuristique, intelligence en essaim, Bees Swarm Optimization (BSO), calcul parallèle, problème MAX-W-SAT.

Abstract

Métaheuristiques are widely recognized to be essential tools to tackle difficult problems in many fields. In fact, métaheuristics offer a practical approach to solve complex problems.

Even by using métaheuristiques, the limits of what could be solved in a reasonable time are quickly reached. At least, too quickly for the increasing research's and industry's needs. Parallel computing offers the prospect to be able to reduce the computing time.

The objective of this work, is to parallelize the metaheuristic "Swarm Bees Optimization" which we proposed within the framework of our end studies project, and whose sequential version gave very satisfactory results, published in LNCS. To evaluate our parallel algorithm, we tested it on MAX-W-SAT problem instances.

Key words: Combinatory Optimization, metaheuristic, swarm intelligence,Bees Swarm Optimization (BSO), parallel calculation, MAX-W-SAT poblem.