Samuel Thomas

Utilisation de la recherche locale pour le problème pseudo-booléen

Travail d'Étude et de Recherche
encadré par
Jean-Marie LAGNIEZ

Master Informatique 1ère Année

Université d'Artois

6 Juin 2011

Remerciements

Je tiens à remercier Monsieur GRÉGOIRE ainsi que l'ensemble du laboratoire de m'avoir accueilli au sein du CRIL.

Je tiens aussi à remercier tout particulièrement Jean-Marie Lagniez pour son encadrement et le temps qu'il m'a accordé lors de ce TER.

Enfin, je remercie mes quatre collègues qui m'ont supporté pendant toute la durée de ce TER.

Table des matières

1 Introduction **5**

2 Formalisme **7**
2.1 La logique propositionelle 7
2.2 Le problème pseudo-booléen 9
2.2.1 Définitions 9
2.2.2 Les contraintes pseudo-booléennes linéraires 10

3 Méthodes de résolution **11**
3.1 Les méthodes complètes 11
3.1.1 La recherche opérationnelle 11
3.1.2 Les solveurs SAT 11
3.2 Les méthodes incomplètes 12
3.2.1 Définition 12
3.2.2 Différents algorithmes 14

4 Contributions **19**
4.1 Le solveur 20
4.2 Choisir efficacement l'interprétation voisine 20
4.2.1 Calcul du problème pour choisir la variable à flipper 21
4.2.2 Se concentrer sur les contraintes 22
4.2.3 Supprimer le maximum de calculs 23
4.2.4 Expérimentations 29
4.3 Stratégies d'échappement des minima locaux 29
4.3.1 Des stratégies issues de SAT 30
4.3.2 Amélioration de la différence 34
4.3.3 Expérimentations 35

5 Conclusion **37**

Bibliographie **39**

Liste des Algorithmes

1 recherche locale . . . 13
2 Le glouton . . . 14
3 Le Recuit Simulé . . . 15
4 GSAT . . . 16
5 WSAT . . . 18

6 WalkPB . . . 19
7 Recalculer le problème . . . 21
8 Concentré sur les contraintes . . . 23
9 éviter les calculs . . . 24
10 condMake égale . . . 26
11 condBreak égale . . . 26
12 MAJ égale . . . 27
13 MAJ inférieur . . . 28
14 Best . . . 31
15 Random Walk Strategy . . . 32
16 Novelty . . . 32
17 RNovelty . . . 33
18 WalkPB Différence . . . 34
19 Best avec différence . . . 35

Chapitre 1

Introduction

Le but de ce TER est d'appliquer des méthodes de recherche locale aux problèmes pseudo-booléens. Afin de l'atteindre, j'ai commencé par m'informer, en cherchant différentes informations, sur les problèmes pseudo-booléens et la recherche locale. Puis j'ai implémenté plusieurs méthodes que j'ai améliorées par la suite.

Les problèmes pseudo-booléens sont des problèmes très répandu en intelligence artificielle, beaucoup de problèmes peuvent être transformés sous cette forme (problème de planification, problème d'ordonnacement, etc...). Le problème pseudo-booléen est, comme le problème SAT, un problème NP-complet, c'est-à-dire qu'il n'existe pas encore d'algorithme efficace en temps polynomial qui permet de résoudre de manière efficace l'ensembe des problèmes de ce type. Cependant, depuis ces dernières années beaucoup d'efforts ont été effectués autour de SAT ce qui a permis d'avoir des solveurs plus performants et qui résolvent plus de problèmes. Sachant que si l'on trouve un algorithme pour un problème NP-complet, il est possible, de résoudre l'ensemble des problèmes NP-complet. Les progrès effectués dans le cadre de SAT laissent penser que des avancées sont encore envisageables dans les solveurs de problèmes pseudo-booléens.

Dans le cadre de ce TER nous nous sommes intéressés à la résolution du problème Pseudo Booléen à l'aide de la recherche locale. La recherche locale est une méthode imcomplète issue de la recherche opérationnelle et qui fournie de bon résultats dans de nombreux domaine (SAT, routage de véhicule, problème du voyageur de commerce, etc...). Malgré tous ses atouts et l'efficacité qu'elle a pu démontrer sur les problèmes NP-Complet, la recherche locale n'a pas beaucoup été étudiée dans le cadre du problème Pseudo Booléen. En effet, nous avons tenté de chercher des solveurs permettant de traiter le problème Pseudo Booléen à l'aide d'une telle méthode et nos recherches se sont révélées infructueuses. Nous avons trouvé uniquement trois applications utilisant cette méthode. La première, appelé WSAT(OIP), est une application implémentée par Joachim Paul Walser en 1997 [12] et qui n'a plus été mise à jour depuis. Ensuite, nous avons trouvé un solveur, appelé LocalSolver, créé en 2007 par des ingénieurs de Bouygues E-lab et des chercheurs du Laboratoire d'Informatique Fondamentale - Université Aix-Marseille II. Celui-ci est encore mis à jour aujourd'hui. Enfin, la solution la plus récente est celle proposée par Cédric Piette à la dernière compétition PB qui s'est déroulée en 2010.

C'est dans ce contexte que s'inscrivent mes travaux. Tout d'abord, afin de com-

prendre le problème sur lequel nous allons travailler, nous allons définir ce qu'est un problème pseudo-booléen. Ensuite, nous analyserons les différentes méthodes permettant de résoudre les problèmes pseudo-booléens, en insistant sur la méthode qui est le thème de ce TER c'est-à-dire la recherche locale. Ensuite, nous présenterons les différents algorithmes que nous avons implémentés, les différentes parties de notre application ainsi que les différentes expérimentations qui ont été menées.

Chapitre 2

Formalisme

2.1 La logique propositionelle

Comme les problèmes pseudo-booléens empruntent quelques notions de logique propositionelle, nous commençons par aborder les notions utilisées par ceux-ci.

Définition 2.1.1 *Atome*
Un ***atome*** *est une variable booléenne. Un variable booléenne peut prendre deux valeurs* ***Vrai****,* ***Faux****. On note aussi ces valeurs respectivement par* $\mathbb{V}$, $\mathbb{F}$ *ou encore 1, 0.*

Définition 2.1.2 *formule*
Soit un alphabet constitué de

- *Un ensemble fini d'atome*
- *Des symboles* $\top$ *(la tautologie) et* $\bot$ *(la contardiction)*
- *des connecteurs*
  - $\neg$ *: la négation*
  - $\wedge$ *: la conjonction*
  - $\vee$ *: la disjonction*
  - $\Rightarrow$ *: l'implication*
  - $\Longleftrightarrow$ *: l'équivalence*
- *les séparateurs (et)*

Une ***formule*** *propositionnelle est un ensemble de mots construits sur l'alphabet tel que :*

- *un atome,* $\top$, $\bot$ *sont des formules*
- *si* $\mathscr{F}$ *est une formule,* $(\mathscr{F})$ *est une formule*
- *si* $\mathscr{F}$ *est une formule,* $\neg\mathscr{F}$ *est une formule*
- *si* $\mathscr{F}$ *et* $\mathscr{F}'$ *sont des formules :*
  - $\mathscr{F} \wedge \mathscr{F}'$ *est une formule*
  - $\mathscr{F} \vee \mathscr{F}'$ *est une formule*
  - $\mathscr{F} \Rightarrow \mathscr{F}'$ *est une formule*
  - $\mathscr{F} \Longleftrightarrow \mathscr{F}'$ *est une formule*

Définition 2.1.3 *littéral*
Un ***littéral*** *représente un atome* (α) *ou sa négation* $(\neg\alpha)$*.* α *est appellé littéral positif et* $\neg\alpha$ *littéral négatif.*

Définition 2.1.4 *interprétation d'une formule*
*L'**interprétation** I d'une formule propositionnelle est une application de l'ensemble des variables propositionnelles dans l'ensemble des valeurs de vérité {0, 1}.*

L'interprétation I($\mathscr{F}$) d'une formule $\mathscr{F}$ est définie par la valeur de vérité donnée à chacun des atomes de $\mathscr{F}$.
L'interprétation d'une formule est définie par :
- $I(\top) = 1$
- $I(\bot) = 0$
- $I(\neg\mathscr{F}) = 1 - I(\mathscr{F})$
- $I(\mathscr{F} \wedge \mathscr{F}') = min(I(\mathscr{F}), I(\mathscr{F}'))$
- $I(\mathscr{F} \vee \mathscr{F}') = max(I(\mathscr{F}), I(\mathscr{F}'))$
- $I(\mathscr{F} \implies \mathscr{F}') = 1$ *ssi* $I(\mathscr{F}) = 0$ *et* $I(\mathscr{F}') = 1$
- $I(\mathscr{F} \iff \mathscr{F}') = 1$ *ssi* $I(\mathscr{F}) = I(\mathscr{F}')$

Il existe plusiseurs façons de répésenter l'interprétation d'une formule. Dans la suite de ce rapport nous adopterons la convention consistant à représenter l'interprétation par l'ensemble des littéraux vrais pour l'interprétation.

Exemple : *Pour la formule* $\mathscr{F} = (a\vee b)\wedge(b\vee c)$ *et l'interprétation* $I(a) = 0,\ I(b) = 1$ *et* $I(c) = 1$*, on notera I comme cela :* $I(\mathscr{F}) = \{\neg a,\ b,\ c\}$

Définition 2.1.5 *Modèle*
Si pour une interprétation I, $I(\mathscr{F}) = 1$ *on dit que la formule* $\mathscr{F}$ *est **satisfaite** par l'interprétation I. Dualement, si* $I(\mathscr{F}) = 0$ *on dit que la formule* $\mathscr{F}$ *est **falsifiée** par l'interprétation I.*
*Une interprétation I est **modèle** d'une formule* $\mathscr{F}$ *si I satisfait la formule.*
*Une interprétation I est **contre-modèle** d'une formule* $\mathscr{F}$ *si I falsifie la formule.*

Définition 2.1.6 *satisfiable, insatisfiable*
*Une formule est **satisfialbe** (consistante) si et seulement si elle admet au moins un modèle.*
*Une formule est **insatisfiable** (inconsistante ou contradictoire) si et seulement si elle n'admet pas de modèle.*

Définition 2.1.7 *Distance de Hamming*
Soit I et I' deux interprétations d'une même formule. La distance de Hamming entre deux interprétations d'une formule représente le nombre de variables pour lesquelles I et I' diffèrent.

Notations :

Dans la suite de ce rapport, nous utiliserons les termes de $\#false(I, \Sigma)$ et $\#true(I, \Sigma)$ qui, étant donnée une interprétaion I donnée et un ensemble Σ de contraintes, indiquent respectivement le nombre de contraintes falsifiées et satisfaites par l'interprétation I.

2.2 Le problème pseudo-booléen

Une fonction pseudo-booléenne est une fonction qui compare n valeurs booléennes à un réel. Le terme pseudo-booléen vient du fait que, bien que ces fonctions ne soient pas booléennes, elles font beaucoup penser aux fonctions booléennes et y empreintes quelques notions. Ces fonctions sont étudiées depuis les années 1960 pour la recherche opérationnelle et la programmation linéaire 0-1 (PL01).

Le problème des fonctions booléennes est un sujet très riche dans lequel il faut optimisé la valeur d'une fonction pseudo-booléenne. Ce formalisme offre un moyen plus naturel et compréhensible d'exprimer des contraintes. Ceci est particulièrement vrai lorsque l'on souhaite prendre en compte les problèmes d'optimisation. Certaines règles d'inférences permettent de résoudre les problèmes polynomiaux une fois transformés en contraintes pseudo-booléennes alors qu'ils demandent un nombre exponentiel d'étapes s'ils sont sous forme de clauses. De plus, ce formalisme étant très proche de celui de SAT il permet de bénéficier des avancées des recherches issues du domaine SAT. En effet, il existe des algorithmes permettant de transposer les contraintes pseudo-booléennes en contraintes SAT [5]. Finalement, les solveurs pseudo-booléens bénéficient aussi des avancées de la recherche opérationnelle dans la programmation linéaire à nombres entiers et plus spécialement de la programmation linéaire 0-1 [2].

2.2.1 Définitions

Une contrainte pseudo-booléenne est définie par un ensemble fini de variables booléennes x_j. Les variables booléennes peuvent prendre deux valeurs, *faux* et *vrai*, qui peuvent être représentées respectivement par $\{\mathbb{F}, \mathbb{V}\}$ ou encore par $\{0, 1\}$. Un littéral l_j est soit une variable booléenne x_j soit son complémentaire $\sim x_j$. un littéral positif (x_j) est évalué à $\mathbb{V}$ si et seulement si la variable x_j correspondante est vrai, un littéral négatif ($\neg x_j$) est évalué à $\mathbb{V}$ si et seulement si la variable correspondante x_j est à faux.

Dans les contraintes pseudo-booléennes, un coefficient entier constant est assigné à chaque littéral (même si un coefficient de 1 est souvent omis dans la représentation de la contrainte). Les opérateurs d'addition et de multiplication ont leur sémantique mathématique habituelle. Un booléen est représenté par les valeurs 0 pour *faux* et 1 pour *vrai*. Puisque l'opération d'addition et de multiplication garde la sémantique habituelle, on a par exemple :
$\forall a \in \mathbb{Z}, \forall b \in \mathbb{Z}, a.\mathbb{V} + b.\mathbb{F} = a.1 + b.0 = a.$

Il existe plusieurs types de contraintes pseudo-booléennes. Dans le cadre de ce TER, nous nous sommes intéressés aux contraintes pseudo-booléennes linéaires.

Un problème pseudo-booléen est représenté de deux manières distinctes. Il est soit une conjonction de contraintes pseudo-booléennes, cela représente les problèmes de décision, soit une conjonction de contraintes pseudo-booléennes avec une fonction d'objectif qui consiste à minimiser ou maximiser une expression, cette forme représente les problèmes d'optimisation. Dans le cadre de ce TER nous nous interressons uniquement à la forme de conjonction de contraintes pseudo-booléennes.

2.2.2 Les contraintes pseudo-booléennes linéraires

La forme d'une contrainte pseudo-booléenne linéaire est la suivante :

$$\sum_i a_i l_i \rhd k$$

Où a_i et k sont des constantes entières, l_i sont des littéraux et $\rhd$ est un opérateur classique de comparaison $(=, >, \geq, <, \leq)$

La partie droite de la contrainte (k) est appelée le *degré* de la contrainte.

Il est possible de modifier la contrainte afin d'obtenir uniquement des littéraux positif. Il s'agit de transformer $\neg l_i$ en $1 - l_i$.

Chapitre 3

Méthodes de résolution

Il existe deux types d'approches de résolution pour les problèmes pseudo-booléens. D'une part, il y a les méthodes complètes, c'est-a-dire, qui déterminent en temps fini la consistance de n'importe quel problème. Elles proviennent principalement de la Recherche opérationnelle et de solveurs SAT modifiés. D'autre part les approches incomplètes, qui se terminent si une solution est trouvée ou si elle n'a pas trouvé de solution. Le but de ces méthodes est de trouver une solution au problème s'il y en existe une. Si une solution est trouvée, l'algorithme s'arrête et peut répondre que le problème est satisfiable, sinon, au bout d'un certain nombre d'opérations ou d'un certain temps, l'algorithme s'arrête sans pouvoir conclure sur la satisfaisabilité du problème. La plupart de ces méthodes sont des méthodes à base de recherche locale.

3.1 Les méthodes complètes

3.1.1 La recherche opérationnelle

En recherche opérationnelle, les problèmes pseudo-booléens sont des problèmes de programmation linéaire en nombre entier en 0-1 (PL01). Ces problèmes sont parmi les plus complexes et les plus étudiés en recherche opérationnelle, différents algorithmes ont été proposés afin de les résoudre le plus efficacement possible. Le principal inconvénient de ces méthodes est qu'elles sont d'une complexité exponentielle, c'est-à-dire que plus le nombre de variables et de contraintes est grand, plus le temps de résolution du problème augmente. Les méthodes de résolutions les plus connues sont la méthode des plans de coupes introduite par R.E.Gomory en 1958 [8] et la méthode de branch and bound introduite par A.H.Land et A.G.Doig en 1960 [10].

3.1.2 Les solveurs SAT

Les solveurs SAT ayant connu de grandes améliorations ces dernières années et puisque SAT et le problème pseudo-booléen sont dans la même classe de complexité (NP-Complet), il est possible de traduire une instance SAT vers une instance du problème pseudo-booléens et *vis versa*. De cette manière, le problème pseudo booléen peut bénéficier des avancées de SAT. Une autre méthode étudiée et d'étendre les solveurs SAT aux problèmes pseudo-booléens. P. Barth [1] a proposé la première

extention de l'algorithme Davis-Putnam-Logeman-Loveland au cas des problèmes pseudo-booléen. Depuis, d'autres solveurs ont été proposés, tel que SAT4JPseudo [11] qui est une extention d'un solveur CDCL (*Conflict Driven Clause Learning*) pour les problèmes pseudo-booléen et Minisat+ [5] qui lui traduit les contraintes pseudo-booléennes en CNF.

3.2 Les méthodes incomplètes

Une autre solution pour résoudre ces problèmes consiste à utiliser des méthodes incomplètes, c'est-à-dire qu'elles n'explorent pas systématiquement l'ensemble des interprétations possibles et est donc capable de répondre uniquement s'il trouve une solution. Les solveurs de ce type sont souvent basés sur la recherche locale. C'est la solution que nous avons envisagé lors de ce TER.

3.2.1 Définition

La recherche locale pour le problème pseudo-booléen est une approche élémentaire qui consiste à se déplacer d'interprétation en interprétation jusqu'à l'obtention d'un modèle. Ces méthodes à base de recherche locale sont des techniques incomplètes se restreignant au traitement de la consistance, c'est-à-dire qu'elles permettent d'exhiber une solution mais ne garantissent pas son obtention, et ne peuvent *a priori* prouver l'inconsistance. Si la recherche échoue il est alors impossible de conclure sur la satisfaisabilité ou non de l'instance. Elle s'appuie sur un parcours (souvent stochastique) de l'espace de recherche, c'est-à-dire de l'espace des interprétations. À chaque pas, seulement quelques interprétations sont examinées (les interprétations voisines). En contrepartie, comparativement aux approches complètes, les algorithmes à base de recherche locale disposent d'une plus grande flexibilité dans l'exploration de cet espace de recherche. Ces méthodes de recherche n'imposent pas, *a priori*, d'ordre sur l'examen des interprétations. Comparativement aux recherches complètes, le nombre d'interprétations visitées par la recherche locale est très réduit. Malgré cela, ces algorithmes se révélent d'une grande efficacité pour la recherche de modèles.

La fonction d'évaluation utilisée est très souvent fonction du nombre de contraintes falsifisées. Pour trouver un point d'altitude minimale, il faut donc minimiser cette fonction d'évaluation, c'est-à-dire le nombre de contraintes falsifiées par l'interprétation courante. Le premier principe est la « descente » c'est-à-dire que tant qu'il existe une interprétation au voisinage de l'interprétation courante améliorant la fonction d'évaluation, il faut se déplacer vers celle-ci. Ce déplacement est appelé réparation de l'interprétation courante.

Le voisinage est à définir et varie suivant les algorithmes de recherche locale. Pour le pseudo-booléen, le voisinage considéré est le plus souvent l'ensemble des interprétations distantes de 1, en terme de distance de Hamming. Autrement dit, il n'y a que la valeur de vérité d'une seule variable qui diffère d'une inerprétation à l'autre.

Appliqué seul, le principe de descente conduit souvent à un minimum local. Ce minimum local est une interprétation qui n'est pas un modèle et pour laquelle il n'existe pas, dans son voisinage, d'interprétation permettant d'améliorer la valeur

de la fonction objective. Il faut alors appliquer un second principe d'échappement pour sortir du minimum local. Il doit permettre de s'éloigner suffisamment de l'interprétation courante afin d'éviter de la revisiter une nouvelle fois. Ce principe autorise donc des dégradations de la valeur de la fonction objective. Il existe différentes stratégies pour s'échapper d'un minimum local (le recuit simulé [9], la méthode tabou [6], algorithme génétique [7], les colonies de fourmis [3], etc.) et différentes manières de les utiliser (tout au long de la recherche, uniquement lorsqu'un minimum local a été atteint, etc.). C'est souvent ce principe qui différencie les multiples algorithmes de recherche locale.

Réaliser le meilleur déplacement (c'est-à-dire choisir le « meilleur » voisin) constitue la motivation principale des méthodes de recherche locale. En effet, déterminer l'interprétation qui parmi un ensemble d'interprétations est celle qui est la plus proche de l'objectif (c'est-à-dire celle qui permettra d'atteindre un modèle avec le plus petit nombre de déplacements), est une tâche délicate étant donné que le nombre d'interprétations pouvant être examinées est extrêmement réduit par rapport au nombre d'interprétations admises par la formule. Par ailleurs, la recherche aboutissant le plus souvent à un minimum local, les techniques d'échappement jouent un rôle primordial pour l'efficacité des méthodes.

Algorithme 1: recherche locale

Données : Un ensemble de contraintes Σ, entier : max_flip (nb flip maxi), entier : max_tries(nb d'essais maxi)
Résultat : le nombre de contraintes falsifiées, 0 si I est un modèle

```
1  début
2      nb_tries = 0;
3      tant que nb_tries < max_tries faire
4          Générer une interprétation initiale I;
5          nb_flip = 0;
6          tant que nb_flip < max_flip faire
7              si I est modèle alors retourner 0;
8              si ∃v ∈ Σ permettant de descendre alors flipper v dans I;
9              sinon
10                 Trouver une nouvelle interprétation I suivant un critère
                   d'échappement pour s'écarter du minimum local;
11     retourner nombre de contraintes falsifiées par I
12 fin
```

L'agorithme 1 décrit les étapes effectuées par un algorithme de recherche locale. Tout d'abord, une interprétation initiale est considérée (ligne 4). Ensuite, tant qu'une solution n'a pas été trouvée ou qu'un certain nombre de flip (*max_flip*) n'a pas été atteint (ligne 6) effectue plusieurs étapes. Tout d'abord, si un modèle du problème est trouvé, alors l'algorithme retourne zéro (ligne 7). Sinon, s'il existe une interprétation voisine permettant de diminuer le nombre de contraintes falsifiées, alors cette interprétation devient l'interprétation courante (ligne 8). Dans le cas contraire (minimum local), on choisit la variable à flipper suivant un certain

critère d'échappement (ligne 10). Ces étapes sont répétées un certain nombre de fois (max_tries) fixé au départ.

3.2.2 Différents algorithmes

Le principe de recherche locale étant une méthode étudiée depuis plusieurs années, beaucoup d'algorithmes ont été proposés. Nous allons présenter une liste non exhaustive des algorithmes les plus connus. Ceux-ci différent principalement dans leurs stratégie d'échappement, c'est-à-dire le moyen de se sortir d'un minimum local.

Algorithme glouton : Hill-Climbing

Dans cette méthode, une configuration initiale est considérée et soumise à des changements locaux améliorant à chaque fois la fonction objective ou d'évaluation, jusqu'à trouver un optimum, le plus souvent local. Cet algorithme ne fait donc que l'étape de descente sans utiliser de mécanisme d'échappement.

L'inconvénient de cet algorithme est qu'il s'arrête dès le premier optimum rencontré. Au vu de l'étendue de l'espace possible des interprétations, il est clair que généralement l'optimum atteint est local. Comme la procédure ne prévoit pas de moyen d'échappement, cet algorithme est très inefficace.

L'algorithme 2 illustre le glouton. Tout comme l'algorithme général de recherche locale, on effectue un certain nombre max_flip de flips. Il diffère uniquement dans le choix de la variable à flipper dans le sens où il limite son choix aux variables qui minimisent le nombre de contraintes falsifiées ligne 8.

Algorithme 2: Le glouton

Données : Un ensemble de contraintes, entier : max_flip (nb flip maxi), entier : max_tries(nb d'essais maxi)
Résultat : le nombre de contraintes falsifiées, 0 si I est un modèle

```
1  début
2    nb_tries = 0;
3    tant que nb_tries < max_tries faire
4      Générer une interprétation initiale I;
5      nb_flip = 0;
6      tant que nb_flip < max_flip faire
7        si I est modèle alors retourner 0;
8        flipper une variable dans I qui minimise le nombre de contraintes
         falsifiées;
9    retourner nombre de contraintes falsifiées par I
10 fin
```

Le Recuit Simulé

L'idée de base de cette heuristique provient de l'opération de recuit, courante en sidérurgie et dans l'industrie du verre. Au lieu de ne permettre que des flips

qui diminuent la fonction objectif, on autorise des augmentations, même importantes, de l'interprétation au début de l'exécution de l'algorithme, puis, à mesure que le temps passe, on autorise ces augmentations de plus en plus rarement. Plus précisément, partant d'une solution courante x_n, on tire au sort une solution voisine $x^* \in V(x_n)$. Si $F(x^*) \leqslant F(x_n)$ (c'est-à-dire x^* minimise le nombre de contraintes falsifiées), alors x^* devient la nouvelle solution courante $x_n + 1$. Si $F(x^*)\, F(x_n)$, on décide en tirant au sort si x^* devient la nouvelle solution courante ou si x_n reste la solution courante. Dans ce dernier cas, on tirera au sort un autre voisin de x_n. La probabilité p d'accepter x^* quand cette solution est moins bonne que la solution courante décroît avec le temps de l'algorithme. La probabilité p est généralement une fonction dépendant d'un paramètre appelé de « température » et de la dégradation de la fonction objectif.

Algorithme 3: Le Recuit Simulé

Données : Un ensemble Σ de contraintes, entier : max_flip, entier : max_tries
Résultat : le nombre de contraintes falsifiées, 0 si I est un modèle

```
1  début
2      proba = 0;
3      nb_tries = 0;
4      tant que nb_tries < max_tries faire
5          Générer une interprétation initiale I;
6          nb_flip = 0;
7          stagnation = 0;
8          tant que stagnation < max_stagnation et nb_flip < max_flip faire
9              si I est modèle alors retourner 0;
10             si ∃v ∈ Σ permettant de descendre alors flipper v dans I;
11             sinon
12                 choisir une variable v à flipper;
13                 proba = f(proba, I, v, Σ);
14                 p = random(0, 1);
15                 si p < proba alors
16                     flipper(v);
17                     stagnation = 0;
18                 sinon
19                     stagnation + +;
20             flipper(v);
21     retourner nombre de contraintes falsifiées par I
22 fin
```

L'algorithme 3 illustre l'approche du recuit simulé. On effectue un nombre max_flip de flip ou un nombre $max_stagnation$ de stagnations (ligne 8), correspondant au nombre de flips pendant lesquels l'interprétation n'a pas changé. Ensuite, s'il existe une variable permettant de descendre, alors on la choisit (ligne 10). Sinon, on choisit une variable au hasard et suivant une certaine probabilité calculée en fonction de plusieurs paramètres, le plus souvent ces paramètres sont la probabilité précédente,

l'interprétation courante, la variable à flipper et le problème, soit la variable choisie est flippée, soit on garde l'interprétation courante. Dans ce dernier cas, le nombre de stagnation est incrémenté (ligne 12).

GSAT

Le principe de cet algorithme est simple, il commence par générer une interprétation des variables du problème. Puis, il effectue des réparations de l'interprétation, c'est-à-dire qu'il inverse la valeur de l'interprétation d'une varaible, pendant un certain nombre d'étapes max_flip fixé au départ. S'il ne trouve pas de solution lors de cette phase, on effectue une nouvelle phase appelée le restart. Cela consiste à répéter le processus depuis le début et permet de diversifier la recherche. On effectue ces deux étapes un certain nombre de fois num_tries fixé lui aussi au départ. Enfin, s'il n'a pas trouvé de solution au bout d'un certain temps ou d'un certain nombre de fois, il retourne qu'il n'a pas trouvé de modèle. GSAT est une des méthodes les plus connues et beaucoup de variantes de cette méthode ont été étudiées. L'algorithme 4 illustre cette stratégie.

Algorithme 4: GSAT

Données : Un ensemble Σ de contraintes, entier : max_flip (nb flip maxi), entier : max_tries(nb d'essais maxi)
Résultat : le nombre de contraintes falsifiées, 0 si I est un modèle

```
1  début
2      nb_tries = 0;
3      tant que nb_tries < max_tries faire
4          Générer une interprétation initiale I;
5          nb_flip = 0;
6          tant que nb_flip < max_flip faire
7              si I est modèle alors retourner 0;
8              sinon
9                  pour chaque variable v de Σ faire
10                     v.va_sat ← nombre de contraintes qui deviendraient
                       satisfaites si v était flippée;
11                     v.va_fals ← nombre de contraintes qui deviendraient
                       falsifiées si v était flippée;
12                     v.score = v.va_sat − v.va_fals;
13                 Ω =ensemble des variables ayant le plus petit score;
14                 v =une variable au hasard dans Ω;
15                 flipper(v);
16     retourner nombre de contraintes falsifiées par I
17 fin
```

La particularité de GSAT est qu'il ne distingue pas les deux phases habituellement considérées dans les approches de recherche locale, la descente et l'échappement. En effet, la phase de descente n'effectue pas uniquement de la descente, elle autorise des remontées. Dans le cadre de GSAT on peut considéré trois phases :

- Une phase de descente, qui consiste à améliorer la valeur de la fonction objective ;
- Une phase de remonté, qui consiste à dégrader la valeur de la fonction objective ;
- Une phase de stagnation (exploration d'un plateau), qui consiste à explorer des solutions ne modifiant pas la valeur de la fonction objective.

La figure suivante 3.1 illustre ces trois phases. Il existe cependant une phase d'échappement dans cette méthode, qui consiste à redémarrer avec une nouvelle interprétation, ce qui permet de diversifier la recherche dans le cas où le solveur reste englué dans une zone de l'espace de recherche ne conduisant pas à une solution.

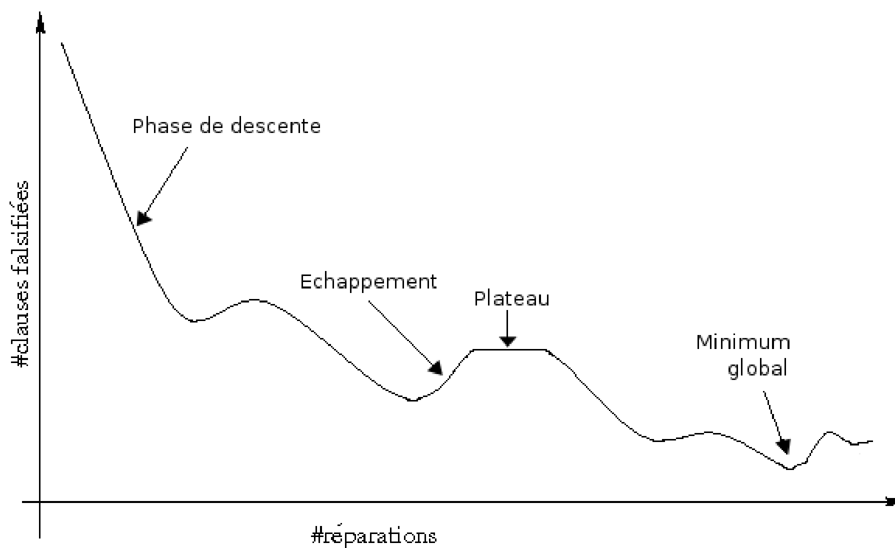

FIGURE 3.1 – Paysage de la recherche locale

WSAT

WSAT est très proche de GSAT. En effet, ces deux approches ne diffèrent que sur la notion de voisinage d'une interprétation. Afin de limiter le nombre d'interprétations à considérer, on se limite à considérer le voisinage vis-à-vis d'une contrainte. Pour cela, WSAT choisit une contrainte au hasard parmi les contraintes falsifiées par l'interprétation courante et il restreint le choix de la variable à flipper aux variables de cette contrainte. Tout comme GSAT, beaucoup de variantes ont été étudiées, de plus, les variantes de GSAT peuvent facilement être transposées à WSAT.

L'algorithme 5 décrit une telle approche. On remarque que l'algorithme change par rapport à GSAT (ligne 9), en effet il se restreint aux variables d'une seule contrainte falsifiée. Ensuite, comme pour GSAT, le score de chaque variable est calculé et on prend une variable au hasard parmi les variables possèdant le plus petit score. On effectue ces opérations pendant un nombre max_flip de flips (ligne 6). Puis, on effectue un certain nombre max_tries de *restarts* (ligne 3), c'est-à-dire qu'on recommence la recherche avec une nouvelle interprétation. Enfin, si aucun

modèle n'est trouvé on retourne le nombre minimum de contraintes falsifiées (ligne 17).

Algorithme 5: WSAT

Données : Un ensemble de contraintes, entier : max_flip (nb flip maxi), entier : max_tries(nb d'essais maxi)

Résultat : le nombre de contraintes falsifiées, 0 si I est un modèle

```
1  début
2      nb_tries = 0;
3      tant que nb_tries < max_tries faire
4          Générer une interprétation initiale I;
5          nb_flip = 0;
6          tant que nb_flip < max_flip faire
7              si I est modèle alors retourner 0;
8              sinon
9                  C ←Une contrainte au hasard parmi les contraintes falsifiées;
10                 pour chaque variable v de C faire
11                     v.va_sat ← nombre de contraintes qui deviendraient
                       satisfaites si v était flippée;
12                     v.va_fals ← nombre de contraintes qui deviendraient
                       falsifiées si v était flippée;
13                     v.score = v.va_sat − v.va_fals;
14                 Ω =ensemble des variables ayant le plus petit score;
15                 v =une variable au hasard dans Ω;
16                 flipper(v);
17     retourner nombre de contraintes falsifiées par I
18 fin
```

Chapitre 4

Contributions

Lors de ce TER nous avons implémenté différents algorithmes afin de trouver celui qui répondra le mieux aux problèmes pseudo-booléens. Comme nous l'avons vu, afin d'être le plus efficace possible, un algorithme de recherche locale demande deux attentions particulères : la méthode permettant de donner le nombre de contraintes falsifiées par le flip d'une variable ainsi que la métaheuristique qui permet de s'échapper des minima locaux. Dans un premier temps nous nous sommes concentré à la recherche d'une méthode permetant de calculer de manière efficace la gestion du flip d'une variable. Pour cela, nous avons implémenté plusieurs approches basées sur l'approche WalkSAT et nous les avons testées sur plusieurs problèmes afin de sélectionner la plus efficace pour les problèmes pseudo-booléens. L'algorithme 6 appellé WalkPB, est un algorithme de recherche locale basé sur WalkSAT (cf. 5) et sera utilisé par la suite dans l'ensemble de nos expérimentations. Après avoir étudié la gestion du flip, nous avons implémenté plusieurs stratégies d'échappement provenant de WSAT. Puis, nous avons ajouté une notion d'échappement que nous avons implémentée dans chacune de ces stratégies.

Algorithme 6: WalkPB

Données : Σ : PB, max_flip : int, max_tries : int
Résultat : le nombre de contraintes falsifiées, 0 si I est un modèle

```
1  début
2      nb_tries = 0;
3      tant que nb_tries < max_tries faire
4          Générer une interprétation initiale I;
5          nb_flip = 0;
6          tant que nb_flip < max_flip faire
7              si I est modèle alors retourner 0;
8              C ←Une contrainte au hasard parmi les contraintes falsifiées;
9              si ∃I' ∈ Voisinage(I) tel que falsifie(I', Σ) < falsifie(I, Σ)
               alors I ← I';
10             sinon I ← critère d'échappement;
11     retourner nombre de contraintes falsifiées par I
```

4.1 Le solveur

Dans cette section nous décrivons les différentes parties du solveur implémenté lors de ce TER. La figure 4.1 donne le diagramme de classe de notre application. En premier lieu, nous avons la classe Probleme qui représente le problème à traiter et qui s'occupe de charger le problème depuis un fichier. Cette classe contient l'ensemble des contraintes qui lui sont rattachées ainsi qu'un tableau *contraintesFalsifiees* qui contient la liste des contraintes falsifiées par l'interprétation courante, ce qui permet de choisir rapidement une contrainte parmi les contraintes falsifiée. Les opérations réalisées lors de la recherche, afin de choisir efficacement la variable à flipper, demandant d'effectuer des opérations différentes selon le type de la contrainte (égale, inférieur ou égale, etc.), les contraintes que le problème possède héritent de la classe abstraite Contrainte. Ensuite, nous avons la classe RechercheLocale qui s'occupe d'effectuer la méthode de recherche locale (solveProbleme et flipper). Elle possède un Probleme et une MetaHeuristique. La métaheuristique représente la méthode avec laquelle nous allons choisir la variable à flipper. Comme il existe plusieurs métaheuristiques, elles héritent toutes de la classe MetaHeuristique. Ceci représente la deuxième partie de notre contribuion lors de ce TER. Les variables n'ont pas de représentation objet, elles ne possèdent qu'un indice, qui est l'indice dans le tableau instance de la classe Recherche locale. Cet indice est choisi comme suis : x1 possède l'indice 0, x2 l'indice 1, etc.

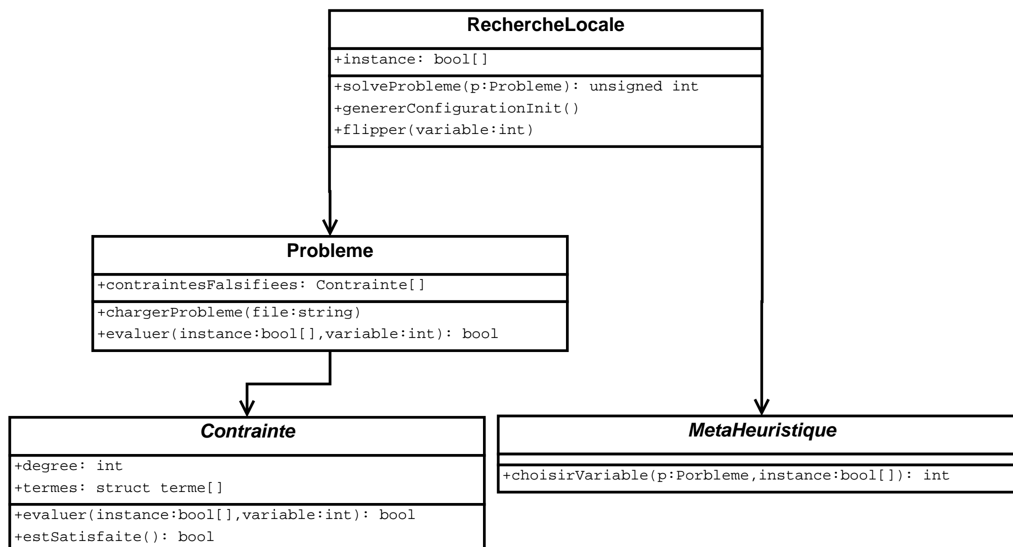

FIGURE 4.1 – Diagramme de classe

4.2 Choisir efficacement l'interprétation voisine

Le but de la recherche locale étant de minimiser une fonction objective, il faut pouvoir choisir la variable à flipper afin de pouvoir diriger la recherche vers la descente. Ce choix est important car il va permettre à la méthode d'effectuer le plus de descente possible. Comme nous avons choisi de minimiser le nombre de contraintes falsifiées, il est intéressant de connaître le nombre de contraintes que va falsifier chaque variable du problème si elle est flippée. Ce calcul peut influencer grandement

le temps nécessaire au choix de la variable. Afin de trouver une méthode rapide et efficace qui nous permet de savoir quelles interprétations voisines peuvent falsifier le moins de contraintes, nous avons implémenté plusieurs algorithmes que nous avons ajouté dans WalkPB puis expérimentés. Tout d'abord, nous avons implémenté un algorithme naïf qui consiste à recalculer l'ensemble des contraintes du problème pour chaque interprétation voisine de l'interprétaion courante. Ensuite, nous avons mis en oeuvre une méthode qui se limite à recalculer uniquement les occurrences dans lesquelles les variables apparaissent. Enfin, afin d'éviter le plus de calculs nous avons implémenté une approche qui utilise des compteurs.

4.2.1 Calcul du problème pour choisir la variable à flipper

Dans le cadre du problème peudo-booléen, la méthode de descente qui paraît évidente, afin de trouver un modèle, est de minimiser le nombre de contraintes falsifiées. Nous avons donc choisi de restreindre le choix des variables à flipper aux variables qui minimisent le nombre de contraintes falsifiées si elles sont flippées. Pour se faire, pour chaque variable de la contrainte séléctionnée on recalcule le problème en considérant qu'elle est flippée et on retient l'ensemble $\mathscr{E}$ des variables pour lesquelles on a trouvé le nombre minimum de contraintes falsifiées possible. Enfin, on choisit au hasard parmi l'ensemble $\mathscr{E}$ la variable que l'on va flipper.

Algorithme 7: Recalculer le problème

Données : Σ : PB, max_flip : int, max_tries : int
Résultat : le nombre de contraintes falsifiées, 0 si I est un modèle

```
1  début
2      nb_tries = 0 , nb_min = BIG_INT;
3      pour nb_tries de 0 à max_tries faire
4          Générer une interprétation initiale I;
5          pour nb_flip de 0 à max_flip faire
6              si I est modèle alors retourner 0;
7              𝓔 = ∅;
8              pour Chaque variable v de Σ faire
9                  I′ =flipper la variable v dans I;
10                 calculer le problème avec l'instance I′;
11                 si #false(I′, Σ) < nb_min alors
12                     nb_min = nombre de contraintes falsifiées par I';
13                     𝓔 = {v};
14                 sinon si #false(I′, Σ) = nb_min alors 𝓔 = 𝓔 ∪ v;
15             si 𝓔 <> ∅ alors
16                 v = une variable au hasard dans 𝓔;
17                 I ← flipper(v, I);
18             sinon I ← critère d'échappement;
19     retourner nombre minimal de contraintes falsifiées trouvé
20 fin
```

L'algorithme 7 illustre cette méthode. La méthode de calcul se situe entre les lignes 8 et 14. Pour chaque variable du problème nous la flippons et recalculons, sur l'ensemble du problème, le nombre de contrainte falsifiées par l'interprétation obtenue (lignes 9-10). Nous retenons uniquement celles qui falsifies le moins de contraintes. Enfin, nous choisissons une variable au hasard parmi l'ensemble des variables qui minimisent le nombre de contraintes falsifiées (ligne 16).

Cet algorithme, bien qu'il semble correct sur le papier n'est pas assez efficace en réalité. Pour cause, recalculer l'ensemble du problème pour chaque variable de celui-ci uniquement pour déterminer le nombre de contranites que va falsifier une variable si elle est flippée n'est pas gênant lorsqu'on a de petits problèmes à résoudre. Mais lorsque le nombre de variables et de contraintes augmentent, cet algorithme passe beaucoup trop de temps à calculer la variable qu'il doit flipper (environt 90% de son temps). La complexité de cet algorithme est en $O(m * n)$ où m représente le nombre de contrainte du problème et n le nombre maximum de variable qu'on peut trouver dans une contrainte.

4.2.2 Se concentrer sur les contraintes

Nous avons constaté que recalculer l'ensemble du problème pour évaluer si le flip de la variable est intéressant prend beaucoup trop de temps. Cependant, on peut remarquer que les variables n'apparaissent pas dans l'ensemble des contraintes du problème. L'idée est qu'au lieu d'évaluer l'ensemble des contraintes du problème pour chaque variable, on peut se limiter au calcul des contraintes dans lesquelles apparaissent les variables. En effet, si une variable n'apparait pas dans une conrainte, elle ne risque pas de falsifier ou satisfaire celle-ci. Ceci permet d'effectuer beaucoup moins de calculs et donc d'améliorer grandement le temps consacré au choix de la variable à flipper.

Pour ce faire, chaque variable possède un ensemble, appelé *occurrence*, qui contient l'ensemble des contraintes dans lesquelles elles apparaissent. Au moment de recalculer le problème, on parcourt uniquement cet ensemble pour chacune des variables. Pour implémenter cette approche nous avons quelque peu modifié notre solveur, nous avons ajouté l'ensemble des occurences pour chaque variable. Ceci est représenté par un tableau appellé *occurrence* ajouté dans la classe Probleme. La figure 4.2 illustre la nouvelle classe Probleme.

Probleme
+occurrence: unsigned int[][] +contraintesFalsifiees: Contrainte[]
+chargerProbleme(file:string) +evaluer(instance:bool[],variable:int): bool

FIGURE 4.2 – Classe Probleme

L'algorithme 8 décrit la méthode utilisée. La méthode de calcul se situe entre les lignes 9 et 16. Pour chaque variable du problème, nous la flippons puis recalculons les contraintes dans lesquelles la variable apparait (ligne 11). Puis, on retient l'ensemble des variables qui minimisent le nombre de contraintes falsifiées si on les

flips (ligne 12). Enfin, nous choisissons une variable au hasard parmi les variables de cet ensemble (ligne 17).

Algorithme 8: Concentré sur les contraintes

Données : Un ensemble Σ de contraintes, entier : max_flip, entier : max_tries
Résultat : le nombre de contraintes falsifiées, 0 si I est un modèle

```
1  début
2    nb_tries = 0 , nb_min = BIG_INT, nb_fals = BIG_INT;
3    pour nb_tries de 0 à max_tries faire
4      Générer une interprétation initiale I;
5      nb_fals = calculer le problème avec I;
6      pour nb_flip de 0 à max_flip faire
7        si I est modèle alors  retourner 0;
8        ℰ = ∅;
9        pour Chaque variable v de Σ faire
10         I′ = flipper la variable v dans I;
11         nb_fals = nb_min + #false(I′, Σ) − #true(I′, Σ);
12         si nb_fals < nb_min alors
13           nb_min = nb_fals;
14           ℰ = {v};
15         sinon si nb_fals = nb_min alors  ℰ = ℰ ∪ v;
16         flipper la variable v dans I;
17       si ℰ <> ∅ alors
18         v = une variable au hasard dans ℰ;
19         I ← flipper(v, I);
20       sinon  I ← critère d'échappement;
21   retourner le nombre minimal de contraintes falsifiées trouvé
22 fin
```

Bien que cela permet, la plupart de temps, de réduire la complexité de l'algorithme qui devient $O(max(occurence)*n)$ où $max(occurence)$ correspond au nombre maximum de contrainte dans laquelle une variable apparait et n le nombre maximum de variable dans une contrainte. Calculer, même uniquement les contraintes dans lesquelles les variables apparaissent, demande beaucoup trop de temps, et reste très coûteux lorsque le nombre de variables et de contraintes croît. L'algorithme passe encore trop de temps à chercher les variables qui falsifies le moins de conraintes si elles sont flippées.

4.2.3 Supprimer le maximum de calculs

Afin d'éviter de perdre beaucoup trop de temps à refaire les calculs, nous avons décidé de retenir les résultats de l'interprétation de départ, puis d'uniquement mettre à jour ces résultats au fur et à mesure que l'on effectue les flips. Ceci permet d'éviter

de recalculer l'ensemble des contraintes, ce qui permet d'améliorer grandement le temps passé à choisir la variable à flipper.

Pour se faire, l'interprétation initiale (générée aléatoirement) est considérée et le nombre de contrainte falsifiées est enregistré. Ensuite, on assigne deux compteurs à chaque variable du problème que l'on appellera *breakcount* et *makecount* chacun représentant respectivement, le nombre de contraintes que la variable va falsifier et satisfaire si on choisit de la flipper. Ces compteurs sont initialisés lors du premier calcul du problème avec l'interprétation de départ. Puis ils sont recalculés à chaque fois qu'une variable est flippée. Pour cela, il suffit, lors d'un flip, de mettre à jour les compteurs de l'ensemble des variables qui apparaissent dans les contraintes qui contiennent la variable flippée et on enregistre l'ensemble des variables qui possèdent le plus petit score. Enfin, nous choisissons une variable au hasard parmi les variables qui possèdent le plus petits score.

L'algorithme 9 décrit cette approche. La fonction objective se situe entre les lignes 8 et 13. On calcul le score de chaque variable du problème, c'est-à-dire l'écart entre son *makecount* et son *breakcount* (ligne 9). Puis, on retient l'ensemble des variables qui possèdent le plus petit score (lignes 10-13). Enfin, on choisie une variable au hasard parmi cet ensemble (ligne 14). Pour terminer on remet à jour les différents compteurs (ligne 15).

Algorithme 9: éviter les calculs

Données : Un ensemble Σ de contraintes, entier : max_flip, entier : max_tries

Résultat : le nombre de contraintes falsifiées, 0 si I est un modèle

```
1  début
2  │ nb_tries = 0, nb_min = BIG_INT, nb_fals = BIG_INT;
3  │ pour nb_tries de 0 à max_tries faire
4  │ │ Générer une interprétation initiale I;
5  │ │ nb_fals = calculer le problème avec I;
6  │ │ pour nb_flip à 0 à max_flip faire
7  │ │ │ si I est modèle alors  retourner 0;
8  │ │ │ pour Chaque variable v du problème faire
9  │ │ │ │ nb_fals = nb_min + v.breakcount − v.makecount;
10 │ │ │ │ si nb_fals < nb_min alors
11 │ │ │ │ │ nb_min = nb_fals;
12 │ │ │ │ └ ℰ = {v};
13 │ │ │ └ sinon si nb_fals = nb_min alors  ℰ = ℰ ∪ v;
14 │ │ │ choisir une variable v au hasard dans ℰ et la flipper;
15 │ │ └ Mettre à jour les compteurs des contraintes et des variables;
   │ └
16 │ retourner nombre minimal de contraintes falsifiées trouvé
17 fin
```

Même si elle n'améliore pas la complexité de l'algorithme précédent, cette méthode permet de réduire au maximum le nombre de calculs à effectuer pour choisir la variable à flipper et donc d'être plus rapide que l'approche précédente. Afin de connaître

les variables qui appartiennent à l'ensemble des variables pour lesquelles on va trouver le nombre minimum de contraintes falsifiées dans la contrainte si on les flips, il suffit de faire la différence entre le *breakcount* et le *makecount* des variables et d'y ajouter le nombre de contraintes falsifiées actuellement, ce qui réduit fortement le nombre d'opérations effectuées lors du choix. Cependant, les calculs que l'on effectuait lorsqu'on recalcule les contraintes sont reportés à la mise à jour des compteurs, sauf que ceux-ci ne sont effectués qu'une fois par flip : lorsque la variable flippée est choisie, ce qui réduit fortement le nombre d'opérations.

Afin de mettre à jour correctement les compteurs à chaque flip, nous avons implémenté une méthode qui permet de les recalculer efficacement suivant le type de contrainte considérée ($=, >, \geq, <, \leq$). Pour cela, chaque type de contrainte dispose d'un ensemble de variables que nous appellons variables importantes (Imp) qui a une signifiaction différente suivant sa valeur (est-elle falsifiée ou satisfaite). Si elle est falsifiée, alors cet ensemble représente l'ensemble des variables pouvant la satisfaire. Sinon (c'est-à-dire qu'elle est satisfaite), il représente l'ensemble des variables qui peuvent falsifier la contrainte. Ensuite, chaque contrainte possède une autre variable que nous appellons *différence*, que Dixon et al nomment [4] *poss* ou encore que d'autres nomment *slack*, cette variable correspond à la différence entre la somme des coefficients des littéraux qui ne sont pas falsifiés et le degré de la contrainte.

Enfin, chaque type de contraintes dipose aussi de trois fonctions principales :

- condMake : fonction qui prend en argument une variable et son interprétation actuelle et retourne vrai si le fait de flipper la variable permet de satisfaire la contrainte, faux sinon ;
- condBreak : fonction qui prend en argument une variable et son interprétation actuelle et retourne vrai si le fait de flipper la variable permet de falsifier la contrainte, faux sinon ;
- MAJ : cette fonction met à jour l'ensemble des compteurs de la contrainte.

Les fonctions condBreak et condMake sont identiques pour chaque type de contrainte, seul le signe du test dans les différents retours diffère.

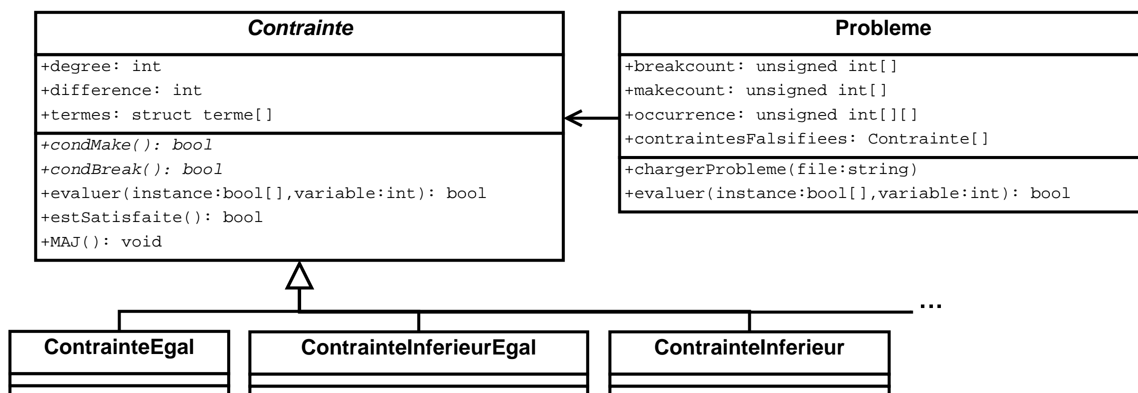

Figure 4.3 – Classes Contraintes

La figure 4.3 illustre les nouvelles classes utilisées pour le solveur. La classe Probleme dispose désormé de deux nouveaux compteurs, le *breakcount* et *makecount* des variables. La classe Contrainte possède un nouveau compteur appellé *différence*,

et trois nouvelles méthodes : condMake, condBreak et MAJ qui sont surchargées par chaque type de contrainte.

La fonction condMake permet de savoir si la variable passée en paramètre peut, en fonction de son affectation courante, satisfaire la contrainte si elle est flippée. Pour qu'une variable satisfait une contrainte il y a plusieurs conditions à prendre en compte. Tout d'abord, ces conditions dépendent de la variable (ligne 2), si elle est négative ce sont les mêmes tests dans lesquels les conditions sont inversées. Ensuite, cela dépend du littéral considéré (ligne 3), si elle est à $\mathbb{V}$ il faut enlever le coefficient de la variable à différence (ligne 4), sinon, il faut y ajouter sont coefficient et enfin tester si la contrainte est satisfaite (ligne 5).

Algorithme 10: condMake égale

Données : I : l'intérprétation courante de la variable, v : la variable flippée
Résultat : vrai si la variable peut satisfaire la contrainte si elle est flippée

```
1  début
2      si v est positive alors
3          si I[v] == vrai alors
4              retourner difference − v.coefficient == 0;
5          sinon  retourner difference + v.coefficient == 0;
6      sinon
7          si I[v] == vrai alors
8              retourner difference + v.coefficient == 0;
9          sinon  retourner difference − v.coefficient == 0;
10 fin
```

La fonction condBreak est exactement la même que condMake pour tous les types de contraintes, il suffit de remplacer le signe à chaque retour par son opposé.

Algorithme 11: condBreak égale

Données : I : l'intérprétation de la variable, v : la variable flippée
Résultat : vrai si la variable peut falsifier la contrainte si elle est flippée

```
1  début
2      si v est positive alors
3          si Y == vrai alors
4              retourner difference − v.coefficient! = 0;
5          sinon  retourner difference + v.coefficient! = 0;
6      sinon
7          si Y == vrai alors
8              retourner difference + v.coefficient! = 0;
9          sinon  retourner difference − v.coefficient! = 0;
10 fin
```

La fonction de mise à jour des compteurs de la contrainte (MAJ) est différente suivant le type de contrainte considérée. Tout d'abord, nous présentons l'algorithme

pour les contraintes de type égale. L'algorithme étant le même pour inférieur, inférieur ou égale, supérieur et supérieur ou égale modulo les tests d'inégalité, nous ne présenterons qu'un algorithme pour les quatres types (le cas inférieur). Ces fonctions sont utilisées pour mettre à jour l'ensemble des compteurs du solveur après chaque flip.

Algorithme 12: MAJ égale

Données : L'interprétation I actuelle des variables, la variable v flippée
Résultat : Mise à jour des compteurs de la contrainte

```
1  début
2      si v est positive alors
3          si I[v] == vrai alors  difference+ = c.coefficient;
4          sinon  difference− = c.coefficient;
5      sinon
6          si I[v] == vrai alors  difference− = c.coefficient;
7          sinon  difference+ = c.coefficient;
8      si contrainte était satisfaite alors
9          Imp = ∅;
10         pour chaque variable v de la contrainte faire
11             v.breakcount − −;
12             si condMake(I[v], v) alors
13                 v.makecount + +;
14                 Imp = Imp ∪ v;
15     sinon
16         pour chaque variable v de Imp faire v.makecount − −;
17         Imp = ∅;
18         si difference == 0 alors Si la contrainte est maintenant satisfaite
19             pour chaque variable v de la contrainte faire
20                 v.breakcount + +;
21                 Imp = Imp ∪ {v};
22         sinon
23             pour chaque variable v de la contrainte faire
24                 si condMake(I[v], v) alors
25                     v.makecount + +;
26                     Imp = Imp ∪ {v};
27 fin
```

La contrainte égale est illustrée par l'algorithme 12. Tout d'abord, entre les lignes 2 et 7, elle met à jour la *difference*. Selon l'interprétation courante de la variable et suivant son signe (positive ou négative), il faut ajouter ou soustraire son coefficient à la différence actuelle. Ensuite, nous avons deux cas possible (ligne 8). Si la contrainte était satisfaite (ligne 9), alors elle ne l'est plus et il faut décrémenter le *breakcount* des variables qui appartiennent à l'ensemble des variables importantes et il faut le remplacer en y ajoutant les variables qui peuvent satisfaire la contrainte

si elle est flippée, variables auxquelles nous incrémentons leurs *makecount*. Sinon (ligne 16), il faut décrémenter le *makecount* des variables importantes puis, nous avons deux possibilitées. Soit elle est devenue satisfaite, c'est-à-dire que *différence* est égale à zéro, il faut alors incrémenter le *breakcount* de l'ensemble des variables de la contrainte et les placer dans l'ensemble Imp (ligne 21). Soit elle est toujours falsifiée, ligne 26, il faut alors ajouter les variables qui peuvent la satifaire dans l'ensemble des variables importantes et incrémenter leur *breakcount*.

La contrainte inférieur est illustrée par l'algorithme 13. Tout d'abord, entre les lignes 2 et 7, tout comme pour la contrainte égale, elle met à jour la *difference*. Ensuite, nous réinitialisons les compteurs pour chaque variable importante (ligne 8). C'est-à-dire que si la contrainte était satisfaite nous décrémentons le *breakcount* de ces variables sinon nous décrémentons leur *makecount*. Enfin, nous avons deux cas. Si la contrainte est falsifiée (ligne 12), c'est-à-dire que *différence* est supérieur ou égale à zéro, alors nous incrémentons le *makecount* des variables de la contrainte qui peuvent la satisfaire et nous les ajoutons dans l'ensemble des variables importantes. Sinon (ligne 16), nous incrémentons le *breakcount* de chaque variable de la contrainte qui peuvent la falsifiée et les ajoutons dans l'ensemble Imp (ligne 21).

Algorithme 13: MAJ inférieur

Données : L'interprétation I actuelle des variables, la variable v flippée
Résultat : Mise à jour des compteurs de la contrainte

```
1  début
2      si v est positive alors
3          si I[v] == vrai alors  difference+ = c.coefficient;
4          sinon  difference− = c.coefficient;
5      sinon
6          si I[v] == vrai alors  difference− = c.coefficient;
7          sinon  difference+ = c.coefficient;
8      pour chaque variable v de Imp faire
9          si la contrainte était satisfaite alors  v.breakcount − −;
10         sinon  v.makecount − −;
11     Imp = ∅;
12     si difference ≥ 0 alors la contrainte est falsifiée
13         pour chaque variable v de la contrainte faire
14             si condMake(I[v], v) alors
15                 v.makecount + +;
16                 Imp = Imp ∪ {v};
17     sinon
18         pour chaque variable v de la contrainte faire
19             si condBreak(I[v], v) alors
20                 v.breakcount + +;
21                 Imp = Imp ∪ {v};
22 fin
```

4.2.4 Expérimentations

Afin de comparer de manière expérimentale les différentes approches proposées nous avons mené un ensemble d'expérimentations. Pour cela nous avons considéré le schéma de recherche locale WalkPB que nous avons tester sur les instances de la compétition PB'10 `http://www.cril.univ-artois.fr/PB10/`. Les résultats reportés dans le tableau 4.2.4 ont été mené sur des machines Intel Core 2 Quad avec un timeout d'une heure et un nombre maximum de 1 million de flips.

On peut clairement apercevoir la différence au niveau du nombre de flips effectués entre ces trois différents algorithmes. Tout d'abord, on peut voir que la première version dépasse le temps imparti pour quasiment l'ensemble des problèmes, seules les petites instances se terminent avant le timeout.

Ensuite, on peut constater une nette amélioration entre la première version ou l'on recalcule l'ensemble du problème pour chaque variable et lorsqu'on ne recalcule uniquement les contraintes dans lesquelles apparaissent les variables. On observe que, bien que le nombre de flips par seconde est nettement amélioré (entre cinq et six fois plus), cette méthode n'est pas encore suffisante, dans le sens où le temps pour calculer le nombre de contraintes qu'une variable falsifie si elle est flippée est encore beaucoup trop grand. Même si pour la plupart des problèmes il termine avant la fin du temps imparti, il dépasse toujours le temps imparti lors de plus grandes instances.

Finalement, on remarque une très nette amélioration entre la deuxième et la dernière version dans laquelle on effectue beaucoup moins de calculs, avec les *breakcount* et *makecount*. Le nombre de flips par seconde est multiplié par quatre voir seize par rapport à la version précédente, suivant les problèmes. On constate aussi que l'algorithme se termine avant la fin du temps imparti pour l'ensemble des problèmes testés.

			Méthode 1		Méthode 2		Méthode 3	
Nom	**#Var**	**#Cont**	**tps**	**#Flips**	**tps**	**#Flips**	**tps**	**#Flips**
army12.21ls	428	644	*time out*	265110	224.5	10^6	**6.77**	10^6
army9.12ls	266	401	*time out*	693901	121.68	10^6	**4.22**	10^6
dbst_v50_e1000_d25	4500	2559501	*time out*	6	*time out*	4745	**648.3**	10^6
elf.rf6.ucl	67	132	339.55	10^6	20.45	10^6	1.28	10^6
fpga10_10_sat_pb	150	130	325.05	271010	2.09	43786	**0.1**	43786
fpga40_40_sat_pb	2400	1720	*time out*	4724	*time out*	10^6	**24.59**	10^6
fpga45_44_sat_pb	2970	2113	*time out*	2869	*time out*	10^6	**31.28**	10^6
prob10	106	2	160.04	1000000	27.09	10^6	6.21	10^6
prob1	77	2	83.65	10^6	16.93	10^6	**4.61**	10^6
prob7	96	2	130.9	10^6	23.51	10^6	**5.68**	10^6
reduced-cuww2	70	2	52.1	753658	11.44	753658	**3.52**	753658
robin10	2025	410	*time out*	11541	1840.54	10^6	26.9	10^6
t2001.13queen13.1110967327	169	101	1183.36	10^6	90.38	10^6	6.72	10^6
t2001.13queen13.1110977464	169	101	1174.23	10^6	90.33	10^6	6.74	10^6
t3002.11tsp11.1900549974	231	2707	*time out*	157504	410.9	10^6	6.68	10^6
t3002.11tsp11.1900553606	231	2707	*time out*	159147	410.74	10^6	6.68	10^6
vdw_k3_l6_n1500	4500	674250	*time out*	15	*time out*	10551	**110.18**	372123

TABLE 4.1 – Résultats des expérimentations

4.3 Stratégies d'échappement des minima locaux

Pour la suite de ce TER, les expérimentations ayant montré l'efficacité de la méthode utilisant les compteurs par rapport aux autres, nous avons décidé de l'utiliser pour implémenter les stratégies d'échappement.

Dans cette partie, nous nous concentrons uniquement sur les méthodes utilisées pour s'échapper des minima locaux.

Le problème avec la recherche locale est qu'il arrive souvent que, lors des phases de descente, on tombe sur une interprétation qui ne permet plus de diminuer le nombre de contraintes falsifiées. On appelle cela un minimum local, c'est la plus petite valeur qu'on puisse avoir avec cette interprétation en ne considérant que les interprétations voisines. Le but du critère d'échappement est de s'éloigner suffisamment de cette interprétation afin de s'échapper du minimum local tout en évitant d'y retomber. Beaucoup de méthodes d'échappement ont été proposées. Lors de ce TER, nous avons adapté puis expérimenté plusieurs métaheuristiques venant de la recherche locale pour le problème SAT.

Contrairement à SAT le fait de flipper une variable dans une contrainte falsifiée ne garantit pas que cette contrainte soit satisfaite. En effet, dans le cadre de SAT puisque les contraintes sont interprétées comme une disjonction de littéraux, le fait de passer à vrai une variable permet de satisfaire la contrainte. Dans le cadre pseudo-booléen cela n'est pas garanti, par exemple, considérons la contrainte $101a + 50b - 2c + 25\overline{d} + 25\overline{e} = 100$ et l'interprétation $I = \{\neg a, \neg b, c, d, e\}$, dans cette configuration aucun flip seul ne permet de satisfaire la contrainte. Pour la satisfaire il sera nécessaire de flipper plus d'une variable.

Pour palier à ce problème nous avons introduit la notion de *différence*. La différence d'une contrainte représente l'écart entre la valeur de la somme des coefficients des littéraux vrais, suivant l'interprétation courante, et le degré de la contrainte. Dixion et al [4] nomment cette différence *poss*. Cette valeur permet de rapidement détecter si la contrainte est satisfaite, puisque, selon le type de contrainte, il suffit de regarder si la différence est négative, positive, ou nulle pour savoir si elle est satisfaite. De plus, cela permet de déterminer rapidement si une variable peut falsifier ou satisfaire une contrainte. En effet, il suffit, suivant le littéral (x ou $\neg x$) et sa valeur d'affectation ($\mathbb{V}$ ou $\mathbb{F}$), d'ajouter ou de soustraire le coefficient de celle-ci à la différence et de tester cette valeur en fonction du type de contrainte considérée. De plus, cette différence nous permet d'estimer si on se rapproche d'une interprétation permettant de satisfaire cette contrainte. Par exemple, considérons la contrainte d'égalité, pour être satisfaite sa différence devra tendre vers zéro. Donc même dans le cas où il n'existe pas d'interprétation voisine I' permetant de satisfaire la contrainte C on peut, à l'aide de ce principe, estimer si I' peut conduire à diminuer le nombre de mouvement utile afin de satisfaire C. Il suffit donc de maintenir à jour cette valeur pendant la recherche pour détecter si une contrainte est satisfaite et quelles variables peuvent la satisfaire ou la falsifier.

Dans un premier temps, nous avons implémenté et adapté plusieurs heuristiques provenant de SAT pour le problème pseudo-booléen. Puis, nous les avons modifiés afin d'y introduire notre notion d'amélioration de la différence afin de s'échapper d'un minimum local. Cette méthoe consiste à choisir aléatoirement une variable parmi l'ensemble des variables améliorant cette différence lorsqu'un minimum local est atteint. Enfin, ces différents algorithmes ont été testés sur plusieurs instances afin de constater si cette méthode permet d'améliorer les premières méthodes proposées.

4.3.1 Des stratégies issues de SAT

La recherche locale est un sujet très étudié dans le cadre de SAT. Par conséquent beaucoup d'algorithmes ont été étudiés. Nous avons sélectionné plusieurs approches

issues de SAT et nous les avons adaptées aux problèmes pseudo-booléen.

Best

Le principe de cette heuristique est de choisir la variable à flipper parmi l'ensemble des variables qui vont falsifier le moins de contraintes. Pour se faire, à chaque fois que l'on souhaite faire une réparation de l'interprétation, on choisit une contrainte au hasard et on retient les variables de cette contrainte qui vont falsifier le moins de contraintes si elles sont flippées. Si on ne peut pas flipper une variable sans falsifier de contrainte (minimum locale), avec une probabilité d'un demi on choisit la variable à flipper au hasard parmi toutes les variables de la contrainte. Sinon, on retourne une variable au hasard parmi l'ensemble calculé.

L'algorithme 14 illustre cette approche. Dans un premier temps, les variables ayant le plus petit breakcount sont sélectionnées (ligne 5). Ensuite, si aucune variable possède un breakcount à zéro, alors avec une probabilité d'un demi on choisit la variable à flipper au hasard parmi celles de la contrainte sélectionnée (ligne 10). Sinon, celle-ci est choisie parmi l'ensemble des variables qui possèdent le plus petit breakcount.

Algorithme 14: Best

Données : L'interprétation courante I
Résultat : La variable à flipper

```
1  début
2      best = BIGINT ;                    /* meilleur berakcount trouvé */
3      Σ = ∅ ;              /* ensemble des variables qu'on peut flipper */
4      C ← une contrainte falsifiée choisie au hasard;
5      pour chaque variable v de c faire
6          si v.breakcount ≤ best alors
7              si v.breakcount < best alors  Σ = ∅;
8              best = v.breakcount  Σ = Σ ∪ v;
9      si best > 0 alors
10         si random(0,1) < 0.5 alors
11             Choisir v une variable au hasard appartenant à C;
12             retourner v;
13     Choisir v une variable au hasard dans Σ;
14     retourner v;
15 fin
```

Random Walk Strategy

L'approche Random Walk Strategy suit une série de déplacements aléatoires afin de s'extraire des minima locaux. C'est-à-dire que suivant une certaine probabilité, généralement un demi, soit la variable à flipper est choisie parmi celles qui minimisent le nombre de contraintes falsifiées par le flippe de celle-ci, soit elle est choisie au hasard parmi celles qui possèdent un *makecount* positif, c'est-à-dire celles qui vont satisfaire au moins une contrainte si elles sont flippées. Cette stratégie est une des approches les plus connues dans le cadre de la recherche locale pour SAT.

L'algorithme 15 décrit cette stratégie. Pour commencer, le score des variables est calculé (ligne 2). Ce score représente la soustraction entre le *makecount* et le *breakcount* de la variable ce qui nous donne le nombre de contraintes falsifiées par celle-ci. Enfin, avec une probabilité de $\frac{1}{2}$ on choisit la variable à flipper au hasard (ligne 6), soit parmi celles qui minimisent le score, soit parmi celles qui possèdent un *makecount* positif.

Algorithme 15: Random Walk Strategy

Données : l'ensemble des contraintes Σ, l'interprétation courante I
Résultat : La variable à flipper

1 **début**
2 **pour chaque** variable v de Σ **faire**
3 $v.score = v.makecount - v.breakcount$;
4 Ω = ensemble des variables ayant le plus petit score;
5 φ = ensemble des variables tel que v.makecount > 0;
6 **si** $random(0, 1) < 0.5$ **alors** v = choisir une variable au hasard dans φ
sinon v = choisir une variable au hasard dans Ω **retourner** v;
7 **fin**

Novelty

Cette stratégie est une variante de *random walk strategy* et est déxrit dans l'algorithme 16.

Algorithme 16: Novelty

Données : L'interprétation courante I
Résultat : La variable à flipper

1 **début**
2 $C \leftarrow$ une contrainte falsifiée choisie au hasard;
3 **pour chaque** $v \in C$ **faire** $v.score = v.makecount - v.breakcount$;
4 $plusVielle \leftarrow$ la variable la plus vieille de C;
5 $meilleur \leftarrow$ la variable avec le plus grand score;
6 $seconde \leftarrow$ la variable avec le deuxième plus grand score;
7 $plusJeune \leftarrow$ la variable la plus récemment flippée;
8 **si** $meilleur.age <> plusJeune.age$ **alors**
9 $meilleur.age + +$;
10 **retourner** $meilleur$;
11 **si** $random(0, 1) < 0.5$ **alors**
12 $second.age + +$;
13 **retourner** $second$;
14 $meilleur.age + +$;
15 **retourner** $meilleur$;
16 **fin**

Elle classe les variables suivant leur score, c'est-à-dire suivant la différence entre le makecount et le breakcount. Un âge est donné à chaque variable correspondant au

dernier moment où elle a été flippée. Les variables sont classées suivant leur score et au momment du choix de la variable, seules les deux variables en tête sont étudiées. Si la meilleure variable n'est pas la plus récemment flippée (la plus vieille) elle est choisie. Sinon, avec une probabilité p, on choisit la seconde variable et avec une probabilité $1 - p$ on choisit la première.

RNovelty

Algorithme 17: RNovelty

```
    Données : L'interprétation courante I
    Résultat : La variable à flipper
 1  début
 2      C ← une contrainte falsifiée choisie au hasard;
 3      pour chaque v in C faire  v.score = v.makecount − v.breakcount;
 4      plusVielle ← la variable la plus vieille de C;
 5      meilleur ← la variable avec le plus grand score;
 6      seconde ← la variable avec le deuxième plus grand score;
 7      plusJeune ← la variable la plus récemment flippée;
 8      si meilleur.age <> plusJeune.age alors
 9          meilleur.age++;
10          retourner meilleur;
11      difference = meilleur.score − seconde.score;
12      p = random(0, 1);
13      si p < 0.5 et difference > 1 alors
14          meilleur.age++;
15          retourner meilleur;
16      si p < 0.5 et difference == 1 alors
17          si 2 * p > random(0, 1) alors
18              seconde.age++;
19              retourner seconde;
20          meilleur.age++;
21          retourner meilleur;
22      si p ≥ 0.5 et difference == 1 alors
23          seconde.age++;
24          retourner seconde;
25      sinon
26          si 2 * (p − 0.5) > random(0, 1) alors
27              seconde.age++;
28              retourner seconde;
29          meilleur.age++;
30          retourner meilleur;
31  fin
```

Cette stratégie est une variante de Novelty. Comme pour celui-ci, les variables sont classées selon leur score et seules les deux premières sont considérées. L'algo-

rithme 17 décrit cette stratégie. Les deux stratégies sont identiques sauf lorsque la meilleure variable est la plus récemment flippée. Dans ce cas, ligne 11, la différence entre les scores de la meilleure variable avec la seconde est considérée. Un nombre p est choisi au hasard et quatre cas sont distingués. Si $p < 0.5$ et que $difference > 1$, alors la meilleure variable est choisie, ligne 13. Sinon, ligne 16, si $p < 0.5$ et que $difference == 1$, alors avec la probabilité $2 * p$ la seconde variable est choisie, sinon la première est choisie. Sinon si $p \geq 0.5$ et que $difference == 1$, alors la seconde meilleure variable est choisie, ligne 22. Sinon, ligne 25, avec la probabilité $2*(p-0.5)$ la seconde variable est choisie, sinon la première est choisie.

4.3.2 Amélioration de la différence

Contrairement à SAT, il n'est pas possible de connaître facilement si le flip d'une variable peut falsifier ou satisfaire une contrainte. En effet, comme celles-ci possèdent des coefficients on ne peut pas connaître à l'avance quelles variables permettent de la satisfaire.

WalkPB avec différence

Dans un premier temps, nous avons ajouté cette stratégie à WalkPB. Nous l'avons implémenté de deux manières différentes. Pour la première, si on tombe dans un minimum local, avec une probabilité de $\frac{1}{2}$, soit une variable de la contrainte est choisie au hasard parmi l'ensemble des variables de celles-ci, soit elle est choisie au hasard parmi celles qui améliorent la différence. L'autre façon implémentée est que lorsqu'on tombe dans un minimum locale, la variable à flipper est toujours choisie au hasard parmi celles qui améliorent la différence.

Algorithme 18: WalkPB Différence

```
   Données : Σ : PB, max_flip : int, max_tries : int
   Résultat : le nombre de contraintes falsifiées, 0 si I est un modèle
1  début
2      pour nb_tries de 0 à max_tries faire
3          Générer une interprétation initiale I;
4          pour nb_flip de 0 à max_flip faire
5              si I est modèle alors retourner 0;
6              C ←Une contrainte au hasard parmi les contraintes falsifiées;
7              si ∃I′ permettant une descente alors I = I′;
8              sinon
9                  Ω = ∅;
10                 pour chaque variable v de Σ faire
11                     v.score = v.makecount − v.breakcount;
12                     si amélioreDifférence(v) alors Ω = Ω ∪ v;
13                 I′ = I où on a flippé la variable v ∈ Ω choisi aléatoirement;
14     retourner nombre de contraintes falsifiées par I
15 fin
```

L'algorithme 18 décrit la première méthode. Lorsqu'on tombe dans un minimum locale (ligne 10) on retient l'ensemble des variables de la contrainte qui améliorent la différence de celle-ci. Puis, on choisit une variable au hasard parmi celles-ci.

Nous avons ensuite appliqué cette approche à l'ensemble des stratégies que nous avons implémentées. Le principe est le même pour l'ensemble des stratégies, c'est-à-dire que lorsqu'on tombe dans un minimum local, avec une probabilité de $\frac{1}{2}$, soit on sélectionne la variable à flipper parmi celles qui améliorent la différence de la contrainte, soit on applique la stratégie d'échappement initiale.

L'algorithme 19 représente cette approche pour le cas de la stratégie Best. Seule la ligne 10 diffère par rapport à Best. Si on tombe dans un minimum local, c'est-à-dire qu'aucune variable ne peut être flippée sans falsifier de contrainte, alors avec une probabilité de $\frac{1}{2}$, soit l'ensemble des variables qui améliorent la différence de la contrainte sélectionnée est calculé, soit on applique la stratégie Best.

Algorithme 19: Best avec différence

Données : L'interprétation courante I
Résultat : La variable à flipper

```
1  début
2      best = BIGINT ;                    /* meilleur breakcount trouvé */
3      Σ = ∅ ;            /* ensemble des variables qu'on peut flipper */
4      C ← une contrainte falsifiée choisie au hasard;
5      pour chaque variable v de c faire
6          si v.breakcount ≤ best alors
7              si v.breakcount < best alors  Σ = ∅;
8              Σ = Σ ∪ v;
9      si best > 0 alors
10         si random(0,1) < 0.5 alors
11             Σ = ∅;         /* ensemble des variables qui améliorent la
               différence */
12             pour chaque variable v de C faire
13                 si le flip de v peut améliorer C alors  Σ = Σ ∪ v;
14     Choisir v une variable au hasard dans Σ;
15     retourner v;
16 fin
```

4.3.3 Expérimentations

Afin de comparer ces différentes stratégies d'échappement, nous avons mené un ensemble d'expérimentations. Pour ce faire, nous avons lancé chacune de ces stratégies sur le cluster du CRIL avec les même conditions que pour la compétition PB'10 (timeout de 1800s). Ceci nous as permis de comparer notre solveur à celui proposé par Cédric PIETTE lors de cette compétition (PB-Wave). La figure 4.3.3 illustre une partie des résultats de cette expérimentation.

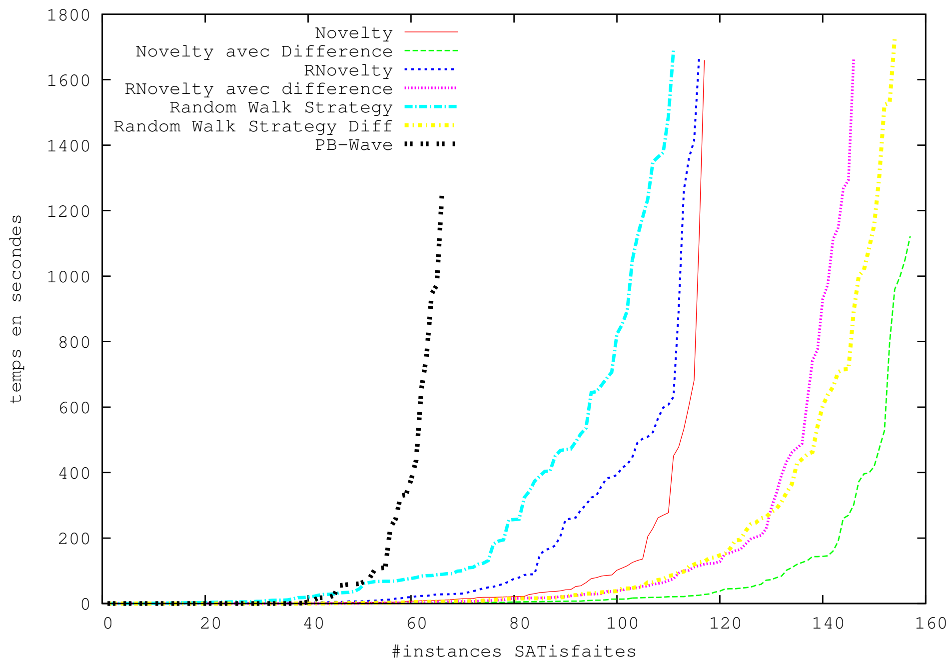

FIGURE 4.4 – Résultats des expérimentations

Nous pouvons constater que, l'ensemble des stratégies proposées permettent de résoudre plus d'instances que l'approche proposée par Cédric PIETTE. En effet, toutes les stratégies (même celles qui n'utilisent pas notre notion de différence) résolvent presque deux fois plus d'instances. De plus, nous pouvons constater que l'ajout de la différence permet d'améliorer sensiblement les résultats obtenus pour chacune des stratégies considérées. Cependant, on peut remarquer que l'amélioration apportée par la notion est plus ou moins notable suivant la stratégie considérée.

Chapitre 5

Conclusion

Lors du stage, nous avons cherché à adapter différents algorithmes de recherche locale aux problèmes pseudo-booléen. Pour cela, nous avons commencé par proposer puis expérimenter plusieurs méthodes pour calculer efficacement le nombre de contraintes qu'une variable va falsifier si elle est flippée. Les expérimentations ont montré que la meilleure façon de connaître rapidement l'état du problème par rapport à l'interprétation courante est d'éviter au maximum le calcul. Cela est possible grâce à l'utilisation de plusieurs compteurs tel que, principalement, le *breakcount* et *makecount* qui retiennent respectivement le nombre de contraintes que la variable va falsifier et satisfaire si elle est flippée. Ainsi que la *différence*, pour les contraintes, qui représente la différence entre la somme des coefficients des littéraux satisfaits et le degré de la contrainte.

Ensuite, nous avons adapté plusieurs stratégies d'échappement venant de la littérature sur le problème SAT au cadre du problème pseudo-booléen. Ces méthodes, comparativement à la méthode de recherche locale PB-wave soumis par Cédric PIETTE, fournissent des résultats encourageant. En effet, elles résolvent deux fois plus d'instances que celui-ci. Puis, nous avons introduit la notion d'amélioration de la différence lorsque l'on tombe dans un minimum local. Cette notion a été ajoutée aux stratégies classiques et a permis d'améliorer fortement les résultats obtenus vis-à-vis de celles-ci. Les expérimentations ont montré que cette notion permet d'améliorer, parfois significativement, ces stratégies pour le problème pseudo-booléen.

Les perspectives à court terme seraient de chercher à améliorer le calcul de l'ensemble des variables améliorant la différence d'une contrainte. Par exemple, de la même manière que pour le *breakcount* et le *makecount*, la contrainte possèderait un nouvel ensemble : l'ensemble des variables qui améliorent sa différence, ce qui éviterait de le recalculer à chaque fois. Une autre voie qui permettrait d'améliorer notre solveur serait de prendre en compte des cas particuliers pour les contraintes sous forme de clause. En effet, les contraintes du type $x_1 + x_2 + x_3 + ... + x_n \geq 1$ forment la clause $x_1 \vee x_2 \vee x_3 \vee ... \vee x_n$. Ce type de contrainte est intéressant, car il permet de bénéficier d'autres études provenant de la recherche locale dans le cadre de SAT permettant de savoir plus facilement si le flip d'une variable peut falsifier ou satisfaire la contrainte.

Ce TER m'a permis d'apprendre de nouvelles choses, les problèmes pseudo-booléens, les algorithmes de recherche permettant de résoudre ces problèmes, bien

entendu, principalement la recherche locale mais j'ai aussi eu l'occasion d'apercevoir d'autres méthodes tel que la méthode complète DPLL que j'ai souvent retrouvée lors de mes recherches. Enfin, cela m'a permis de découvrir une partie du monde de la recherche et m'a donné envie d'aller plus loin dans ce domaine.

Bibliographie

[1] Peter Barth. A davis-putnam based enumeration algorithm for linear pseudo-boolean optimization. 1995.

[2] Alexander Bockmayr and Friedrich Eisenbrand. Combining logic and optimization in cutting plane theory. In FroCos, pages 1–17, 2000.

[3] A Colorni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant colonies. In European Conference on Artificial Life, 1991.

[4] Heidi E. Dixon and Matthew L. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver. In Processing of The Eighteenth National Conference on Artificial Intelligence (AAAI-2002), pages 635–640, 2002.

[5] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into sat. Journal on Satisfiability, Boolean Modeling and Computation, 2 :1–26, 2006.

[6] Fred Glover and Manuel Laguna. Tabu Search. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[7] D. E. Goldberg. Algorithmes génétiques. exploration, optimisation et apprentissage automatique. Addison-Wesley France, France, 1994.

[8] R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. Bulletin of the American Society, 64 :275–278, 1958.

[9] Scott Kirkpatrick, D. Gelatt Jr., and Mario P. Vecchi. Optimization by simulated annealing. Science, 220(4598) :671–680, 1983.

[10] A. H. Land and A. G Doig. An automatic method of solving discrete programming problems. Econometrica, 28(3) :497–520, 1960.

[11] Daniel Le Berre and Anne Parrain. À propos de l'extension d'un solveur SAT pour traiter des contraintes pseudo-booléennes. In Troisièmes Journées Francophones de Programmationpar Contraintes (JFPC07), JFPC07, INRIA, Domaine de Voluceau, Rocquencourt, Yvelines France, June 2007.

[12] Joachim Paul Walser. Solving linear pseudo-boolean constraint problems with local search. In Proceedings of the fourteenth National Conference on Artificial Intelligence (AAAI-97), 1997.