ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Projet de semestre

Département de mathématiques

Méthode de recherche locale pour 3-SAT

David SCHINDL — Thomas GAUGLHOFER

Sous la direction de Monsieur le Professeur
Alain HERZ
Assistant: Michael GERBER

Le 15 février 1999

Table des matières

I Le projet 3

1 Introduction 5
- 1.1 Problèmes de satisfiabilité 5
- 1.2 Résolution heuristique: GSAT 5
- 1.3 Transition de phase 6
- 1.4 But du projet 6

2 Méthodes de résolution utilisées 7
- 2.1 Algorithme de Davis-Putnam [4] 7
- 2.2 Algorithme S-G-Sat 8

3 Les programmes S-G-Sat et D-Put 11
- 3.1 Génération des problèmes: generation.h 11
- 3.2 S-G-Sat 12
- 3.3 D-Put 14

4 Description des tests effectués et résultats 15
- 4.1 Pondérations constantes 15
- 4.2 Changements des paramètres max_essais et max_iter 16
- 4.3 Solution initiale 16
- 4.4 Pondérations variables 17

5 Analyse des résultats et conclusion 19
- 5.1 Analyse 19
- 5.2 Conclusion 20

II Annexes 21

A Code source: generation.h 23

B Code source: S-G-Sat.c 25

C Code source: D-Put.c 33

Première partie

Le projet

Chapitre 1

Introduction

1.1 Problèmes de satisfiabilité

Les problèmes de satisfiabilité se rencontrent dans de nombreux domaines des mathématiques et de l'informatique (par exemple la coloration de graphes, "N-queens" ou l'intelligence artificielle [1],[2]). De quoi s'agit-il?

Un problème de satisfiabilité est une expression booléenne que l'on cherche à satisfaire. Cette expression contient plusieurs clauses séparées par $\wedge$ ("et") et chaque clause se compose de littéraux (une variable logique ou sa négation) séparés par $\vee$ ("ou"). Pour satisfaire une telle expression, il faut qu'au moins un littéral soit satisfait dans chaque clause.

Exemple:

$$(v_3 \vee \overline{v_2} \vee v_4 \vee \overline{v_1}) \wedge (v_1 \vee v_2) \wedge (v_2 \vee v_1 \vee \overline{v_4}) \wedge (v_2 \vee \overline{v_4})$$

Solution admissible:
$v_1 = 1, v_2 = 1, v_3 = 1, v_4 = 1$

Solution non-admissible:
$v_1 = 0, v_2 = 0, v_3 = 1, v_4 = 1$

Dans notre cas (3-SAT), les clauses contiennent chacune trois littéraux différents; une variable et sa négation ne pouvant pas être dans la même clause. Il a été montré que ce problème est NP-complet (par S.A. Cook [3]), c'est à dire. un algorithme de résolution exact prendra au moins un temps proportionnel à l'exponentielle de la "taille" du problème.

1.2 Résolution heuristique: GSAT

Il existe deux approches pour résoudre ce genre de problème: la manière exacte et l'heuristique. Cette dernière consiste à sacrifier quelque chose: en restreignant le champ d'application (dans notre cas on n'admettrait que certains problèmes), en admettant des réponses erronées (mais peu fréquentes), en utilisant des réponses par défaut si on se trouve dans un cas "dur"...

B. Selman, H. Levesque et D. Mitchell ont mis au point une heuristique pour des problèmes de satisfiabilité qu'ils ont appelée GSAT [1]. Cette heuristique cherche un minimum local de la fonction objectif définie par le nombre de clauses non satisfaites, à partir d'une solution initiale choisie aléatoirement. A chaque étape de l'algorithme, GSAT change l'affectation d'une seule variable (de "vrai" à "faux" ou vice-versa),

de manière à ce que le nombre de clauses satisfaites soit le plus grand possible. Cette étape est répétée jusqu'à ce que GSAT trouve une solution ou jusqu'à ce que le nombre de répétitions dépasse une certaine borne prédéfinie ("MAX-FLIPS"). Si nécessaire cette procédure est répétée avec une nouvelle solution initiale aléatoire jusqu'à ce que GSAT trouve une solution ou jusqu'à ce que le nombre "MAX-TRIES" de répétitions soit atteint.

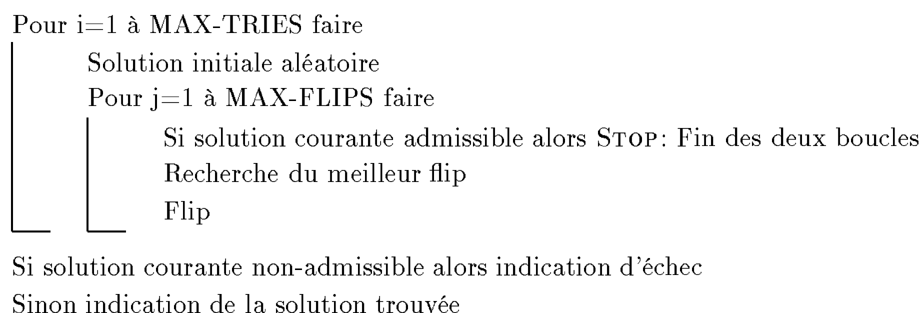

FIG. 1.1 – *Schéma de l'algorithme GSAT*

1.3 Transition de phase

B. Selman, H. Levesque et D. Mitchell ont montré statistiquement [2] que pour des problèmes 3-SAT, il existe une forte transition de phase; c'est à dire que si l'on fixe un nombre n de variables et que l'on fait varier le nombre de clauses m, on observe le comportement suivant:

Pour m entre zéro et une certaine valeur critique, la probabilité qu'il existe au moins une affectation des variables qui satisfait le problème est très grande (le problème est sous-contraint). Puis, autour de cette valeur critique, la probabilité de satisfiabilité diminue rapidement, de sorte que la probabilité est proche de zéro pour des m supérieurs à la valeur critique (le problème est sur-contraint).

B. Selman, H. Levesque et D. Mitchell ont observé que ce saut de probabilité de satisfiabilité ne dépend que du rapport m/n et qu'il se produit autour de $m/n \approx 4.3$ (pour 3-SAT), le point où (en moyenne) 50% des problèmes sont satisfiables. Il est évident qu'avec une méthode heuristique, on arrive facilement à trouver une solution admissible pour des problèmes sous-contraints, d'autre part, avec un algorithme exact, on montre vite la non-satisfiabilité de problèmes sur-contraints. Pour notre projet, on va donc se restreindre à des problèmes de rapport $m/n = 4.3$.

1.4 But du projet

Le but de notre projet est de tester un nouveau voisinage. En introduisant la possibilité d'affecter une variable à "vrai et faux", on espère augmenter le nombre de problèmes résolus. En autorisant cette double affectation on satisfera plus de clauses. Mais, comme une solution avec des doubles affectations n'est pas admissible, il faut introduire une pénalité. On propose une fonction objectif de la forme "$a \times \#$(clauses non satisfaites)$+b \times \#$(variables doublement affectées)". On essayera de trouver les meilleurs paramètres a et b, éventuellement de les faire varier avec le nombre d'itérations. Pour savoir quel pourcentage des problèmes solvables est trouvé par notre algorithme S-G-Sat, on programmera aussi un algorithme exact; celui de M. Davis et H. Putnam [4].

Chapitre 2

Méthodes de résolution utilisées

2.1 Algorithme de Davis-Putnam [4]

Cet algorithme détermine si un problème de satisfiabilité admet une solution ou non, et cela de manière certaine. Si le problème possède au moins une solution, il en donne une, sinon il indique que le problème n'est pas satisfiable. Il se base sur un branchement des variables en utilisant récursivement les règles de simplification suivantes:

1. *la règle tautologique*: toute clause contenant une variable et sa négation est éliminée (cette règle n'est jamais appliquée dans notre cas, car les problèmes générés ne contiennent pas de telles clauses).

2. *la règle du littéral pur*: lorsqu'un littéral n'apparaît que sous une forme dans tout le problème, alors il est affecté à "vrai".

3. *la règle de propagation unitaire*: s'il existe une clause unitaire, alors on affecte la variable de cette clause à "vrai" et on élimine toutes les clauses la contenant ainsi que toutes les négations de cette variable, si celles-ci ne se trouvent pas dans une clause unitaire. Dans le cas contraire, on doit quitter ce branchement car une solution est alors impossible.

4. *la règle de séparation*: cette règle n'est appliquée que lorsqu'il n'y a plus d'autres simplifications possibles. Elle choisit un littéral pas encore affecté et le fixe une fois à "vrai" et une fois à "faux". L'algorithme reprend alors au point 2.

Si toutes les clauses sont éliminées, alors on a trouvé une solution admissible (il se produit parfois que certaines variables ne sont pas affectées, dans ce cas n'importe quelle affectation de ces variables conduit à une solution).

Sur la page suivante se trouve le schéma de l'algorithme qu'on a utilisé pour le mettre en code (la règle tautologique figure sur le schéma mais elle n'a pas été utilisée dans le code).

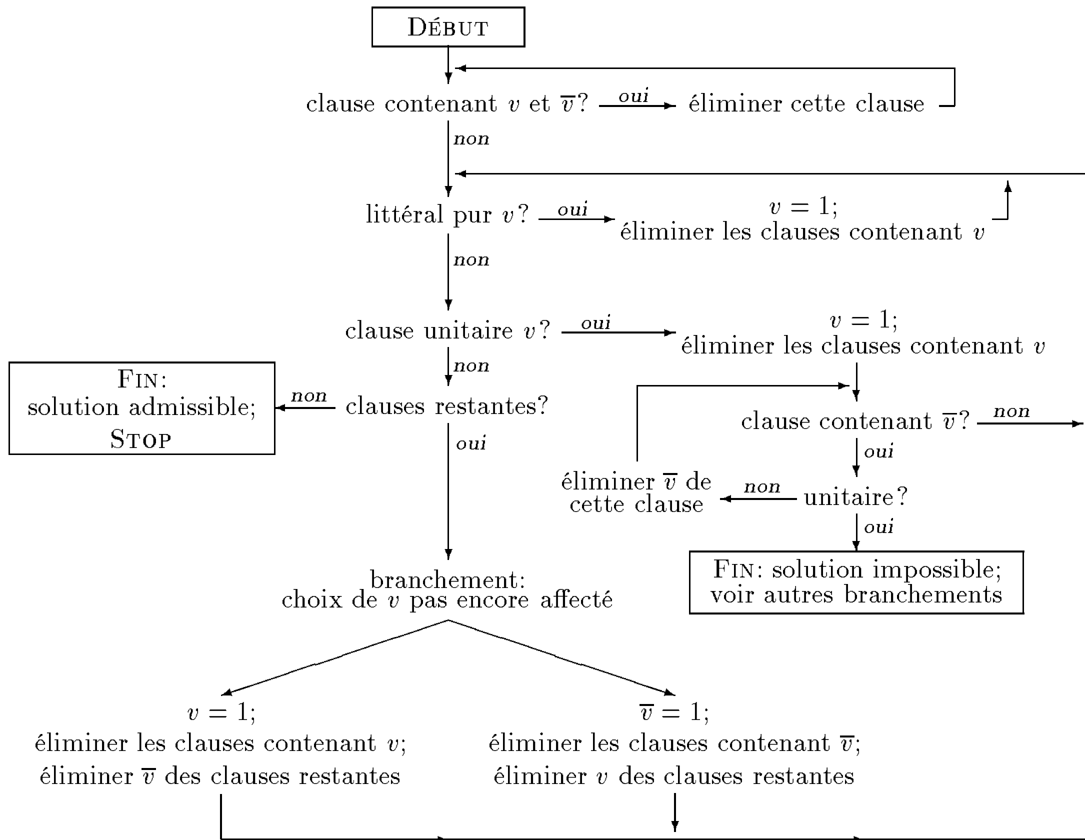

FIG. 2.1 – *Schéma de l'algorithme Davis-Putnam*

2.2 Algorithme S-G-Sat

Comme GSAT, sa modification (qu'on appellera S-G-Sat) est une heuristique qui cherche un minimum local d'une fonction objectif à partir d'une solution initiale choisie aléatoirement. Le changement principal est qu'on admet des affectations "doubles", c'est à dire des variables qui sont affectées à la fois à "vrai" et "faux". De plus, au lieu de minimiser le nombre de clauses non satisfaites, elle minimise la somme du nombre de clauses non satisfaites et du nombre de variables doublement affectées, pondérée par les coefficients a et b:

$$a \times \#(\text{clauses non satisfaites}) + b \times \#(\text{variables doublement affectées})$$

A chaque étape de l'algorithme, S-G-Sat change l'affectation d'une seule variable, de manière à ce que la fonction objectif diminue le plus possible. Cette étape est répétée jusqu'à ce que S-G-Sat trouve une solution (c'est à dire. une affectation qui satisfait toutes le clauses sans qu'il y ait une variable doublement affectée). Si l'algorithme ne trouve pas de telle solution avant que le nombre de répétitions ne dépasse une certaine borne prédéfinie ("max_iter"), alors la procédure est répétée avec une nouvelle solution initiale aléatoire (dans ces solutions initiales on admet a priori aussi des variables doublement affectées) jusqu'à ce que S-G-Sat trouve une solution ou jusqu'à ce qu'on ait répété cette procédure un nombre "max_essais" de fois.

```
Pour i=1 à "max_essais" faire
    Solution initiale aléatoire
    Pour j=1 à "max_iter" faire
        Si solution courante admissible alors STOP: Fin des deux boucles
        Recherche du meilleur flip
        Flip
Si solution courante non-admissible alors indication d'échec
Sinon indication de la solution trouvée
```

FIG. 2.2 – *Schéma de l'algorithme S-G-Sat*

Chapitre 3

Les programmes S-G-Sat et D-Put

3.1 Génération des problèmes: generation.h

Le fichier "generation.h" contient le code source en C++ (voir annexes) d'un générateur de nombres aléatoires entre zéro et un, ainsi qu'une fonction "Genere3sat" qui crée un problème de 3-satisfiabilité.

Ce problème à n variables et m clauses est stocké de deux façons différentes dans les variables globales "Variable" et "Clause" où n et m sont des constantes entières (que l'on définit avant la compilation).

Le tableau "Clause", de dimensions $(m+1) \times 4$, contient dans chaque ligne (*à partir de la ligne 1!*) en position zéro l'entier "3" (utilisé dans le programme D-Put), puis une clause du problème. Cette clause est stockée de la manière suivante:

- pour la variable v_i on met l'entier positif i,
- pour une négation on multiplie l'entier par -1.

La clause "$v_8 \vee \overline{v_6} \vee v_3$" est donc stockée dans "Clause[i][0], Clause[i][1], Clause [i][2], Clause[i][3]" sous la forme "3, 8, -6, 3", où i est le numéro de la clause.

Le tableau "Variable" contient dans les lignes $2i-1$ et $2i$ l'ensemble des clauses dans lesquelles apparaît le littéral correspondant à la variable i, respectivement à sa négation. En position zéro de chacune de ces lignes se trouve le nombre de clauses ou apparaît cette variable (ou sa négation). Par exemple, les nombres "3, 2, 7, 16; 1, -5" stockés dans "Variable[$2i-1$][0], Variable[$2i-1$][1], Variable[$2i-1$][2], Variable[$2i-1$][3]; Variable[$2i$][0], Variable[$2i$][1]" signifient que la variable i apparaît dans 3 clauses: la numéro 2, la numéro 7 et la numéro 16; sa négation dans une seule clause: la numéro 5. Remarquons que dans les lignes paires de "Variable", les numéros des clauses sont multipliés par -1, pour rappeler qu'elles contiennent des négations de variables. Pour faire ceci, on a utilisé l'allocation dynamique de place de mémoire (définition de "Variable" par "`int **Variable`"; initialisation de la colonne zéro; remplissage du reste au fur et à mesure).

Les littéraux sont générés aléatoirement, de manière équiprobable (grâce à la fonction "uniforme_reel"), en prenant garde de ne jamais avoir deux fois le même littéral dans une même clause, que ce soit avec le même signe ou non (d'abord, une des n variables est choisie uniformément; cette étape est répétée jusqu'à ce que l'on ait trois variables distinctes, puis chacune d'elles devient une négation avec une chance sur deux).

On a employé le générateur de nombre aléatoire “uniforme_reel” qui crée une suite de nombres pseudo-aléatoires (“germe”) à partir desquels sont calculés des réels compris entre zéro et un.

Dans “generation.h”, on définit en outre la constante “max_prob” (le nombre de problèmes que l'on essayera de résoudre) et les variables globales “Succes” (qui compte le nombre de problèmes résolus) et “Germe_inst” (le “germe” pour les problèmes).

3.2 S-G-Sat

Le programme S-G-Sat.c est la mise en code de l'algorithme GSAT modifié, décrit en chapitre 2.

Avant la description des différentes fonctions de ce code, on va présenter les plus importantes constantes et variables globales. On s'est servi entre autres des deux constantes suivantes:

1. “indice_init” qui détermine la façon dont on génère la solution initiale: La variable “i” est affectée à zéro avec probabilité “indice_init”, à “i” et “$-i$”avec probabilités (1-“indice_init”)/2 chacunes;
2. “indice_fonc” qui détermine le type de fonction objectif:

   0 : pour que les pondérations a et b soient constantes,

   1 : pour que b croisse linéairement avec le nombre d'itérations,

   2 : pour que b croisse exponentiellement avec le nombre d'itérations.

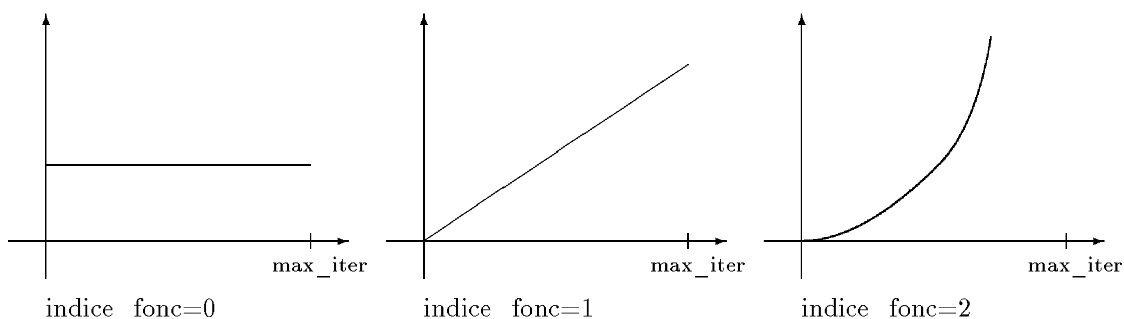

FIG. 3.1 – *b en fonction du nombre d'itérations*

Les variables globales que l'on a utilisées sont, entre autres:

`Sol`**:** vecteur de longueur $n+1$ contenant la solution courante sous la forme suivante: “i” en position i donne l'affectation “vrai” à la $i^{\text{ème}}$ variable, “$-i$” lui donne l'affectation “faux” et “0” lui donne l'affectation “vrai et faux”;

`Sat_cl`**:** vecteur de longueur $m+1$ donnant en position i le nombre de littéraux satisfaits dans la $i^{\text{ème}}$ clause;

`Unsat`**:** vecteur de longueur 4, contenant en position i le nombre de clauses i fois satisfaites, pour i allant de 0 à 2, et en position 3 le nombre de variables doublement affectées;

`Flip_pos` **:** tableau de dimensions $3 \times (n+1)$ contenant dans la case $[i][j]$ la variation du nombre de clauses qui seraient satisfaites i fois si on changeait la $j^{\text{ème}}$ variable en positif (cette case prend la valeur "infini" si la variable a déjà l'affectation positive);

Les variables globales que l'on a utilisées sont, entre autres:

`Sol` **:** vecteur de longueur $n+1$ contenant la solution courante sous la forme suivante: "i" en position i donne l'affectation "vrai" à la $i^{\text{ème}}$ variable, "$-i$" lui donne l'affectation "faux" et "0" lui donne l'affectation "vrai et faux";

`Sat_cl` **:** vecteur de longueur $m+1$ donnant en position i le nombre de littéraux satisfaits dans la $i^{\text{ème}}$ clause;

`Unsat` **:** vecteur de longueur 4, contenant en position i le nombre de clauses i fois satisfaites, pour i allant de 0 à 2, et en position 3 le nombre de variables doublement affectées;

`Flip_pos` **:** tableau de dimensions $3 \times (n+1)$ contenant dans la case $[i][j]$ la variation du nombre de clauses qui seraient satisfaites i fois si on changeait la $j^{\text{ème}}$ variable en positif (cette case prend la valeur "infini" si la variable a déjà l'affectation positive);

`Flip_neg`, `Flip_libre` **:** analogue à "Flip_pos";

`Best_flip` **:** tableau de dimensions $3 \times (2n+1)$ contenant une liste des meilleurs flips: dans la case $[1][j]$ le numéro de la variable et dans la case $[2][j]$ la future affectation. Dans la case $[0][0]$ se trouve le nombre de ces meilleurs flips.

Le code consiste en une partie principale ("main") qui règle l'application de l'algorithme à plusieurs problèmes et dans laquelle sont appelées les trois fonctions suivants:

1. "Genere3sat"(voir section précédente)
2. "Ecrit3sat" qui effectue les affichages des principaux tableaux à l'écran (sert essentiellement à contrôler le bon fonctionnement du code);
3. "Pensat" qui règle le bon déroulement de l'algorithme en utilisant différentes fonctions, soit dans l'ordre de programmation:

   `remplit_flip(i)` **:** remplit les $i^{\text{èmes}}$ colonnes des tableaux "Flip_pos", "Flip_neg" et "Flip_libre";

   `Sol_init` **:** crée une solution initiale aléatoire dépendant du choix de "indice_init";

   `Effectue_flip` **:** met à jour les variables "Sat_cl", "Unsat", "Flip_pos", "Flip_neg" et "Flip_libre" en faisant appel à "remplit_flip";

   `Determine_meill_flips` **:** remplit le tableau "Best_flip";

   `Ponda` et `Pondb` **:** définissent la fonction objectif selon "indice_fonc". La valeur a (dans "Ponda") et la valeur maximale de b (dans "Pondb") sont à choisir avant la compilation. Cette dernière valeur signifie

   - pour "indice_fonc"=0 : la valeur de la fonction constante,
   - pour "indice_fonc"=1 : la valeur b("max_iter") de la fonction linéaire qui vaut zéro en zéro,
   - pour "indice_fonc"=2 : la valeur b("max_iter") de la fonction exponentielle qui vaut zéro en zéro.

3.3 D-Put

Comme S-G-Sat, D-Put (qui est la mise en code de l'algorithme Davis-Putnam) consiste en une partie principale ("main") qui règle l'application de l'algorithme à plusieurs problèmes en appelant les trois fonctions suivantes:

1. "Genere3sat" (voir section 3.1)
2. "Ecrit3sat" qui effectue l'affichage du problème à l'écran (sert essentiellement à contrôler le bon fonctionnement du code);
3. "Algorithme" qui est rien d'autre que la traduction en C++ du tableau 2.1; les fonctions appelées s'expliquent par leurs noms sauf peut-être la fonction "non_v" qui regroupe les étapes concernant la négation d'une variable lorsque celle-ci (la variable, pas sa négation) se trouve dans une clause unitaire.

La plupart des variables utilisées sont des variables locales vu la récursivité de l'algorithme, mais il ne sont en gros que des copies des variables globales suivantes:

`Sol`: un vecteur de taille $n+1$ contenant la solution courante ("0" signifie que la variable n'a pas encore été affectée);

`Fin`: indique la fin de l'algorithme:

- "0" si l'algorithme n'a pas encore fini,
- "-1" si l'algorithme a trouvé un branchement impossible,
- "1" si l'algorithme a trouvé une solution admissible;

`Clause`: défini en "generation.h"; sa colonne zéro sert à indiquer le nombre de littéraux se trouvent à chaque étape dans les clauses.

Notons aussi la variable locale "variab" de taille $2n+1$ qui est une copie de la colonne zéro de "Variable" (définie en "generation.h"). Son $2v-1^{\text{ème}}$ coefficient est le nombre de clauses dans lesquelles il reste le littéral "v", son $2v^{\text{ème}}$ coefficient le nombre de clauses dans lesquelles il reste le littéral "$\overline{v}$".

Chapitre 4

Description des tests effectués et résultats

Pour trouver les paramètres, on a effectué quatre séries de tests:

1. à pondérations constantes;
2. en changeant les paramètres max_essais et max_iter;
3. en changeant la façon de générer la solution initiale.;
4. avec des pondérations variables.

Tous ces tests ont étés effectués sur la même suite aléatoire de 1000 problèmes à 100 variables et 430 clauses (sauf indication contraire max_iter = 500, max_essais = 100, indice_init = 0.33..., indice_fonc = 0) et on a comparé les résultats avec ceux du programme D-Put (470 succès en 41713 sec. = 11h 35min 13sec.). On a indiqué les temps d'exécution pour l'ensemble des 1000 problèmes.

4.1 Pondérations constantes

La question que l'on se posait était de savoir quel quotient (a/b; cf. chapitre 2) serait le plus efficace. Nous avons donc effectué au total 7 tests, dont trois en maintenant a à 1 et en variant b, trois en maintenant b à 1 et en variant a puis un en posant a=b=1.

Les résultats de ces tests se trouvent dans le tableau suivant:

a	b	a/b	nbre de succès	temps en sec.
1	10	0.1	438	5759
1	4	0.25	433	5767
1	2	0.5	435	5792
1	1	1	352	6862
2	1	2	174	7434
4	1	4	150	8824
10	1	10	150	8830

TAB. 4.1 – *Pondérations constantes*

4.2 Changements des paramètres max_essais et max_iter

Ici, il s'agissait de voir si une augmentation du nombre d'itérations ou du nombre d'essais produirait une amélioration significative du nombre de succès, avec une augmentation raisonnable du temps d'exécution. Nous avons pour cela testé les valeurs 500 et 1000 pour le nombre d'itérations (avec max_essais = 100) puis 100, 500 et 1000 pour le nombre d'essais (avec max_iter = 500).

Nous avons effectué chacun de ces tests avec $b = 2$, 4 et 10 ($a = 1$), d'une part pour avoir plus de tests et d'autre part dans l'espoir de trouver une différence d'efficacité entre ces trois possibilités, car les résultats du tableau 4.1 ne permettaient pas de trancher entre ces trois valeurs.

Dans les tableaux qui suivent, nous avons répertorié le nombre moyen d'itérations effectuées lors d'un essai menant à un succès (iter_moy), le nombre maximal de telles itérations (iter_max, à ne pas confondre avec max_iter décrit au chapitre 2), le nombre moyen d'essais effectués pour trouver une solution admissible (essais_moy) et le nombre maximal de tels essais (essais_max).

Voici les résultats que nous avons obtenus:

a	b	max_ iter	iter_ moy	iter_ max	max_ essais	essais_ moy	essais_ max	succès	temps en sec.
1	10	500	275	500	100	19	100	438	5759
1	4	500	286	500	100	19	100	433	5767
1	2	500	289	500	100	19	99	435	5792
1	10	1000	433	997	100	14	100	462	10525
1	4	1000	450	991	100	13	96	459	10486
1	2	1000	471	999	100	14	96	454	10774
1	10	500	278	495	500	31	432	470	24904
1	4	500	276	500	500	28	448	467	24946
1	2	500	288	500	500	27	350	468	25013
1	10	500	280	500	1000	31	553	468	48631
1	4	500	282	500	1000	26	456	470	48384
1	2	500	290	500	1000	32	476	470	46018

TAB. 4.2 – *Modifications de max_iter et max_essais*

4.3 Solution initiale

L'idée ici était de vérifier si une plus ou moins grande proportion de variables affectées à "0" dans la solution initiale avait une influence notable sur le nombre de succès. Nous avons donc fait des tests en mettant indice_init (cf section 3.2) à 0.8, puis à 0 (aucune variable doublement affectée dans la solution initiale).

Voici le tableau des résultats:

a	b	indice_ init	iter_ moy	iter_ max	essais_ moy	essais_ max	succès	temps en sec.
1	10	0.33...	275	500	19	100	438	5759
1	4	0.33...	286	500	19	100	433	5767
1	2	0.33...	289	500	19	99	435	5792
1	20	0.8	313	498	18	98	439	5722
1	10	0.8	313	498	18	98	439	5727
1	4	0.8	304	500	19	96	443	5711
A suivre...								

Suite:								
a	b	indice init	iter_ moy	iter_ max	essais_ moy	essais_ max	succès	temps en sec.
1	2	0.8	310	500	21	100	433	5906
1	10000	0	299	499	19	100	330	6456
1	10	0	299	499	19	100	330	6469
1	4	0	299	497	19	99	333	6452
1	2	0	310	500	19	96	323	6591

TAB. 4.3 – *Variations de la solution initiale*

4.4 Pondérations variables

Cette fois-ci, nous avons voulu tester si le fait de faire varier b avec le nombre d'itérations pourrait donner des résultats intéressants. Pour ceci, nous avons utilisé des fonctions croissantes, car il est important, à la fin d'un essai, de ne plus avoir de variables doublement affectées dans la solution. Nous avons donc choisi une fonction linéaire (b proportionnel au nombre d'itérations) et une fonction exponentielle (b proportionnel à exp(nombre d'itérations); cf section 3.2).

a	b(max_iter)	indice_ fonc	iter_ moy	iter_ max	essais_ moy	essais_ max	succès	temps en sec.
1	10	0	275	500	19	100	438	5759
1	4	0	286	500	19	100	433	5767
1	2	0	289	500	19	99	435	5792
1	20	1	298	500	18	100	447	5624
1	8	1	310	499	20	100	442	5735
1	4	1	346	499	21	99	425	5893
1	3.7×10^{109}	2	Batch pas encore arrivé					
1	1.6×10^{109}	2	385	500	22	99	418	5984
1	7.4×10^{108}	2	385	500	23	100	424	6007

TAB. 4.4 – *Pondérations variables*

Chapitre 5

Analyse des résultats et conclusion

5.1 Analyse

On a effectué la première série de tests sur un spectre assez large de rapports (a/b) et on voit clairement que les trois premiers tests du tableau 4.1 donnent des résultats beaucoup plus satisfaisants que les quatre autres. On n'a donc considéré par la suite que des rapports inférieurs à 1. On a essayé dans les tests suivants de voir des différences de qualité entre les tests avec les valeurs $b = 2$, 4 et 10, en maintenant a à 1.

Concernant les tests sur max_iter, on constate que le nombre de problèmes satisfiables non résolus (470 - #(problèmes résolus)) a considérablement diminué quand on est passé à max_iter=1000, avec une multiplication du temps de calcul par deux (environ).

Dans les tests suivants, on a commencé par faire marcher l'algorithme avec le paramètre max_essais à 1000, ce qui a donné de très bons résultats (maximum de succès atteint), mais dans un temps démesuré, même supérieur à celui pris par l'algorithme Davis-Putnam pour les mêmes problèmes. Cependant les valeurs bien inférieures à 1000 de essais_max nous ont permis de conclure que cette valeur (1000) était trop élevée et on a décidé de refaire les tests avec max_essais=500, dont les résultats (plus satisfaisants) se trouvent dans le tableau 4.2.

Les tests faisant intervenir différentes constructions de solutions initiales (cf section 3.2; indice_init) nous ont fourni des résultats assez révélateurs: l'algorithme GSAT (S-G-Sat avec indice_init = 0 et b très grand) a livré l'un des pires résultats de l'ensemble des tests effectués. D'autres valeurs pour b (toujours avec indice_init = 0) n'ont d'ailleurs pas non plus donné de résultats satisfaisants. Par contre, les tests avec indice_init = 0.8 ont confirmé le bon fonctionnement de notre approche, même si l'amélioration par rapport aux résultats obtenus avec indice_init = 0.33... est insignifiante. On a d'abord pensé que l'on pourrait arriver à plus de succès (avec indice_init = 0.8) en choisissant un b plus important; l'expérience nous a prouvé le contraire.

Finalement, les tests avec pondérations variables ont mis en évidence une sensible amélioration de l'algorithme en faisant croître b linéairement avec le nombre d'itérations. En revanche les tests avec une fonction exponentielle pour b se sont révélés décevants. Dans le cas linéaire comme dans le cas exponentiel, on a choisi b(max_iter) de sorte que b(max_iter/2) corresponde au b que l'on a utilisé dans le cas constant, pour pouvoir faire une comparaison. D'autres valeurs inférieures pour b(max_iter) auraient peut-être amélioré le cas exponentiel, mais le temps CPU

pour l'exécution de ces tests nous a malheureusment manqué, étant donné que les stations étaient passablement occupées par les "batchs" d'autres étudiants.

5.2 Conclusion

L'ensemble des résultats montrent clairement que S-G-Sat est plus efficace que GSAT. Pour MAXFLIPS = max_iter = 500 et MAXTRIES = max_essais = 100, la différence apparaît clairement, pour d'autres valeurs on la devine.

Remarquons que ces tests, s'ils donnent individuellement des résultats intéressants, une combinaison n'en donnnera pas forcément de meilleurs. Nous avons par exemple essayé la combinaison suivante:

max_iter	=	1000
max_essais	=	100
indice_fonc	=	1
indice_init	=	0.33...
a	=	1
b(max_iter)	=	20

Elle nous a fourni 456 succès en 10570 secondes, ce qui entre dans l'ordre des résultats avec max_iter=1000. D'après nous, l'idée la plus convaincante reste celle de faire croître b linéairement avec le nombre d'itérations, mais il convient de bien choisir les autres paramètres (notamment max_iter et max_essais) avant de procéder aux tests.

S'il fallait encore approfondir le sujet, une direction que nous privilégerions serait d'essayer d'autres fonctions croissantes pour b(nombre d'itérations) comme une fonction logarithmique ou une bonne combinaison de celle-ci avec la fonction linéaire, voire exponentielle.

Un autre affinage que l'on pourrait apporter serait d'additioner à la fonction objectif le nombre de clauses avec un seul littéral satisfait et le nombre de clauses avec exactement deux littéraux satisfaits, bien sûr avec des pondérations adéquates. Cette alternative permettrait de trancher lorsque, dans S-G-Sat, deux manières de "flipper"auraient amené à la même amélioration de la fonction objectif. Elle diminuerait donc la part de hasard dans l'algorithme, mais il reste encore à vérifier que c'est dans cette direction qu'il faut progresser.

Deuxième partie

Annexes

Annexe A

Code source: generation.h

```
#define n 100 /*nbre de variables*/
#define m 430 /*nbre clauses*/
#define max_prob 1000 /*nombre de problemes*/

int Germe_inst; /*generer probleme*/
int Succes;
int **Variable;
int Clause[m+1][4];


/* Generateur de nombres aleatoires Unif[0,1[ ---------------------------------- */
double uniforme_reel(int &germe)
{
  long mm,a,b,c;
  long k1;
  double tempo;

  mm=2147483647L;
  a=16807;
  b=127773L;
  c=2836;

  k1=germe/b;
  germe=a*(germe%b)-k1*c;
  if (germe<0)germe+=mm;
  tempo=((double)germe)/((double)mm);
  return tempo;
}
/* ---------------------------------------------------------------------------- */


/* Generation des m clauses a n variables -------------------------------------- */
void Genere3sat(void)
{
    int i,j;
    int signe;

    free(Variable);
    Variable=(int**)malloc((2*n+1)*sizeof(int*));
    for(i=1;i<=2*n;i++)
        {
            Variable[i]=(int*)malloc(sizeof(int));
            Variable[i][0]=0;
        }

    for(i=1;i<=m;i++)
        {
            Clause[i][0]=3;
            for(j=1;j<=3;j++)
                {
                    do
                        {
                            /* variable */
                            Clause[i][j]= (int)(n*(uniforme_reel(Germe_inst))+1);
                        }
                    while ((j>1)&&((abs(Clause[i][j])==abs(Clause[i][j-1]))||
                             ((j==3)&&(abs(Clause[i][j])==abs(Clause[i][j-2])))));
```

```
                if (uniforme_reel(Germe_inst)<0.5)
                    Clause[i][j]= -Clause[i][j]; /* signe du litteral */

                signe=(Clause[i][j]>0) ? 1 : 0;

                Variable[abs(Clause[i][j]*2)-signe][0]++;

                Variable[abs(Clause[i][j]*2)-signe]=
                    (int*)realloc(Variable[abs(Clause[i][j]*2)-signe],
                       (Variable[abs(Clause[i][j]*2)-signe][0]+1)*sizeof(int));

                Variable[abs(Clause[i][j]*2)-signe]
                    [Variable[abs(Clause[i][j]*2)-signe][0]]
                    =i*(int)((signe-0.5)*2);
            }
        }
}
/* ---------------------------------------------------------------------------- */
```

Annexe B

Code source: S-G-Sat.c

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<string.h>
#include<limits.h>

#include"generation.h"

#define infini 10000 /*place vide*/
#define max_essais 1000 /*nombre max. d'essais par probleme*/
#define max_iter 500 /*nombre max. d'iterations par essai*/
#define indice_init 0.33333333 /*choix du type de la solution initiale*/
#define indice_fonc 0 /*choix du type de la fonction objective*/

int Germe_sol_init; /*generer sol initiale*/
int Germe_pensat; /*nombre aleatoire utilise dans Pensat */

int Iter_min,Iter_max,Essai_min,Essai_max;
double Iter_moy,Essai_moy;
int Sol[n+1];
int Sat_cl[m+1];
int Unsat[4];
int Flip_pos[3][n+1];
int Flip_neg[3][n+1];
int Flip_libre[3][n+1];
int Best_flip[3][2*n+1];



/* Affichage ------------------------------------------------------------------ */
void Ecrit3sat(void)
{
    int i,j;

    printf("Voila le Probleme:\n");
    for(i=1;i<=m;i++)
        {
            for(j=1;j<=3;j++)
                {
                    printf("%3d",Clause[i][j]);
                }
            printf("\n");
        }
    printf("\n");
    /*
      printf("Et voici les variables:\n");
      for(i=1;i<=2*n;i++)
      {
      for(j=1;j<=Variable[i][0];j++)
      {
      printf("%3d",Variable[i][j]);
      }
      printf("\n");
      }
      printf("\n");
    */
    printf("Solution courante:\n");
    for (i=1;i<=n;i++)
```

```
        {
            printf("%3d",Sol[i]);
        }
    printf("\n\n");
    /*
      printf("Nombre de litteraux satisfaits dans chaque clause:\n");
      for(i=1;i<=m;i++)
      {
      printf("%3d",Sat_cl[i]);
      }
      printf("\n\n");
      printf("Nombre de litteraux satisfaits 0 fois: %3d\n",Unsat[0]);
      printf("                                1 fois: %3d\n",Unsat[1]);
      printf("                                2 fois: %3d\n",Unsat[2]);
      printf("\n");
      printf("Nombre de variables doublement affectees: %3d\n",Unsat[3]);
      printf("\n");
      printf("Flip_pos:\n");
      for (i=0;i<=2;i++)
      {
      for (j=1;j<=n;j++)
      if (Flip_pos[i][j]!=infini)
      printf("%3d",Flip_pos[i][j]);
      else printf("  -");
      printf("\n");
      }
      printf("\n");
      printf("Flip_neg:\n");
      for (i=0;i<=2;i++)
      {
      for (j=1;j<=n;j++)
      if (Flip_neg[i][j]!=infini)
      printf("%3d",Flip_neg[i][j]);
      else printf("  -");
      printf("\n");
      }
      printf("\n");
      printf("Flip_libre:\n");
      for (i=0;i<=2;i++)
      {
      for (j=1;j<=n;j++)
      if (Flip_libre[i][j]!=infini)
      printf("%3d",Flip_libre[i][j]);
                else printf("  -");
            printf("\n");
        }
    printf("\n");
    */
}
/* ----------------------------------------------------------------------------- */


/* remplissage de Flip_pos, Flip_neg et Flip_libre pour la variable i ------------ */
void remplit_flip(int i)
{
    int var,j;

    for (j=0;j<=2;j++)
        {
            Flip_pos[j][i]=0;
            Flip_neg[j][i]=0;
            Flip_libre[j][i]=0;
        }

    if (Sol[i]<0)
        {
            for (j=0;j<=2;j++)
                    Flip_neg[j][i]=infini;
            for (j=1;j<=Variable[2*i-1][0];j++)
                {
                    var=Variable[2*i-1][j]; /* var = clause contenant +i */
                    Flip_pos[Sat_cl[var]][i]--; /* on diminue de 1 le nombre de clauses
                                                   satisfaites Sat_cl[var] fois */
                    Flip_libre[Sat_cl[var]][i]--;
                    if (Sat_cl[var]!=2)
                        {
                            Flip_pos[Sat_cl[var]+1][i]++;
                            Flip_libre[Sat_cl[var]+1][i]++;
                        }
                }
```

```
            for (j=1;j<=Variable[2*i][0];j++)
                {
                    var=-Variable[2*i][j]; /* var = clause contenant -i */
                    if (Sat_cl[var]!=3) Flip_pos[Sat_cl[var]][i]--;
                    Flip_pos[Sat_cl[var]-1][i]++;
                }
        }


    if (Sol[i]>0)
        {

            for (j=0;j<=2;j++)
                {
                    Flip_pos[j][i]=infini;
                }

            for (j=1;j<=Variable[2*i-1][0];j++)
                {

                    var=Variable[2*i-1][j]; /* var = clause contenant +i */

                    if (Sat_cl[var]!=3) Flip_neg[Sat_cl[var]][i]--;
                    Flip_neg[Sat_cl[var]-1][i]++;

                }


            for (j=1;j<=Variable[2*i][0];j++)
                {
                    var=-Variable[2*i][j];
                    Flip_neg[Sat_cl[var]][i]--;
                    Flip_libre[Sat_cl[var]][i]--;
                    if (Sat_cl[var]!=2)
                        {
                            Flip_neg[Sat_cl[var]+1][i]++;
                            Flip_libre[Sat_cl[var]+1][i]++;
                        }
                }

        }

    if (Sol[i]==0)
        {
            for (j=0;j<=2;j++)
                {
                    Flip_libre[j][i]=infini;
                }
            for (j=1;j<=Variable[2*i-1][0];j++)
                {
                    var=Variable[2*i-1][j];
                    if (Sat_cl[var]!=3) Flip_neg[Sat_cl[var]][i]--;
                    Flip_neg[Sat_cl[var]-1][i]++;
                }
            for (j=1;j<=Variable[2*i][0];j++)
                {
                    var=-Variable[2*i][j];
                    if (Sat_cl[var]!=3) Flip_pos[Sat_cl[var]][i]--;
                    Flip_pos[Sat_cl[var]-1][i]++;
                }
        }

}
/* ---------------------------------------------------------------------------- */


/* Generation de la solution initiale ------------------------------------------- */
void Sol_init(void)
{
    int i,j;
    double aux;

    /* initialisation */
    for(i=1;i<=m;i++)
        {
            Sat_cl[i]=0;
                for (j=0;j<=3;j++)
                    {
                        Unsat[j]=0;
                    }
```

```
        }

    /* remplissage de Sol */
    for (i=1;i<=n;i++)
        {
            aux=uniforme_reel(Germe_sol_init);
            if (aux<=indice_init)
                aux=0;
            else
                {
                    if (aux>((1+indice_init)/2))
                        aux=-1;
                    else
                        aux=1;
                }
            Sol[i]=i*aux;
        }

    /* remplisaage de Sat_cl */
    for(i=1;i<=n;i++)
        {
            if (Sol[i]>=0)
                {
                    for(j=1;j<=Variable[2*i-1][0];j++)
                        {
                            Sat_cl[Variable[2*i-1][j]]++;
                        }
                }
            if (Sol[i]<=0)
                {
                    for(j=1;j<=Variable[2*i][0];j++)
                        {
                            Sat_cl[-Variable[2*i][j]]++;
                        }
                }
        }

    /* remplissage de Unsat */
    for (i=1;i<=m;i++)
        {
            if (Sat_cl[i]!=3)
                {
                    Unsat[Sat_cl[i]]++;
                }
        }
    for (i=1;i<=n;i++)
        {
            if (Sol[i]==0) Unsat[3]++;
        }


    /* remplissage de Flip_pos, Flip_neg et Flip_libre */
    for (i=1;i<=n;i++)
        {
            remplit_flip(i);
        }
}
/* ---------------------------------------------------------------------------- */



/* Flip ----------------------------------------------------------------------- */
void Effectue_flip(int var,int signe)
{
    int i,j;
    int aux[n+1];


    /* baisse le nombre des litteraux qui ne sont plus satisfaits  */
    if (Sol[var]>=0)
        for (i=1;i<=Variable[2*var-1][0];i++)
            Sat_cl[Variable[2*var-1][i]]--;

    if (Sol[var]<=0)
        for (i=1;i<=Variable[2*var][0];i++)
            Sat_cl[-Variable[2*var][i]]--;

    if (Sol[var]==0) Unsat[3]--;
```

```
    /* augmente le nombre des litteraux qui sont satisfaits maintenant */
    Sol[var]=signe*var;

    if (Sol[var]>=0)
        for (i=1;i<=Variable[2*var-1][0];i++)
            Sat_cl[Variable[2*var-1][i]]++;

    if (Sol[var]<=0)
        for (i=1;i<=Variable[2*var][0];i++)
            Sat_cl[-Variable[2*var][i]]++;

    if (Sol[var]==0) Unsat[3]++;


    /* remplit de nouveau le tableau Unsat (sauf [3]) */

    for (i=0;i<=2;i++)
        Unsat[i]=0;

    for (i=1;i<=m;i++)
        {
            if (Sat_cl[i]!=3)
                {
                    Unsat[Sat_cl[i]]++;
                }
        }


    /* remplit de nouveau les colonnes des tableaux Flip qui sont affectees */
    for (i=1;i<=n;i++)
        aux[i]=0;

    for (i=1;i<=Variable[2*var-1][0];i++)
        {
          for (j=1;j<=3;j++)
              aux[abs(Clause[Variable[2*var-1][i]][j])]=1;
        }

    for (i=1;i<=Variable[2*var][0];i++)
        {
          for (j=1;j<=3;j++)
              aux[abs(Clause[-Variable[2*var][i]][j])]=1;
        }

    for (i=1;i<=n;i++)
        {
            if (aux[i])
            remplit_flip(i);

        }
}
/* ------------------------------------------------------------------------------ */


/* Meilleur flip ----------------------------------------------------------------- */
void Determine_meill_flips(double a,double b)
{
    int i,k,l;
    double val;
    int libre[n+1];

    /* initialisation du vecteur libre */
    for (i=1;i<=n;i++)
        {
            if (Sol[i]==0)
                libre[i]=1;
            else libre[i]=0;
        }

    /* recherche du minimum */
    k=1;
    l=1;
    val=a*Flip_pos[0][1]-b*libre[1];
    for (i=2;i<=n;i++)
        {
            if (a*Flip_pos[0][i]-b*libre[i]<val)
                {
                    val=a*Flip_pos[0][i]-b*libre[i];
                    l=i;
                }
```

```
        }

    for (i=1;i<=n;i++)
        {
            if (a*Flip_neg[0][i]-b*libre[i]<val)
                {
                    val=a*Flip_neg[0][i]-b*libre[i];
                    l=i;
                    k=-1;
                }
        }

    for (i=1;i<=n;i++)
        {
            if (a*Flip_libre[0][i]+b<val)
                {
                    val=a*Flip_libre[0][i]+b;
                    l=i;
                    k=0;
                }
        }

    /* remplit le tableau Best_flip */
    Best_flip[0][0]=0;
    switch(k)
        {
        case 1:
            for (i=1;i<=n;i++)
                if (a*Flip_pos[0][i]-b*libre[i]==val)
                    {
                        Best_flip[0][0]++;
                        Best_flip[1][Best_flip[0][0]]=i;
                        Best_flip[2][Best_flip[0][0]]=1;
                    }
        case -1:
            for (i=1;i<=n;i++)
                if (a*Flip_neg[0][i]-b*libre[i]==val)
                    {
                        Best_flip[0][0]++;
                        Best_flip[1][Best_flip[0][0]]=i;
                        Best_flip[2][Best_flip[0][0]]=-1;
                    }
        case 0:
            for (i=1;i<=n;i++)
                if (a*Flip_libre[0][i]+b==val)
                    {
                        Best_flip[0][0]++;
                        Best_flip[1][Best_flip[0][0]]=i;
                        Best_flip[2][Best_flip[0][0]]=0;
                    }
        }
}
/* ------------------------------------------------------------------------------ */


/* Ponderation du nombre de clauses non satisfaites ------------------------------ */
double Ponda(int iter,int essai)
{
    return 1;
}
/* ------------------------------------------------------------------------------ */


/* Ponderation du nombre de variables doublement affectees ----------------------- */
double Pondb(int iter,int essai)
{
    double aux;

    if (indice_fonc==0)
        aux=10; /* b dans le cas constant */
    if (indice_fonc==1)
        aux=20*(((double)(iter))/((double)(max_iter))); /* b(max_iter) dans le cas lineaire */
    if (indice_fonc==2)
       aux=(2*3.74645e108)*((exp((double)(iter))-1)/(exp((double)(max_iter))-1));
                            /* b(max_iter) dans le cas exp */
    return aux;
}
/* ------------------------------------------------------------------------------ */
```

```
/* Iterations -------------------------------------------------------------------- */
void Pensat(void)
{
    int iter;
    int choix;
    int essai;
    double a, b;

    essai=0;
    while((essai<max_essais)&&((Unsat[0]>0)||(Unsat[3]>0)))
        {
            essai++;

            if (essai>1) Sol_init();
            iter=0;
            while((iter<max_iter)&&((Unsat[0]>0)||(Unsat[3]>0)))
                {
                    iter++;
                    a=Ponda(iter,essai);
                    b=Pondb(iter,essai);
                    Determine_meill_flips(a,b);
                    choix=(int)(uniforme_reel(Germe_pensat)*Best_flip[0][0]+1);
                    Effectue_flip(Best_flip[1][choix],Best_flip[2][choix]);
                }
        }
    if ((Unsat[0]==0)&&(Unsat[3]==0))
        {
            Essai_moy=Essai_moy+essai;
            Iter_moy=Iter_moy+iter;
            if (Essai_max<essai)
                Essai_max=essai;
            if (Essai_min>essai)
                Essai_min=essai;
            if (Iter_max<iter)
                Iter_max=iter;
            if (Iter_min>iter)
                Iter_min=iter;
        }
}
/* ------------------------------------------------------------------------------ */


/* ------------------------------------------------------------------------------ */
void main(void)
{
    int /*i,*/p;
    Germe_inst=241734159;
    Germe_sol_init=1490397746;
    Germe_pensat=1539566639;
    Essai_min=max_essais;
    Iter_min=max_iter;
    for (p=1;p<=max_prob;p++)
        {
            Genere3sat();
            Sol_init();
            /*if((m<=50)&&(n<=10))Ecrit3sat();*/
            Pensat();
            /*
              if((m<=50)&&(n<=10))Ecrit3sat();
              printf("Solution finale:\n");
              for (i=1;i<=n;i++)
                {
                   printf("%4d",Sol[i]);
                }
              printf("\n\n");
            */
            if ((Unsat[0]==0)&&(Unsat[3]==0))
                {
                    /*
                    printf("La solution marche.");
                    */
                    Succes++;
                }
            /*
            else
                printf("La solution ne marche pas.");
            printf("\n\n");
            */
        }
```

```
    if (Succes>0)
        {
            Iter_moy=Iter_moy/Succes;
            Essai_moy=Essai_moy/Succes;
        }

    printf("Nombre de problemes:%3d\n",max_prob);
    printf("Nombre de succes:%3d\n",Succes);
    printf("# succes / # problemes:%5f\n",((double)Succes)/((double)max_prob));
    printf("Nombre moyen d'essais:%5f\n",Essai_moy);
    printf("Minimum d'essais:%3d\n",Essai_min);
    printf("Maximum d'essais:%3d\n",Essai_max);
    printf("Nombre moyen d'iterations:%5f\n",Iter_moy);
    printf("Minimum d'iterations:%3d\n",Iter_min);
    printf("Maximum d'iterations:%3d\n",Iter_max);
    printf("Germe_inst:%d\n",Germe_inst);
}
/* ----------------------------------------------------------------------------- */
```

Annexe C

Code source: D-Put.c

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<string.h>
#include<limits.h>

#include"generation.h"


int Fin;
int Sol[n+1];

/* Affichage ------------------------------------------------------------------- */
void Ecrit3sat(int claus[m+1][4],int solution[n+1])
{
    int i,j;

    printf("Probleme courrant:\n");
    for(i=1;i<=m;i++)
        {
            printf("%3d  ",claus[i][0]);
            for(j=1;j<=3;j++)
                printf("%3d",claus[i][j]);
            printf("\n");
        }
    printf("\n");

    printf("Solution courante:\n");
    for (i=1;i<=n;i++)
        printf("%3d",solution[i]);
    printf("\n\n");
}
/* ---------------------------------------------------------------------------- */



/* elimine les clauses qui contiennent v --------------------------------------- */
void elim_clause(int v, int &k, int variab[2*n+1],int claus[m+1][4])
{
    int signe,i;

    if (v>0) signe=1;
    else signe=0;
    v=abs(v);

    for (i=1;i<=Variable[v*2-signe][0];i++)
        {
            if (claus[abs(Variable[v*2-signe][i])][0]!=0)
                {
                    claus[abs(Variable[v*2-signe][i])][0]=0;
                    k--;
                }
        }
    variab[v*2-signe]=0;
}
/* ---------------------------------------------------------------------------- */
```

```
/* elimine v des clauses ------------------------------------------------------- */
void elim_var(int v,int variab[2*n+1],int claus[m+1][4])
{
    int i,j,l,signe,deja;

    if (v>0) signe=1;
    else signe=0;
    v=abs(v);

    for (i=1;i<=Variable[v*2-signe][0];i++)
        {
            deja=1;
            for (j=1;j<=3;j++)
                if (abs(claus[abs(Variable[v*2-signe][i])][j])==v)
                    {
                        l=1;
                        while (abs(claus[abs(Variable[v*2-signe][i])][l])==v)
                            l++;
                        claus[abs(Variable[v*2-signe][i])][j]
                            =claus[abs(Variable[v*2-signe][i])][l];

                        if ((claus[abs(Variable[v*2-signe][i])][0])&&(deja))
                            {
                                claus[abs(Variable[v*2-signe][i])][0]--;
                                deja--;
                            }
                    }
        }
    variab[v*2-signe]=0;
}
/* ---------------------------------------------------------------------------- */


/* determine un litteral pur (v=0 si y en n'existe pas) ------------------------ */
int litpur(int variab[2*n+1],int solution[n+1])
{
    int i,v;

    v=0;
    i=1;
    while ((v==0)&&(i<=n))
        {
            if ((solution[i]==0)&&((variab[i*2-1]==0)||(variab[i*2]==0)))
                {
                  if (variab[i*2-1]==0) v=-i;
                  else v=i;
                }
            i++;
        }
    return v;
}
/* ---------------------------------------------------------------------------- */


/* controle les clauses contenantes non-v lorsqu'on elimine une clause unitaire */
void non_v(int v,int k, int variab[2*n+1], int claus[m+1][4], int &fini)
{
    int signe,i,j,l,deja;

    if (v>0) signe=0;
    else signe=1;
    v=abs(v);

    i=1;
    while (((variab[v*2-signe])&&(fini==0))&&(k))
        {
            if (claus[abs(Variable[v*2-signe][i])][0]==1)
                fini=-1;
            else
                deja=1;
                for (j=1;j<=3;j++)
                    if (abs(claus[abs(Variable[v*2-signe][i])][j])==v)
                        {
                            l=1;
                            while (abs(claus[abs(Variable[v*2-signe][i])][l])==v)
                                l++;
                            claus[abs(Variable[v*2-signe][i])][j]
                                =claus[abs(Variable[v*2-signe][i])][l];
```

```
                              if ((claus[abs(Variable[v*2-signe][i])][0])&&(deja))
                                  {
                                      claus[abs(Variable[v*2-signe][i])][0]--;
                                      deja--;
                                  }
                          }
            i++;
            variab[v*2-signe]--;
        }
}
/* ---------------------------------------------------------------------------- */


/* regarde s'il extste une clause unitaire ------------------------------------- */
void clause_unitaire(int &v,int claus[m+1][4])
{
    int i;

    i=1;
    v=0;
    while ((i<=m)&&(claus[i][0]!=1))
        i++;
    if (i<m)
        v=claus[i][1];
    else
        {
            if (claus[i][0]!=1)
                v=0;
            else
                v=claus[m][1];
        }
}
/* ---------------------------------------------------------------------------- */


/* Algorithme de Davis-Putnam ------------------------------------------------- */
void Algorithme(int k, int varia[2*n+1],int clau[m+1][4],
                       int &fini,int solution[n+1])
{
    int v,s,i,j,k_aux;
    int variab[2*n+1];
    int claus[m+1][4];
    int sol_aux[n+1];

    /*
    Ecrit3sat(clau,solution);
    */
    if (fini==0)
        {
            for (i=1;i<=m;i++)
                for (j=0;j<=3;j++)
                    claus[i][j]=clau[i][j];

            for(i=1;i<=2*n;i++)
                variab[i]=varia[i];

            if (v=litpur(variab,solution))
                {
                    solution[abs(v)]=v;
                    elim_clause(v,k,variab,claus);
                    Algorithme(k,variab,claus,fini,solution);
                }
            else
                {
                    clause_unitaire(v,claus);
                    if (v)
                        {
                            solution[abs(v)]=v;
                            elim_clause(v,k,variab,claus);
                            non_v(v,k,variab,claus,fini);
                            Algorithme(k,variab,claus,fini,solution);
                        }

                    else
                        {
                            if (k==0)
                                {
                                    fini=1;
                                }
                            else
```

```
                                {
                                    s=1;
                                    while (solution[s]!=0)
                                            s++;
                                    for (i=1;i<=n;i++)
                                        sol_aux[i]=solution[i];
                                    k_aux=k;

                                    solution[s]=s;
                                    elim_clause(s,k,variab,claus);
                                    elim_var(-s,variab,claus);
                                    Algorithme(k,variab,claus,fini,solution);

                                    if (fini<0)
                                        {
                                            fini=0;
                                            for (i=1;i<=m;i++)
                                                for (j=0;j<=3;j++)
                                                    claus[i][j]=clau[i][j];
                                            sol_aux[s]=-s;
                                            elim_clause(-s,k_aux,variab,claus);
                                            elim_var(s,variab,claus);
                                            Algorithme(k_aux,variab,claus,fini,sol_aux);
                                            for (i=1;i<=n;i++)
                                                solution[i]=sol_aux[i];
                                            k=k_aux;
                                        }
                                }
                        }
                }
        }
}
/* ---------------------------------------------------------------------------- */


/* ---------------------------------------------------------------------------- */
void main(void)
{
    int i,j;
    int variab[2*n+1];

    Germe_inst=241734159;
    for (j=1;j<=max_prob;j++)
        {
            Fin=0;
            Genere3sat();
            for(i=1;i<=2*n;i++)
                variab[i]=Variable[i][0];
            for(i=1;i<=n;i++)
                Sol[i]=0;
            /*
            Ecrit3sat(Clause,Sol);
            */
            Algorithme(m,variab,Clause,Fin,Sol);
            if (Fin>0)
                {
                    Succes++;
                    /*
                      printf("Solution admissible:\n");
                      for (i=1;i<=n;i++)
                      {
                      printf("%3d",Sol[i]);
                      }
                      printf("\n\n");
                    */
              }
            /*
            else
                {
                    printf("Pas de solution admissible\n\n");
                }
            */
        }
        printf("Nombre de succes:%3d\n",Succes);
        printf("# succes / # problemes:%5f\n",((double)Succes)/((double)max_prob));
}
/* ---------------------------------------------------------------------------- */
```

Bibliographie

[1] B. Selman, H. Levesque et D. Mitchwell, *A New Method for Solving Hard Satisfiability Problems*, PROC. AAAI-92, 1992, 440-446.

[2] B. Selman, H. Levesque et D. Mitchwell, *Hard and Easy Distributions of SAT Problems*, PROC. AAAI-92, 1992, 456-465.

[3] S.A. Cook, *The Complexity of Theorem-prooving Procedures*, PROC. 3rd ANN. ACM SYMP. ON THEORY OF COMPUTING, 1971, 151-158.

[4] M. Davis et H. Putnam, *A Computing Procedure for Quantification Theory*, J. ASSOC. COMPUT. MACH., 1960, 7:201-215.

[5] M. Walz et J. Godjevac, *Introduction au langage C*, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE.

[6] B. Bayart, *Joli manuel pour LaTeX2ε, Guide local de l'ESIEE*, ÉCOLE SUPÉRIEURE D'INGÉNIEURS EN ÉLEKTROTECHNIQUE ET ÉLECTRONIQUE, 1995.

[7] J.-P. Braquelaire, *Méthodologie de la programmation en C*, MASSON,2e édition, 1995.