



Département Informatique 5A
Rapport final de PFE 2015

Étude et résolution du problème SAT. Utilisation de réduction polynomiales sur différents problèmes NP-difficiles

Auteur(s)

Valentin Montmirail

[valentin.montmirail@gmail.com]

Encadrant(s)

Christophe Lenté

[christophe.lente@univ-tours.fr]

Ville: Tours

Table des matières

1	Remerciements	1
2	Introduction	2
2.1	Bibliographie	2
2.2	Étude des autres solveurs	2
2.3	SATyr : le solveur SAT + UNSAT	2
2.4	Réductions polynomiales	3
2.5	Benchmarks	3
2.6	Méthodologies utilisées pour le développement	3
3	Bibliographie	4
3.1	Réductions Polynomiales	4
3.1.1	A Very Efficient Algorithm to Representing JSSP as Satisfiability Problem	4
3.1.2	A Local Search Algorithm for a SAT Representation of Scheduling Problems	5
3.1.3	Planning as Satisfiability	5
3.1.4	Planification SAT : Amélioration des codages	6
3.1.5	A Reduced Codification for the Logical Representation of JSSP	8
3.2	Algorithmes Évolutionnaires et SAT	11
3.2.1	Comparing the performance of the genetic and LS algorithms for solving SAT	11
3.2.2	GASAT : a genetic local search algorithm for the satisfiability problem	12
3.2.3	FlipGA : Evolutionary Algorithms for the Satisfiability Problem	12
3.2.4	Local Search with Three Truth Values for SAT Problems	13
3.2.5	An improved deterministic local search algorithm for 3-SAT	13
3.2.6	A Superior Evolutionary Algorithm for 3-SAT	13
3.3	SAT insatisfiable : UNSAT	15
3.3.1	GUNSAT : a Greedy Local Search Algorithm for Unsatisfiability	15
3.3.2	Learning in local search	15
3.3.3	On the limit of branching rules for hard random unsatisfiable 3-SAT	16
3.3.4	Clauses critiques et extraction de MUS	16
3.3.5	A Machine-Oriented Logic Based on the Resolution Principle	17
3.4	Livre de référence dans le domaine	18
3.4.1	Problème SAT : progrès et défis	18
3.4.2	Handbook of Satisfiability	20
3.4.3	Introduction to Mathematics of Satisfiability	22
3.4.4	The Satisfiability Problem	23

4	Étude des autres solveurs	24
4.1	Introduction	24
4.2	WalkSAT (Selman, Kautz, 1996)	25
4.3	FlipGA (Marchiori, Rossi, 1999)	26
4.4	zChaff (Moskewicz, Madigan, 2001)	27
4.5	SAT4j (Le Berre, Parrain, 2004)	29
4.6	MiniSAT (Eén et Sörensson, 2005)	31
4.7	CDLS (Audemard, Lagniez, 2009)	32
4.8	Glucose (Audemard, Simon, 2014)	33
5	SATyr : le solveur génétique SAT + UNSAT	34
5.1	Introduction	34
5.2	Lecture de fichier CNF au format DIMACS	34
5.2.1	Explication du format DIMACS	35
5.3	SAT : trouver une solution à un problème	36
5.3.1	Représentation et espace de recherche	37
5.3.2	La fonction de fitness	37
5.3.3	L'opérateur de sélection	38
5.3.4	L'opérateur de croisement uniforme	39
5.3.5	L'opérateur de Mutation	41
5.3.6	Adaptation	42
5.3.7	Réinsertion dans certaines conditions	43
5.4	Algorithme de la partie SAT de SATyr	44
5.4	UNSAT : prouver qu'un problème n'a pas de solution	44
5.4.1	Principe de résolution	44
5.4.2	SATyr + Principe de résolution	45
5.4.3	Comment est réalisé mettreAJourTableau ?	47
5.4.4	Comment sont réalisées les structures de données ?	48
5.5	SATyr : SAT + UNSAT	51
6	Réductions polynomiales	52
6.1	Le problème des N-Reines	52
6.2	Le problème d'emploi du temps	54
6.2.1	1 ^{re} version : cours	55
6.2.2	2 ^e version : cours + salles	56
6.2.3	3 ^e version : cours + salles + professeurs	56
6.2.4	4 ^e version : cours + salles + professeurs + groupes d'étudiants	57
6.2.5	Remarques et problèmes rencontrés	57
6.3	Le problème de Job Shop	58
6.3.1	Fichier d'entrée et solution attendue	59
6.3.2	Mécanisme pour trouver le C_{max} optimal avec un problème de décision	60
6.3.3	Les différentes structures utilisées	61
6.3.4	Les différentes contraintes utilisées	62
6.4	Le problème de Flow Shop	64
6.5	Problèmes rencontrés dans Job Shop et Flow Shop	65
7	Benchmarks	66
7.1	Rappel des benchmarks du cahier des spécifications	67

7.2	Benchmarks sur les SATISFIABLEs	68
7.2.1	Podium	69
7.3	Benchmarks restants non tracés	70
7.4	Benchmarks sur les UNSATISFIABLEs	71
7.4.1	Podium	71
8	Méthodologies utilisées pour le développement du projet	72
8.1	La documentation	72
8.2	Le versionning	73
8.3	Méthodes agiles : les sprints de développement	74
8.4	Les tests unitaires	75
8.4.1	Script pour les résultats	75
8.4.2	La preuve par résolution	76
8.5	Intégration continue : Jenkins	77
8.6	Analyse de code : SonarQube	78
9	Bilan personnel	79
9.1	Connaissances et savoir-faire	79
9.2	Découvrir les méthodes pour un projet de recherche	79
9.3	Découvrir le monde des SAT...	79
10	Conclusion	81
A	Cahier des spécifications systèmes	85

Chapitre 1

Remerciements

Avant toute chose, je tiens à remercier [Nicolas Ragot](#) et [Christophe Lenté](#) pour avoir bien voulu respectivement accepter et encadrer mon projet de fin d'études.

Je tiens aussi à remercier [Claudio Rossi](#), créateur du solveur FlipGA qui aura été l'inspiration principale de SATyr, ainsi que [Jean Marie Lagniez](#) et [Cédric Piette](#) pour bien avoir accepté de passer du temps à répondre à mes différentes questions.

Je remercie également [Amit Gross](#), utilisateur très assidu du site StackOverFlow et qui aura posé la première pierre qui a permis de réaliser la réduction polynomiale du problème d'emploi du temps vers SAT.

J'adresse finalement un très grand merci à [Gilles Audemard](#) qui m'aura finalement donné goût aux problèmes SAT, qui aura passé énormément de temps sur mes questions, qui m'aura fourni des pistes de recherches afin de pouvoir en définitif réussir les objectifs de mon PFE et obtenir un solveur fonctionnel et complet¹.

1. capable de trouver une solution quand elle existe et prouver que ce n'est pas le cas dans le cas contraire.

Chapitre 2

Introduction

Dans le cadre de ma cinquième et dernière année à Polytech Tours au sein du département informatique, j'ai été amené à réaliser le projet de fin d'études *Étude et résolution du problème SAT* dont les objectifs étaient les suivants :

- Étude bibliographique et étude des solveurs existants.
- Développement de métaheuristiques destinées à trouver rapidement une solution, quand elle existe.
- Réglage des métaheuristiques.
- Ajout de procédures destinées à détecter des cas d'infaisabilité.
- Benchmarks et comparaison aux solveurs existants.
- Utilisation de réductions polynomiales pour s'attaquer à la résolution d'autres problèmes NP-difficiles.

2.1 Bibliographie

L'étude bibliographique (voir chapitre 3) aura nécessité la lecture de bon nombre d'articles.

Même si ces derniers sont de très bonne qualité la plus part du temps, j'ai fait le choix de ne garder que les principaux à mes yeux, ceux qui ont contribué à faire avancer mon projet de fin d'étude ou qui permettrai de poursuivre ce projet.

2.2 Étude des autres solveurs

Dans le chapitre 4, étant donné qu'il est impossible de faire une liste exhaustive de l'ensemble des solveurs existants, je présenterai brièvement les principaux en insistant d'avantage sur les solveurs auxquels je vais me comparer dans le chapitre 7.

Le but de ce chapitre est de montrer qu'il est possible de résoudre le problème SAT de différentes manières.

2.3 SATyr : le solveur SAT + UNSAT

Dans le chapitre 5, en plus d'expliquer concrètement comment fonctionne mon solveur, j'ai fait aussi le choix de décrire l'ensemble des notions qu'il est nécessaire de connaître afin

de pouvoir comprendre comment sont résolus ces problèmes SAT.

2.4 Réductions polynomiales

Dans le chapitre 6, je vais détailler ce qu'est une réduction polynomiale mais je vais surtout expliquer en quoi consistent les problèmes NP-difficiles qui ont été utilisés.

Ce chapitre parle finalement plus de Proofs of Concept plutôt que de réelles réductions polynomiales utilisables avec des instances du "monde réel".

2.5 Benchmarks

Dans le chapitre 7, des résultats visuels seront présentés à travers différentes courbes GnuPlot réalisées directement depuis la lecture des fichiers solutions des solveurs.

Je trouve qu'il est bien plus parlant d'afficher en graphique à la même échelle, le temps pour exécuter l'ensemble des fichiers résolus par les solveurs. Ainsi, on voit très facilement : qui est le plus performant et aussi à partir de quelle taille certains solveurs perdent en efficacité.

2.6 Méthodologies utilisées pour le développement

Dans le chapitre 8, je vais parler de l'ensemble des méthodes génies logiciels qui ont été utilisées durant ce projet. Nous allons voir comment j'ai assuré la qualité du code rédigé ainsi que la pertinence des résultats retournés par le solveur.

Chapitre 3

Bibliographie

3.1 Réductions Polynomiales

3.1.1 A Very Efficient Algorithm to Representing Job Shop Scheduling as Satisfiability Problem [1]

Cet article explique comment il est possible de transformer un problème JSSP en problème 3-SAT et comment le résoudre pour arriver jusqu'au Gantt. Voici comment fonctionne globalement leur algorithme :

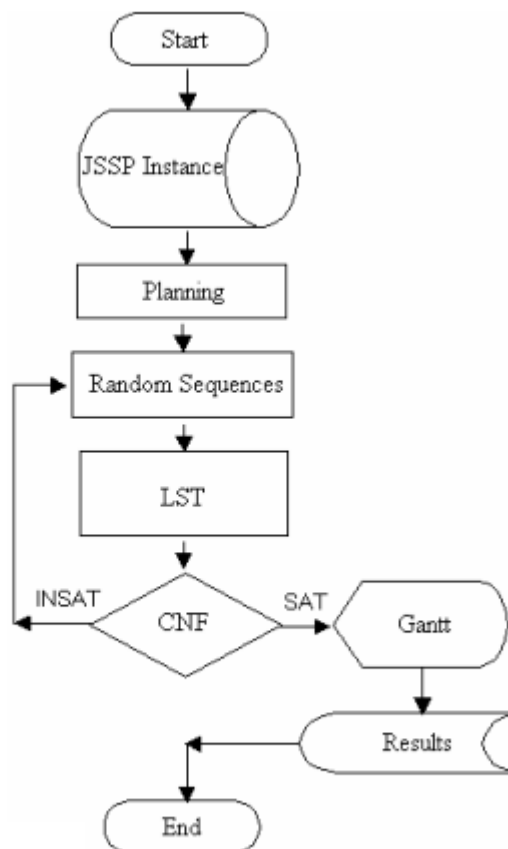


FIGURE 3.1 – Algorithme RandTaut

Le JSSP est transformé en 3-SAT avec l'ensemble des contraintes possibles, et ils tentent de ramener ce 3-SAT à 1-SAT via des transformations logiques. Une fois le problème en 1-SAT, ils ont leur affectation pour chaque machine et donc leur planification à afficher dans

le Gantt.

Il est tout à fait concevable de résoudre à l'aide d'un solveur classique le 3-SAT d'origine et ainsi obtenir d'une autre manière l'affectation pour chaque machine.

3.1.2 A Local Search Algorithm for a SAT Representation of Scheduling Problems [2]

Cet article propose la mise en place d'un algorithme de recherche locale afin de résoudre des problèmes JSSP formalisés en SAT. Cette article explique aussi comment on passe d'un problème JSSP en problème SAT via ce petit tableau :

Clauses of R	CNF	Interpretation
$sa_{i,t} \wedge pr_{i,j} \rightarrow sa_{j,t+pi}$	$\sim sa_{i,t} \vee \sim pr_{i,j} \vee sa_{j,t+pi}$	If i Starts at or after t and j follows i then j cannot start until i is finished.
$sa_{j,t} \wedge pr_{j,i} \rightarrow sa_{i,t+pj}$	$\sim sa_{j,t} \vee \sim pr_{j,i} \vee sa_{i,t+pj}$	If j Starts at or after t and i follows j then i cannot start until j is finished

FIGURE 3.2 – Table de correspondance entre JSSP et SAT

$sa_{j,t}$ signifie que l'opération j démarre à (ou après) le temps t .

Si $Pr_{i,j}$ vaut vrai, alors $Pr_{i,j}$ signifie que l'opération i précède j .

Si $Pr_{i,j}$ vaut faux, alors $Pr_{i,j}$ signifie que l'opération j précède i .

Si l'on veut pouvoir trouver une affectation valide pour JSSP, il est nécessaire de trouver un assignement de variables qui satisfait le problème SAT défini par la table ci-dessus, bien évidemment, il faut connaître la date de démarrage de chaque tâche.

3.1.3 Planning as Satisfiability[3]

Cet article parle d'une modélisation des problèmes de Planification sous forme de problème SAT. Il arrive à la conclusion que les clauses d'un problème de planification sont composées à 99% de clauses de Horn.

3.1.4 Planification SAT : Amélioration des codages, automatisation de la traduction et étude comparative[4]

Ce rapport présente des travaux sur la planification par satisfaction de bases de clauses (planification SAT). Ils y détaillent les améliorations apportées aux codages classiques dans les espaces d'états, de plans et au codage du graphe de planification. La représentation des problèmes de planification est basé sur le modèle STRIPS.

Le modèle STRIPS est basé sur une restriction de la logique du premier ordre : un état du monde y est représenté par un ensemble de fluents.

Tout fluent qui n'est pas modifié par l'application d'une action est présent dans l'état du monde résultant de l'application de cette action. Les effets d'une action de type STRIPS ne traitent donc que ce qui est modifié par l'application de cette action. Une description d'action de type STRIPS est constituée par un triplet d'ensembles de fluents :

- **Les pré-conditions** : ce sont les fluents qui déterminent l'applicabilité de l'action. L'action est applicable sur un état du monde si et seulement si ses pré-conditions sont incluses dans cet état.
- **Les ajouts** : ce sont les fluents qui sont ajoutés à l'état du monde résultant de l'application de l'action à un état donné.
- **Les retraits** : ce sont les fluents qui sont retirés de l'état du monde résultant de l'application de l'action à un état donné.

Voici l'exemple que les auteurs utilisent pour illustrer le modèle STRIPS :

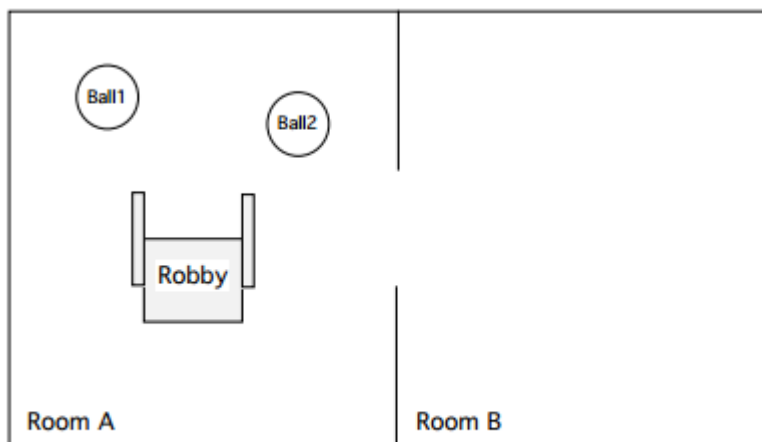


FIGURE 3.3 – Etat initial du problème dans le monde du Gripper

- $\text{room}(x)$: l'objet x est une salle.
- $\text{ball}(x)$: l'objet x est une balle.
- $\text{gripper}(x)$: l'objet x est l'une des pinces de Robby.
- $\text{at-roby}(x)$: Robby est dans la salle x .
- $\text{at}(x, y)$: la balle x est dans la salle y .
- $\text{free}(x)$: la pince x de Robby est libre.
- $\text{carry}(x, y)$: la balle x est dans la pince y de Robby.

Dans l'état initial, les deux balles Ball1 et Ball2 sont dans la salle RoomA ainsi que Robby. Cet état est représenté par l'ensemble de fluents suivant :

$I = \text{room} \{ (\text{RoomA}), \text{room}(\text{RoomB}), \text{ball}(\text{Ball1}), \text{ball}(\text{Ball2}), \text{gripper}(\text{Left}), \text{gripper}(\text{Right}), \text{at}(\text{Ball1}, \text{RoomA}), \text{at}(\text{Ball2}, \text{RoomA}), \text{at-robby}(\text{RoomA}), \text{free}(\text{Left}), \text{free}(\text{Right}) \}$

Dans l'état final du problème, les balles Ball1 et Ball2 sont dans la salle RoomB ainsi que Robby ; ce qui se traduit par l'ensemble de fluents suivant :

$G = \{ \text{at}(\text{Ball1}, \text{RoomB}), \text{at}(\text{Ball2}, \text{RoomB}), \text{at-robby}(\text{RoomB}) \}$

Lors de la description du problème, nous avons vu que plusieurs actions étaient envisageables. Nous allons ici utiliser trois opérateurs :

Move(x, y) : (* Robby se déplace de la salle x à la salle y *)

Préconditions = { room(x), room(y), at-robby(x) }

Ajouts = { at-robby(y) }

Retraits = { at-robby(x) }

Pick(x, y, z) : (* Robby prend la balle x qui est dans la salle y dans sa pince z *)

Préconditions = { ball(x), room(y), gripper(z), at(x, y), at-robby(y), free(z) }

Ajouts = { carry(x, z) }

Retraits = { at(x, y), free(z) }

Drop(x, y, z) : (* Robby pose la balle x qui est dans sa pince z dans la salle y *)

Préconditions = { ball(x), room(y), gripper(z), carry(x, z), at-robby(y) }

Ajouts = { at(x, y), free(z) }

Retraits = { carry(x, z) }

Les auteurs ont ensuite créé un langage (qu'ils appellent "TSPL" qui vient de "Tunable SatPlan Language") et une implémentation de ce langage (qu'ils appellent "TSP" pour "Tunable SatPlan").

TSP permet d'automatiser la traduction des différents codages pour passer d'un problème de planification STRIPS à une base de clauses. Cette traduction automatique nous permet ensuite d'utiliser un solveur SAT pour trouver un modèle de la base de clauses et, par une traduction inverse, de fournir un plan solution lorsque la base est satisfiable.

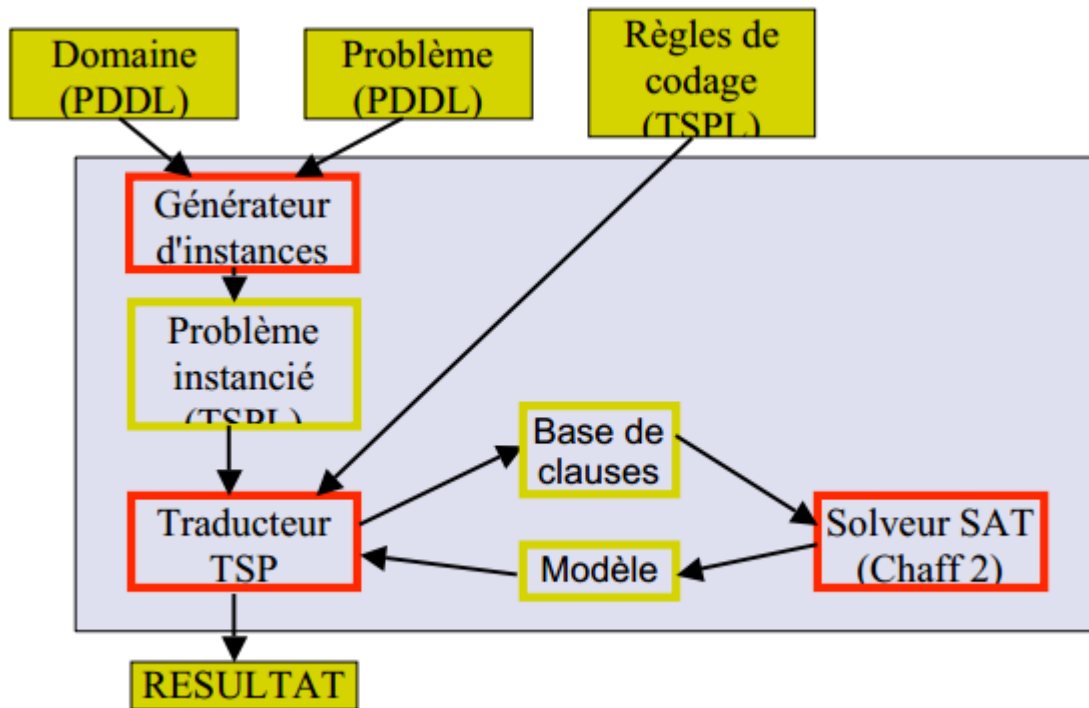


FIGURE 3.4 – Schéma de la réduction polynomiale de Planification vers SAT

3.1.5 A Reduced Codification for the Logical Representation of Job Shop Scheduling Problems[5]

Cet article propose la représentation que ses auteurs appellent Reduced SAT Formula, qui améliore significativement les réductions polynomiales classiques de JSSP vers SAT. Ils fournissent aussi des résultats expérimentaux de leur algorithme, et comparent le nombre de clauses créées par la représentation classique et par leur algorithme RSF.

Tout d'abord, les auteurs reviennent sur comment passer facilement d'un problème de Job Shop (JSSP) à un problème de satisfaction de contraintes (CSP).

Constraint	Interpretation
$s_i \geq 0$	<i>Starting time constraint:</i> The starting time of the operation i must be non-negative.
$s_i + p_i \leq s_j$	<i>Precedence constraint:</i> The operation i must be complete before j can begin.
$s_i + p_i \leq s_j \vee s_j + p_j \leq s_i$	<i>Resource capacity constraint:</i> The operations i and j are in conflict. They require the same resource and they cannot be scheduled concurrently.
$r_i \leq s_i$	<i>Ready time constraint:</i> The operation i cannot begin before its ready time.
$s_i + p_i \leq d_i$	<i>Deadline constraint:</i> The operation i cannot finish after its deadline.

FIGURE 3.5 – Tableau d'équivalence entre JSSP et CSP

Les auteurs expliquent ensuite comment passer de ce CSP a une version en SAT.

Logical clauses for the JSSP represented as a CSP

Constraint	CNF
$s_i + p_i \leq s_j$	$pr_{i,j}$
$s_i + p_i \leq s_j \vee s_j + p_j \leq s_i$	$pr_{i,j} \vee pr_{j,i}$
$s_i \geq r_i$	sa_{i,r_i}
$s_i + p_i \leq d_i$	eb_{i,d_i}

The coherence conditions for the SAT codification of a JSSP

Coherence condition	CNF	Interpretation
$sa_{i,t} \rightarrow sa_{i,t+1}$	$\sim sa_{i,t} \vee sa_{i,t+1}$	<i>Coherence of sa:</i> If i starts at or after time t , then it starts at or after the time $t+1$.
$eb_{i,t} \rightarrow eb_{i,t+1}$	$\sim eb_{i,t} \vee eb_{i,t+1}$	<i>Coherence of eb:</i> If i ends by time t , then it ends by time $t+1$.
$sa_{i,t} \rightarrow \sim eb_{i,t+p_i-1}$	$\sim sa_{i,t} \vee \sim eb_{i,t+p_i-1}$	<i>Coherence of p_i:</i> If i starts at or after time t , then it cannot end before time $t+p_i$.
$sa_{i,t} \wedge pr_{i,j} \rightarrow sa_{j,t+p_i}$	$\sim sa_{i,t} \vee \sim pr_{i,j} \vee sa_{j,t+p_i}$	<i>Coherence of $pr_{i,j}$:</i> If i starts at or after time t , and j follows i , then j cannot start until i is finished.

FIGURE 3.6 – Tableau d'équivalence entre JSSP et SAT

Cette représentation fonctionnerait sans soucis et fournirait effectivement, une équivalence entre le problème de JSSP et le problème SAT créé.

Seulement les auteurs ont trouvés qu'il existait des contraintes qui pouvaient être éliminé (afin de réduire la taille du problème SAT et d'augmenter la vitesse de sa résolution). Les auteurs ont donc réalisés un algorithme qu'ils appellent "Reduced SAT Formula ou RSF". Cet algorithme fournit une réduction du nombre de clauses de 75.9% !

Bien évidemment, ils justifient pourquoi certaines contraintes peuvent être supprimées, en voici les tableaux de justification :

Conditions for eliminating the scheduling constraints in a reduced codification

Constraints which can be eliminated	Rationale
Precedence constraints (pr_{ij})	For any JSSP, the precedence constraints are considered to define the problem and the equivalent clause pr_{ij} is always true. These clauses can be eliminated from the codification because their truth-values are always true.
Resource capacity constraints ($pr_{ij} \vee pr_{kl}$)	Because the resource capacity constraints specify the use of the same machine by two operations, and because the schedule for the problem is defined, the resource capacity constraints can be exchanged for the precedence constraints in the logical representation. When this is done, it is possible to see that these clauses can be eliminated from the codification because their truth-values are always true.
Ready time constraints ($sa_{i,n}$) and deadline constraints ($eb_{i,d}$)	If the LST of each operation in the defined schedule is found, the ready times and the deadlines of each operation can be determined. If these times are known, the clauses in the codification that contain $sa_{i,n}$ or $eb_{i,d}$ will be always true, so they can be eliminated from the codification.

FIGURE 3.7 – Contraintes de planification supprimées

Justification for eliminating the coherence conditions in a reduced codification

Constraints which can be eliminated	Rationale
Coherence of sa	Because t is the LST, then $t=r_i$. For clauses with the form $\sim sa_{i,t} \vee sa_{i,t}$, the literal $\sim sa_{i,t}$ is false, and $sa_{i,t}$ is true. The literal $sa_{i,t}$ is true because the operation i starts at r_i which is equal to t and s_i is after $r_{i,t}$. Therefore, the clauses are always true and they can be eliminated.
Coherence of eb	Because t is the LST, then $t=r_i$. For clauses with the form $\sim eb_{i,t} \vee eb_{i,t}$, the literal $\sim eb_{i,t}$ is true and $eb_{i,t}$ is false. Because the operation i cannot finish at its ready time or before, the literal $eb_{i,t}$ is false and $eb_{i,t}$ is also false. Therefore the clauses are always true and they can be eliminated.
Coherence of p_i	Because t is the LST, $t=r_i$. For clauses in the form $\sim sa_{i,t} \vee \sim eb_{i,t+n_i}$, the literal $\sim sa_{i,t}$ is false and $sa_{i,t}$ is true. The literal $\sim eb_{i,t+n_i}$ is true and $eb_{i,t+n_i}$ is false because the operation i cannot finish before of r_i+p_i . Therefore the clauses are always true and they can be eliminated.

FIGURE 3.8 – Conditions de cohérences supprimées

3.2 Algorithmes Évolutionnaires et SAT

3.2.1 Comparing the performance of the genetic and local search algorithms for solving the satisfiability problems [6]

Cet article donne une comparaison très détaillée des performances des algorithmes génétiques ainsi que des algorithmes en recherche locale dans le cas d'instances SAT.

Les auteurs ont utilisés 2 solveurs représentatif de chaque catégorie :

- **Recherche Locale** : DLM et ESG.
- **Algorithme Génétique** : SAT-WAGA et GASAT.

Parmi d'autres, voici les instances qu'ils ont considérés pour leurs benchmarks :

Instance	vars	Cls
N queens		
10 queen	100	1480
20 queen	400	12,560
50 queen	2,500	203,400
100 queen	10,000	1,646,800

FIGURE 3.9 – Sous-ensemble des benchmarks utilisés pour comparer RL et AG

Ils expliquent les avantages des recherches locales sur le peu d'algorithmes génétiques existants traitant de SAT et fournissent les résultats suivant qui sont sans appel :

Instance	Succ	Time	Instance	Succ	Time
N queens problem			N queens problem ($P = 0.2$ and $PS = 1$):		
10 queens	20/20	0.01	10 queens	20/20	0.02
20 queens	20/20	0.04	20 queens	20/20	0.04
50 queens	20/20	3.68	50 queens	20/20	1.03
100 queens	20/20	174.18	100 queens	20/20	15.23

FIGURE 3.10 – Résultat des solveurs DLM et ESG (Recherche locale)

Instance	Succ	Time	Instance	Succ	Time
N queens problem			N queens problem		
10queen	20/20	827.9	10queen	20/20	0.02
20queen	18/20	10127.28	20queen	20/20	0.23
50queen	0/20	—	50queen	20/20	4580.66
100queen	0/20	—	100queen	0/20	—

FIGURE 3.11 – Résultat des solveurs SAT-WAGA et GASAT (Algorithme génétique)

Les temps de calculs sont une moyenne des succès (/20) et le matériel utilisé pour afficher ces résultats est le suivant : PC Pentium III 800 MHz avec 256 MB de mémoire.

3.2.2 GASAT : a genetic local search algorithm for the satisfiability problem [7]

Cet article présente GASAT, un solveur hybride pour SAT. C'est un solveur qui est basé sur les algorithmes génétiques. L'hybridation est réalisée avec des algorithmes de recherche par liste tabou.

GASAT n'est pas forcément ce qui se fait de mieux en terme de rapidité d'exécution, en revanche l'hybridation réalisé avec une recherche par liste tabou permet de réduire le nombre de flip de variable a effectué pour trouver une solution.

Voici le tableau qui récapitule les résultats de GASAT face à d'autres solveurs génétiques. Les instances ont un ratio de $\frac{nbClauses}{nbVariables} = 4.3$.

- La suite A, 4 groupes de 3 instances avec 30, 40, 50 et 100 variables.
- La suite B, 3 groupes de 50 instances avec 50, 75 et 100 variables.
- La suite C, 5 groupes de 100 instances avec 20, 40, 60, 80 et 100 variables.

Voici un résumé des résultats :

Instances				GASAT		FlipGA		ASAP	
Suite	N.B.	Var	Clauses	% success	nb Flips	% success	nb Flips	% success	nb Flips
A	3	100	430	95	7550	100	127900	100	52276
B	50	100	430	69	28433	57	20675	59	43601
C	100	100	430	74	1533	62	20319	61	34548

Les auteurs déduisent aussi que GASAT n'a pas de bon résultat sur des instances générées aléatoirement.

3.2.3 FlipGA : Evolutionary Algorithms for the Satisfiability Problem [8]

Cet article parle de la meilleure représentation possible d'un individu pour utiliser des algorithmes évolutionnaires dans le cas de SAT.

Les auteurs ont étudiés :

- La représentation en chaîne de bits.
- La représentation en point flottant.
- La représentation en chemin.

La représentation en chaîne de bits est la plus simple et la plus intuitive : on associe à chaque variable une seule variable.

La représentation en point flottant fonctionne de la manière suivante :

Chaque individus est représenté par un vecteur $y \in [-1, 1]^n$, la transformation est réalisé en remplaçant les variables x_j et \bar{x}_j par $(y_j - 1)^2$ et $(y_j + 1)^2$ et en substituant les opérateurs ET et OU booléens par leurs équivalents arithmétiques - et +.

La représentation en chemin fonctionne de la manière suivante :

Les auteurs partent du fait qu'il faut qu'il y ait au moins 1 variable vrai dans chaque clause. Cela signifie qu'il doit exister un chemin qui passe au travers de toutes les clauses. Cette représentation permet de repérer non pas juste 1 seule solution, mais un ensemble de solution (vu qu'il suffit de changer le chemin si un choix quelque part nous le permet).

Il réalise des benchmarks des différentes représentations possibles. Selon leur conclusion, la représentation en chaîne de bit serait supérieure aux autres représentations testées.

Cet article est la description du solveur FlipGA.

3.2.4 Local Search with Three Truth Values for SAT Problems [9]

Cet article propose une nouvelle interprétation des problèmes SAT et MAX-SAT. Au lieu de considérer simplement vrai et faux, ils ajoutent la possibilité de considérer une variable comme "undefined" dans le but d'améliorer l'efficacité de la résolution. En utilisant cette idée, ils ont amélioré le solveur WalkSAT ainsi qu'une recherche tabou.

3.2.5 An improved deterministic local search algorithm for 3-SAT [10]

Sachant qu'un brute-force tournerait en $O * (2^n)$, les auteurs de cet article propose une recherche locale déterministe avec une complexité en $O * (1.473^n)$.

Bien évidemment, avec une telle complexité, l'algorithme est une méthode exacte. Leur méthode vient simplement du fait que les auteurs ne créent pas l'arbre de recherche sans réflexion, ils cherchent à chaque fois quel serait le meilleur branchement en fonction des différentes clauses au sein du problème.

Les auteurs réalisent une petite recherche locale et réalisent les branchements de leur arbre de recherche sur les clauses fausses, l'arbre de recherche se retrouve ainsi plus petit.

3.2.6 A Superior Evolutionary Algorithm for 3-SAT [11]

Dans cet article, les auteurs comparent l'heuristique WGSAT, qui est une heuristique simple mais incomplète¹, et 2 algorithmes évolutionnaires. Les résultats montrent que l'adaptation à son environnement pendant l'exécution est la meilleure approche.

Typiquement une adaptation simple dans un algorithme génétique consiste à faire varier la probabilité de réaliser un croisement et la probabilité de faire une mutation au cours de l'exécution de l'algorithme.

1. Ne garantie pas de trouver l'optimum

La puissance de l'approche des auteurs provient du fait que le mécanisme d'adaptation est complètement indépendant du problème.

Les auteurs appellent cette adaptation dans leur article : SAW pour "Stepwise Adaptation of Weights".

Le principe est le suivant :

- On affecte des poids à chaque clauses
- On possède une fonction de pénalité
- On pénalise les clauses qui restent fausses un certains nombre de génération
- On tente de résoudre ces clauses en priorité en forçant certaines mutations

3.3 SAT insatisfiable : UNSAT

3.3.1 GUNSAT : a Greedy Local Search Algorithm for Unsatisfiability [12]

GUNSAT est un solveur qui utilise le principe de résolution pour obtenir une clause vide. Si une clause vide est obtenue de cette manière, cela signifie que l'instance CNF en cours n'a pas de solution et que le problème est UNSAT.

Algorithm : GUNSAT

```

Data :  $\Sigma$  a CNF formula
Result : UNSAT if a derivation of  $\perp$  is found, UN-
          KNOWN otherwise
begin
  for  $i=1$  to MaxTries do
    for  $j=1$  to MaxFlips do
      if 2-Saturation( $\Sigma$ ) returns UNSAT then
        return UNSAT ;
      if  $|\Sigma| > \text{MaxSize}$  then
        Remove-One-Clause( $\Sigma$ )
        Add-One-Clause( $\Sigma$ ) ;
        Add-Extended-Variables( $\Sigma$ ) ;
        Simplify-Look-Ahead( $\Sigma$ ) ;
      end
      Replace  $\Sigma$  by all its vital clauses;
    end
  return UNKNOWN;
end

```

FIGURE 3.12 – gunSAT

Facile à mettre en place, seulement GUNSAT ne réalise qu'une seule résolution à chaque étape et de manière gloutonne, là où les solveurs actuels peuvent faire jusqu'à 100 étapes. GUNSAT n'était qu'un proof of concept.

3.3.2 Learning in local search [13]

Cet article parle d'une amélioration de GUNSAT basée sur le principe de CDLS ([Conflict Driven Local Search](#)). L'apprentissage des clauses est basé sur le même principe que les solveurs CDCL et on apprend donc des clauses beaucoup plus intéressantes.

Ces clauses apprises rendent le problème équisatisfiable, elles peuvent donc être ajoutés au problème sans aucun soucis.

Algorithm : CDLS

Input: Σ a CNF formula
Output: *SAT* if Σ is satisfiable, *UNSAT* if Σ is unsatisfiable, else *UNKNOWN*

```

1 for  $i \leftarrow 1$  to  $MaxTries$  do
2    $\mathcal{I}_c \leftarrow \text{completePUInterpretation}(\Sigma);$ 
3   for  $j \leftarrow 1$  to  $MaxFlips$  do
4     if  $\mathcal{I}_c \models \Sigma$  then
5       return SAT;
6      $\Gamma = \{\alpha \in \Sigma \mid \mathcal{I}_c \not\models \alpha\};$ 
7     while  $\Gamma \neq \emptyset$  do
8        $\alpha \in \Gamma;$ 
9       if  $\exists x \in \alpha$  allowing a descent then
10         $flip(x);$ 
11        break;
12      else
13         $\Gamma \leftarrow \Gamma \setminus \{\alpha\};$ 
14      if  $\Gamma = \emptyset$  then /* local minimum */
15         $\alpha \in \Sigma$  such that  $\mathcal{I}_c \not\models \alpha;$ 
16         $\beta \leftarrow \text{conflictAnalysisRL}(\Sigma, \mathcal{I}_c, \alpha);$ 
17        if  $\beta = \perp$  then
18          return UNSAT;
19         $\Sigma \leftarrow \Sigma \cup \{\beta\};$ 
20 return UNKNOWN;

```

FIGURE 3.13 – CDLS

3.3.3 On the limit of branching rules for hard random unsatisfiable 3-SAT [14]

Cet article parle de la limite des règles de branchement dans la procédure DLL (Davis-Logemann-Loveland) et essaie de répondre à la question : Quelle serait la taille de l'arbre de recherche si chaque variable de branchement était la meilleure possible ? En effet, la conception de meilleures règles de branchement a un énorme intérêt pratique dans la résolution des problèmes UNSAT.

3.3.4 Clauses critiques et extraction de MUS [15]

Le chapitre 4 de cette thèse parle de 2 choses très importantes si l'on veut optimiser la preuve du UNSAT : Les clauses critiques et les MUS.

Clause Critique : Soit Σ une formule CNF. Une clause $c \in \Sigma$ invalidé par une interprétation ω est dite critique si et seulement si l'opposé de chaque littéral de c apparaît dans au moins une clause de Σ unisatisfait par ω . Ces clauses unisatisfaites relatives à une clause critique c sont dites liées à c .

Une clause C est unisatisfait par une interprétation I si et seulement si exactement un littéral de C est satisfait par I . Prenons un exemple, les clauses $\{\neg a \vee b$ et $\neg b \vee c$ et $\neg c \vee a\}$ sont unisatisfaites par $I = \{a, b, c\}$.

MUS : Formule Minimale Insatisfaisable Un MUS Γ d'une instance SAT Σ est un ensemble de clauses tel que :

- $\Gamma \subseteq \Sigma$
- Γ est insatisfaisable
- $\forall \Delta \subset \Gamma, \Delta$ est satisfaisable

L'idée du chapitre 4, qui est le seul qui nous intéresse ici, est de se dire : "Si un problème SAT n'a pas de solution, c'est à cause d'un sous-ensemble de clauses, si on trouve ce sous-ensemble, cela limite les possibilités de faire des résolutions inutiles."

3.3.5 A Machine-Oriented Logic Based on the Resolution Principle [16]

Cet article parle de la règle de résolution afin de réussir à prouver qu'un problème SAT n'a aucune solution. C'est sur cette règle que reposent des solveurs actuels.

La règle de résolution s'applique sur deux clauses, c'est-à-dire des formules composées de disjonctions de littéraux, un littéral étant un atome (positif) ou un atome précédé d'une négation (négatif).

Prenons un exemple :

$$C_1 = L_1 \vee \dots \vee L_i \vee \dots \vee L_m$$

$$C_2 = M_1 \vee \dots \vee M_j \vee \dots \vee M_n$$

où les littéraux L_i et M_j sont l'un positif et l'autre négatif et qu'ils portent sur le même atome.

Le résolvant de C_1 et C_2 sur L_i et M_j est la clause

$$C_R = \left(L_1 \vee \dots \vee L_{i-1} \vee L_{i+1} \vee \dots \vee L_m \right) \vee \left(M_1 \vee \dots \vee M_{j-1} \vee M_{j+1} \vee \dots \vee M_n \right)$$

Si par résolution successive, on arrive à une clause \perp alors le problème n'a pas de solution.

3.4 Livre de référence dans le domaine

3.4.1 Problème SAT : progrès et défis [17]



FIGURE 3.14 – Problème SAT : progrès et défis

Ce livre est une mine d'or pour quiconque s'intéresse à SAT. Il fournit une théorie très solide, un aperçu de ce qui se fait dans le monde de la recherche en ce moment même (2015).

Il précise différents types d'algorithmes pour résoudre des problèmes SAT, que ce soit la procédure DP, la recherche locale, la recherche stochastique sans mémoire, avec mémoire à court/long terme, et même des algorithmes à base de populations.

Enfin, il fournit aussi des applications concrètes de SAT à travers la vérification de circuits intégrés ou encore des problèmes de planification.

Je vais donc maintenant résumer les chapitre qui traitent directement de mon projet de fin d'études :

Chapitre 1 : Phénomènes de seuil

Ce chapitre traite d'un phénomène qui est observé sur les problèmes SAT générés aléatoirement. En effet, quand on regarde le ratio $(\frac{nbClauses}{nbVariables})$, on se rend compte qu'il y a une valeur très particulière qui est de 4.25. A cette valeur nous avons autant de chance d'avoir une instance SAT ou une instance UNSAT.

C'est donc une valeur à prendre en compte lorsque l'on réalise des benchmarks.

Chapitre 2 : Classes polynomiales

Ce chapitre fait un rapide résumé des concepts relatifs aux problèmes SAT, à savoir : qu'est-ce qu'un Graphe d'implication, que veut dire être polynomiale ? être non-déterministe polynomiale ? Qu'est-ce qu'une clause de Horn ? Qu'est-ce qu'un CSP ?

Chapitre 3 : Techniques de simplifications

Ce chapitre parle des différents pré-traitements possibles à réaliser sur un problème SAT afin de simplifier ce dernier. En effet, il est possible de rechercher les clauses redondantes, de supprimer certaines variables, ...

Chapitre 4 : Algorithmes de recherche systématique

Ce chapitre parle des méthodes de résolution qui permettent de voir une certaine "progression". En effet, on va parler de résolution logique (comme l'a réalisée la procédure DP en 1960 même si cette dernière est $\exp(N)$ en espace), on va aussi parler de la méthode DPLL qui est une amélioration à travers une recherche arborescente et enfin de l'algorithme des solveurs actuels : l'algorithme CDCL.

Chapitre 5 : Algorithmes de recherche stochastique

Ce chapitre parle à la fois des algorithmes génétiques, mais aussi des recherches stochastiques simples comme le recuit simulé par exemple. Il va aussi traiter l'algorithme GSAT qui est une référence dans les recherches stochastiques, et il va expliquer en quoi cet algorithme a été une avancée dans le domaine de la recherche.

Chapitre 8 : Génération de sous-formules minimalement inconsistantes (MUS)

Ce chapitre parle de la génération des MUS, un MUS étant le sous-ensemble de clauses qui fait qu'un problème SAT n'aura pas de solution. Il existe des méthodes pour trouver ce sous-ensemble, et les mettre en place permet d'améliorer la preuve du UNSAT par résolution successive (étant donné que le choix des clauses pour les résolutions est réduit).

Chapitre 11 : Planification par satisfaction de bases de clauses

Ce chapitre parle de la réduction polynomiale des problèmes de Planification en SAT. Il explique comment réaliser l'encodage et pourquoi le faire de cette manière là. Il fournit aussi des notes bibliographiques sur les différents articles de recherche ayant traité le sujet.

3.4.2 Handbook of Satisfiability [18]

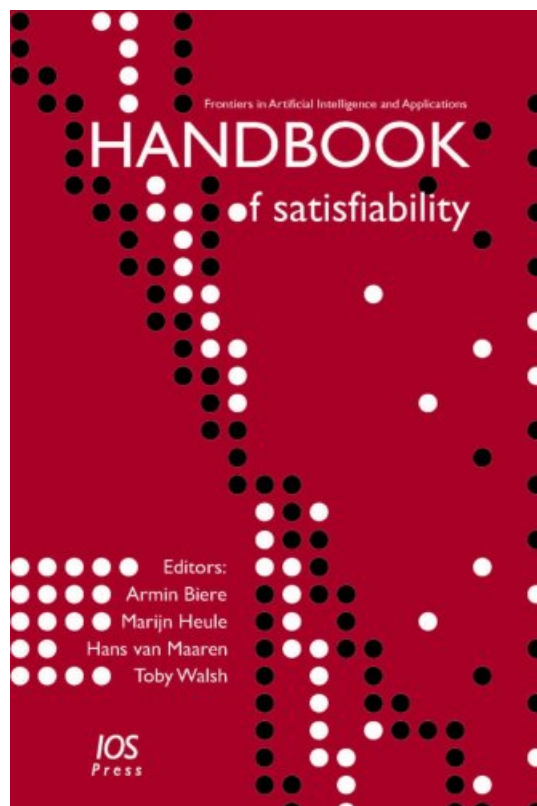


FIGURE 3.15 – Handbook of Satisfiability

Ce livre est considéré comme LA référence sur les problèmes SAT et cela s'explique parfaitement.

Il analyse et explique comment implémenter énormément de méthodes pour résoudre SAT. Il explique par exemple, la procédure DP ainsi que son amélioration, la procédure CDCL, qui est la procédure la plus utilisée de nos jours (2014). Il explique aussi comment fonctionnent les solveurs Look-Ahead et comment ces derniers peuvent être améliorés.

Il fournit énormément d'applications possibles que ce soit le Bounded Model Checking, la planification, la vérification de circuit intégré...

Et enfin ce livre ne s'intéresse pas qu'aux problèmes SAT, il s'intéresse aussi aux problèmes similaires comme MaxSAT, les QBF (Quantified Boolean), les SMT (Satisfiability Modulo Theories), ...

Chapitre 2 : CNF Encodings

Ce chapitre fait un tour complet sur qu'est-ce que le format CNF ? pourquoi encoder les problèmes SAT de cette manière ? Qu'est-ce que cela amène comme décisions à faire pour réaliser un solveur SAT ?

Chapitre 3 : Complete Algorithms

Ce chapitre parle des différentes manières de réaliser un solveur complet. Ils insistent beaucoup sur l'algorithme DPLL qui est la genèse de beaucoup des solveurs utilisés de nos jours.

Chapitre 4 : CDCL Solvers

Ce chapitre explique tout ce qu'il y a à savoir sur les solveurs CDCL. Comment l'algorithme fonctionne, comment utiliser la propagation unitaire pour le cas UNSAT à travers ce type d'algorithmes, comment faire une analyse de conflits pour pouvoir apprendre des clauses intéressantes...

Chapitre 6 : Incomplete Algorithms

Ce chapitre montre différentes méthodes pour réaliser des algorithmes incomplets fonctionnels, cela va de la simple recherche gloutonne à des méthodes extrêmement complexes comme la recherche par Méthodes de Lagrange discrets.

Un algorithme incomplet pour résoudre SAT signifie que cet algorithme ne sera pas capable de garantir qu'il va trouver une solution ou si jamais aucune solution existe, de prouver que le problème n'a pas de solution.

Typiquement, ces algorithmes possèdent un temps limite avant d'arrêter de chercher, au quel cas, ils afficheront que la réponse au problème est toujours indéfinie.

An incomplete method for solving the propositional satisfiability problem (or a general constraint satisfaction problem) is one that does not provide the guarantee that it will eventually either report a satisfying assignment or declare that the given formula is unsatisfiable. In practice, most such methods are biased towards the satisfiable side : they are typically run with a pre-set resource limit, after which they either produce a valid solution or report failure ; they never declare the formula to be unsatisfiable.

Chapitre 11 : Minimal Unsatisfiability and Autarkies

Ce chapitre est quasiment exhaustif sur tout ce qu'il faut savoir sur les problèmes UNSAT et comment les résoudre. Il présente le principe d'"autarcie" qui représente une manière générale pour traiter la redondance dans l'ensemble des clauses d'un problème SAT. Éliminer la redondance d'un problème permet d'améliorer la preuve par résolution.

Chapitre 15 : Planning and SAT

Ce chapitre explique différentes manières pour réaliser une réduction polynomiale des problèmes de planification vers problème SAT. Il explique comment faire varier la réduction polynomiale pour passer d'un problème de planification séquentielle à un problème de planification parallèle.

3.4.3 Introduction to Mathematics of Satisfiability [19]

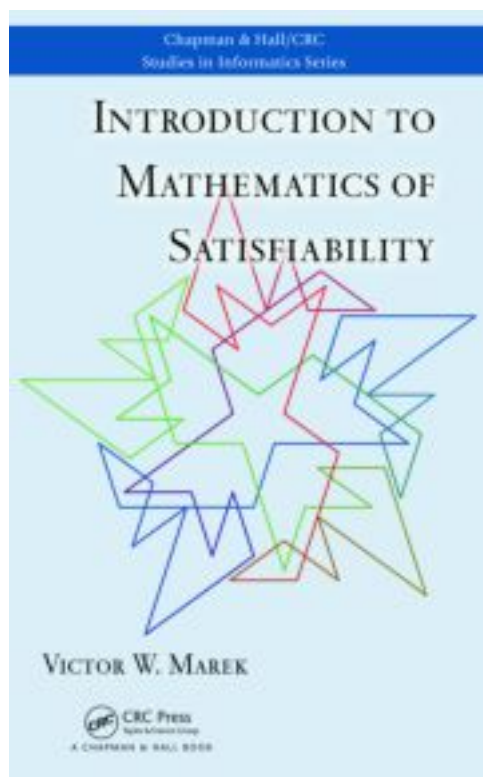


FIGURE 3.16 – Introduction to Mathematics of Satisfiability

Chapitre 2 : Resolution Calculus

Ce chapitre traite du principe de résolution ainsi que la preuve du UNSAT en utilisant ce principe. Il parle aussi d'un ensemble d'outils pour améliorer cette preuve comme les clauses unitaires, les clauses subsumées, les littéraux purs...

Chapitre 4 : Backtracking and DPLL Algorithms

Ce chapitre traite des algorithmes basés sur DPLL. Par exemple il parle de l'algorithme MS (Monien-Speckenmeyer), de l'algorithme PPZ (Paturi-Pudlak-Zane), ainsi que de comment réaliser l'apprentissage des clauses à l'aide de propagation unitaire.

Chapitre 5 : Local Search and Hamming Balls

Ce chapitre parle d'utiliser la distance de Hamming au lieu du nombre de clauses insatisfaites comme mesure pour améliorer la recherche locale. En utilisant ce principe, ils introduisent l'idée d'une balle d'Hamming. L'idée est de chercher dans un certain sous-ensemble de voisins s'il existe ou non une solution et ainsi avancer de balle en balle.

3.4.4 The Satisfiability Problem [20]

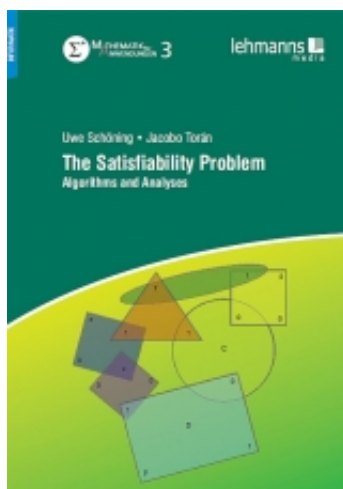


FIGURE 3.17 – The Satisfiability Problem

Chapitre 3 : Normal forms of formulas

Ce chapitre explique de manière théorique et pratique ce qu'est et comment utiliser le format CNF des problèmes SAT afin de pouvoir résoudre ces derniers.

Chapitre 7 : Clausal logic and resolution

Ce chapitre fournit une solide théorie sur le principe de résolution et en quoi ce dernier est complet et permet de fournir une preuve que le problème initial n'a pas de solution. Il parle aussi du lemme de Davis-Putnam qui permet de détecter un sous-ensemble de variables sur lesquelles on doit appliquer le principe de résolution.

Chapitre 10 : Embedding SAT into integer programming and into matrix algebra

Ce chapitre est très intéressant au niveau pratique, au lieu de voir les clauses comme ensemble de littéraux reliés par des OU logique, ils ont décidé de voir les clauses comme des inégalités.

Il en résulte une toute autre manière de visualiser et donc de résoudre le problème SAT à travers l'algèbre matricielle.

Chapitre 12 : Computational knowledge representation with SAT

Ce n'était pas tout à fait le but de ce chapitre, mais il en ressort une très bonne explication de la réduction polynomiale. En effet, durant ce chapitre, le principe de la réduction polynomiale et une réduction polynomiale du Sudoku $N \times N$ sont expliqués et très détaillés afin de bien visualiser comment passer d'un problème à l'autre.

Chapitre 4

Étude des autres solveurs

4.1 Introduction

L'un des objectifs de ce PFE était de réaliser une étude comparative entre le solveur réalisé et ceux déjà existant.

L'un des reproches que je me fais à posteriori sur cette partie, c'est que le choix des solveurs au moment de réaliser les spécifications n'était pas assez diversifié.

Les solveurs ne sont pas tous complet, et certains utilisent la même manière fondamentale de résoudre les problèmes.

Difficile de comparer de manière précise si les adversaires ont beaucoup de similarités.

J'ai d'ailleurs pris la liberté d'ajouter GaSAT qui est un autre solveur SAT qui utilise des algorithmes génétiques, même si ce dernier n'était pas demandé par le cahier des spécifications.

Ainsi cela donne aussi la position de SATyr par rapport à un solveur qui serait dans la même catégorie que lui.

4.2 WalkSAT (Selman, Kautz, 1996)

WalkSAT fait partie des tout premiers solveurs performants, il n'est pas capable de prouver UNSAT, le solveur est donc incomplet, en revanche il est assez performant sur les instances certifiées SAT. En voici son pseudo-code :

Algorithm 1 WalkSAT

Require: Une instance SAT F
Ensure: Renvoyer une valuation qui satisfait F si l'algorithme termine

loop

 $T :=$ une valuation tirée aléatoirement

 for $i := 1$ **to** $Maxiter$ **do**

 if T satisfait F **then**

 return T

 else

 $C :=$ une clause de F aléatoirement choisie non satisfaite par T

 if $rand(0, 1) < p$ **then**

 Flipper dans T une variable aléatoirement choisie dans C

 else

 Flipper dans T une variable dans C qui minimise le nombre de clauses rendues fausses après le flip

 end if

 end if

 end for
end loop

FIGURE 4.1 – Algorithme du solveur WalkSAT

WalkSAT décrit ici ne garantit pas la terminaison, mais la validité de la solution est évidente en cas de terminaison. Dans la pratique, il y a souvent une contrainte de temps de calcul pour que l'algorithme ne boucle pas indéfiniment.

4.3 FlipGA (Marchiori, Rossi, 1999)

FlipGA aura été l'inspiration principale pour SATyr durant ce PFE, en voici l'article de présentation : <http://www.cs.ru.nl/~elenam/fsat.pdf>.

Feature	SAWEA	RFEA	FlipGA	ASAP
replacement	$(1, \lambda^*)$	steady-state	generational	$(1 + 1)$
parent selection	–	tournament	fitness proportional	–
fitness	f_{SAW}	f_{REF}	f_{MAXSAT}	f_{MAXSAT}
initialization	random	random	random	random
crossover	–	–	uniform	–
mutation	MutOne	knowledge-based	random	random adaptive
local search	–	–	flip heuristic	flip heuristic
adaptation	fitness	fitness	–	tabu list

FIGURE 4.2 – Comparaison des différents algorithmes évolutionnaires

FlipGA explique en quoi chacune de ses fonctionnalités permet d'améliorer les résultats, c'est donc en ça que SATyr en est une inspiration.

Replacement : Generational Le principe étant de se servir des meilleurs (selon la fitness) de la population précédente pour générer les enfants de la population suivante.

Parent Selection : Fitness proportional La sélection des parents dans FlipGA fonctionne selon le principe de roulette wheel ¹

Fitness : f_{maxSAT} La fitness de chaque individu se calcule de la manière suivante : $\left(\frac{nbClausesVraies}{nbClausesTotal}\right)$ ainsi un individu ayant une fitness de 1.00 sera un individu solution du problème.

Initialization : Random très simplement, pour initialiser chaque individu au sein de la population, on leur affecte une assignation choisie aléatoirement. La sélection naturelle va faire en sorte que les mauvais disparaissent au profit des bons.

Crossover : uniform FlipGA, tout comme SATyr, utilise un croisement d'individus uniforme. Cela signifie que lorsque l'on prend 2 parents, l'enfant va posséder en moyenne 50% du parent 1 et 50% du parent 2.

Mutation : random Pour réaliser une mutation sur un individu, on va sélectionner une variable au hasard, et on va inverser sa valeur.

Local search : flip heuristic fonctionne comme la mutation, sauf que les changements de valeur de variables sont réalisés dans l'optique d'améliorer la fitness de l'individu.

1. http://en.wikipedia.org/wiki/Fitness_proportionate_selection

4.4 zChaff (Moskewicz, Madigan, 2001)

zChaff est un solveur complet et extrêmement performant. Très simplement, il correspond à un algorithme DPLL² avec un système d'apprentissage. En voici le code source :

```
while(1) {
    if (decide_next_branch()) { //Branching
        while(̄deduce()==conflict) { //Deducing
            blevel = analyze_conflicts(); //Learning
            if (blevel < 0)
                return UNSAT;
            else back_track(blevel); //Backtracking
        }
    }
    else //no branch means all variables got assigned.
        return SATISFIABLE;
}
```

FIGURE 4.3 – Algorithme de zChaff

L'algorithme est basé sur une méthode de backtracking. Il procède en choisissant un littéral, lui affecte une valeur de vérité, simplifie la formule en conséquence, puis vérifie récursivement si la formule simplifiée est satisfaisable.

Si c'est le cas, la formule originale l'est aussi, dans le cas contraire, la même vérification est faite en affectant la valeur de vérité contraire au littéral. Dans la terminologie de la littérature DPLL, c'est la conséquence d'une règle dite "splitting rule" (règle de séparation), et sépare le problème en deux sous problèmes.

L'étape de simplification consiste essentiellement en la suppression de toutes les clauses rendues vraies par l'affectation courante, et tous les littéraux déduits faux à partir de l'ensemble des clauses restantes.

DPLL étend l'algorithme de backtracking par l'utilisation des deux règles suivantes :

La propagation unitaire

Si une clause est unitaire, c'est-à-dire qu'elle contient un et un seul littéral, elle ne peut être satisfaite qu'en affectant l'unique valeur qui la rend vraie à son littéral. Il n'y a par conséquent plus à choisir. En pratique, son application entraîne une cascade d'autres clauses unitaires de manière déterministe, et évite donc d'explorer une grande part de l'espace de recherche. Elle peut être vue comme une forme de propagation de contraintes.

L'élimination des littéraux dits purs

Si une variable propositionnelle apparaît seulement sous une forme (soit seulement positive, soit seulement négative) ses littéraux sont dits purs. Les littéraux purs peuvent être affectés d'une manière qui rend toutes les clauses qui les contiennent vrais. Par conséquent

2. http://fr.wikipedia.org/wiki/Algorithme_DPLL

ces clauses ne contraignent plus l'espace de recherche et peuvent être éliminées. L'incohérence d'une affectation partielle des variables est détectée quand une clause devient vide, c'est-à-dire quand toutes ses variables sont affectées de manière à ce que les littéraux correspondants soient faux. La satisfiabilité d'une formule est déterminée quand toutes les variables sont affectées sans qu'une clause ne devienne vide, ou bien, dans les implantations modernes de l'algorithme, quand toutes les clauses sont satisfaites. L'incohérence de la formule complète ne peut être déterminée qu'après une recherche exhaustive.

La preuve du UNSAT chez zChaff est réalisé, comme pour beaucoup d'autre solveur à l'aide de la règle de résolution qui peut s'illustrer de la manière suivante :

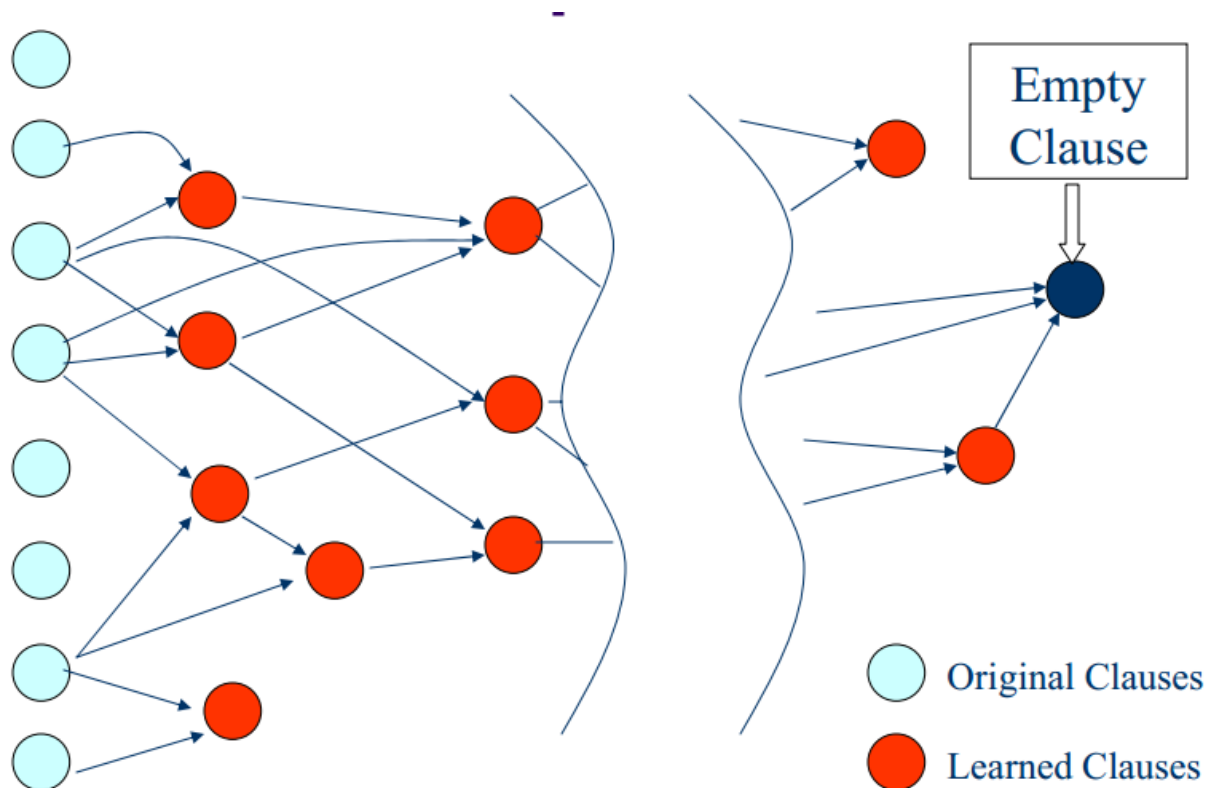


FIGURE 4.4 – Illustration de la règle de résolution

Très simplement, si par résolution successive, on obtient une clause vide, cela signifie que le problème n'a pas de solution.

4.5 SAT4j (Le Berre, Parrain, 2004)

Le solveur SAT4J (accessible ici : <http://www.sat4j.org/>) est un cas très intéressant de solveur SAT vu que ses créateurs ont réussi à intéresser des industriels.

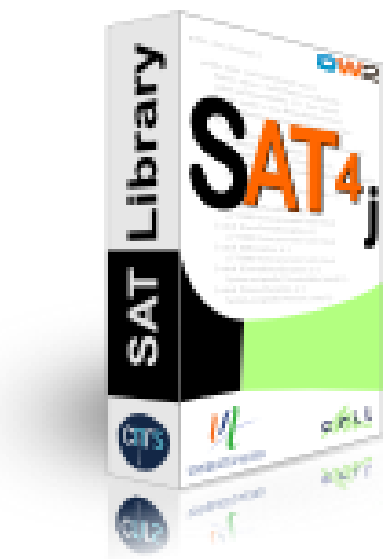


FIGURE 4.5 – Logo du solveur SAT4J

À l'origine, SAT4J n'était qu'une implémentation en Java du solveur MiniSAT. Cela a rapidement évolué et SAT4J est devenu aujourd'hui une librairie Java capable de résoudre des problèmes SAT ainsi que des problèmes d'optimisations.

Bien évidemment, le fait que le code soit en Java rend ce solveur quasiment inutilisable dans des compétitions de rapidité (Java est environ 3.25x plus long qu'un équivalent C++ selon eux). En revanche, le code a été pensé extrêmement flexible, il possède énormément de design pattern tels que le DP Decorator et le DP Strategy par exemple.

En preuve de sa qualité de solveur voici la liste des industriels qui l'utilisent (on remarquera l'IDE [Eclipse](#) qui l'utilise pour s'assurer qu'il n'y a aucun soucis dans les dépendances de ses plugins)

Our most famous adopter!



Software Engineering



VissBIP

GAVS+

Formal Verification



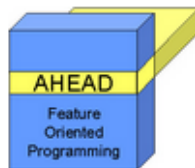
KODKOD

a constraint solver for relational logic



Software Product Lines

C2O 2.0



TypeCheck



Bioinformatic

GNA. sim



FIGURE 4.6 – Utilisateurs du solveur SAT4J

4.6 MiniSAT (Eén et Sörensson, 2005)

Pour apprendre beaucoup plus en détails comment fonctionne ce solveur, en voici l'article de recherche : <http://minisat.se/downloads/MiniSat.pdf>.

Ce solveur était le plus performant de la compétition des solveurs qui a eu lieu en 2005 comme le prouve ces différentes coupes gagnées :



FIGURE 4.7 – Trophées gagnés par MiniSAT en 2005

MiniSAT est, comme tous les bons solveurs de nos jours, basés sur la procédure DPLL. En voici l'algorithme :

```

loop
  propagate()  - propagate unit clauses
  if not conflict then
    if all variables assigned then
      return SATISFIABLE
    else
      decide()  - pick a new variable and assign it
  else
    analyze()   - analyze conflict and add a conflict clause
    if top-level conflict found then
      return UNSATISFIABLE
    else
      backtrack() - undo assignments until conflict clause is unit

```

FIGURE 4.8 – Algorithme du solveur miniSAT

MiniSAT a la particularité, en comparaison avec les autres solveurs DPLL, de générer beaucoup de conflits au début de la résolution, et ainsi être très performant sur les problèmes "évidemment UNSAT" (là où les conflits sont rapidement identifiables).

4.7 CDLS (Audemard, Lagniez, 2009)

CDLS, les initiales viennent de Conflict Driven Local Search. Le solveur est un solveur capable d'apprendre et est basé sur une recherche locale. Il fonctionne selon les mêmes principes que les solveurs CDCL mais adapté à une recherche locale.

Le solveur est donc lui aussi basé sur la technique de propagation unitaire pour construire un graphe d'implication. Ce solveur est donc capable, même au travers de la recherche locale, de prouver l'inconsistance des problèmes.

$$\begin{array}{llll} \phi_1 : (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) & \phi_2 : (x_1 \vee \overline{x_4} \vee \overline{x_5}) & \phi_3 : (x_2 \vee \overline{x_1}) & \phi_4 : (x_4 \vee \overline{x_7} \vee \overline{x_6}) \\ \phi_5 : (x_3 \vee \overline{x_5}) & \phi_6 : (x_5 \vee \overline{x_7}) & \phi_7 : (x_6 \vee \overline{x_8}) & \phi_8 : (x_7 \vee \overline{x_8}) \\ \phi_9 : (x_8 \vee \overline{x_4}) & \phi_{10} : (x_1 \vee \overline{x_8}) & \phi_{11} : (x_7 \vee \overline{x_9}) & \end{array}$$

$$\mathcal{I} = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9\}$$

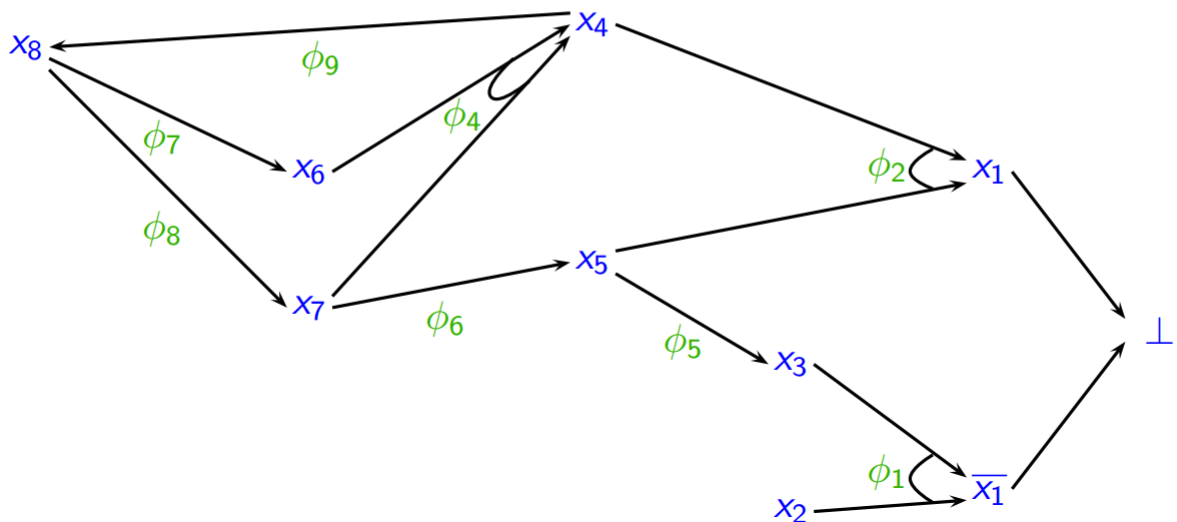


FIGURE 4.9 – Illustration du graphe d'implication généré par propagation unitaire

Extraire des informations de ce graphe n'est pas évident.

- Des cycles peuvent être présents (Le principe de résolution a créé des tautologies)
- Comment s'arrêter? Traiter toutes les variables (cela coûte du temps)? Considérer juste un sous-ensemble (lequel)?

Cependant CDLS y arrive, et grâce à ces informations, il est capable de prouver l'inconsistance UNSAT et bien évidemment de trouver une solution quand cette dernière existe (comme n'importe quel autre solveur en recherche locale).

4.8 Glucose (Audemard, Simon, 2014)

Glucose est le solveur le plus récent que j'ai manipulé durant mon PFE. Il existe la version Glucose-syrup, qui correspond à une programmation parallèle du solveur Glucose. Maintenant, qu'est-ce que Glucose ?

Glucose tire son nom du jeu de mots avec "glue clauses" (concept détaillé ensuite).



FIGURE 4.10 – Illustration du logo du solveur Glucose

La force de Glucose vient du fait qu'il cherche en priorité la preuve du UNSAT la plus courte possible. Pour cela il a un système d'apprentissage de bonne clause et de suppression de mauvaise clause extrêmement agressif (pour la compétition SAT en 2011, sur l'ensemble des clauses que Glucose avait appris, 93% d'entre elles avaient été supprimées).

Glucose, comme son nom l'indique, est à la recherche de clause glue afin d'améliorer l'apprentissage/suppression de ses clauses. Une clause glue est par définition une clause qui a un LBD de 2. Le LBD étant le "Literals Blocks Distance". Très simplement, pour une clause donnée C , il y a une partition de ses littéraux à l'intérieur de N sous-ensembles en fonction de l'affectation des variables courantes, le LBD de C est exactement N .

Très simplement, et comme tous les solveurs basés sur la procédure CDCL, Glucose est capable d'apprendre du passé (Conflict Driven Clause Learning). Par exemple, quand il s'agit de choisir une variable, Glucose choisit une variable qui est apparu récemment dans des conflits, ainsi le conflit remonte le plus haut possible dans l'arbre de recherche et Glucose peut ainsi couper des branches inutiles.

Et c'est avec cette procédure CDCL ainsi que le côté agressif de Glucose que ce solveur arrive à obtenir d'aussi bons résultats.

Chapitre 5

SATyr : le solveur génétique SAT + UNSAT

5.1 Introduction

Après avoir longuement parlé de ce qu'ont rédigés les différents acteurs du domaine des problèmes SAT (chapitre 3), après avoir aussi expliqué comment fonctionne un certain nombre de solveurs déjà existants (chapitre 4), il est maintenant temps d'expliquer ce qui a été réalisé durant ce PFE et surtout comment et pourquoi.

5.2 Lecture de fichier CNF au format DIMACS

Tout d'abord, avant de parler de comment fonctionne le solveur, il faut définir de quoi est composé un problème et comment il est représenté informatiquement.

On dit qu'une formule logique, sous forme normale conjonctive, parfois abrégée CNF¹ sigle de l'anglais Conjunctive Normal Form, est satisfaisable s'il est possible d'associer une valeur logique booléenne à chacune de ses variables afin que cette formule soit logiquement vraie.

Si la formule propositionnelle n'est pas sous forme normale conjonctive, il est nécessaire de la normaliser pour que le problème soit qualifié de SAT. Généralement, la réponse à cette question fournit également un exemple d'assignation des variables pour laquelle la formule sous forme normale conjonctive soit logiquement vrai.

Prenons un exemple :

Soit l'ensemble de variables $\{v_1, v_2, v_3\}$ et la formule

$$f = (v_1 \vee v_2) \wedge (\neg v_1 \vee v_3) \wedge (\neg v_2 \vee \neg v_1).$$

f est satisfaisable puisque, si on pose $v_1 = \text{vrai}$, $v_2 = \text{faux}$, $v_3 = \text{vrai}$, alors f est logiquement vraie.

En revanche, $f' = (v_1 \vee v_2) \wedge (\neg v_1 \vee v_3) \wedge (\neg v_2 \vee v_1) \wedge (\neg v_2 \vee v_3) \wedge (\neg v_1 \vee \neg v_3)$ n'est pas satisfaisable, car f' sera évalué comme fausse quelles que soient les valeurs attribuées à v_1, v_2 et v_3 .

1. pour en savoir plus sur CNF : http://fr.wikipedia.org/wiki/Forme_normale_conjonctive

5.2.1 Explication du format DIMACS

Cet exemple insatisfiable serait représenté en machines dans un fichier de cette manière :

```
c l'exemple au dessus représenté en CNF via le format DIMACS
p cnf 3 5
 1  2 0
-1  3 0
-2 -1 0
-2  3 0
-1 -3 0
```

- La ligne `c l'exemple au dessus représenté en CNF via le format DIMACS` et plus largement toutes les lignes commençant par c seront des lignes de commentaire.
- La ligne `p cnf 3 5` est très importante, elle doit se lire de cette manière : le p pour indiquer que la définition du problème se trouve à cette ligne, le cnf pour indiquer que le problème est en CNF, le premier entier correspond au nombre de variables présentes dans le problème et le deuxième entier représente le nombre de clauses.
- La ligne `1 2 0` et plus largement, toutes les lignes qui suivent la ligne démarrant par p et qui ne sont pas des commentaires sont les lignes qui servent à représenter les clauses. Elles se lisent de cette manière -i représente $\neg v_i$ et i représente v_i et ce, qu'importe la valeur de i. La ligne doit aussi se terminer par un 0 afin d'indiquer au solveur entrain de lire ce fichier que c'est la fin de la définition de la clause.

Le but de n'importe quel solveur est donc de trouver une valeur pour chacun des i entre {vrai et faux} de telle sorte qu'il y ait au moins une valeur vraie dans chacune des clauses, nous allons maintenant voir comment fait SATyr pour trouver cette affectation de chacun des i.

5.3 SAT : trouver une solution à un problème

SATyr est un solveur SAT qui utilise des algorithmes génétiques afin de trouver une solution quand cette dernière existe. Cette partie va donc utiliser le vocabulaire classique des algorithmes génétiques à savoir : population, fitness (ou évaluation), individu ...

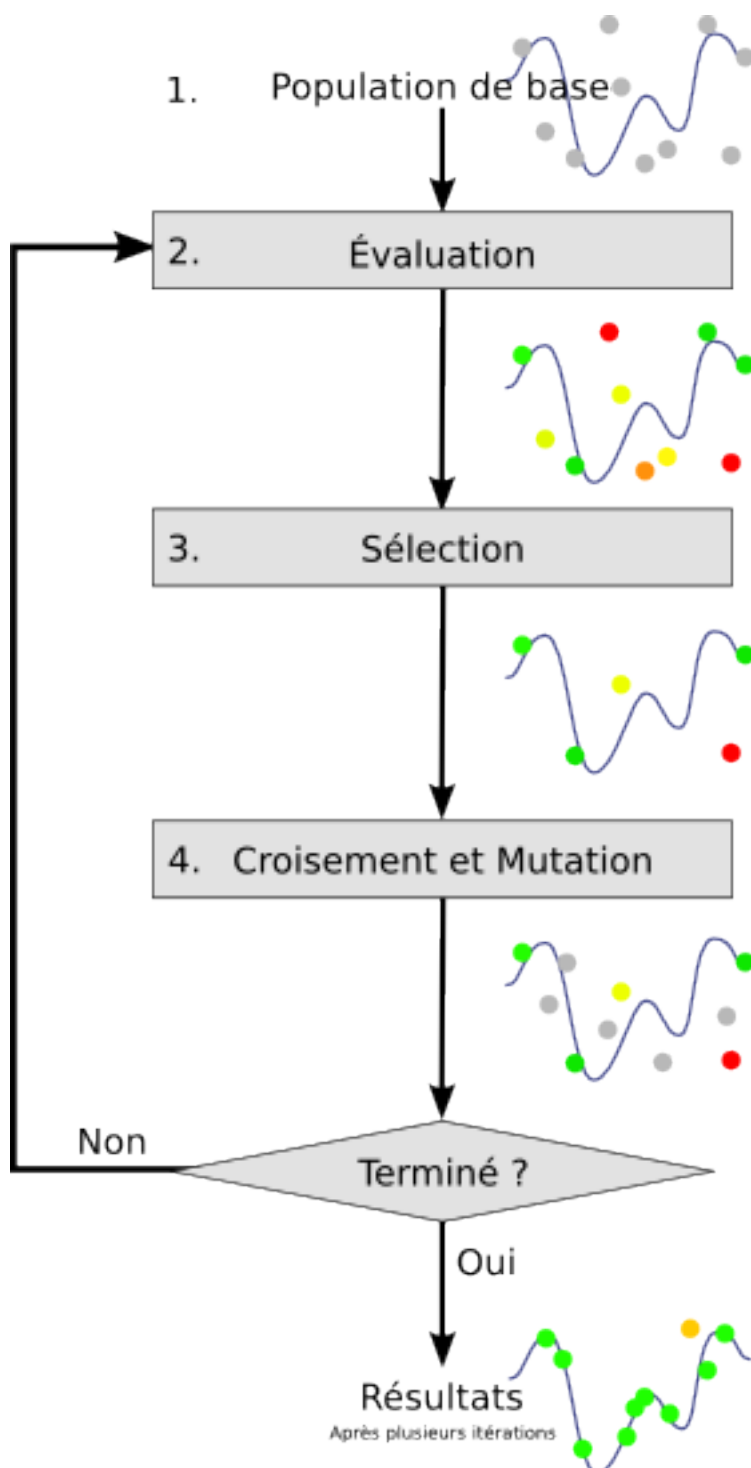


FIGURE 5.1 – Schéma général d'un algorithme génétique

5.3.1 Représentation et espace de recherche

SATyr utilise la manière la plus évidente mais aussi la plus pertinente (selon l'étude dans l'article [8]) de représenter un individu pour un problème SAT. Cette représentation correspond à une chaîne de n bits où chaque bit est associé à une seule variable.

Dans cette représentation, un individu X correspond donc à une affectation logique des variables qui sera soit solution, soit non solution du problème.

A partir de là, l'espace de recherche est donc l'ensemble suivant : $S = \{0, 1\}^n$ (toutes les chaînes possibles de n bits).

5.3.2 La fonction de fitness

Imaginons un problème SAT appelé P et X un individu. La fitness de X selon P est définie par le nombre de clauses de P qui sont satisfaites par X :

$$\begin{aligned} \text{Fitness} : S &\rightarrow \mathbb{R} \\ X &\mapsto \frac{(\text{NbClausesVraies})}{(\text{NbClausesTotales})} \end{aligned} \quad (5.1)$$

Avec une telle équation, il est donc trivial de dire qu'un individu qui aurait une fitness égale à 1.00 serait solution du problème.

Cette fonction permet de manière mathématique d'évaluer la qualité d'un individu au sein d'une population.

Il aurait été tout à fait possible d'utiliser une autre manière d'analyser la fitness d'un individu, il existe par exemple dans la littérature le f_{SAW} où le but est globalement d'associer un poids à chacune des clauses, et du coup : moins une clause est satisfaite plus le poids associé augmente (voir article numéro [8] pour plus d'informations).

Le but étant de favoriser la satisfaction des clauses qui le sont rarement. Malheureusement cette méthode n'a pour le moment jamais été testée avec SATyr.

5.3.3 L'opérateur de sélection

La fonction sélection a donné lieu à de nombreuses modifications (les méthodes utilisées ne fournissaient pas assez de diversifications pour prouver qu'un problème n'a pas de solution (voir section 5.4). Différentes techniques ont donc été réalisées pour trouver celle qui fournit le plus de diversités sans perdre en efficacité pour trouver une solution quand elle existe.

Sélection par Rank

Le principe est de garder triés les individus en fonction de leur fitness et de prendre toujours ceux qui sont les mieux adaptés au problème. Cette méthode est bonne pour trouver une solution mais catastrophique pour la diversité lorsque l'on cherche à prouver qu'un problème n'a pas de solution.

Sélection par Roulette Wheel

Le principe est d'affecter, en fonction de la fitness, une certaine valeur sur 360 (qui correspond aux 360 degrés de la roue). Plus la fitness est grande, plus la valeur sera grande (et réciproquement).

On tire ensuite 1 nombre au hasard sur 360 et l'on regarde sur quel partie de la roue on est, la partie correspond à un seul individu, qui sera donc l'individu sélectionné.

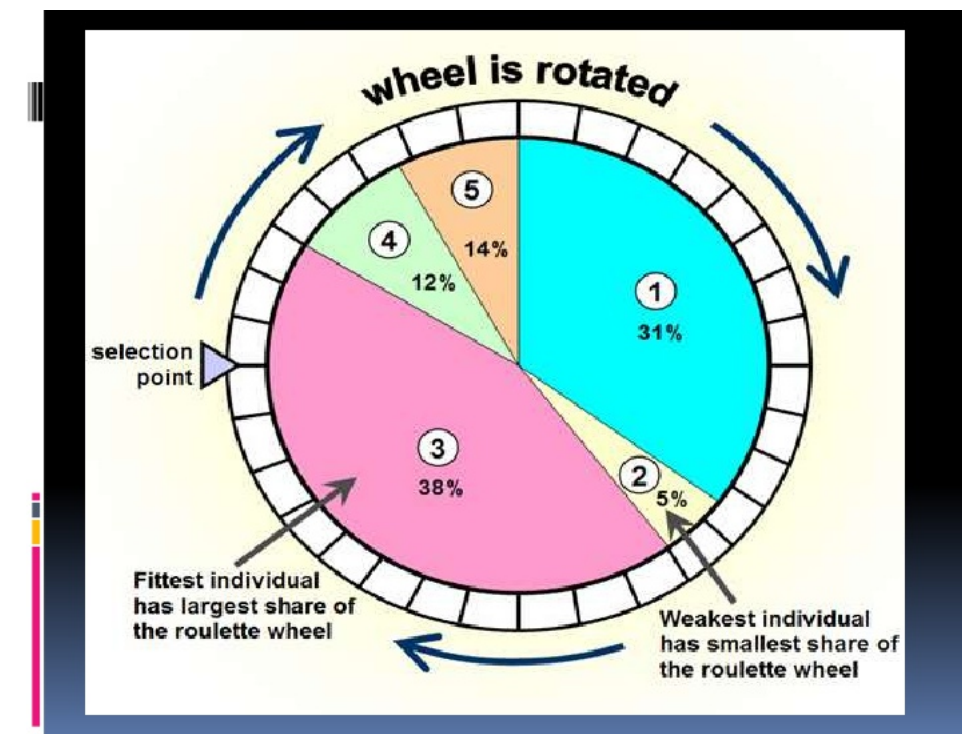


FIGURE 5.2 – Illustration du principe de Roulette Wheel

Sélection Uniforme

Le principe est finalement "on ne peut plus simple", on prend au hasard un individu parmi la population. Si on a N individus dans la population, on a donc $\frac{1}{N}$ pourcent de chance d'être choisi, qu'importe notre fitness.

Aussi surprenant que cela puisse paraître, c'est la méthode la plus simple, à savoir la Sélection Uniforme qui donne les meilleurs résultats sur les instances benchmarkées en terme de SAT et de UNSAT.

Données : Une population P d'une taille N

Résultat : l'individu sélectionné

début

/ On génère une position aléatoire dans la population */*

$\text{indice} \leftarrow (\text{random}()) \bmod N$

/ On retourne le Indice-ième individu de la population */*

retourner $P[\text{indice}]$

fin

Algorithme 1 : Sélection Uniforme d'un individu

5.3.4 L'opérateur de croisement uniforme

Données : 2 parents X et Y

Résultat : l'enfant Z

début

/ On croise uniformément l'ensemble des variables possibles */*

pour $(\forall x \in Y)$ **faire**

/ On a une chance sur 2 d'hériter de Y pour la variable x */*

si $(\text{random} \leq 50\%)$ **alors**

/ La variable x du fils sera la variable x du père Y . */*

$Z_x \leftarrow Y_x$

sinon

/ La variable x du fils sera la variable x du père X . */*

$Z_x \leftarrow X_x$

fin

fin

fin

Algorithme 2 : Croisement uniforme entre X et Y

Pour illustrer cet algorithme, rien ne vaut une image :

Voici le résultat enfant en fonction des parents avec des chaînes de bits.

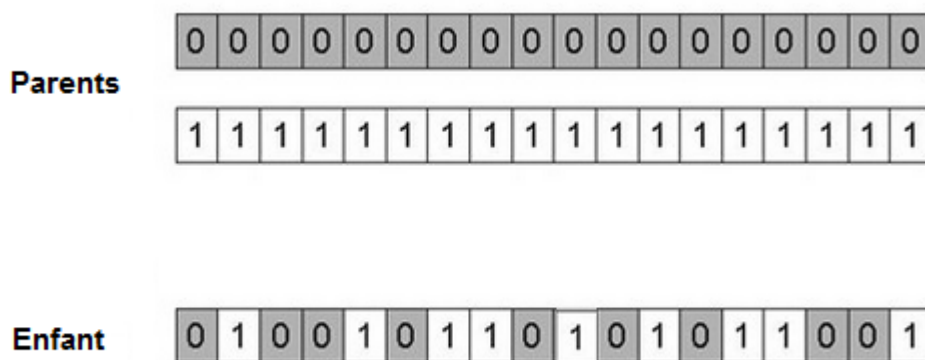


FIGURE 5.3 – Illustration du croisement uniforme

D'autres manières de croiser les individus ont été testées tel que :

- Le croisement à "un point de croisement" (voir Figure 5.4)
- Le croisement à "deux points de croisement" (voir Figure 5.5)

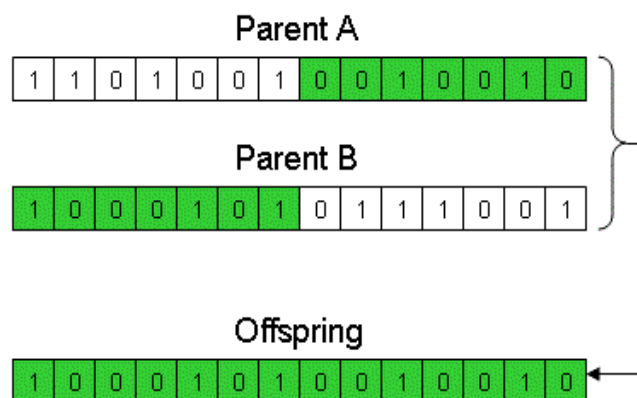


FIGURE 5.4 – "un point de croisement"

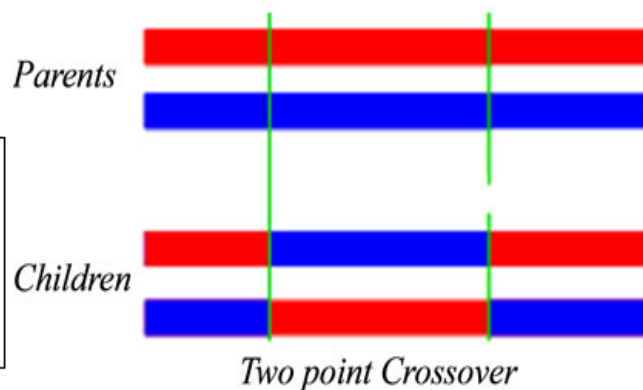


FIGURE 5.5 – "deux points de croisement"

Mais parmi ces 3 types de croisements testés, celui qui donne les résultats les plus efficaces était le croisement uniforme et donc le choix d'utiliser ce croisement au sein de SATyr a été réalisé.

5.3.5 L'opérateur de Mutation Adaptation

Traditionnellement, les algorithmes génétiques possèdent une fonction de mutation qui se produit aléatoirement afin d'assurer une diversité au sein de la population et de ne pas rester bloqué dans un extremum local.

Dans SATyr, le choix a été fait de pas utiliser de mutation pour les raisons suivantes :

- La méthode de mutation aléatoire d'un bit a été codée et n'a apporté aucun avantage visible.
- Une adaptation de l'enfant à l'aide d'une liste taboue est bien plus intéressante que de muter un gène.

En effet, SATyr possède un système d'adaptation de l'enfant à l'aide d'une liste taboue qui amène des individus de plus en plus robustes au sein de la population.

L'adaptation par liste taboue a été réalisée car il m'avait déjà été donné de réaliser un solveur SAT à l'aide seulement d'une liste taboue. Le principe est très simple à coder et donne de bons résultats. C'est donc pour ces raisons que l'adaptation a été réalisée ainsi.

Le principe est très simple : on possède une liste de taille T , un individu X ainsi qu'un nombre de mouvements max autorisé. On va donc à chaque fois chercher la variable qui améliore le plus la fitness de l'individu X (et qui n'est pas déjà dans la liste taboue), changer cette variable et l'ajouter dans la liste taboue. Si on a atteint la taille de la liste taboue, on supprime son élément le plus ancien. On réalise cette méthode tant que l'on n'a pas atteint le nombre de changements max autorisé.

Données : Un individu X , une liste L de taille T et un nombre max de flip M

Résultat : X a amélioré sa fitness

début

```

/* On ne va réaliser que M mouvements en tout pour améliorer X */
tant que  $(nbMouvement < M)$  faire
    /* On prend la variable x qui améliore le plus X et pas déjà dans L */
     $x \leftarrow CeQuiADeMieux(X, L)$ 

    /* On améliore ensuite X avec x */
     $X_x \leftarrow \neg(X_x)$ 

    /* x fini dans la taboue, on n'a plus le droit d'y toucher pour le moment */
     $ajouterDansListe(L, x)$ 

```

fin

fin

Algorithme 3 : Adaptation par liste taboue

5.3.6 Réinsertion dans certaines conditions

La réinsertion de l'enfant qui a été amélioré à l'aide d'une liste taboue, va se faire si et seulement si, sa fitness est meilleure que le premier tiers dans les classements des fitness des individus de la population.

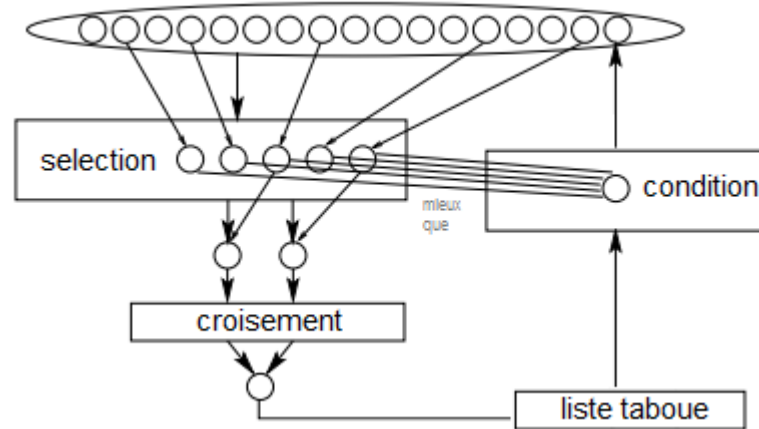


FIGURE 5.6 – Illustration de la ré-insertion

Si ce n'est pas le cas, on va simplement garder la population telle qu'elle est recommencer à nouveau une sélection d'individus.

La valeur de $\frac{1}{3}$ a été choisie parfaitement arbitrairement, c'est la valeur qui, après avoir testé plusieurs autres valeurs, donne les individus les plus robustes tout en conservant une diversité dans la population.

5.3.7 Algorithme de la partie SAT de SATyr

Maintenant que l'on connaît les fonctions nécessaires pour prouver SAT avec SATyr, voici l'algorithme de la partie SAT du solveur.

Données : ϕ : Le problème au format CNF *Population* : La population d'individus

Résultat : *VRAI* si une solution trouvée *FAUX* sinon

début

```

/* On sélectionne 2 parents au hasard... */
Parent1, Parent2 ← Selection(Population) ;

/* On réalise le croisement de manière uniforme. */
Fils ← Croisement(Parent1,Parent2) ;

/* On améliore notre enfant avec de la RL. */
Fils ← ListeTaboue(Fils) ;

/* On le ré-insère si sa fitness  $\geq \frac{1}{3}$  des meilleurs. */
Population ← Réinsertion(Fils,Population) ;

/* Fils a une fitness de 1.00 alors il est solution. */
si (Fils satisfait  $\phi$ ) alors
    retourner VRAI ;
sinon
    retourner FAUX ;
fin

```

fin

Algorithme 4 : essayerDeProuverSAT(ϕ)

Cette fonction ne répondra VRAI que si et seulement si, un des individus de la population est solution du problème.

5.4 UNSAT : prouver qu'un problème n'a pas de solution

Prouver UNSAT à l'aide d'algorithmes génétiques aura été la partie la plus compliquée de ce PFE. En effet, très peu de gens ont réussi (ou du moins ont voulu faire) un solveur capable de prouver UNSAT à l'aide d'une recherche locale.

Il a donc été très compliqué de s'inspirer d'articles sur le sujet, étant donné que les articles traitant de UNSAT sont la plus part du temps, des articles utilisant des solveurs CDCL².

Cependant, suite à des discussions téléphoniques avec [Mr. Jean Marie Lagniez](#), ce dernier m'a donné des pistes pour pouvoir utiliser au mieux mes individus.

Toute la partie UNSAT de SATyr repose donc sur le principe de résolution.

5.4.1 Principe de résolution

La règle de résolution ou principe de résolution de Robinson est une règle d'inférence logique que l'on peut voir comme une généralisation du modus ponens. Cette règle est principalement utilisée dans les systèmes de preuves automatiques, elle est à la base du langage de programmation logique Prolog.³

La règle de résolution s'applique sur deux clauses, c'est-à-dire des formules composées de disjonctions de littéraux, un littéral étant un atome (positif) ou un atome précédé d'une négation (négatif).

Étant données deux clauses

$$C_1 = L_1 \vee \dots \vee L_i \vee \dots \vee L_m$$

$$C_2 = M_1 \vee \dots \vee M_j \vee \dots \vee M_n$$

où les littéraux L_i et M_j sont l'un positif et l'autre négatif et qu'ils portent sur le même atome.

Le résolvant de C_1 et C_2 sur L_i et M_j (noté $C_1 \otimes_{L_i} C_2$) est la clause

$$C_R = \left(L_1 \vee \dots \vee L_{i-1} \vee L_{i+1} \vee \dots \vee L_m \right) \vee \left(M_1 \vee \dots \vee M_{j-1} \vee M_{j+1} \vee \dots \vee M_n \right)$$

Le principe de résolution étant complet, si l'ensemble de clauses considéré est inconsistant, on arrive toujours à générer la clause vide. **Autrement dit, si le problème SAT n'a pas de solution, une succession de résolutions le prouvera forcément.**

Par contre, le problème de la consistance (satisfaisabilité) n'étant pas décidable en logique des prédicats, il n'existe pas de méthode pour nous dire quelles résolutions effectuer et dans quel ordre pour arriver à la clause vide.

2. Conflict Driven Clause Learning

3. http://fr.wikipedia.org/wiki/Règle_de_résolution

5.4.2 SATyr + Principe de résolution

Nous l'avons vu plus haut, il n'existe pas pour le moment (et peut être jamais) d'algorithme capable de nous dire quelles résolutions effectuer et dans quel ordre pour arriver à la clause vide (notée \perp).

Ce que fait SATyr, c'est qu'il réalise un tour de boucle en essayant de trouver SAT. Les individus auront tous un certain nombre de clauses restées insatisfaites. On compte le nombre de fois pour chaque clause que cette dernière est restée insatisfaite pour l'ensemble de population. On va ensuite récupérer les $\frac{1}{3}$ premiers du classement.

Et avec ce sous-ensemble de clauses insatisfaites, on va initialiser un tableau de taille $[NbClauses * 2][NbClauses * 2]$ et l'on va stocker les tailles que donneraient les résolutions si on les effectuait vraiment.

On va donc se retrouver, au premier tour de boucle, avec un tableau qui sera rempli à $\frac{1}{3}$.

Et l'on va effectuer pour de vrai, la résolution qui nous donne la plus petite clause résultante. Cette clause va être ajoutée au problème (qui reste équisatisfiable), on dit que cette clause est apprise. On initialise aussi la ligne dans le tableau qui correspond à cette nouvelle clause. Et l'on va répéter l'opération jusqu'à ce que l'on arrive à prouver l'inconsistance, c'est à dire, une fois que l'on va trouver une taille qui nous donnera 0.

Bien évidemment, comme le risque de partir dans une mauvaise direction et de faire trop de résolutions inutiles est très élevé, SATyr possède un système de redémarrage qui va vider le tableau de son contenu et recommencer, comme si nous venions d'arriver au premier tour de boucle.

SATyr vérifie aussi, que la clause qu'il apprend n'est pas une tautologie, cela n'apporte rien à l'inconsistance d'apprendre une clause qui est tout le temps vraie. Pire encore, cela ralentit le solveur.

Données : ϕ : Le problème au format CNF *Population* : La population d'individus

Résultat : *VRAI* si une preuve du UNSAT est trouvée *FAUX* sinon

début

```

si (nbResolutions  $\geq$  nbResolutionsMAX) alors
    /* On réinitialise la resolutionTable[ ][ ] */
    redémarrage();
fin
min  $\leftarrow$  BIG;

/* On récupère les  $\frac{1}{3}$  clauses fausses qui reviennent souvent chez nos individus. */
pour (i  $\in$  clausesFaussesSurLaPopulation) faire
    pour (j  $\in$  clausesFaussesSurLaPopulation) faire
        si (min  $\geq$  resolutionTable[i][j]) alors
            /* On veut i et j où resolutionTable[i][j] est minimal */
            min  $\leftarrow$  resolutionTable[i][j];
            c1  $\leftarrow$  i;
            c2  $\leftarrow$  j;
        fin
    fin
fin

/* On a besoin d'une variable commune pour faire une résolution */
variable  $\leftarrow$  variableCommuneEntre(c1,c2);

/* On réalise la résolution qui fournit le plus petit */ lastResolution  $\leftarrow$ 
resolution(c1,c2,variable,VRAI);

/* si lastResolution =  $\perp \rightarrow$  UNSAT */
si (taille(lastResolution) = 0) alors
    retourner VRAI;
fin

si (lastResolution  $\neq$  Tautologie) alors
    apprendreClause();
    mettreAJourTableau(lastResolution);
    nbResolutions  $\leftarrow$  nbResolutions + 1;
fin
retourner FAUX;

```

fin

Algorithme 5 : essayerDeProuverUNSAT(ϕ)

5.4.3 Comment est réalisé mettreAJourTableau ?

Données : ϕ : Le problème au format CNF

i : l'indice de la résolution

début

```

    /* on met à jour ligne et colonne de notre nouvelle résolution */
    pour (j de 0 à nbClauses) faire
        var ← variableCommuneEntre(i,j) ;
        /* Si la résolution est possible */
        si (var ≠ 0) alors
            resolutionTable[i][j] ← taille(resolution(i,j,var,FAUX)) ;
            /* le tableau est symétrique */
            resolutionTable[i][j] ← resolutionTable[j][i] ;
        fin
    fin
fin

```

Algorithme 6 : mettreAJourTableau()

5.4.4 Comment sont réalisées les structures de données ?

Tout d'abord, prenons un exemple :

```
p cnf 2 3
1 2
-2
-1
```

FIGURE 5.7 – Exemple UNSAT

Voici comment il est représenté en mémoire au travers de différents tableaux.

On va stocker dans un tableau la taille de chacune des clauses (le tableau est 2x plus grand que prévu, pour pouvoir stocker aussi la taille des futures résolutions).

2
1
1
?
?
?

TABLE 5.1 – Tableau de size

Il va aussi falloir représenter les clauses concrètement, ainsi que les variables qu'elles contiennent.

Imaginons que maintenant, seules les clauses C1 et C2 soient retournées comme fausses par la population. Cela signifie que nous aurons les informations sur les résolutions possibles entre C1 et C2 en revanche, tout ce qui concerne C3 nous est totalement inconnu.

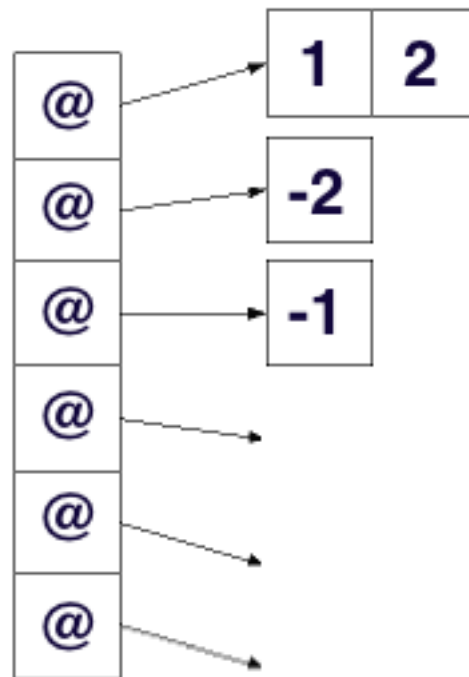


FIGURE 5.8 – Représentation du problème

	C1	C2	C3	R1	R2	R3
C1		1				
C2	1					
C3						
R1						
R2						
R3						

TABLE 5.2 – ResolutionTable

L'algorithme va parcourir ce "ResolutionTable" afin de chercher la valeur minimale, il trouve donc {C1,C2} et il va réaliser une résolution entre C1 et C2 sur une variable qu'elles ont en commun (ici la variable 2).

$$C_1 \otimes_2 C_2 = R_1 \iff \{1 \vee 2\} \otimes_2 \{-2\} = \{1\}$$

On obtient donc $R_1 = \{1\}$

Voici comment sont modifiées les structures de données à la suite de cette résolution réalisée :

La clause peut être ajoutée au problème et ce dernier reste équisatisfiable, on dit qu'elle a été apprise.

Il ne reste qu'à appliquer *mettreAJourTableau*(R1).

2
1
1
1
?
?

TABLE 5.3 – Tableau de size

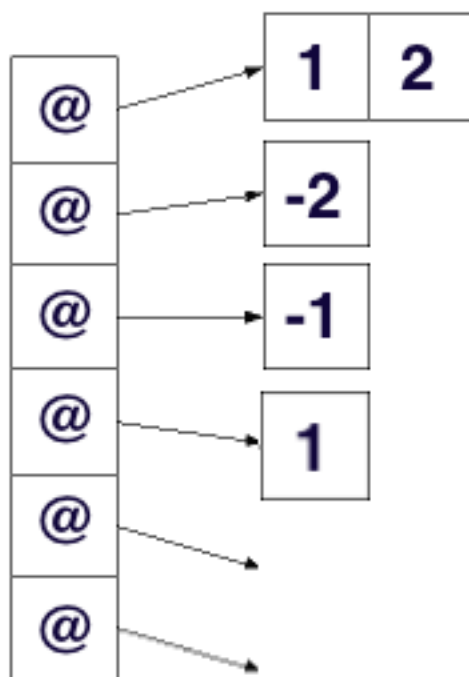


FIGURE 5.9 – Ce qu'est devenu le problème, la clause a été apprise

	C1	C2	C3	R1	R2	R3
C1		1		1		
C2	1					
C3				0		
R1	1		0			
R2						
R3						

TABLE 5.4 – Ce qu'est devenu ResolutionTable

Le tableau ResolutionTable contient des 0, cela signifie que la clause est une \perp et donc le problème n'a pas de solution.

5.5 SATyr : SAT + UNSAT

Données : ϕ : Le problème au format CNF

Résultat : SAT ou UNSAT

début

```

/* On ne connaît pas la solution encore... */
FOUND ← UNKNOWN ;

/* C'est ici qu'on initialise notre population... */
InitialiserLesDifferentesParametres() ;

tant que (FOUND = UNKNOWN) faire
    /* FOUND ← SAT une solution est trouvée. (voir Algorithme 4) */
    si (essayerDeProuverSAT( $\phi$ ) = VRAI) alors
        FOUND ← SAT ;
    sinon
        /* FOUND ← UNSAT pas de solution. (voir Algorithme 5) */
        si (essayerDeProuverUNSAT( $\phi$ ) = VRAI) alors
            FOUND ← UNSAT ;
        fin
    fin

    /* Affiche le nombre de flips et le numéro de la génération */
    afficherStatistiques() ;
fin

/* Affiche le temps, le nombre de flips total ... */
afficherResultatFinal() ;

```

fin

Algorithme 7 : Algorithme globale de SATyr

Chapitre 6

Réductions polynomiales

6.1 Le problème des N-Reines

Le but du problème des N-reines est de placer N dames d'un jeu d'échecs sur un échiquier de $N \times N$ cases sans que les reines ne puissent se menacer mutuellement, conformément aux règles du jeu d'échecs (la couleur des pièces étant ignorée). Par conséquent, deux dames ne devraient jamais partager la même rangée, colonne, ou diagonale.

Ce problème appartient au domaine des problèmes mathématiques et non à celui de la composition échiquéenne. Simple mais non trivial, ce problème sert souvent d'exemple pour illustrer des techniques de programmation.¹

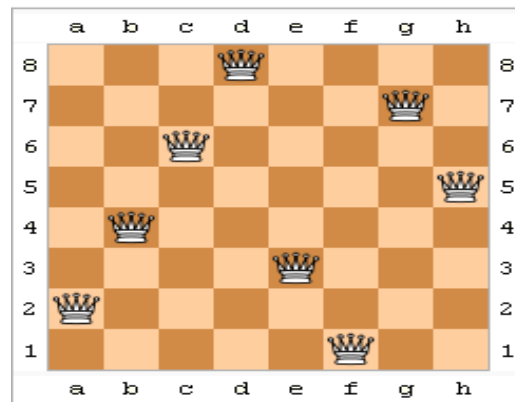
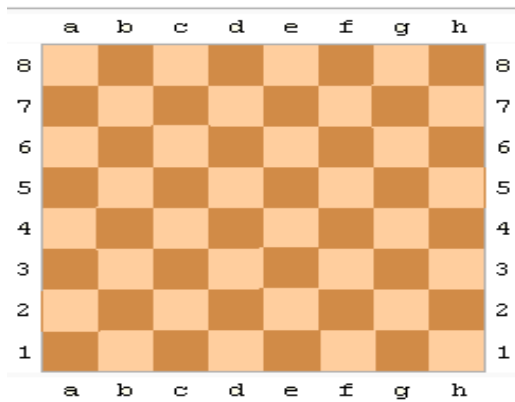


FIGURE 6.1 – Echiquier des 8 reines vide

FIGURE 6.2 – Echiquier des 8 reines solution

L'échiquier $N \times N$ est en réalité constitué de $N \times N$ variables booléennes.

x11	x12	x13	x14
x21	x22	x23	x24
x31	x32	x33	x34
x41	x42	x43	x44

À partir de là, il ne reste qu'à écrire les 5 contraintes classiques du problème des N-Reines, à savoir :

- Une reine présente sur chaque ligne.
- Pas d'attaque de reine de manière horizontale.
- Pas d'attaque de reine de manière verticale.
- Pas d'attaque de reine sur la diagonale Sud-Est.
- Pas d'attaque de reine sur la diagonale Sud-Ouest.

1. http://fr.wikipedia.org/wiki/Problème_des_huit_dames

6.2 Le problème d'emploi du temps

L'idée de base que j'ai utilisé pour réaliser la réduction polynomiale du problème d'emploi du temps vers SAT était de considérer tous les créneaux horaires utilisable comme des variables booléennes à vrai si elles sont réellement utilisées, faux sinon.

-----	Monday	Tuesday	Wednesday	Thursday	Friday
08:15 - 10:15					
10:15 - 12:30					
----- MIAM -----					
14:00 - 16:00					
16:15 - 18:15					

FIGURE 6.4 – Emploi du temps vide

On peut voir sur la figure 6.4 que si l'on a un certain nombre de matières N qui peuvent tous s'exécuter sur l'ensemble de ces créneaux, on va donc se retrouver avec $N * 20$ variables.

$$Input = \{S1, S2, \dots, Sn \mid Si = \{(x_{i1}, y_{i1}), (x_{i2}, y_{i2}), \dots, (x_{ik}, y_{ik}) \mid 0 \leq x_{ij} < y_{ij} \leq M\}\}$$

L'Input est un ensemble de cours, chaque cours est un ensemble d'intervalles ouvert de la forme (x,y) , (M étant une constante qui décrit plus ou moins "la fin de la semaine").

On va donc définir une variable booléenne pour chacun des intervalles : V_{ij} . On obtient donc :

$$F1 = (V_{1,1} \vee V_{1,2} \vee \dots \vee V_{1,k_1}) \wedge \dots \wedge (V_{n,1} \vee V_{n,2} \vee \dots \vee V_{n,k_n}) \quad (6.1)$$

De manière informelle, $F1$ sera vrai si et seulement si au moins un des intervalles par cours est vrai.

Ensuite, on définit $Smaller(x, y) = true$ si et seulement si $x \leq y$, afin de pouvoir empêcher que des intervalles se chevauchent.

Maintenant, il faut définir ces nouvelles clauses :

$$G_{c,d} = (\neg V_{a,c} \vee \neg V_{b,d} \vee Smaller(y_{ac}, x_{bd}) \vee Smaller(y_{bd}, x_{ac})) \quad (6.2)$$

À partir de là, on peut donc avoir notre version finale qui ne gère que les cours :

$$F = F1 \wedge \{G_{c,d} \mid \forall c, d\} \quad (6.3)$$

Avec cette équation, on s'assure :

- Pour chaque cours, 1 intervalle au moins est choisi.
- Pour 2 intervalles c et d , s'ils sont tous les 2 à vrai, ils ne se chevauchent pas.

6.2.1 1^{re} version : cours

Imaginons maintenant que nous ayons N h de Java a placé dans l'emploi du temps, que nous n'ayons qu'un seul enseignant et qu'une seule salle de cours possible. Tant que le N est inférieur à 20, il nous est possible de générer un emploi du temps, car le problème SAT aura une solution logique.

- We authorized 1 rooms.
- We have 1 teachers.
- We have week of 5 days.
- Java_ with 20 * 2h
- Teacher n° 1 can teach : [Java_]

Generation of the Class Scheduling by [Valentin Montmirail](#)

Creating file + Solving + Generation HTML time : 23 ms

-----	Monday	Tuesday	Wednesday	Thursday	Friday
08:15 - 10:15	Teacher : 1 Java_grp 1 Room : 1 -----	Teacher : 1 Java_grp 1 Room : 1 -----	Teacher : 1 Java_grp 1 Room : 1 -----	Teacher : 1 Java_grp 1 Room : 1 -----	Teacher : 1 Java_grp 1 Room : 1 -----
10:15 - 12:30	Teacher : 1 Java_grp 1 Room : 1 -----	Teacher : 1 Java_grp 1 Room : 1 -----	Teacher : 1 Java_grp 1 Room : 1 -----	Teacher : 1 Java_grp 1 Room : 1 -----	Teacher : 1 Java_grp 1 Room : 1 -----
----- MIAM -----					
14:00 - 16:00	Teacher : 1 Java_grp 1 Room : 1 -----	Teacher : 1 Java_grp 1 Room : 1 -----	Teacher : 1 Java_grp 1 Room : 1 -----	Teacher : 1 Java_grp 1 Room : 1 -----	Teacher : 1 Java_grp 1 Room : 1 -----
16:15 - 18:15	Teacher : 1 Java_grp 1 Room : 1 -----	Teacher : 1 Java_grp 1 Room : 1 -----	Teacher : 1 Java_grp 1 Room : 1 -----	Teacher : 1 Java_grp 1 Room : 1 -----	Teacher : 1 Java_grp 1 Room : 1 -----

FIGURE 6.5 – Emploi du temps complètement plein

Au dessus de 20, le problème SAT devient UNSAT, le solveur renvoie donc UNSATISFIABLE, ce qui est affiché dans l'interface graphique par ce message.

- We authorized 1 rooms.
- We have 1 teachers.
- We have week of 5 days.
- Java_ with 21 * 2h
- Teacher n° 1 can teach : [Java_]

Generation of the Class Scheduling by [Valentin Montmirail](#)

Creating file + Solving + Generation HTML time : 24 ms

This problem has been proved without any solution.

-----	Monday	Tuesday	Wednesday	Thursday	Friday
08:15 - 10:15					
10:15 - 12:30					
----- MIAM -----					
14:00 - 16:00					
16:15 - 18:15					

FIGURE 6.6 – Emploi du temps impossible à générer

Les seules contraintes à respecter sont que 2 créneaux ne doivent pas se chevaucher.

6.2.2 2^e version : cours + salles

Les contraintes ajoutées pour pouvoir prendre en compte les salles sont toujours selon le principe de chevauchement : [8 :00 - 10 :00] le lundi en salle i ne chevauche pas [8 :00 - 10 :00] le lundi en salle j (si $i \neq j$).

6.2.3 3^e version : cours + salles + professeurs

On reprend toujours le principe fondamentale de chevauchement pour réaliser cette réduction polynomiale, sauf que ce coup-ci : un professeur ne peut pas se retrouver à la fois en salle 1 et en salle 2 de [8 :00 - 10 :00] le lundi.

A chaque fois que l'on rajoute un nouveau type de contrainte, beaucoup de variables booléennes sont créées.

- En version 1, nous avons une variable pour un créneau.
- En version 2, nous avons une variable pour un créneau et une salle donnée.
- En version 3, nous avons une variable pour un créneau et une salle et un professeur donnée.

6.2.4 4^e version : cours + salles + professeurs + groupes d'étudiants

Cette version finalement n'a rien changé fondamentalement à la version 3. A ceci près qu'en version 3, on ne considérait que le cours X, enseigné par certains professeurs dans certaines salles.

En version 4, on considère des matières X, mais des cours Y différents pour une même matière X. Autrement dit, le cours "Java CM" est différent du cours "Java TD grp1" qui est différent du cours "Java TD grp2".

Les contraintes sont gérées ensuite que si CM est vrai, alors rien d'autre ne peut être vrai en même temps. On se retrouve donc en résultat final avec quelque chose de similaire à ceci :

- We authorized 2 rooms.
- We have 2 teachers.
- We have week of 5 days.
- Java_TD_ with 6 * 2h
- C#_CM with 8 * 2h
- C++_TD_ with 6 * 2h
- PHP_CM with 6 * 2h
-
- Teacher n° 1 can teach : [Java_TD_ C#_CM C++_TD_ PHP_CM]
- Teacher n° 2 can teach : [Java_TD_ C#_CM C++_TD_ PHP_CM]

Generation of the Class Scheduling by [Valentin Montmirail](#)

Creating file + Solving + Generation HTML time : 2012 ms

	Monday	Tuesday	Wednesday	Thursday	Friday
08:15 - 10:15	Teacher : 1 PHP_CM Room : 1 -----	Teacher : 1 PHP_CM Room : 1 -----	Teacher : 2 Java_TD_grp 2 Room : 2 ----- Teacher : 1 C++_TD_grp 1 Room : 1 -----	Teacher : 1 C#_CM Room : 1 -----	Teacher : 1 C#_CM Room : 1 -----
10:15 - 12:30	Teacher : 1 PHP_CM Room : 1 -----	Teacher : 1 PHP_CM Room : 1 -----	Teacher : 1 C++_TD_grp 1 Room : 1 -----	Teacher : 1 C#_CM Room : 1 -----	Teacher : 1 C#_CM Room : 1 -----
	----- MIAM -----				
14:00 - 16:00	Teacher : 1 PHP_CM Room : 1 -----	Teacher : 1 C++_TD_grp 1 Room : 1 -----	Teacher : 1 C++_TD_grp 1 Room : 1 -----	Teacher : 1 C#_CM Room : 1 -----	Teacher : 1 C#_CM Room : 1 -----

FIGURE 6.7 – Emploi du temps final avec l'ensemble des contraintes demandées

6.2.5 Remarques et problèmes rencontrés

Rien que pour cette image (figure 6.7), le problème SAT était de 2080 variables et 362266 clauses...

Malheureusement, vu l'ensemble des contraintes demandées, cela crée des problèmes énormes, tellement énormes qu'avec les réelles contraintes de Polytech le solveur ne s'en sort pas...

Je pense que ma modélisation est fonctionnelle mais absolument pas la plus optimale, finalement, cela fonctionne comme un Proof of Concept et je ne pense pas avoir les compétences pour réaliser une meilleure modélisation pour cette réduction polynomiale.

L'avantage étant que : si quelqu'un veut reprendre cette réduction polynomiale et l'améliorer, le coeur qui est le solveur, n'a pas besoin d'être touché.

6.3 Le problème de Job Shop

Les Job Shops sont des unités manufacturières traitant une variété de produits individuels dont la production requiert divers types de machines dans des séquences variées.

L'une des caractéristiques d'un atelier à cheminement multiple est que la demande pour un produit particulier est généralement d'un volume petit ou moyen.

Une autre caractéristique est la variabilité dans les opérations et un mix produit constamment changeant.

Ainsi, il est nécessaire que le système soit de nature flexible. Dans un sens général, la flexibilité est la capacité d'un système de répondre aux variations dans l'environnement.

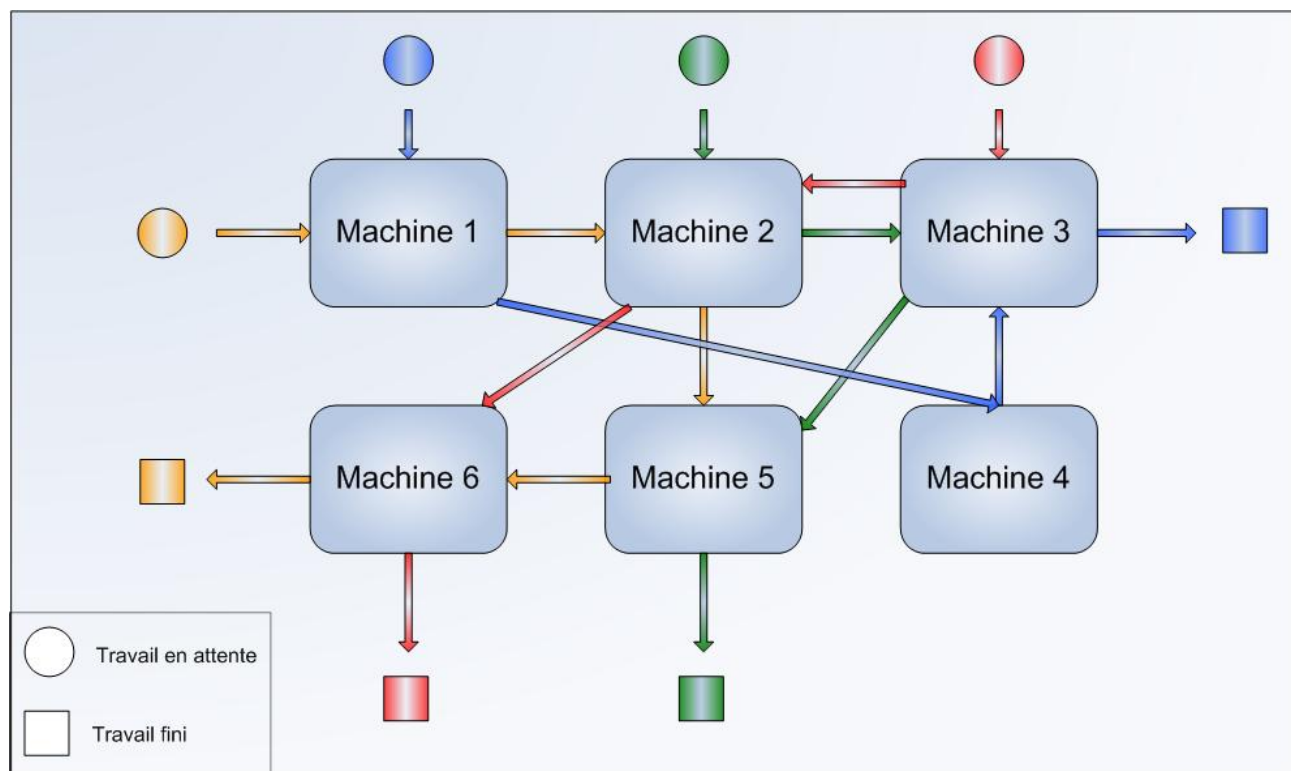


FIGURE 6.8 – Illustration d'un Job Shop

Parmi les caractéristiques d'un problème d'ordonnancement de type Job Shop :

le nombre de solutions possibles est de l'ordre de $(n!)m$, où n est le nombre de tâches à effectuer et m le nombre de machines.

Notons qu'une tâche veut dire la même chose qu'un travail. le problème est NP-difficile et est considéré parmi les problèmes les plus difficiles à traiter.²

Les problèmes Job Shops traités au sein de ce PFE sont les : $J_n | C_{max}$ en faisant varier n . Etant donné que le fait de réaliser une réduction polynomiale est une tâche assez complexe, j'ai tenu à traiter les problèmes de Job Shops les plus simples possible.

Je dis polynomiale, mais en réalité la solution trouvée est pseudo-polynomiale. Le fait que Job Shop est un problème d'optimisation NP-difficile et que SAT est un problème de

2. http://fr.wikipedia.org/wiki/Ordonnancement_d%27atelier

décision NP-Complet, à part si $P = NP$, il ne sera pas possible de trouver mieux en terme de complexité.

6.3.1 Fichier d'entrée et solution attendue

En fichier d'entrée, la réduction pseudo-polynomiale prend des fichiers du type :

```

3 2
1 1 1
1 1 1

```

Pour représenter, par exemple ici, 3 jobs, 2 machines et la durée de chaque job sur chaque machine. Le but final étant de trouver le C_{max} optimal, c'est à dire la date de fin au plus tôt pour que toutes les tâches soient exécutées. Ici on pourrait donc obtenir le Gantt suivant (désolé pour le Paint.exe)

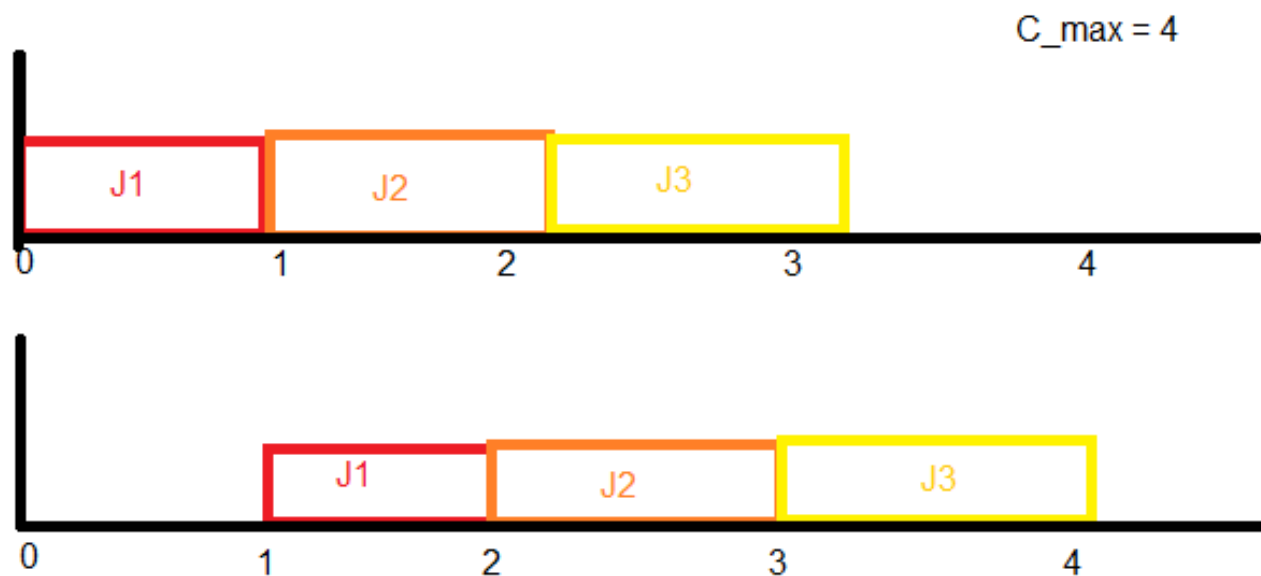


FIGURE 6.9 – Gantt solution de l'exemple

6.3.2 Mécanisme pour trouver le C_{max} optimal avec un problème de décision

Données : Un solveur $\text{solv}()$, un fichier JobShop JobShopProblem , une borne C_{param}

Résultat : Le C_{max} optimal du problème

début

```

/* On créer notre premier problème SAT pour avoir un bon début */
SATproblem ← reduction(JobShopProblem,  $C_{param}$ )

/* On résout cette version du problème pour démarrer notre boucle */
result ← solv(SATproblem)

/* On va decrementer le  $C_{param}$  jusqu'à avoir l'optimal */
tant que ( $result \neq UNSAT$ ) faire
     $C_{param} \leftarrow C_{param} - 1$ 
    /* On recréer un problème SAT */
    SATproblem ← reduction(JobShopProblem,  $C_{param}$ )
    /* On le résout, si on est SAT, on continue, en UNSAT on a trouvé l'optimal */
    result ← solv(SATproblem)

```

fin

afficher("Le C_{max} optimal est : C_{param} ")

fin

Algorithme 8 : Trouver C_{max} via une réduction vers SAT

La fonction $result \leftarrow \text{solv}(\text{SATproblem})$ revient finalement à ce qui a été réalisé dans SATyr.

Toute la difficulté de cette partie vient de la fonction :

$\text{SATproblem} \leftarrow \text{reduction}(\text{JobShopProblem}, C_{param})$

6.3.3 Les différentes structures utilisées

Pour cela, j'ai créé des structures afin de représenter le plus simplement possible ce problème JobShop et pour pouvoir le réduire à SAT plus tard.

```
typedef struct _resource {  
  
    unsigned int numShop;  
    unsigned int numJob;  
    unsigned int value;  
  
} Resource;  
  
typedef struct _problem {  
  
    unsigned int nbShops;  
    unsigned int nbJobs;  
    Resource** resources;  
  
} Problem;
```

FIGURE 6.10 – Les Structures pour la réduction de JobShop vers SAT

à partir de ces structures, il ne reste qu'à mettre au point le modèle mathématique, déterminer ce que sont les variables booléennes et les différentes contraintes qui permettent de passer du problème de Job Shop au problème SAT.

Pour les variables j'ai choisi de modéliser le problème de la manière suivante : $S_{1,2,3}$ par exemple, représente le fait que la tâche 1 se trouve sur la machine 2 au temps 3. Avec une telle représentation, je me retrouve donc avec $J * M * C$ **variables booléennes**, où J est le nombre de jobs, M le nombre de machine et C la valeur du C_{param} courant.

On se rend bien compte ici, que la réduction est donc pseudo-polynomiale, le nombre de variables ne dépend pas que de la taille du problème (J et M) mais aussi des valeurs du problème (le C_{param} que l'on est en train de tester)

6.3.4 Les différentes contraintes utilisées

- Un job i ne peut être que sur une seule machine à un temps k .

$$\begin{aligned} C1 &= \forall(i, j, k) \bigwedge_l^M (\neg S_{i,j,k} \implies S_{i,l,k}) \\ &= \forall(i, j, k) \bigwedge_l^M (\neg S_{i,j,k} \vee \neg S_{i,l,k}) \end{aligned} \quad (6.4)$$

- Une machine j ne peut gérer qu'une seule tâche à un temps k

$$\begin{aligned} C2 &= \forall(i, j, k) \bigwedge_l^J (\neg S_{i,j,k} \implies S_{l,j,k}) \\ &= \forall(i, j, k) \bigwedge_l^J (\neg S_{i,j,k} \vee \neg S_{l,j,k}) \end{aligned} \quad (6.5)$$

- Les tâches ne sont pas **préemptives**, une tâche qui démarre doit finir.

$$\begin{aligned} C3 &= \forall(i, j, k) \bigwedge_l^{Problem[i][j]} (S_{i,j,k} \Leftrightarrow S_{i,j,l}) \\ &= \forall(i, j, k) \bigwedge_l^{Problem[i][j]} (S_{i,j,k} \implies S_{i,j,l}) \wedge (S_{i,j,l} \implies S_{i,j,k}) \\ &= \forall(i, j, k) \bigwedge_l^{Problem[i][j]} (\neg S_{i,j,l} \vee S_{i,j,k}) \wedge (\neg S_{i,j,k} \vee S_{i,j,l}) \end{aligned} \quad (6.6)$$

- Si on demande N temps d'exécution, il doit y avoir N temps d'exécution.

$$C4 = \forall(i, j, k) \bigvee_l^{Problem[i][j]} S_{i,j,l} \quad (6.7)$$

- Notre problème de Job Shop est donc $SAT_{problem} = C1 \wedge C2 \wedge C3 \wedge C4$

Ainsi en résolvant ce problème SAT, nous allons pouvoir avoir exactement le bon nombre de variables à vrai, et à chaque variable, on pourra associer le tuple (job,machine,date) pour savoir quelle tâche est sur quelle machine et à quel temps.

Prenons un exemple très simple :

Nombre de machines : 1

Nombre de jobs : 10

1 1 1 1 1 1 1 1 1 1

```
[INFO] - Solver      : ../../satyr
[INFO] - Filename    : ./test.shop
[INFO] - NbVariables : 120
[INFO] - NbClauses   : 1162
[INFO] - C_max       : 12
[INFO] - SATISFIABLE
[INFO] - Time        : 14 ms

[INFO] - Solver      : ../../satyr
[INFO] - Filename    : ./test.shop
[INFO] - NbVariables : 110
[INFO] - NbClauses   : 1072
[INFO] - C_max       : 11
[INFO] - SATISFIABLE
[INFO] - Time        : 16 ms

[INFO] - Solver      : ../../satyr
[INFO] - Filename    : ./test.shop
[INFO] - NbVariables : 100
[INFO] - NbClauses   : 982
[INFO] - C_max       : 10
[INFO] - SATISFIABLE
[INFO] - Time        : 17 ms

[INFO] - Solver      : ../../satyr
[INFO] - Filename    : ./test.shop
[INFO] - NbVariables : 90
[INFO] - NbClauses   : 892
[INFO] - C_max       : 9
[INFO] - UNSATISFIABLE
[INFO] - Time        : 18 ms

s C_MAX* : 10

8
16
23
37
50
52
69
71
84
95
```

FIGURE 6.11 – Execution de la réduction avec SATyr et un C_{init} à 20

On peut voir qu'en 18ms, le programme a créé et résolu 10 problèmes SAT et à conclu que la dernière valeur à laquelle ce problème a une solution, c'est quand le $C_{max} = 10$, quand toutes les tâches sont les unes à la suite des autres...

6.4 Le problème de Flow Shop

L'un des objectifs principaux dans le cas du Flow Shop est de trouver une séquence des tâches en main qui respecte un ensemble de contraintes et qui minimise le temps total de production. Parmi les caractéristiques d'un problème de cette catégorie :

- il existe au minimum $n!$ différentes solutions où n est le nombre de travaux à réaliser.
- le problème est NP-difficile à l'exception des versions avec deux machines et certains cas particuliers avec trois machines.
- une grande productivité mais une faible flexibilité.³

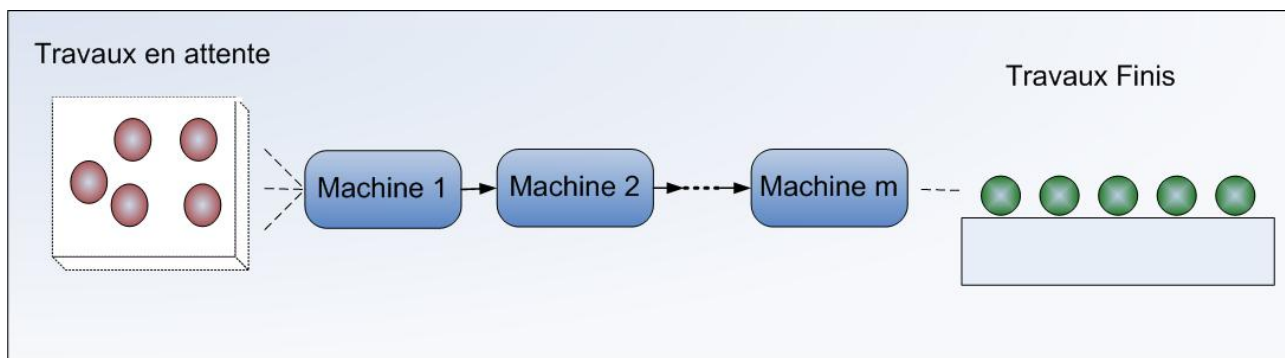


FIGURE 6.12 – Illustration d'un Flow Shop

Flow Shop est tout simplement un cas particulier du problème de Job Shop, où il faut en plus de minimiser le C_{max} , il faut qu'il y ait un ordre pour l'exécution sur les différentes machines.

Il faudrait donc pouvoir ajouter une nouvelle contrainte :

- L'ordre d'exécution est le même sur toutes les machines.

$$C5 = \forall(i, j, k) \text{ ???} \quad (6.8)$$

Et le mécanisme serait finalement exactement le même que pour Job Shop.

3. http://fr.wikipedia.org/wiki/Ordonnancement_d%27atelier

6.5 Problèmes rencontrés dans Job Shop et Flow Shop

Ces réductions auraient dû être analysées et comprises par Thomas Lorand durant son projet de GPF. Seulement, le fait qu'il ait abandonné ses études a rallongé énormément le temps que prend cette tâche dans mon PFE. Cette tâche était d'ailleurs tellement longue que je n'ai pas pu la finir.

Ce qui fonctionne pour le moment, sont les Job Shops à 1 machine et N jobs. Malheureusement, trouver un C_{max} dans ce genre de cas n'a aucun intérêt, il suffit d'additionner la somme des durées d'exécution pour l'obtenir.

Je pense selon moi, que l'analyse mathématique est correcte, mais le code pour l'implémenter ne l'est pas, il y a une subtilité quelque part, que je n'ai pas cernée, et qui ne me permet pas de gérer M machines.

Ce qui reste encore à trouver par contre, c'est la contrainte dans l'équation 6.8 qui reste encore inconnue pour moi à l'heure où j'écris ce rapport.

Chapitre 7

Benchmarks

Les benchmarks sont une tâche très coûteuse en temps pour finalement très peu d'effort, vu qu'il suffit simplement de laisser l'ensemble des solveurs s'exécuter sur l'ensemble des fichiers d'instances que l'on considère pour obtenir un résultat.

Ces benchmarks ont été optimisés à l'aide de [Mr. Christophe Forycki](#) que je ne remercierai jamais assez. Le but de son PFE était très grossièrement de pouvoir paralléliser l'exécution de plusieurs programmes sur l'ensemble des ordinateurs d'une/plusieurs salles machines de l'école à l'aide de [HTCondor](#).



FIGURE 7.1 – Logo de HTCondor

Étant donné que mes benchmarks sont peu coûteux en mémoire mais énormément en temps CPU, son projet m'a fait gagner plusieurs heures (si ce n'est jours, vu la rapidité de HTCondor).

Les calculs ont été effectués sur un ensemble de machine identiques dont voici la configuration :

- **Machines physiques**
 - Windows 7 Pro 64bits
 - RAM 8Go
 - [CPU Intel Xeon E3-1230 v3 @ 3.30GHz](#)
- **Machines virtuelles**
 - Ubuntu 14.04 64bits
 - 6 cores
 - 5.8Go RAM

7.1 Rappel des benchmarks du cahier des spécifications

TABLE 7.1 – Benchmarks utilisés : <http://www.cs.ubc.ca/hoos/SATLIB/benchm.html>

Nom du benchmarks	# instances	clauses	# Variables	# Clauses	SAT ou UNSAT
<i>uf20-91</i>	1000	3	20	91	<i>SAT</i>
<i>uf50-218 / uuf50-218</i>	1000	3	50	218	<i>SAT et UNSAT</i>
<i>uf75-325 / uuf75-325</i>	1000	3	75	325	<i>SAT et UNSAT</i>
<i>uf100-430 / uuf100-430</i>	1000	3	100	430	<i>SAT et UNSAT</i>
<i>uf125-538 / uuf125-538</i>	1000	3	125	538	<i>SAT et UNSAT</i>
<i>uf150-645 / uuf150-645</i>	1000	3	150	645	<i>SAT et UNSAT</i>
<i>uf175-753 / uuf175-753</i>	1000	3	175	753	<i>SAT et UNSAT</i>
<i>uf200-860 / uuf200-860</i>	1000	3	200	860	<i>SAT et UNSAT</i>
<i>uf225-960 / uuf225-960</i>	1000	3	225	960	<i>SAT et UNSAT</i>
<i>uf250-1065 / uuf250-1065</i>	1000	3	250	1065	<i>SAT et UNSAT</i>
<i>hole6.cnf</i>	1	6	42	133	<i>UNSAT</i>
<i>hole7.cnf</i>	1	7	56	204	<i>UNSAT</i>
<i>hole8.cnf</i>	1	8	72	297	<i>UNSAT</i>
<i>CBS_k3_n100_m449_b90</i>	1000	3	100	449	<i>SAT</i>

Les courbes sont donc tracées pour les SAT sur 10000 points de la manière suivante :

- { 0000 - 1000 } *uf020-0091*
- { 1000 - 2000 } *uf050-0218*
- { 2000 - 3000 } *uf075-0325*
- { 3000 - 4000 } *uf100-0430*
- { 4000 - 5000 } *uf125-0538*
- { 5000 - 6000 } *uf150-0645*
- { 6000 - 7000 } *uf175-0753*
- { 7000 - 8000 } *uf200-0860*
- { 8000 - 9000 } *uf225-0960*
- { 9000 - 10000 } *uf250-1065*

et pour les UNSAT sur 9000 points de la manière suivante :

- { 0000 - 1000 } *uuf050-0218*
- { 1000 - 2000 } *uuf075-0325*
- { 2000 - 3000 } *uuf100-0430*
- { 3000 - 4000 } *uuf125-0538*
- { 4000 - 5000 } *uuf150-0645*
- { 5000 - 6000 } *uuf175-0753*
- { 6000 - 7000 } *uuf200-0860*
- { 7000 - 8000 } *uuf225-0960*
- { 8000 - 9000 } *uuf250-1065*

Les *CBS_k3_n100_m449_b90* et les *holesX* ont été testées, résolues et benchmarkées, mais j'ai choisi de ne pas les faire apparaître en courbe afin de pouvoir avoir des courbes parfaitement homogènes dans leur progression.

7.2 Benchmarks sur les SATISFIABLEs

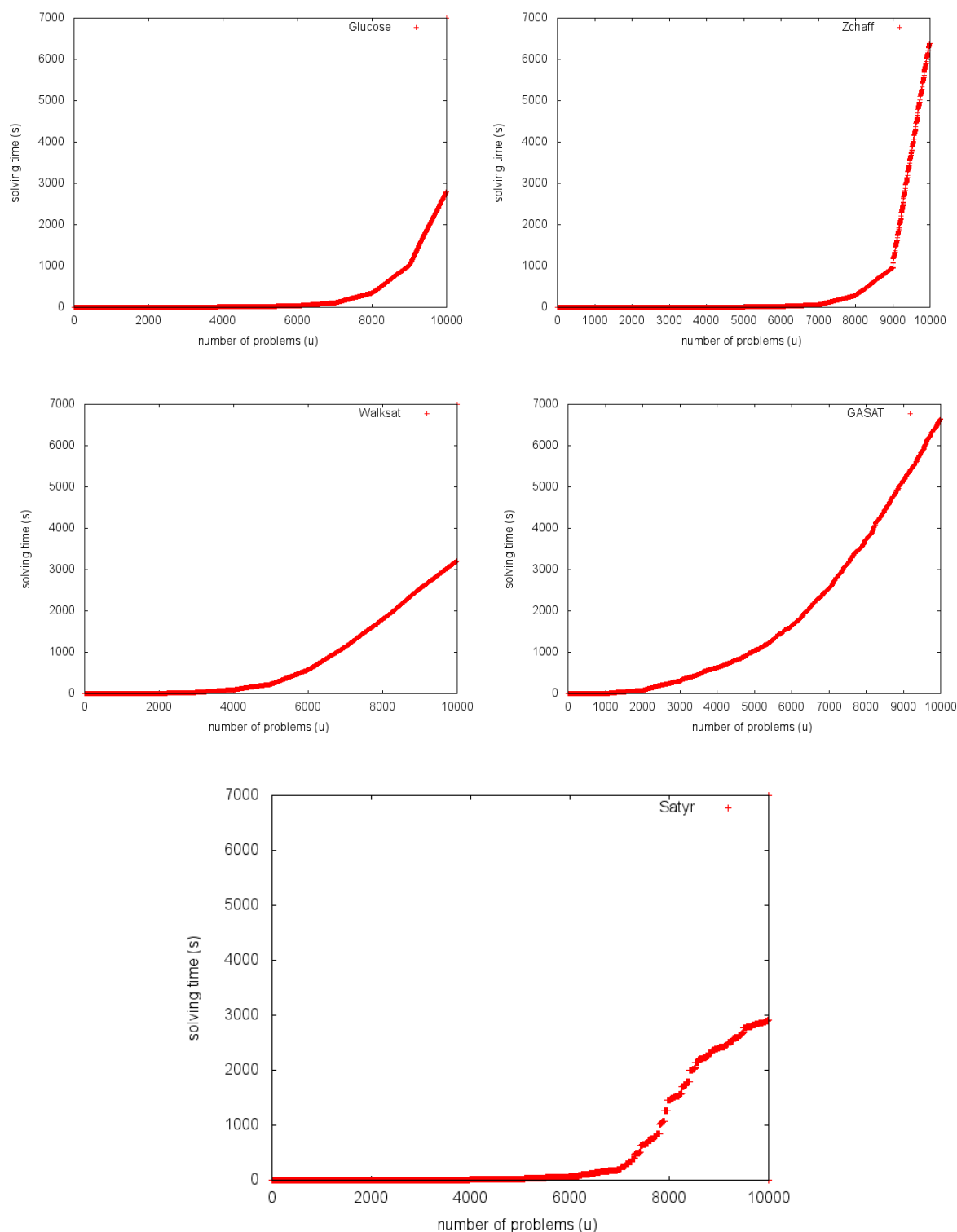


FIGURE 7.2 – Résultats de l'ensemble des solveurs sur les instances SATISFIABLEs

Étant donné que les échelles ne sont pas bonnes pour départager premier et deuxième à l'oeil. Voici Satyr et Glucose dans une échelle différente :

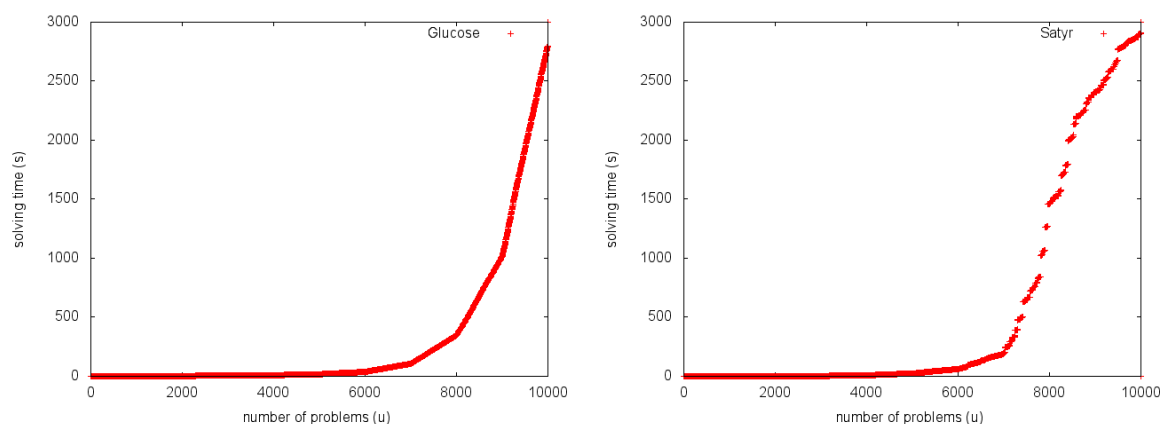


FIGURE 7.3 – Glucose 1^{er}/5 et SATyr 2^e/5

7.2.1 Podium

Voici le tableau podium pour récapituler les temps d'exécutions finaux :

Solveur	Glucose	SATyr	WalkSAT	Zchaff	GaSAT
Temps Final (s)	2797.401664	2910.950000	3211.000000	6412.874758	6644.6900000

On peut en tout cas conclure sur la partie SAT, que SATyr n'a rien à envier aux solveurs de référence dans le monde des problèmes SAT.

Mais bien évidemment, des résultats pareil s'obtiennent sur des instances générées aléatoirement, sur des instances de type industriel, la recherche locale de SATyr ne serait clairement pas aussi efficace.

Petit mot sur les résultats de Zchaff, je n'ai pas réussi à expliquer pourquoi il arrive à être aussi performant jusqu'à une certaine taille d'instance et d'un coup être extrêmement lent pour résoudre des instances plus difficiles... C'est cela qui a plombé son score et l'a fait arriver à la 4^e place.

D'après une analyse de Mr. Gilles Audemard (créateur de Glucose) les résultats de Zchaff s'expliquent par le fait que le solveur apprend trop de clauses et doit donc avoir le taux de propagation unitaire en continuelle chute au fur et à mesure que les problèmes deviennent gros (Glucose supprime beaucoup de clauses apprises et n'a donc pas ce handicap).

Mr. Jean Marie Lagniez explique quant à lui, que Glucose est meilleur que Zchaff car il a une stratégie de redémarrage beaucoup plus agressive.

Le problème des instances aléatoires est que les modèles forment des grappes. Un mauvais choix au départ entraîne donc une lenteur sur la résolution de l'instance.

7.3 Benchmarks restants non tracés

Voici le temps d'exécution de chacun des solveurs sur les benchmarks non tracés en graphe, les résultats sont moyennés sur 10 exécutions :

Nom du solveur / instances	Glucose	Satyr	Zchaff	WalkSAT	GaSAT
<i>CBS_k3_n100_m449_b90</i>	0.6920	1.2099	0.7440	2.1274	46.6487
<i>hole6</i>	0.007917	0.960641	0.006136	—	—
<i>hole7</i>	0.11025	15.383548	0.018763	—	—
<i>hole8</i>	1.42976	117.869913	0.103547	—	—

Un petit mot sur pourquoi avoir choisi ces fichiers en plus des U(U)FXX. les **HolesX** sont des instances très connues du monde SAT, elles correspondent au problème du pigeon hole.

Pour chaque pigeon i nous avons une variable x_{ij} qui signifie que le pigeon i a été placé dans le trou j .

Ensuite nous avons $n + 1$ clauses qui dit qu'un pigeon doit être placé dans un trou.

Nous avons aussi un ensemble de clauses qui certifie qu'un trou ne contient qu'un seul et unique pigeon.

Cet encodage nous amène à $n * (n + 1)$ variables booléennes et $(n + 1) + n * (n * (n + 1) / 2)$ clauses.

Ces problèmes sont forcément sans solution car le problème contient plus de pigeons que de trous, un humain détecte instantanément que le problème n'a pas de solution, c'est un donc un bon moyen pour tester la rapidité d'un solveur.

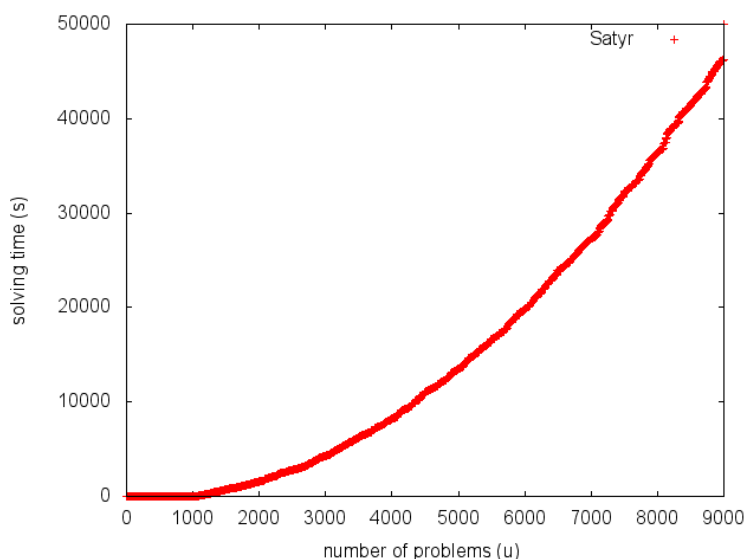
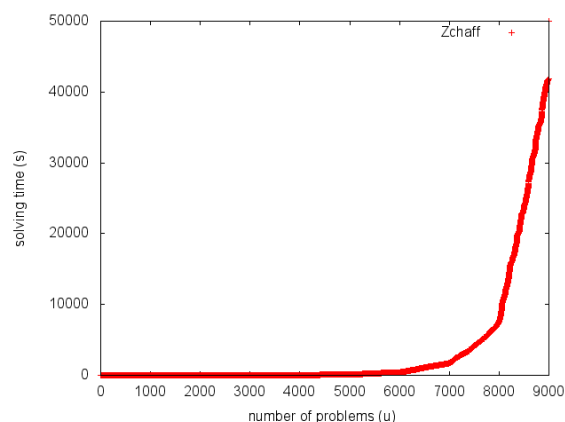
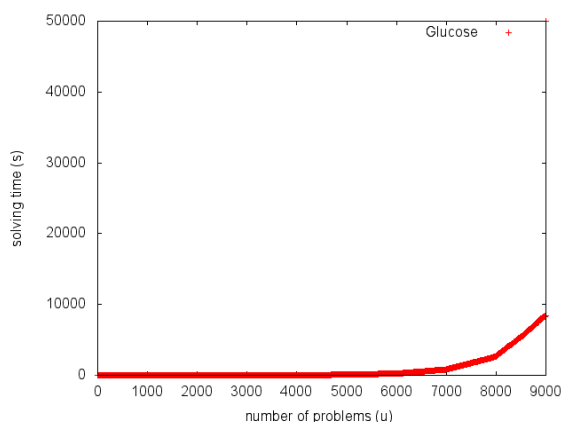
Pour ce qui est **des problèmes CBS**, ils correspondent à des problèmes SAT générés aléatoirement mais avec une taille de backbone contrôlé.

Le backbone est défini de cette manière : "The backbone of a satisfiable SAT instance is the set of entailed literals and so the backbone size is the number of entailed literals."

Ils sont générés de cette manière :

- Générer m clauses aléatoires, pour chaque clause :
 - Sélectionner aléatoirement k variables distinctes sur l'ensemble des variables disponibles.
 - Prendre la négation de 50% d'entre elles. Cela construit les clauses elles mêmes.
- Si l'instance n'a pas de solution, redémarrer le processus.
- Déterminer la taille du backbone. Si ce n'est pas égal à celui que l'on cherchait, on redémarre le processus.

7.4 Benchmarks sur les UNSATISFIABLEs



7.4.1 Podium

Voici le tableau podium pour récapituler les temps d'exécutions finaux :

Solveur	Glucose	Zchaff	SATyr	WalkSAT	GaSAT
Temps Final (s)	8396.256832	41891.387636	46278.185367	—	—

On peut conclure sur la partie UNSAT, que SATyr est fonctionnel, contrairement à WalkSAT et GaSAT qui sont incapables de prouver l'inconsistance d'un problème.

Malheureusement, c'est un des points à améliorer, la méthode utilisée n'est pas assez compétitive avec les solveurs de référence. SATyr est donc bien plus long que ses concurrents, qu'importe la taille de l'instance...

Chapitre 8

Méthodologies utilisées pour le développement du projet

8.1 La documentation

Durant l'ensemble de ce projet, une méthodologie de génie logiciel a été utilisée. Une documentation de l'ensemble du code généré (que ce soit le solveur ou les différentes réductions polynomiales) a été documentée et générée selon la norme Doxygen.

SATyr

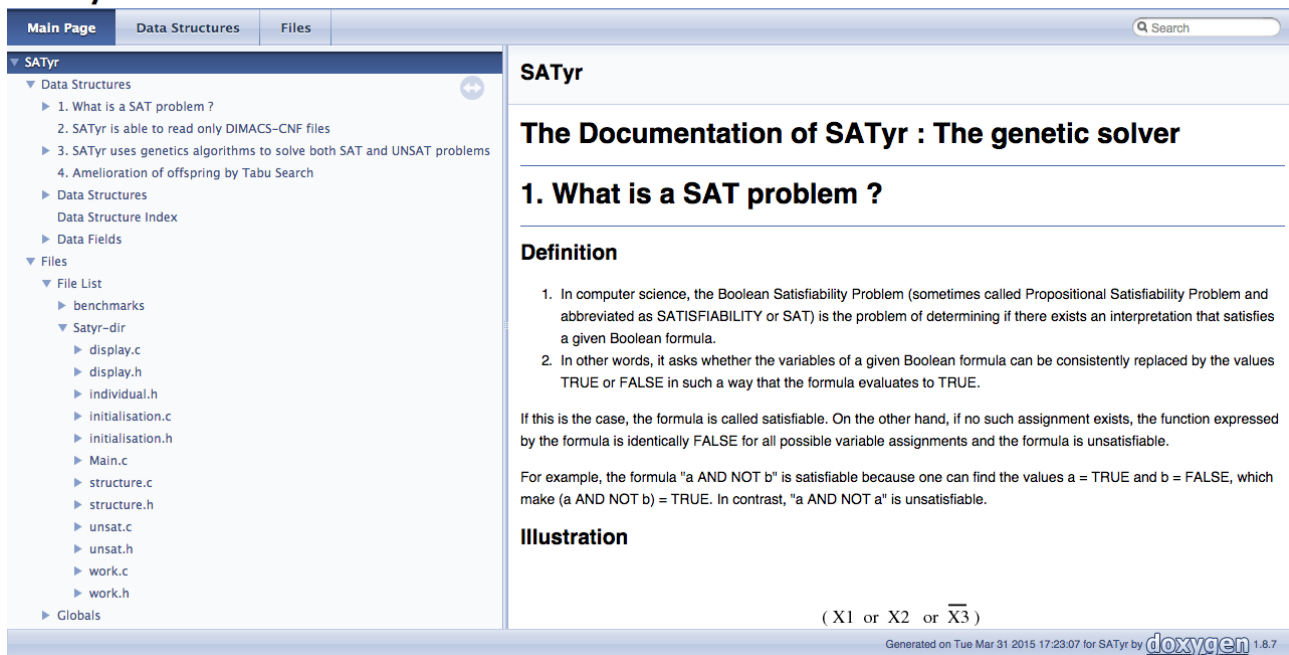


FIGURE 8.1 – Page d'accueil de la documentation Doxygen du solveur

Par habitude, et surtout parce qu'il est beaucoup plus conventionnel de le faire ainsi, l'ensemble du code et de la documentation est rédigé en anglais.

8.2 Le versionning

L'ensemble du processus de développement a été versionné à l'aide de l'outil git. Ainsi lors des quelques soucis de codes qu'il y a pu avoir durant le déroulement de ce projet, la possibilité de revenir en arrière a toujours été possible, ce qui rend le développement beaucoup plus serein.

The screenshot shows the GitHub repository page for 'Mystelven / SATyr'. The repository has 448 commits, 2 branches, 2 releases, and 1 contributor. The current branch is 'master'. A merge pull request #64 from 'Mystelven/valentin' is shown, with the latest commit '1e11464f6a'.

File	Description	Time
Reductions	l'emploi du temps fonctionne bien, il ne lui manque que la notion de ...	5 days ago
Satyr-dir	l'emploi du temps fonctionne bien, il ne lui manque que la notion de ...	5 days ago
benchmarks	la réduction polynomiale N creneaux S salles fonctionne	14 days ago
images	correction git	a month ago
.DS_Store	mise à jour du dépôt	a month ago
.gitignore	correction git	a month ago
Doxyfile	correction git	a month ago
Makefile	l'emploi du temps fonctionne bien, il ne lui manque que la notion de ...	5 days ago
README.md	Modifications des règles de cppcheck pour ajouter plus de vérifications	7 months ago
satyrSAT	optimisation de la fonction de résolution	a month ago
sonar-project.properties	correction du doxygen sur Jenkins	2 months ago
sortie.txt	la réduction polynomiale N creneaux S salles fonctionne	14 days ago
test.sh	la réduction polynomiale N creneaux S salles fonctionne	14 days ago

The README.md file contains the following text:

Etude et résolution du problème SAT

- Etude bibliographique et étude des solveurs existants
- Développement de métaheuristiques destinées à trouver rapidement une solution, quand elle existe

On the right side of the page, there are links for 'Code', 'Issues', 'Pull requests', 'Wiki', 'Pulse', 'Graphs', and 'Settings'. Below these links, there is a section for 'HTTPS clone URL' with the URL 'https://github.com/1e11464f6a' and a button to 'Clone in Desktop'. There is also a 'Download ZIP' button.

FIGURE 8.2 – Page d'accueil du GitHub du projet

Tout est hébergé de manière publique sur GitHub étant donné que j'avais moi-même déposé mon sujet, je n'ai pas cherché à rendre ce dernier confidentiel. De plus, je compte continuer ce projet après la fin de ce PFE. Le code source sera accessible pendant encore longtemps à l'adresse : <https://github.com/Mystelven/SATyr>.

8.3 Méthodes agiles : les sprints de développement

L'ensemble de ce projet aura été développé selon des méthodes agiles. A chaque sprint, un e-mail était envoyé à Mr. Christophe Lenté afin de lui spécifier :

- Ce qui a été réalisé durant la semaine.
- Les problèmes qui ont été rencontrés.
- Ce qui est prévu pour la semaine suivante.

Mail			1-38 of 38
<input type="checkbox"/> ☆ □	Christophe Lenté	PFE Re : Re: [SAT - Semaine 24] Compte Rendu - Bonjour, Je suis absent jusqu'au mardi 07 :	16:37
<input type="checkbox"/> ☆ □	me	Inbox PFE Re: [SAT - Semaine 24] Compte Rendu - jours en semaine pour faire les achats n	16:37
<input type="checkbox"/> ☆ □	Christophe Lenté	Inbox PFE Re: [SAT - Semaine 24] Compte Rendu - : [SAT - Semaine 24] Compte Rendu B	16:31
<input type="checkbox"/> ☆ □	me	Inbox PFE [SAT - Semaine 24] Compte Rendu - durant cette semaine : * * La gestion des gr	27 Mar
<input type="checkbox"/> ☆ □	me	PFE [SAT - Semaine 23] Compte Rendu - durant cette semaine : * Comme prévu, je suis toujo	22 Mar
<input type="checkbox"/> ☆ □	me	PFE [SAT - Semaine 22] Compte Rendu - durant cette semaine : * * Les réductions polynomia	14 Mar
<input type="checkbox"/> ☆ □	me	PFE [SAT - Semaine 20 - 21] Compte Rendu - ces deux semaine : * La version quasi finale de	26 Feb
<input type="checkbox"/> ☆ □	me	PFE [SAT - Semaine 18 - 19] Compte Rendu - ordonnancement vers SAT. Cordialement, *Val	11 Feb
<input type="checkbox"/> ☆ □	me	PFE [SAT - Semaine 17] Compte Rendu - fin de semaine un peu complexe. Voici ce qui a été	2 Feb
<input type="checkbox"/> ☆ □	me	PFE [SAT - Semaine 16] Compte Rendu - durant cette semaine :	22 Jan
<input type="checkbox"/> ☆ □	Christophe Lenté	PFE RE: [SAT - Semaine 14 - 15] Compte Rendu - : [SAT - Semaine 14 - 15] Compte Rendu	16 Jan
<input type="checkbox"/> ☆ □	me	PFE [SAT - Semaine 14 - 15] Compte Rendu - la prochaine semaine: * * Trouver les articles im	16 Jan
<input type="checkbox"/> ☆ □	me	PFE [SAT - Semaine 13] Compte Rendu - durant la semaine (ainsi que pendant les vacances):	10 Jan
<input type="checkbox"/> ☆ □	me	PFE [SAT - Semaine 12] Compte Rendu - durant la semaine: * Le livre "The Satisfiability Prob	19/12/2014

FIGURE 8.3 – Page de GMail listant les mails des sprints

Ainsi, Mr. Lenté savait exactement ce qui était entrain d'être fait, et en se référant au Gantt initial du cahier des spécifications, pouvait savoir si une tâche était en avance ou en retard.

8.4 Les tests unitaires

8.4.1 Script pour les résultats

Mis à part la fonction de lecture du fichier CNF d'entrée, peu de fonctions sont réellement complexes au sein de SATyr, elles ont donc été testées à la main sans forcément utiliser d'outils de tests.

En revanche, tester que ce que renvoie SATyr est conforme avec ce qui est attendu à était testé a été un script shell.

```
satyrSAT. : ../instances/uf100/uf100-0973.cnf
satyrSAT. : ../instances/uf100/uf100-0974.cnf
satyrSAT. : ../instances/uf100/uf100-0975.cnf
satyrSAT. : ../instances/uf100/uf100-0976.cnf
satyrSAT. : ../instances/uf100/uf100-0977.cnf
satyrSAT. : ../instances/uf100/uf100-0978.cnf
satyrSAT. : ../instances/uf100/uf100-0979.cnf
satyrSAT. : ../instances/uf100/uf100-0980.cnf
satyrSAT. : ../instances/uf100/uf100-0981.cnf
satyrSAT. : ../instances/uf100/uf100-0982.cnf
satyrSAT. : ../instances/uf100/uf100-0983.cnf
satyrSAT. : ../instances/uf100/uf100-0984.cnf
satyrSAT. : ../instances/uf100/uf100-0985.cnf
satyrSAT. : ../instances/uf100/uf100-0986.cnf
satyrSAT. : ../instances/uf100/uf100-0987.cnf
satyrSAT. : ../instances/uf100/uf100-0988.cnf
satyrSAT. : ../instances/uf100/uf100-0989.cnf
satyrSAT. : ../instances/uf100/uf100-0990.cnf
satyrSAT. : ../instances/uf100/uf100-0991.cnf
satyrSAT. : ../instances/uf100/uf100-0992.cnf
satyrSAT. : ../instances/uf100/uf100-0993.cnf
satyrSAT. : ../instances/uf100/uf100-0994.cnf
satyrSAT. : ../instances/uf100/uf100-0995.cnf
satyrSAT. : ../instances/uf100/uf100-0996.cnf
satyrSAT. : ../instances/uf100/uf100-0997.cnf
satyrSAT. : ../instances/uf100/uf100-0998.cnf
satyrSAT. : ../instances/uf100/uf100-0999.cnf
satyrSAT. : ../instances/uf100/uf100-01000.cnf
satyrSAT : 1000 instances en 11384 ms

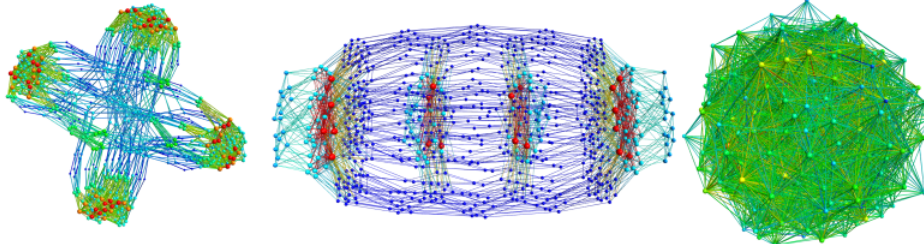
satyrSAT a résolu 4000 instances en 31857 ms
```

FIGURE 8.4 – Affichage du script de test

Le script considère qu'une instance est résolue, si le résultat renvoyé par SATyr est bien le résultat qui est attendu. Le script fait varier la taille des instances en terme de nombre de variables et de nombre de clauses.

8.4.2 La preuve par résolution

La preuve par résolution est très sensible et est la seule manière de prouver qu'un problème n'a pas de solution. Il a fallu s'assurer que la preuve était solide et "logiquement correct".



SAT Competition 2013

affiliated with the [SAT 2013](#) conference, July 8-12 in Helsinki, Finland
jointly organized by [Ulm University](#), [University of Helsinki](#),
[University College Dublin](#), [University of Texas at Austin](#),

[HOME](#)

[Results](#)

[Important dates](#)

[Submission](#)

[Competition Tracks](#)

[Certified UNSAT](#)

[Rules](#)

[Execution Environment](#)

[Call for Benchmarks](#)

[Organization](#)

[Contact information](#)

[Downloads](#)

[Proceedings](#)

[Sponsors](#)

[Credits](#)

Certified UNSAT

We are accepting 3 different UNSAT certificate types. The first format [trace-check \(stc.c\)](#) by Armin Biere is based on resolution. The second format [DRUP-check \(drup-check.c\)](#) by Marijn Heule and Nathan Wetzler is based on reverse unit propagation (RUP). The third format [BDRUP](#) by Marijn Heule is based on a conversion to the DRUP format.

The [EDACC verifier](#) which is used in the competition accepts all these formats. **All solver outputs should be on stdout, which is redirected in EDACC to file, which is then passed to the verifier.**

The output of the solver though should conform to the following rules:

- Comment lines are to be prepended by "c "
- Solution line "**s SATISFIABLE**" or "**s UNSATISFIABLE**"
- In case of unsat answer the certification format that will follow in the form "**o proof [format]**" where format can be one of DRUP, BDRUP or TC (for trace check).
- The proof as specified by each individual format

The solution line can appear anywhere in the output of the solver, though

- the proof type should appear before the proof (which should be no problem to print at the beginning of the output)
- the solution "v ..." in case of SAT should appear after the solution line "s SATISFIABLE" as it is the case in the original specification

Certified UNSAT solver output example

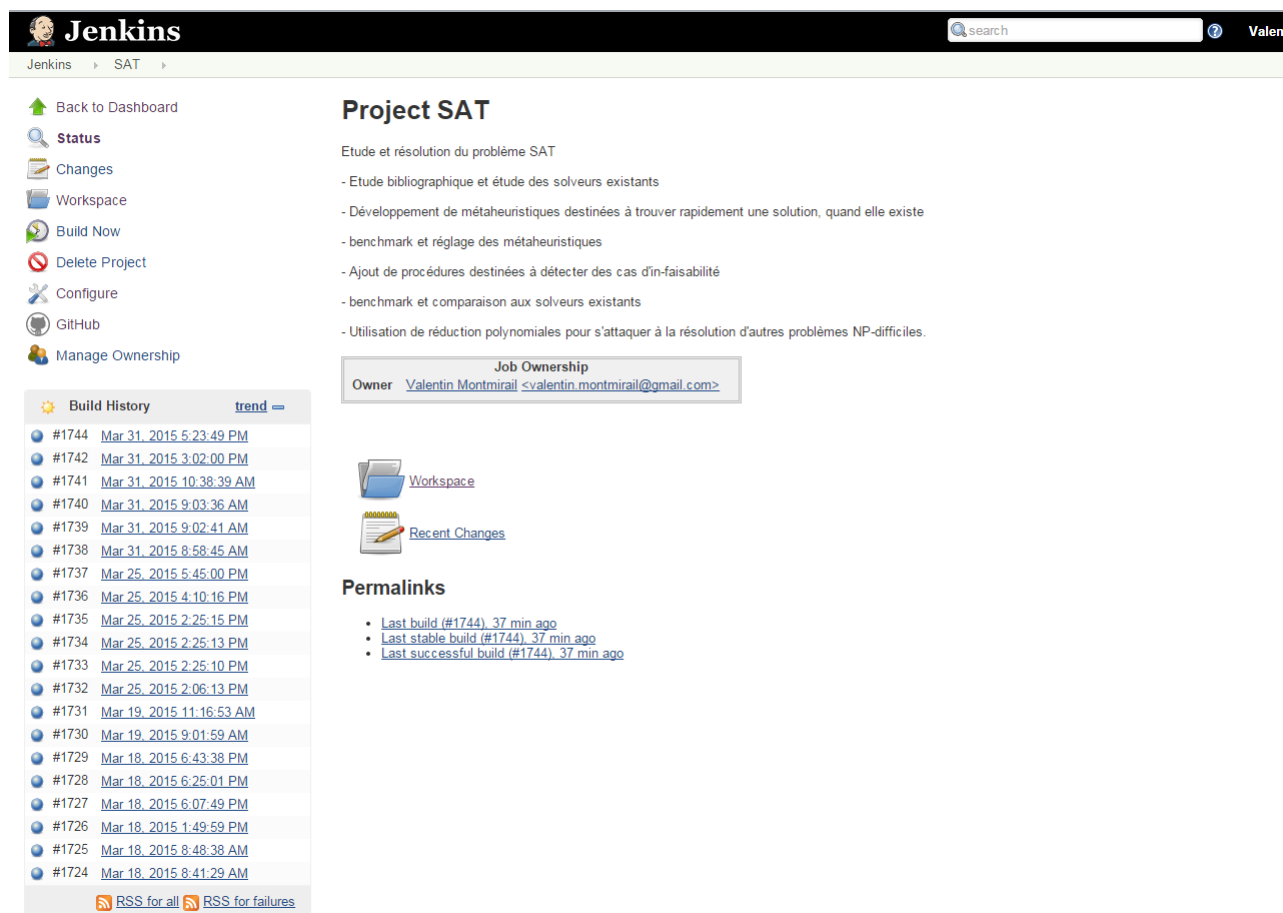
```
c this is a solver output example
c computing ....
o proof DRUP
  1 2 0
d 1 2 -3 0
  1 0
d 1 2 0
d 1 3 4 0
d 1 -2 -4 0
  2 0
  0
s UNSATISFIABLE
```

FIGURE 8.5 – Page expliquant comment fonctionne DRUP-Check

Il existe pour cela un outil appelé [DRUP-Check](#). Cet outil demande à ce que l'on écrive dans un certain formalisme l'ensemble des résolutions effectuées dans un fichier. Il va ensuite prendre le fichier et le problème SAT utilisé et va tester réellement si la preuve est correcte ou non.

8.5 Intégration continue : Jenkins

Le code étant réalisé, documenté et testé : il ne restait plus qu'à mettre tout ça en place au sein d'un système d'intégration continue.



The screenshot shows the Jenkins web interface for a project named 'SAT'. The top navigation bar includes the Jenkins logo, a search bar, and the user name 'Valen'. Below the navigation bar, the left sidebar contains links for 'Back to Dashboard', 'Status', 'Changes', 'Workspace', 'Build Now', 'Delete Project', 'Configure', 'GitHub', and 'Manage Ownership'. The main content area is titled 'Project SAT' and includes a description: 'Etude et résolution du problème SAT'. It lists several tasks: 'Etude bibliographique et étude des solveurs existants', 'Développement de métaheuristiques destinées à trouver rapidement une solution, quand elle existe', 'benchmark et réglage des métaheuristiques', 'Ajout de procédures destinées à détecter des cas d'in-faisabilité', 'benchmark et comparaison aux solveurs existants', and 'Utilisation de réduction polynomiales pour s'attaquer à la résolution d'autres problèmes NP-difficiles'. Below this, there is a 'Job Ownership' section showing the owner as 'Valentin Montmirail <valentin.montmirail@gmail.com>'. Further down, there are links for 'Workspace' and 'Recent Changes'. The 'Permalinks' section provides links for the 'Last build (#1744)', 'Last stable build (#1744)', and 'Last successful build (#1744)'. At the bottom, there is a 'Build History' table with columns for build number, timestamp, and status. The table lists builds from #1724 to #1744, with timestamps ranging from Mar 18, 2015, to Mar 31, 2015. At the bottom of the build history, there are links for 'RSS for all' and 'RSS for failures'.

FIGURE 8.6 – Page d'accueil du Jenkins du projet

Jenkins fonctionnait de la manière suivante : dès que quelque chose était placé sur la branche master, Jenkins téléchargeait le nouveau contenu, et repassait l'ensemble du code au test du script et du DRUP-Check vu plus haut.

Si le code passait l'ensemble des tests, Jenkins générerait la documentation Doxygen sans que j'ai besoin de faire une action.

Autre avantage qui est finalement un "effet de bord" : le serveur d'intégration est sur un serveur GNU/Linux sous Debian. Le développement étant réalisé sous Mac OS X. Si aucune erreur n'était générée nulle part, je m'assurais ainsi que le code était valide pour 2 systèmes différents.

8.6 Analyse de code : SonarQube

Et enfin, afin d'assurer un code de qualité pour ce solveur, une analyse de code a été réalisée via SonarQube.

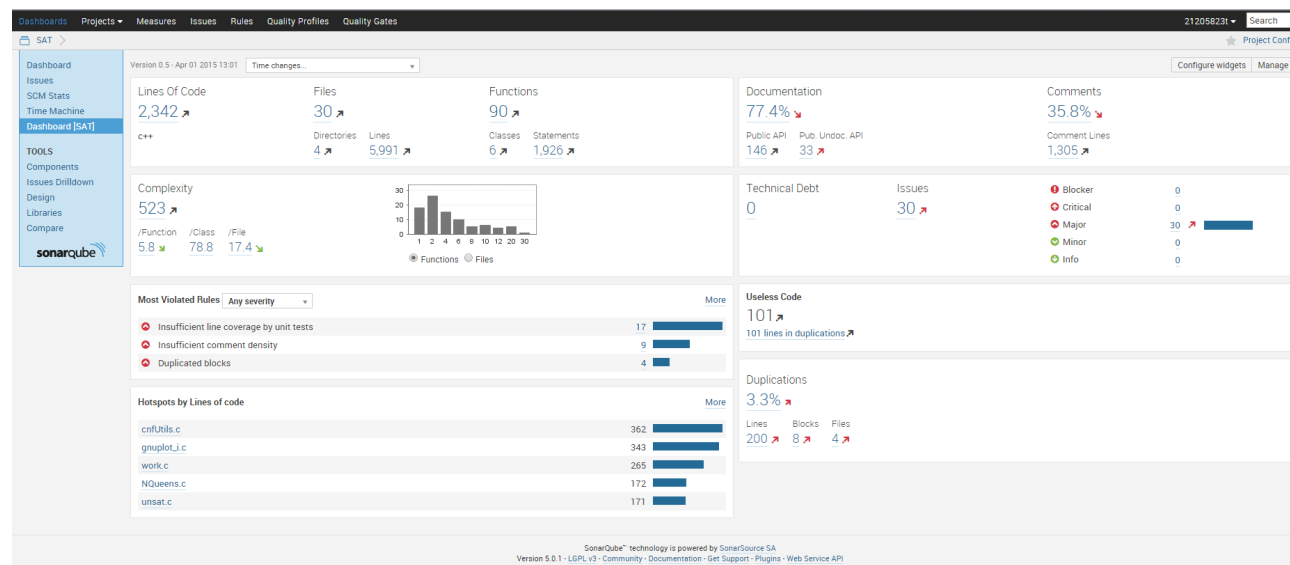


FIGURE 8.7 – Page d'accueil du SonarQube du projet

Je tiens d'ailleurs à remercier [Mr. Florent Clarret](#) pour cette partie, qui m'a accompagné dans l'intégration de mon projet sur le SonarQube de l'école (installé par ses soins durant son PFE). Ainsi pas besoin d'installer et de configurer moi même, un serveur additionnel sur l'une de mes machines, l'école l'héberge et je sais ainsi qu'il sera accessible depuis Polytech.

On peut voir sur cette analyse, que la complexité des fonctions est élevée malgré le peu de lignes de code. Cette analyse reflète assez bien ce qui aura été réalisé durant ce PFE.

Chapitre 9

Bilan personnel

Ce projet de fin d'études m'a énormément apporté, que ce soit au niveau des connaissances (théoriques et pratiques), ou que ce soit au niveau personnel. Dans cette partie je vais détailler les connaissances et savoir-faire acquis durant ce projet ainsi que leurs conséquences, avant de terminer sur un bilan plus personnel de cette aventure.

9.1 Connaissances et savoir-faire

Le thème des problèmes SAT est un thème que j'avais abordé lors de mes 2 premières années d'enseignement supérieure, durant mon DUT à [l'IUT de Lens](#). A l'époque, il aurait été totalement impossible de réaliser ce qui a été fait durant ce PFE. Les notions étaient beaucoup trop complexe...

Je trouve extrêmement satisfaisant le fait, que 3 ans plus tard, réaliser un solveur complet et manipuler des problèmes NP-difficile ne soit plus de l'ordre de l'impossible.

J'ai pu aussi réaliser du code C de manière optimisée et ainsi voir les limites au niveau de la rapidité d'exécution d'un code "genie log compliant".

De plus, ce projet étant un projet assez conséquent, nous avons ainsi pu passer à travers toutes les phases, des spécifications systèmes à la livraison chez le "client" durant la soutenance de PFE.

9.2 Découvrir les méthodes pour un projet de recherche

J'ai pu, de manière extrêmement légère, toucher un peu le monde de la recherche. Effectuer une recherche bibliographique, me servir de ce qui a été lu pour réaliser quelque chose qui à ma connaissance n'existait pas encore : Un solveur génétique complet.

De plus, j'ai pu découvrir le monde des réductions polynomiales qui est très intéressant, je n'étais pas un grand fan des problèmes de Flow Shop, Job Shop et autres problèmes mathématiques. Mais il suffit simplement de comprendre comment les transformer en SAT pour pouvoir en trouver une solution. Cela permet de mettre au point beaucoup de notions, comme celle des différentes classes de complexités par exemple.

9.3 Découvrir le monde des SAT...

J'ai ainsi pu découvrir que le monde de la recherche, au moins dans le domaine des problèmes SAT, est un monde très actif et finalement très ouvert à la discussion, tout le

monde était prêt à aider "le p'tit nouveau avec son solveur" que je suis, alors que la plupart des gens que j'ai contacté ne me connaissait même pas.

De plus, ce PFE m'a tellement plu que j'ai depuis contacté [Mr. Daniel Leberre](#) au sujet d'une poursuite d'étude à travers une offre de thèse au [Centre de Recherche en Informatique de Lens](#).



FIGURE 9.1 – Logo du CRIL

De manière très grossière, le but de cette thèse serait d'étudier, de comprendre et de réaliser un solveur Modal-SAT. La seule différence avec un solveur SAT comme celui réalisé durant ce PFE, est qu'au lieu d'utiliser la logique booléenne, le solveur Modal-SAT devrait utiliser la logique modale.

La logique modale est un type de logique qui permet de formaliser des éléments modaux, c'est-à-dire de spécifier des qualités du vrai. Exemple d'éléments modaux : "il est possible", "il est nécessaire", etc...

Chapitre 10

Conclusion









Je suis très content d'avoir eu cette expérience de PFE. Je faisais parti de ceux qui en début d'année, préféreraient simplement avoir moins de PFE pour plus de stage. Mais je me rend compte aujourd'hui que l'expérience de travailler aussi longtemps et seul sur un projet est une très bonne expérience.










Je pense avoir réussi à mener à bien les objectifs que l'on avait fixé en commun avec [Mr. Christophe Lenté](#). SATyr est fonctionnel et plutôt compétitif avec les différents solveurs du domaine.

J'ai aussi pu toucher à certaines réductions polynomiales, dont celle du problème d'emploi du temps qui est aujourd'hui entièrement fonctionnel.

Si ce projet m'était donné à refaire, je passerais peut être moins de temps sur la gestion de projet, et plus sur la phase de recherche, puisque ce projet étant finalement, plus un projet de recherche qu'un projet de développement, mais je referais ce projet sans hésiter.

Bibliographie

- [1]  Felix Martinez-Rios and Marco Antonio Cruz-Chavez.
[A Very Efficient Algorithm to Representing Job Shop Scheduling as Satisfiability Problem.](#) (English).
- [2]  Marco Antonio Cruz-Chávez and Rafael Rivera-López.
[A Local Search Algorithm for a SAT Representation of Scheduling Problems.](#) (English).
- [3]  Henry Kautz and Bart Selman. [Planning as Satisfiability.](#) (English).
- [4]  Frédéric MARIS and Pierre REGNIER and Vincent VIDAL.
[Planification SAT : Amélioration des codages](#) (Français).
- [5]  Marco Antonio Cruz-Chávez and Juan Frausto-Solis.
[A Reduced Codification for the Logical Representation of Job Shop Scheduling Problems..](#) (English).
- [6]  Yousef Kilani.
[comparing the performance of the genetic and local search algorithms for solving sat.](#)
[Applied Soft Computing, 10\(1\) :198 – 207, 2010.](#)
- [7]  Frederic Lardeux and Frederic Saubion and Jin-Kao Hao.
[GASAT: a genetic local search algorithm for the satisfiability problem.](#) (English).
- [8]  Jens Gottlieb and Elena Marchiori and Claudio Rossi.
[Evolutionary Algorithms for the Satisfiability Problem.](#) (English).

- [9]  Frédéric Lardeux and Frédéric Saubion and Jin-Kao Hao.
[local search with three truth values for sat problems](#). In Proc. of the 6th Metaheuristics International Conference (MIC'05), Vienna, Austria, aug 2005.
- [10]  Tobias Brueggemann and Walter Kern.
[an improved deterministic local search algorithm for 3-sat](#). Theoretical Computer Science, 329(1–3) :303 – 313, 2004.
- [11]  Thomas Bäck and Agoston E. Eiben and Marco E. Vink.
[a superior evolutionary algorithm for 3-sat](#). In Proceedings of the 7th Annual Conference on Evolutionary Programming, number 1477 in LNCS, pages 125–136. Springer, 1998.
- [12]  Gilles Audemard and Laurent Simon.
[GUNSAT: a Greedy Local Search Algorithm for Unsatisfiability](#). (English).
- [13]  Gilles Audemard and Jean-Marie Lagniez and Bertrand Mazure and Lakhdar Saïs. [Learning in local search](#). (English).
- [14]  Chu-Min Li and Sylvain Gérard.
[on the limit of branching rules for hard random unsatisfiable 3-sat](#). Discrete Applied Mathematics, 130(2) :277 – 290, 2003. The Renesse Issue on Satisfiability.
- [15]  Piette Cédric.
[Techniques algorithmiques pour l'extraction de formules minimales inconsistantes](#). PhD thesis, Université d'Artois, Lens, nov 2007.
- [16]  Robinson, J. A.
[A Machine-Oriented Logic Based on the Resolution Principle](#). J. ACM, 12(1) :23–41, January 1965.
- [17]  Saïs, Lakhdar. [Problème SAT : Progrès et Défis \(Français\)](#). Hermes, Londres, 2008. 352 pages.



- [18] Biere, A. and Biere, A. and Heule, M. and van Maaren, H. and Walsh, T. Handbook of Satisfiability : Volume 185 Frontiers in Artificial Intelligence and Applications (English). IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.



- [19] Marek, V.W. Introduction to Mathematics of Satisfiability. Chapman & Hall/CRC Studies in Informatics Series. Taylor & Francis, 2009.



- [20] Schöning, U. and Torán, J. The Satisfiability Problem. Mathematik für Anwendungen. Lehmanns Media, 2013.



École Polytechnique de l'Université de Tours
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. +33 (0)2 47 36 14 14
www.polytech.univ-tours.fr

Département Informatique – 5A

Cahier de spécification système & plan de développement			
Projet :	Étude et résolution du problème SAT		
Émetteur :	Valentin Montmirail	Coordonnées : <valentin.montmirail@etu.univ-tours.fr>	
Encadrant :	Christophe Lenté	Coordonnées : <christophe.lente@univ-tours.fr>	
Date d'émission :	27 avril 2015		
Validation			
Nom	Date	Valide (O/N)	Commentaires

C. Lenté : 09/10/2014 ; *N* ; Imprécisions sur les benchmarks, sur la détection de cas impossibles et sur les entrées/sorties des différents problèmes NP. Le solveur va focaliser sur les problèmes de Planification, de Job Shop et de Flow Shop.

C. Lenté : 23/10/2014 ; *N* ; Imprécisions sur les entrées/sorties des problèmes de Flow Shop. Plus de précisions à apporter sur les benchmarks.

C. Lenté : 08/12/2014 ; *O* ; Je suis ok avec cette version

Historique des modifications		
Version	Date	Description de la modification

0.1 : 02/10/2014 ; Le planning n'est pas encore fait, et le cahier de spécification est réalisé.

0.2 : 16/10/2014 ; Les imprécisions de la version 0.01 ont été corrigées.

0.3 : 06/11/2014 ; Le descriptif des benchmarks est maintenant plus précis et des exemples d'entrée/sorties pour les problèmes NP-difficile sont donnés.

0.4 : 06/11/2014 ; Le diagramme de Gantt et la liste des tâches ont été refait afin d'étaler les tâches complexes.

0.5 : 27/11/2014 ; Le Gantt a été intégré au projet, la description des tâches a été revue ainsi que la liste des livrables du projet.

1.0 : 08/12/2014 ; La date à laquelle le rapport s'est terminé.

Table des matières

Cahier de spécification système	6
1.1 Introduction	6
1.2 Contexte de la réalisation	6
1.2.1 Contexte	6
1.2.2 Objectifs	6
1.2.3 Hypothèses	7
1.2.4 Bases méthodologiques	8
1.3 Description générale	9
1.3.1 Environnement du projet	9
1.3.2 Fonctionnalités et structure générale du système	9
1.3.3 Contraintes de développement, d'exploitation et de maintenance	10
1.3.4 Problèmes ϕ considérés	10
1.4 Description des interfaces externes du logiciel	12
1.4.1 Interfaces matériel/logiciel	12
1.4.2 Interfaces homme/machine	12
1.4.3 Interfaces logiciel/logiciel	12
1.5 Architecture générale du système	13
1.6 Description des fonctionnalités	13
1.6.1 Définition de la création d'un Problème	13
1.6.2 Définition de la résolution d'un Problème	14
1.6.3 Définition du test de vérification de Solution	14
1.6.4 Définition de transformerEnFichierSAT(string filenameProblemeNP)	14
1.6.5 Définition de lireSolution(string filenameSolutionSAT)	15
1.7 Conditions de fonctionnement	15
1.7.1 Performances	15
1.7.2 Contrôlabilité	15
 Plan de développement	 17
2.1 Planning	17
2.2 Découpage du projet en tâches	18
2.2.1 tâche 1 : Gestion de projet (tâche continue)	18
2.2.2 tâche 2 : Rédiger les spécifications	18
2.2.3 tâche 3 : Rédiger la bibliographie	18
2.2.4 tâche 4 : Mettre en place les outils de développement	18
2.2.5 tâche 5 : Documenter le projet (tâche continue)	19
2.2.6 tâche 6 : Résoudre SAT avec algorithmes génétiques	19
2.2.7 tâche 7 : Réaliser les tests unitaires sur solveur SAT	19
2.2.8 tâche 8 : Mettre en place méta-heuristiques pour UNSAT	19
2.2.9 tâche 9 : Réaliser les tests unitaires sur solveur SAT+UNSAT	20
2.2.10 tâche 10 : Réaliser les réductions polynomiales vers SAT	20
2.2.11 tâche 11 : Réaliser les tests performances et optimisations	20
2.2.12 tâche 12 : Réaliser les benchmarks	20



2.2.13 tâche 13 : Rédiger le rapport final et préparer la soutenance	20
2.3 Date et noms des différents livrables	21
Glossaire	22
A Références	23

Cahier de spécification système

1.1 Introduction

Ce document constitue une référence pour l'ensemble des spécifications du projet.

Il définit de manière précise l'environnement de notre logiciel, les spécifications fonctionnelles qui devront être implémentées ainsi que les contraintes de développement avancées par la maîtrise d'ouvrage.

La maîtrise d'ouvrage, l'équipe **Ordonnancement et Conduite**, est représenté par M.**Christophe Lenté**, enseignant chercheur au sein de l'école Polytechnique de l'Université de Tours et du laboratoire d'Informatique.

La maîtrise d'oeuvre est représentée par moi-même, M.**Valentin Montmirail**, élève en 5ème année à l'école Polytechnique de l'Université de Tours.

L'encadrement de ce projet est assuré par Mr.**Christophe Lenté**.

1.2 Contexte de la réalisation

1.2.1 Contexte

L'équipe Ordonnancement et Conduite de l'école Polytech Tours effectue des recherches sur des applications variées.

Ces recherches menées au sein de cette équipe portent principalement sur l'étude et la résolution des problèmes d'ordonnancement et de planification, que ce soit dans le cadre de problématiques issues du monde de l'industrie ou du service, ou de problèmes académiques.

L'équipe s'intéresse également de plus en plus aux problèmes d'optimisation liés au transport de biens ou de personnes ainsi qu'aux problèmes d'optimisation apparaissant dans les systèmes de santé.

1.2.2 Objectifs

Le but de ce projet est de fournir une solution, qui sera appelée par la suite un solveur, permettant la résolution de problèmes **SAT**, et de le tester à travers une réduction polynomiale de problèmes **NP-difficiles**.

L'objectif de ce projet est de pouvoir fournir :

- Une étude bibliographique des problèmes **SAT**, des réductions polynomiales et des solveurs existants, permettant à un néophyte de pouvoir avoir assez de connaissances pour reprendre et améliorer ce projet.
- Le développement du dit solveur au travers de méta-heuristiques destinées à trouver rapidement une solution.
- Des procédures destinées à détecter les cas d'infaisabilité. La procédure qui sera utilisée est la suivante :
 - Le problème initial va être réécrit différemment en ajoutant/supprimant/modifiant les clauses du problème. Le principe est donc, à l'aide de la règle de résolution et via des réécritures, de créer une clause vide si possible et ainsi démontrer que le problème est **UNSAT**.
- Des benchmarks afin de tester la qualité de ce solveur par rapport à ceux déjà existants. Ces solveurs sont les suivant :

- **zChaff**¹ (Moskewicz, Madigan, Zhao, Zhang) qui est l'implémentation la plus connue de solveur SAT.
- **WalkSat**² (Bart Selma et Henry Kautz), qui est un solveur en recherche locale de manière stochastique. Ce solveur est réputé pour être très efficace sur les problèmes de planification.
- **Glucose**³ (Audemard et Simon), basé sur Minisat et utilisant les algorithmes **Conflict-Driven Clause Learning** actuels. Ce solveur a fini 1er dans la compétition des SAT/UNSAT en 2012.
- Les benchmarks seront réalisés sur différents types de problème **SAT** :
 - Des instances 3-SAT générées de manière aléatoire, en faisant varier le ratio nombre de clause/ nombre de variables.
 - Des instances certifiées **UNSAT** mais l'information ne sera pas communiquée aux solveurs.
 - Des instances certifiées **SAT** mais l'information ne sera pas communiquée aux solveurs.
 - Les problèmes **NP-Complet** que l'on aura réduits en **SAT**.

TABLE 1.1 – Benchmarks utilisés : <http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html>

Nom du benchmarks	# instances	clauses	# Variables	# Clauses	SAT ou UNSAT
<i>uf20-91</i>	1000	3	20	91	<i>SAT</i>
<i>uf50-218 / uuf50-218</i>	1000	3	50	218	<i>SAT et UNSAT</i>
<i>uf75-325 / uuf75-325</i>	1000	3	75	325	<i>SAT et UNSAT</i>
<i>uf100-430 / uuf100-430</i>	1000	3	100	430	<i>SAT et UNSAT</i>
<i>uf125-538 / uuf125-538</i>	1000	3	125	538	<i>SAT et UNSAT</i>
<i>uf150-645 / uuf150-645</i>	1000	3	150	645	<i>SAT et UNSAT</i>
<i>uf175-753 / uuf175-753</i>	1000	3	175	753	<i>SAT et UNSAT</i>
<i>uf200-860 / uuf200-860</i>	1000	3	200	860	<i>SAT et UNSAT</i>
<i>uf225-960 / uuf225-960</i>	1000	3	225	960	<i>SAT et UNSAT</i>
<i>uf250-1065 / uuf250-1065</i>	1000	3	250	1065	<i>SAT et UNSAT</i>
<i>hole6.cnf</i>	1	6	42	133	<i>UNSAT</i>
<i>hole7.cnf</i>	1	7	56	204	<i>UNSAT</i>
<i>hole8.cnf</i>	1	8	72	297	<i>UNSAT</i>
<i>CBS_k3_n100_m449_b90</i>	1000	3	100	449	<i>SAT</i>

- La documentation du solveur afin qu'en plus des connaissances théoriques obtenu grâce à l'étude bibliographique, le code soit plus facilement améliorable.

Bien évidemment, l'ensemble des choix réalisés durant ce projet seront justifiés dans le rapport final afin qu'aucun point ne soit obscur pour quiconque voudrait améliorer ce projet.

1.2.3 Hypothèses

Les problèmes de classe **NP**-difficiles étant pour la plupart, compliqués à résoudre et à comprendre, le nom des problèmes de classe **NP**-difficiles qui seront utilisés pour tester le solveur ne sont pas spécifiés afin de pouvoir être, même en cas d'imprévu, conforme au cahier des spécifications. (Sauf en cas de mise en exemple concret comme lors d'un cas d'utilisation).

Cependant, afin que l'équipe Ordonnancement et Conduite puisse tirer un maximum d'avantage de ce projet, les problèmes de classe **NP**-difficiles qui serviront de test du solveur seront probablement des problèmes d'ordonnancement et de planification.

1. <https://www.princeton.edu/~chaff/zchaff.html>
 2. <http://www.cs.rochester.edu/u/kautz/walksat/>
 3. <http://www.labri.fr/perso/lsimon/glucose/>

1.2.4 Bases méthodologiques

La méthodologie retenue pour la conduite de ce projet est une approche par cycle en V, permettant une structuration de la réalisation de la solution. Cependant, le laboratoire d'Ordonnancement et Conduite étant situé au même emplacement que l'équipe de développement, la maîtrise d'ouvrage pourra avoir un suivi constant de l'avancement du projet et ainsi ajuster ses demandes.

Pour le développement de ce solveur, le langage de programmation n'est pas encore défini, la nécessité de rapidité étant très présente dans ce projet, le langage sera sûrement un langage compilé (langage C) qui permet une réalisation portable et compréhensible par la plupart des personnes ayant des bases en programmation.

La compilation ainsi que les tests unitaires du solveur pouvant être très chronophages, un système d'intégration continu ([Jenkins](#)) sera mis en place afin de pouvoir décharger le coût en temps des tests sur un serveur distant.

De plus, afin de pouvoir garantir une qualité de code tout le long du projet, un logiciel de mesure de qualité de code source ([SonarQube](#)) sera mis en place.

1.3 Description générale

1.3.1 Environnement du projet

Pour utiliser le solveur au maximum de ses capacités, une machine avec un processeur rapide pourrait avoir un avantage certain, mais cet environnement n'est qu'un environnement de confort, étant donné qu'une simple machine personnelle permettra d'utiliser le solveur dans des temps d'exécutions raisonnables.

1.3.2 Fonctionnalités et structure générale du système

Le système, une fois lancé par l'utilisateur, devra être capable d'effectuer la résolution d'un problème de classe **NP**-difficile via une réduction polynomiale vers **SAT**.

Attention, les fonctions de transformation du problème considéré vers **SAT** et de la solution **SAT** vers la solution du problème considéré (représenté en bleu sur le diagramme de cas d'utilisation ci-dessous) devront avoir été préalablement codées.

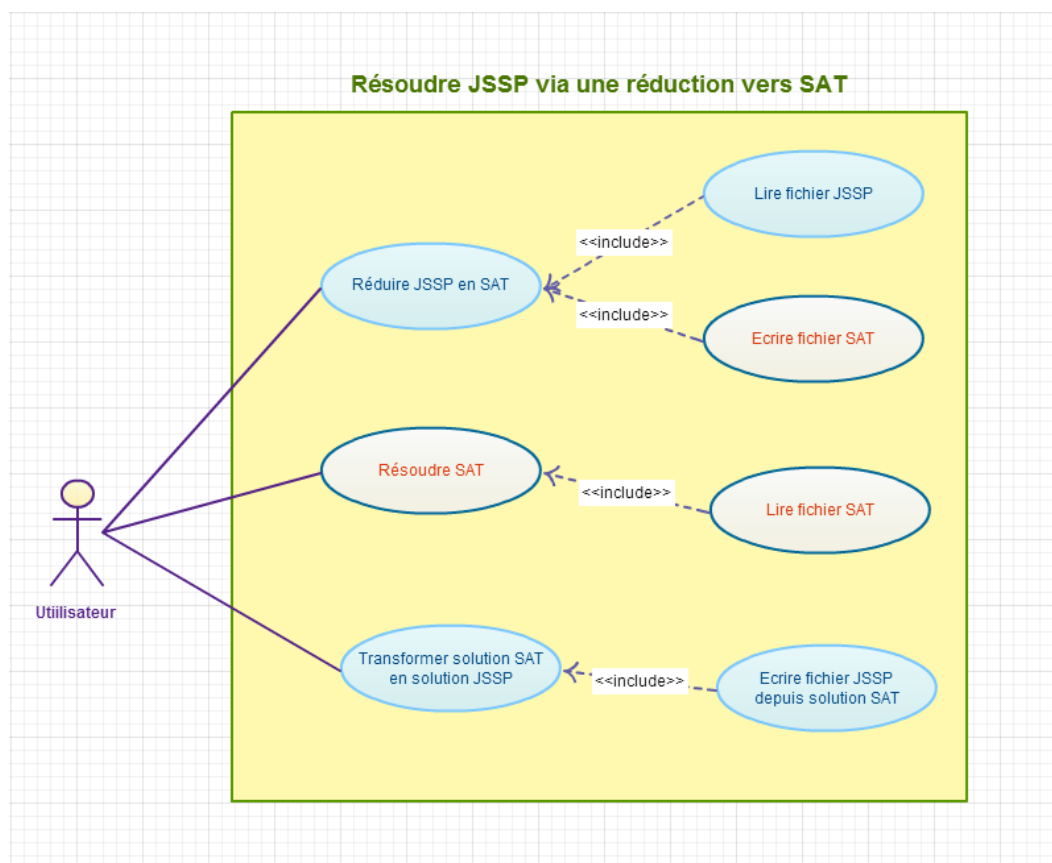


FIGURE 1.1 – Diagramme de cas d'utilisation dans la résolution de **Job Shop Scheduling Problem**

N'importe quel problème de classe **NP**-complet (noté ϕ) peut donc être résolu à l'aide de ce solveur, moyennant un coût de traduction de ϕ vers SAT et de solution/non-solution SAT vers solution/non-solution ϕ .

1.3.3 Contraintes de développement, d'exploitation et de maintenance

Contraintes de développement

Le projet final est à développer avant le 15 mai 2015.

Contraintes d'exploitation

Le projet final sera valide sous les systèmes d'exploitations de type Unix.

Maintenance et évolution du système

- Pour quiconque voudrait améliorer ce solveur et le rendre encore plus général, il faut pour chaque problème de classe NP-difficile (noté ϕ) que le solveur doit résoudre,
- ajouter les fonctions de transformation de ϕ vers SAT.
 - Le fichier d'entrée dépendra du problème ϕ considéré
 - Le fichier de sortie devra être un fichier en forme normale conjonctive valide.
 - ajouter les fonctions de transformation de SAT vers ϕ .
 - Le fichier d'entrée devra être un fichier SAT en état SATISFIABLE ou UNSATISFIABLE.
 - Le fichier de sortie dépendra du problème ϕ considéré.

1.3.4 Problèmes ϕ considérés

Le solveur pouvant théoriquement résoudre un ensemble très vaste de problèmes NP-Complet, il a bien fallu restreindre le développement à certains problèmes. Voici ceux qui ont été validés par le client :

Planning Problem

Les instances de planification seront formalisées comme des fichiers de type Planning Domain Definition Language

Ce genre de fichier possède les composants suivants :

- Objets : Les choses dans le monde qui nous intéressent.
 - Prédicats : Les propriétés des objets qui peuvent être vrais ou faux.
 - L'état initial : l'état dans lequel le monde se trouve au début.
 - Le but : Les choses que l'on voudrait vraies en sortie.
 - Action/Opérateurs : Les manières de changer l'état du monde.
- Nous ne tiendrons compte que de ces éléments pour les problèmes de Planifications.

Job Shop Scheduling Problem

Chaque instance sera formalisé comme suit :

Ji	Di	Ordre sur les M	Temps sur la machine Mi
J1	D1	M1 M3 M4 ... Mm	T1 T2 Tm
J2	D2	M5 M1 M4 ... Mm	T1 T2 Tm
...
Jn	Dn	M1 M2 M3 ... Mm	T1 T2 Tm

Nous réaliserons des instances à n jobs et m machines, la seule contrainte sur n et m est juste que l'instance SAT générée ne doit pas être trop grande. Selon cet article voici les équations qui nous donnent

le nombre de clauses et variables en fonction du nombre de jobs (pour $n = m$) :

$$Clauses = 2n^2 - 2n \quad (1.1)$$

$$Variables = 6n^2 - 6n \quad (1.2)$$

La fonction objectif sera C_{max} , la date de fin du dernier job.

Flow Shop Scheduling Problem

Le flow-shop définit un ensemble de n tâches et m machines. Les fichiers seront formalisés comme suit :

Ji	Di	Temps sur la machine Mi
J1	D1	T1 T2 Tm
J2	D2	T1 T2 Tm
...
Jn	Dn	T1 T2 Tm

La fonction objectif sera C_{max} , la date de fin de la dernière tâche.

Les ajouts possibles

Si le temps du projet le permet, il serait intéressant d'ajouter :

- le problème du voyageur du commerce
- le problème de tournées de véhicules

car elles peuvent être des améliorations très intéressantes dans l'ensemble des problèmes que le solveur peut résoudre.

1.4 Description des interfaces externes du logiciel

1.4.1 Interfaces matériel/logiciel

Pas d'interface matériel n'est à signaler.

1.4.2 Interfaces homme/machine

L'interface homme/machine sera donc une ligne de commande affichant, à l'aide d'un paramètre d'utilisation `-help`, comment utiliser le solveur.

Chaque message d'erreur sera aussi explicite que possible, toujours dans une optique de pouvoir reprendre et améliorer ce projet en très peu de temps d'adaptation.

Une interface graphique permettant d'aller chercher le fichier d'instance de ϕ pourrait être mis en place afin de faciliter l'utilisation du projet.

Cette interface graphique ressemblerait au Balsamiq Mockup suivant :

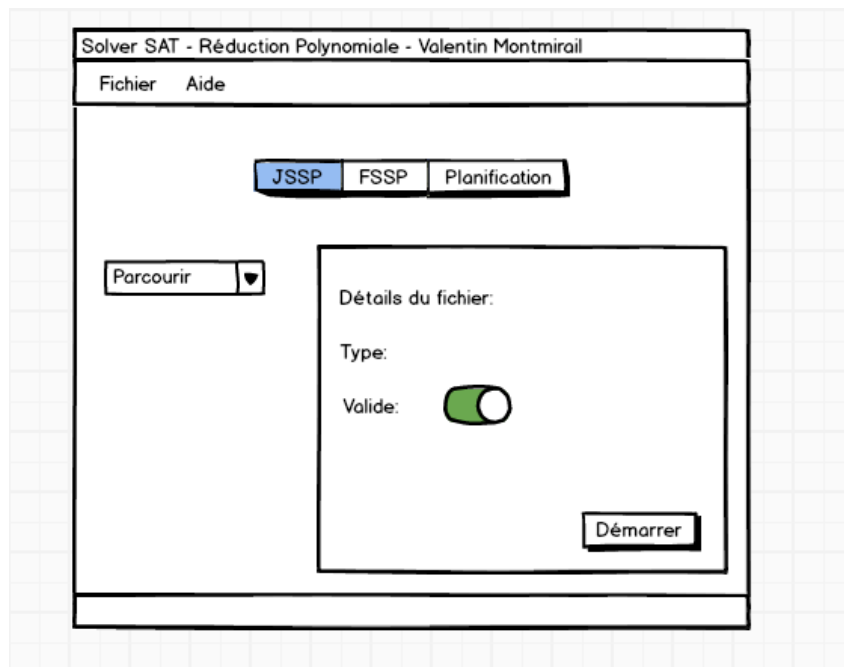


FIGURE 1.2 – Balsamiq Mockup de l'interface graphique à réaliser

1.4.3 Interfaces logiciel/logiciel

Pas d'interface logiciel/logiciel n'est à signaler.

1.5 Architecture générale du système

L'architecture générale du projet peut être résumée avec le diagramme de classes suivant :

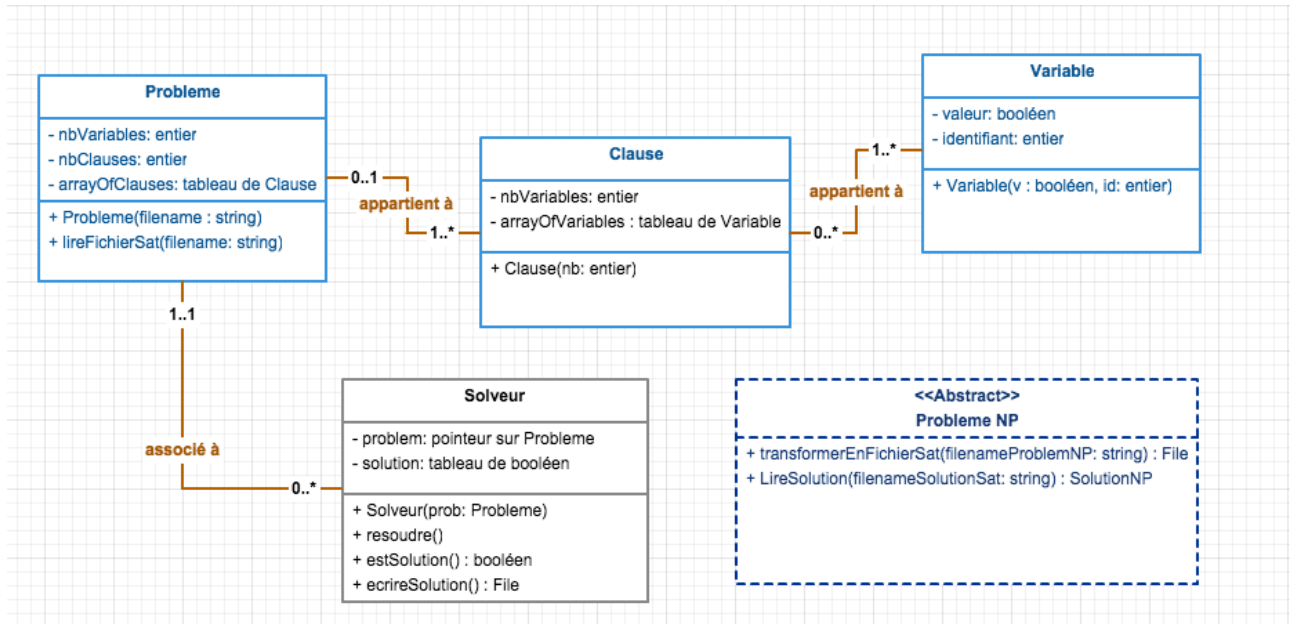


FIGURE 1.3 – Diagramme de classe du projet

Comme on peut le voir il y a 3 grandes parties :

- le triplet Problème ; Clause ; Variable, pour coder un problème SAT.
- la classe Solvreur : qui résout un problème SAT.
- la classe abstraite ProblemeNP qui spécifie que pour qu'un problème de classe NP-difficile soit résolvable par le solveur, il doit pouvoir fournir un fichier formalisé en forme normale conjonctive et être capable de lire une solution SAT pour fournir une solution du problème de classe NP-difficile d'origine.

Cette architecture va devenir de plus en plus complexe, en fonction des nouveaux problèmes de classe NP-difficiles que le solveur sera apte à résoudre. Le fonctionnement sera toujours le même qu'importe le nouveau problème que l'on voudra résoudre :

- On crée une classe héritant de ProblemeNP
- On code la fonction de transformation en problème SAT
- On code la fonction de lecture de la solution SAT en solution du problème de classe NP.

1.6 Description des fonctionnalités

1.6.1 Définition de la création d'un Problème

Identification de la fonction

- nom : Problème(string filename).
- utilité : permet de créer un Problème à partir d'un fichier en forme normale conjonctive.
- priorité : primordiale.

Description de la fonction

- Le paramètre 'filename' correspond au chemin pour accéder au fichier **forme normale conjonctive**, la fonction va le lire et créer les Clauses et les Variables en conséquence.
- Cette fonction est donc dépendante de la classe Clause et la classe Variable comme vu plus haut dans le diagramme de classe du projet. (13)
- Les cas anormaux mais prévisible de fonctionnement sont :
 - Le fichier n'existe pas ou le chemin est incorrect.
 - Le fichier n'est pas formalisé comme un fichier en **forme normale conjonctive**.Chaque erreur sera donc relevée et gérée dans la mesure du possible et on indiquera le cas d'erreur avec un message d'erreur explicite.

1.6.2 Définition de la résolution d'un Problème

Identification de la fonction

- nom : resoudre(Probleme p).
- utilité : permet de répondre à la question : "Le problème p a-t-il une solution ?"
- priorité : primordiale.

Description de la fonction

- Le paramètre 'p' correspond à une référence vers le problème où l'on essaie de terminer si ce dernier a une solution ou non.
- Cette fonction est donc dépendante de la classe Problème comme vu plus haut dans le diagramme de classes du projet. (13)
- Les cas anormaux mais prévisibles de fonctionnement sont :
 - Le problème est bien trop gros et nous fait faire un 'OutOfMemory'
 - Un signal d'interruption a été envoyé, le problème est donc dans une phase "UNKNOWN"Chaque erreur sera donc relevée et gérée dans la mesure du possible et on indiquera le cas d'erreur avec un message d'erreur explicite.

1.6.3 Définition du test de vérification de Solution

Identification de la fonction

- nom : estSolution(Problem p).
- utilité : permet de savoir si l'affectation des littéraux dans le solveur fournit une solution au problème
- priorité : haute.

Description de la fonction

- Le paramètre 'p' correspond à une référence vers le problème où l'on essaie de savoir si le solveur a fourni une solution ou non.
- Cette méthode fait partie de la classe Solveur et est dépendante de la classe Problème comme vu plus haut dans le diagramme de classes du projet. (13)
- Cette dernière va vérifier si pour chaque clause d'un problème, il existe au moins un littéral à vrai.

1.6.4 Définition de transformerEnFichierSAT(string filenameProblemeNP)

Identification de la fonction

- nom : transformerEnFichierSAT(string filenameProblemeNP) : string

- utilité : correspond à la méthode abstraite que l'on "appellera" lorsque l'on voudra résoudre un problème de classe NP via une réduction polynomiale vers SAT.
- priorité : haute.

Description de la fonction

- Le paramètre 'filenameProblemeNP' correspond au chemin vers le fichier qui caractérise le problème de classe NP que l'on veut résoudre. Cette méthode va retourner un fichier correspondant à ce même problème, transformer en *forme normale conjonctive*.
- Cette méthode fait partie de la classe abstraite ProblemeNP. (13)

1.6.5 Définition de lireSolution(string filenameSolutionSAT)

Identification de la fonction

- nom : lireSolution(string filenameSolutionSAT) : string
- utilité : correspond à la méthode abstraite que l'on "appellera" lorsque l'on transforme la solution SAT en solution du problème de classe NP que l'on veut résoudre.
- priorité : haute.

Description de la fonction

- Le paramètre 'filenameSolutionSAT' correspond au chemin vers la solution au format *forme normale conjonctive* de notre problème de classe NP. Le but de cette méthode est donc de transformer la solution SAT en solution lisible pour le problème de classe NP.
- Cette méthode fait partie de la classe abstraite ProblemeNP. (13)

1.7 Conditions de fonctionnement

1.7.1 Performances

Les problèmes SAT étant toujours un thème actif de recherche, il faut bien prendre conscience que la taille d'un problème est limitée de part le fait que ces problèmes soient de classe NP-Complet. Comme indiqué dans le livre "Problème SAT - progrès et défis"⁴, la taille des instances n'ayant pas de solution (celles qui sont donc les plus longues à résoudre) n'arrive de nos jours pas à dépasser les 400 variables de moyenne, Ceci vient du fait que les méthodes de résolution connues ont une complexité de nature exponentielle.

Une deuxième limitation sera aussi la valeur de la mémoire vive de la machine exécutant le solveur. Les tests seront réalisés sur une machine possédant 8 Go de RAM, ce qui permet de monter jusqu'à 12 000 variables et 20 000 clauses, au delà, le fait de représenter le problème et de tenter de le résoudre déclenchera rapidement un OutOfMemory.

1.7.2 Contrôlabilité

Ce projet possède une contrôlabilité élevée :

- une analyse de code sera réalisée afin de s'assurer de la conformité du code pour les règles de développement classiques. Le résultat de cette analyse sera affiché sur une interface Sonar afin de pouvoir contrôler de manière graphique cette conformité.

4. [ISBN:2746218860](#)

Plan de développement

2.1 Planning

Voici donc l'ensemble des tâches "principales" du projet, chacune d'entre elle sera détaillée dans les parties suivantes.

Id	Nom de la Tâche	Durée	Date début	Prédécesseurs
1	Kickoff		01/10/2014	
2	Gestion de projet (tâche continue)	150	01/10/2014	1
3	Rédiger les spécifications	35	01/10/2014	1
4	Rédiger la bibliographie	29	01/10/2014	1
5	Mettre en place les outils de développement	5	01/10/2014	1
6	Documenter le projet (tâche continue)	80	19/11/2014	3
7	Résoudre SAT avec algorithmes génétiques	30	19/11/2014	3
8	Réaliser les tests unitaires sur solveur SAT	5	31/12/2014	7
9	Livraison du solveur pour SAT		06/01/2015	8
10	Mettre en place métaheuristiques pour UNSAT	35	07/01/2015	9
11	Réaliser les tests unitaires sur solveur SAT+UNSAT	5	25/02/2015	10
12	Livraison du solveur pour SAT + UNSAT		03/03/2015	11
13	Réaliser les réductions polynomiales vers SAT	25	04/03/2015	12
14	Réaliser les tests performances & optimisations	10	08/04/2015	13
15	Livraison en version 1.0		21/04/2015	14
16	Réaliser les benchmarks	10	22/04/2015	15
17	Rédiger le rapport final & préparer la soutenance	5	06/05/2015	16
18	Fin du projet		12/05/2015	17

Et voici l'ensemble de ces tâches sous le format d'un diagramme de Gantt :

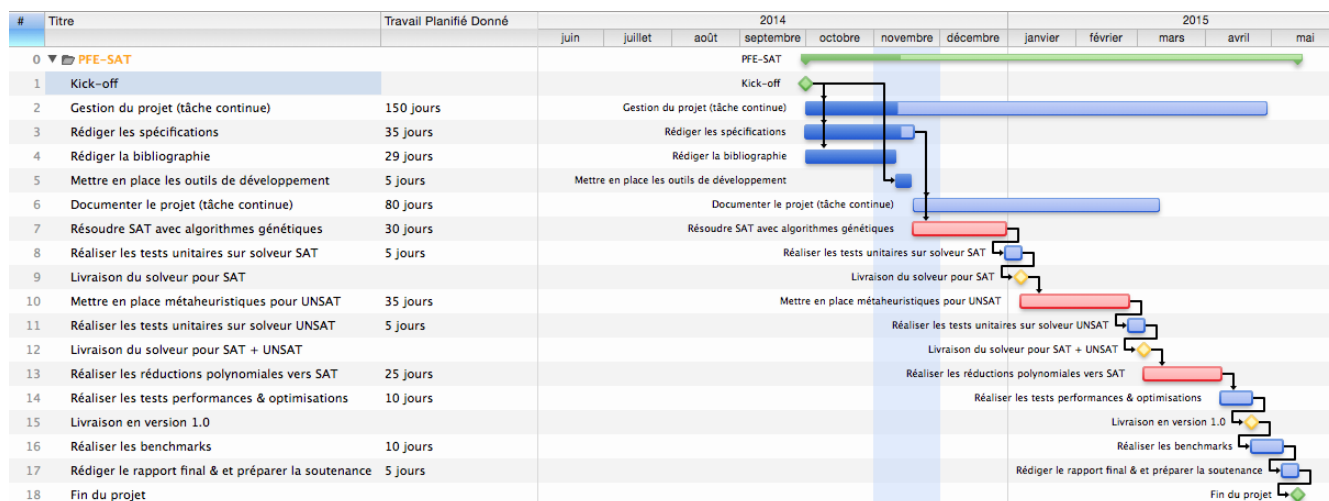


FIGURE 2.4 – Diagramme de Gantt

2.2 Découpage du projet en tâches

2.2.1 tâche 1 : Gestion de projet (tâche continue)

Description de la tâche

Cette tâche est une tâche continue précisant que tout le long du projet il y a eu des rendez-vous régulier avec la maîtrise d'ouvrage, que des mails, afin de tenir cette maîtrise d'ouvrage informé, ont été envoyés.

Qu'il a fallu des fois éclaircir certains points afin de préciser concrètement comment ces points aller être réalisés.

C'est donc tout ce temps passé qui est symbolisé par cette "tâche" continue.

2.2.2 tâche 2 : Rédiger les spécifications

Description de la tâche

Cette tâche correspond à la rédaction du cahier des spécifications du projet ainsi que de la planification de ce dernier. La version définitive doit être rendu dans le cadre de ce projet de fin d'étude pour le 9 Janvier 2015.

Dire en quoi consiste la tâche, quels sont les composants, parties ou fonctions du système qu'elle intègre, le degré de finalisation souhaité (version alpha, bêta, etc.), etc.

Livrables

Cette tâche fournira donc en livrable le Cahier des Spécifications ainsi que le Plan de Développement en version 1.0

2.2.3 tâche 3 : Rédiger la bibliographie

Description de la tâche

Cette tâche correspond à la rédaction d'un document listant et expliquant en quelque mot les éléments de la bibliographie de ce projet. L'ensemble des livres lus et articles liés à ce projet.

Livrables

Cette tâche fournira en livrable un document de Bibliographie en version 1.0

Contraintes temporelles

Certains livres ont été commandés via la Bibliothèque de l'École Polytechnique de l'Université de Tours, la lecture et le résumé de ces livres ne peuvent être réalisés qu'une fois les livres arrivés.

2.2.4 tâche 4 : Mettre en place les outils de développement

Description de la tâche

Cette tâche correspond à la mise en place du versionning du projet via Git. Une fois le projet versionné, une intégration continue sera réalisé grâce à l'outil Jenkins. Il faut donc mettre en place le serveur correspondant.

Et enfin, une analyse qualité du code sera réalisé via Sonar qui sera accessible depuis l'interface de Jenkins, il faut donc aussi mettre en place le serveur Sonar et les fichiers de configuration correspondant.

2.2.5 tâche 5 : Documenter le projet (tâche continue)

Description de la tâche

Cette tâche correspond à la réalisation de la documentation du projet grâce à l'outil Doxygen. L'ensemble des fonctions réalisés seront commentés et documenté.

Livrables

Cette tâche fournira en livrable une documentation Doxygen sur l'ensemble du projet.

2.2.6 tâche 6 : Résoudre SAT avec algorithmes génétiques

Description de la tâche

Cette tâche correspond à la charge principale de ce projet, et celle qui va déterminer l'efficacité du projet. Elle consiste à réaliser à base d'algorithmes évolutionnaires, un solveur SAT. Il va donc régler précisément les pourcentages de mutations et de croisement de manière à être le plus efficace dans la résolution des problèmes SAT.

Livrables

Cette tâche fournira en livrable, le solveur capable de résoudre efficacement les instances SAT qui possèdent forcément une solution.

2.2.7 tâche 7 : Réaliser les tests unitaires sur solveur SAT

Description de la tâche

Cette tâche correspond à la mise en place des tests unitaires qui vont

- Tester que chacune des fonctions réalisent bien ce qui est attendu
- Tester le solveur qui sur des fichiers dont le résultat est connu (SAT) va tester doit renvoyer le même résultat.

Livrables

Un rapport détaillé indiquant combien de pourcentage du projet a été testé.

2.2.8 tâche 8 : Mettre en place méta-heuristiques pour UNSAT

Description de la tâche

Cette tâche correspond à la mise en place de méta-heuristique capable de détecter les cas impossibles lors d'instances SAT ne possédant pas de solutions. Ces méta-heuristiques seront basés sur le principe de résolution où des résolutions vont être effectué jusqu'à trouver une clause vide et prouver ainsi que le problème ne possède pas de solutions.

Livrables

Cette tâche fournira en livrable, un solveur capable de résoudre les instances **UNSAT** en plus des instances SAT déjà possible depuis le livrable précédent.

2.2.9 tâche 9 : Réaliser les tests unitaires sur solveur SAT+UNSAT

Description de la tâche

Cette tâche correspond à la mise en place des tests unitaires qui vont

- Tester que chacune des fonctions réalisent bien ce qui est attendu
- Tester le solveur qui sur des fichiers dont le résultat est connu (SAT/UNSAT) va tester doit renvoyer le même résultat.

Livrables

Un rapport détaillé indiquant combien de pourcentage du projet a été testé.

2.2.10 tâche 10 : Réaliser les réductions polynomiales vers SAT

Description de la tâche

Cette tâche correspond à la réalisation des réductions polynomiales de **Job Shop Scheduling Problem** vers SAT, de **Flow Shop Scheduling Problem** vers SAT ainsi que des **Planning Problem** vers SAT (et réciproquement). Le programme devra donc être capable à partir de fichier **Job Shop Scheduling Problem**, **Flow Shop Scheduling Problem** et des **Planning Problem**, de fournir l'équivalent en **forme normale conjonctive** et de fournir la solution à ces problèmes en passant par **SAT**

Livrables

Cette tâche fournira le solveur en version 1.0

2.2.11 tâche 11 : Réaliser les tests performances et optimisations

Description de la tâche

Cette tâche correspond à la mise en place de tests de performances calculant pour des fichiers de tests dont le résultat est connu, en combien de temps le solveur est capable d'obtenir le même résultat.

Puis une fois cette observation réalisée, la tâche consistera à optimiser la résolution des instances (à travers la modification des paramètres des algorithmes évolutionnaires).

2.2.12 tâche 12 : Réaliser les benchmarks

Description de la tâche

Cette tâche correspond à la réalisation des benchmarks (précisés dans cette sous-partie : 6)) entre le solveur et les solveurs existant précisés dans la partie "Objectif" (sous-partie : 6)

Livrables

Cette tâche fournira en livrable un tableau pour les résultats des benchmarks réalisés. Ce tableau précisera pour chaque solveur si ce dernier a trouvé ou non un résultat, ainsi que le temps qu'il a mis pour le trouver.

2.2.13 tâche 13 : Rédiger le rapport final et préparer la soutenance

Description de la tâche

Cette tâche correspond à la rédaction du rapport final du projet ainsi qu'à la réalisation de la présentation pour la soutenance de projet.

Il s'agit donc de tout rassembler en un seul projet afin que si quelqu'un veuille reprendre ce projet dans les années suivante, tout lui soit accessible à un seul endroit.

Livrables

Cette tâche fournira en livrable le rapport final du projet ainsi que les slides de présentation pour la soutenance de projet.

2.3 Date et noms des différents livrables

Nom du livrable	Date de livraison
Document de bibliographie	13/11/2014
Solveur SAT	06/01/2015
Cahier de spécification	09/01/2015
Solveur SAT+UNSAT	03/03/2015
Rapport des tests unitaires	08/04/2015
Solveur en version 1.0	21/04/2015
Documentation du projet	21/04/2015
Tableau des benchmarks	06/05/2015
Rapport final de PFE	12/05/2015
Présentation pour soutenance	12/05/2015

Glossaire

Conflict-Driven Clause Learning L'algorithme est basé sur une méthode de backtracking. Il procède en choisissant un littéral, lui affecte une valeur de vérité, simplifie la formule en conséquence, puis vérifie récursivement si la formule simplifiée est satisfaisable. Si c'est le cas, la formule originale l'est aussi, dans le cas contraire, la même vérification est faite en affectant la valeur de vérité contraire au littéral. 7

Flow Shop Scheduling Problem est un problème d'optimisation défini par un ensemble de n tâches et m machines. toutes les tâches doivent passer sur toutes les machines, de la machine 1 à la machine m et une machine ne peut traiter qu'une tâche à la fois. 11, 20

forme normale conjonctive est une normalisation d'une expression logique qui est une conjonction de clauses, autrement dit une conjonction de disjonction de littéraux. Les formules en CNF sont utilisées dans le cadre démonstration automatique de théorèmes ou encore dans la résolution du problème SAT. 10, 13–15, 20

Job Shop Scheduling Problem est un problème d'optimisation où chaque tâche idéale est assigné à des ressources à un temps t particulier. 9, 10, 20

NP La classe NP est une classe très importante de la théorie de la complexité. Un problème de décision est dans NP s'il peut être décidé sur une machine de Turing non-déterministe en temps polynomial par rapport à la taille de l'entrée. 7, 9, 10, 13, 15

Planning Domain Definition Language Encodage standard pour les problèmes de planification "classique". 10

Planning Problem est un problème d'optimisation où la tâche consiste à trouver une séquence d'actions qui réalisera un objectif. 10, 20

SAT signifie **SAT**isfiability problem ou encore problème de satisfaisabilité booléenne. 6, 7, 9, 10, 13, 15, 20

UNSAT Résultat possible d'un problème en CNF. Cela signifie que le problème ne possède aucune solution. 6, 7, 19

Références

- A Reduced Codification for the Logical Representation of Job Shop Scheduling Problems.
- Planification SAT : Amélioration des codages, automatisation de la traduction et étude comparative
- GUNSAT: a Greedy Local Search Algorithm for Unsatisfiability
- Solving hard UNSAT Problems via Local Search Reasoning
- Problème SAT progrès et défis
- <http://www.satlive.org/>
- Flow Shop Scheduling Problem
- Solveur: ZChaff
- Solveur: WalkSAT
- Solveur: Glucose

Étude et résolution du problème SAT. Utilisation de réduction polynomiales sur différents problèmes NP-difficiles

Rapport de PFE 2015

Résumé : Rapport du projet de fin d'études de Valentin Montmirail, encadré par Mr Christophe Lenté Ragot, durant l'année 2014-2015 et portant sur le sujet "Etude et résolution du problème SAT".

Mots clé : SAT, UNSAT, preuve par résolution, JSSP, Flow Shop, Planification, SATyr, Glucose, WalkSAT, FlipGA, CDLS, zChaff, Jenkins, Git

Abstract : Report of the project of the end of studies of Valentin Montmirail, supervised by Mr Christophe Lenté, during year 2014-2015 and concerning the subject "Study and Resolution of the SATisfiability Problem".

Keywords : SAT, UNSAT, proof resolution, JSSP, Flow Shop, Planning, SATyr, Glucose, WalkSAT, FlipGA, CDLS, zChaff, Jenkins, Git

Auteur(s)

Valentin Montmirail

[valentin.montmirail@gmail.com]

Encadrant(s)

Christophe Lenté

[christophe.lente@univ-tours.fr]

Ville: Tours