UNIVERSITÉ D'ARTOIS

Résolution du jeu Eternity 2 avec les technologies SAT

Rapport de TER

Master 1 (Spécialité informatique) Université d'Artois

Réalisé par

Joffrey **CUVILLIER**
et
Rémi **SZYMKOWIAK**

Responsables de stage : Jean-Marie LAGNIEZ
Stéphane CARDON

UNIVERSITÉ D' ARTOIS : CENTRE DE RECHERCHE EN INFORMATIQUE DE LENS
Faculté Jean Perrin - Rue Jean Souvraz - SP 18 - 62307 Lens Cedex
Tél : +33 (0)3 21 79 32 74 – Fax : +33 (0)3 21 79 32 72

Remerciements

Nous tenons d'abord à remercier Éric GRÉGOIRE, Professeur d'Informatique à l'Université d'Artois, pour nous avoir accueilli au sein du CRIL.
Nous remercions également nos encadrants :

- Jean-Marie LAGNIEZ, Doctorant à la faculté des Sciences Jean-Perrin à Lens
- Stéphane CARDON, Maître de Conférence à la faculté des Sciences Jean-Perrin à Lens
- Daniel LE BERRE, Maître de Conférence à la faculté des Sciences Jean-Perrin à Lens

pour leur soutien et leurs conseils avisés, et cela malgré leurs emplois du temps chargés.

Enfin, Nous remercions le CRIL pour son accueil.

Table des matières

Table des figures

Liste des tableaux

Introduction générale

Eternity II est un jeu mi-puzzle, mi-logique. Il repose sur des règles simples : poser toutes les pièces sur son plateau en faisant correspondre les couleurs des pièces côte à côte. Cependant, sa complexité est telle qu'il existe une instance, proposée le 28 Juillet 2007 et composée de 256 pièces et de 21 couleurs différentes, qui n'a toujours pas été résolue à l'heure actuelle. Dans ce rapport, nous présenterons un codage du jeu Eternity II vers SAT.

Le travail consista à réaliser une application permettant de :

- Jouer au jeu Eternity à l'aide d'une application graphique (effectuée en Java) ;
- Définir un codage vers une logique pour la résolution du jeu Eternity ;
- Générer des instances (des parties) Eternity de manière aléatoire et définir de manière empirique un seuil de difficulté (avec combien de variables et de couleurs le problème ne possède que peu de solutions) pour de telles instances ;
- Construire des ponts entre l'application graphique et le codage proposé.

Ce rapport comporte quatre chapitres. Le premier a pour but d'introduire le cadre formel du problème SAT, c'est-à-dire la logique propositionnelle et de donner un bref état de l'art des méthodes de résolution les plus utilisées pour sa résolution. Le deuxième présente le jeu Eternity II et définit le codage utilisé afin de transcrire les contraintes du jeu sous forme CNF. Le troisième traite de nos travaux. Enfin, dans le quatrième chapitre, nous utilisons différents solveurs pour fournir un ensemble d'expérimentations montrant les différentes performances de ces derniers sur le problème Eternity II.

Chapitre 1

Logique propositionnelle et SAT

Sommaire

Ce chapitre est consacré à la logique propositionnelle booléenne et à la présentation du problème SAT.

La première section décrit brièvement le cadre général de la logique propositionnelle en introduisant les différentes notions (syntaxique et sémantique) du problème. Enfin, dans la dernière partie nous présentons le problème SAT et différentes méthodes de résolution.

1.1 Logique propositionnelle

En logique mathématique, le calcul des propositions est la première étape dans la définition de la logique et du raisonnement. Il définit les règles qui relient les propositions entre elles, sans examiner le contenu. Il est ainsi une première étape dans la construction du calcul des prédicats qui lui, s'intéresse au contenu des propositions et qui est un formalisme achevé du raisonnement mathématique.

Dans cette section nous décrivons les aspects syntaxique et sémantique de la logique propositionnelle.

1.1.1 Syntaxe

Définition 1.1 (atome)
Une proposition atomique (ou atome) est une variable booléenne, c'est-à-dire une variable qui prend ses valeurs dans l'ensemble $\mathbb{B}$ ou {FAUX,VRAI} ou {F,V} ou encore {0,1}.

Définition 1.2 (formule)
Soit un alphabet constitué d'un ensemble fini d'atomes et de deux symboles particuliers $\top$, $\bot$.
Soient les connecteurs :

- *de négation : $\neg$ (non ...) ;*
- *de conjonction : $\wedge$ (... et ...) ;*
- *de disjonction : $\vee$ (... ou ...) ;*
- *d'implication : $\Rightarrow$ (si ... alors ...) ;*
- *d'équivalence : $\Leftrightarrow$ (... si et seulement si ...).*

et les opérateurs de parenthésage : «(» et «)».

Une formule propositionnelle $\mathcal{F}$ se construit récursivement en appliquant un nombre fini de fois les règles suivantes :

1. *un atome, $\top$ et $\bot$ sont des formules ;*
2. *si $\mathcal{F}$ est une formule alors $(\mathcal{F})$ est une formule ;*
3. *si $\mathcal{F}$ est une formule alors $\neg\mathcal{F}$ est une formule ;*
4. *si $\mathcal{F}$ et $\mathcal{F}'$ sont des formules alors :*
   - *$\mathcal{F} \wedge \mathcal{F}'$ est une formule ;*
   - *$\mathcal{F} \vee \mathcal{F}'$ est une formule ;*
   - *$\mathcal{F} \Rightarrow \mathcal{F}'$ est une formule ;*
   - *$\mathcal{F} \Leftrightarrow \mathcal{F}'$ est une formule.*

Définition 1.3 (littéral)
On désigne par littéral un atome l ou sa négation $\neg l$ où l est appelé littéral positif et $\neg l$ littéral négatif. On dit que l et $\neg l$ sont complémentaires et on note $\sim \lambda$ le complémentaire du littéral λ.

1.1.2 Sémantique

Définition 1.4 (interprétation)
Une interprétation $\mathcal{I}$ en calcul propositionnel est une application de l'ensemble des formules propositionnelles dans l'ensemble des valeurs de vérité $\{V, F\}$.
L'interprétation $\mathcal{I}(\mathcal{F})$ d'une formule $\mathcal{F}$ est définie par la valeur de vérité donnée à chacun des atomes de $\mathcal{F}$. On calcule $\mathcal{I}(\mathcal{F})$ grâce à l'interprétation des règles suivantes :

1. *$\mathcal{I}(\top) = V$;*
2. *$\mathcal{I}(\bot) = F$;*
3. *$\mathcal{I}(\neg\mathcal{F}) = V$ si et seulement si $\mathcal{I}(\mathcal{F}) = F$;*
4. *$\mathcal{I}(\mathcal{F} \wedge \mathcal{G}) = V$ si et seulement si $\mathcal{I}(\mathcal{F}) = \mathcal{I}(\mathcal{G}) = V$;*
5. *$\mathcal{I}(\mathcal{F} \vee \mathcal{G}) = F$ si et seulement si $\mathcal{I}(\mathcal{F}) = \mathcal{I}(\mathcal{G}) = F$;*
6. *$\mathcal{I}(\mathcal{F} \Rightarrow \mathcal{G}) = F$ si et seulement si $\mathcal{I}(\mathcal{F}) = V$ et $\mathcal{I}(\mathcal{G}) = F$;*
7. *$\mathcal{I}(\mathcal{F} \Leftrightarrow \mathcal{G}) = V$ si et seulement si $\mathcal{I}(\mathcal{F}) = \mathcal{I}(\mathcal{G})$.*

Définition 1.5 (Interprétation partielle, incomplète et complète)
Soit $\mathcal{I}$ une interprétation complète (vue comme un ensemble de littéraux) d'une formule Σ.
On dit que :
- *$\mathcal{I}'$ est une interprétation partielle de Σ si et seulement si $\mathcal{I}' \subseteq \mathcal{I}$;*
- *$\mathcal{I}'$ est une interprétation incomplète de Σ si et seulement si $\mathcal{I}' \subset \mathcal{I}$.*

1.1.3 Consistance, inconsistance, validité et invalidité d'une formule

Définition 1.6 (formule satisfaite)
On dit qu'une formule propositionnelle $\mathcal{F}$ est satisfaite par une interprétation $\mathcal{I}$ si et seulement si $\mathcal{I}(\mathcal{F}) = V$.

Définition 1.7 (formule insatisfaite)
Dualement, on dit qu'une formule propositionnelle $\mathcal{F}$ est falsifiée (ou contredite) par une interprétation $\mathcal{I}$ si et seulement si $\mathcal{I}(\mathcal{F}) = F$.

Définition 1.8 (modèle)
On appelle modèle d'une formule $\mathcal{F}$, une interprétation $\mathcal{I}$ telle que $\mathcal{I}$ satisfait $\mathcal{F}$.

Définition 1.9 (contre-modèle)
On appelle contre-modèle d'une formule $\mathcal{F}$, une interprétation $\mathcal{I}$ telle que $\mathcal{I}$ falsifie $\mathcal{F}$.

Définition 1.10 (consistance, inconsistance)
Une formule est dite consistante si et seulement si elle admet au moins un modèle. Une formule est dite inconsistante si et seulement si elle n'admet pas de modèle, c'est-à-dire :

$$\forall \mathcal{I} \in \mathcal{S}_{\mathcal{I}}(\mathcal{F}), \mathcal{I}(\mathcal{F}) = F.$$

1.1.4 Clauses, monômes et formes normales

Définition 1.11 (clause)
On appelle clause une formule constituée d'une disjonction finie de littéraux.

Définition 1.12 (monôme ou cube)
Dualement, on appelle monôme une formule constituée d'une conjonction finie de littéraux.

Définition 1.13 (clause et monôme fondamentaux)
On appelle clause fondamentale une clause qui ne contient pas de littéraux complémentaires.
On appelle monôme fondamental un monôme qui ne contient pas de littéraux complémentaires.

Définition 1.14 (longueur d'une clause ou d'un monôme)
On appelle longueur de la clause c (respectivement du monôme m), le nombre de littéraux différents contenus dans c (respectivement m). Cette longueur sera notée l(c) (respectivement l(m)).

Définition 1.15 (CNF)
Une formule est sous forme normale conjonctive (CNF) si et seulement si $\mathcal{F}$ est une conjonction finie de clauses, c'est-à-dire une conjonction de disjonction finie de littéraux.

Exemple 1.1 (forme CNF)
$(a \vee b) \wedge (\neg a \vee c) \wedge (\neg a \vee \neg b)$ est une CNF.

1.2 Présentation du problème SAT

Le problème de satisfaisabilité (SAT) est un problème de décision visant à déterminer s'il existe une valuation sur un ensemble de variables propositionnelles telle qu'une formule propositionnelle donnée soit logiquement vraie. Ce problème est très important en théorie de la complexité et possède de nombreuses applications en planification classique (STRIPS), model checking, diagnostic, etc. Le problème SAT fut le premier à être classé NP-Complet en 1971 par Stéphane Cook, permettant ainsi par réduction polynomiale de classer beaucoup d'autres problèmes. Dans la pratique, on s'intéresse au problème SAT pour des formules données en CNF, ce qui est justifié par le fait que de nombreux problèmes se modélisent ainsi. Dans le cas où le problème ne peut se modéliser sous forme CNF, des transformations sont alors possibles.
La partie qui suit a pour objectif de présenter le problème SAT. Pour se faire, nous commençons par le définir de façon formelle. Une fois cette définition établie, nous présentons deux types d'approches par la résolution d'un tel problème.

1.2.1 Définition du problème SAT

1.2.1.1 Définition

Soit une formule logique sous forme normale conjonctive (CNF), posons $\mathcal{F}$ une telle formule.
On dit que $\mathcal{F}$ est satisfaisable si et seulement s'il est possible d'associer une valeur logique (booléenne) à chaque variable de $\mathcal{F}$ de telle manière que $\mathcal{F}$ soit logiquement vraie, ce qui revient à trouver un modèle $\mathcal{I}$, tel que $\mathcal{I}(\mathcal{F}) = vrai$.

Définition 1.16 (SAT)

Soit $\mathcal{S}$ un ensemble fini de symboles propositionnels et $\mathcal{F}$ un ensemble fini de clauses construites sur $\mathcal{S}$.
La question de trouver une interprétation de $\mathcal{S}$ qui satisfasse l'ensemble des clauses de $\mathcal{F}$ est qualifiée de problème SAT.

Exemple 1.2
Soit $\{a, b, c, d\}$ un ensemble de variables, et soit $\mathcal{F}$ une formule logique définie par :

$$\mathcal{F} = (a \vee b \vee \neg d \vee c) \wedge (b \vee d) \wedge (\neg a \vee \neg d \neg c)$$

alors, le problème de trouver un modèle à $\mathcal{F}$ est une instance du problème SAT.

Exemple 1.3 (satisfaisabilité d'une formule)
Soit l'ensemble de variables $\{a, b, c\}$ et $\mathcal{F} = (a \vee b) \wedge (\neg a \vee c) \wedge (\neg a \vee \neg b)$.
$\mathcal{F}$ est satisfaisable, puisqu'en posant $a = vrai$, $b = faux$, $c = vrai$ on a $\mathcal{I}(\mathcal{F}) = vrai$.

En revanche, $\mathcal{F}' = (a \vee b) \wedge (\neg a \vee c) \wedge (\neg b \vee a) \wedge (\neg b \vee c) \wedge (\neg a \vee \neg b)$ n'est pas satisfaisable !
Pour montrer que $\mathcal{F}'$ n'est pas satisfaisable, il suffit de parcourir la table de vérité de $\mathcal{F}'$:

a	b	c	$a \vee b$	$\neg a \vee c$	$\neg b \vee a$	$\neg b \vee c$	$\neg a \vee \neg c$	$\mathcal{F}'$
0	0	0	0	1	1	1	1	0
0	0	1	0	1	1	1	1	0
0	1	0	1	1	0	0	1	0
0	1	1	1	1	0	1	1	0
1	0	0	1	0	1	1	1	0
1	0	1	1	1	1	1	0	0
1	1	0	1	0	1	0	1	0
1	1	1	1	1	1	1	0	0

TABLE 1.1 – Table de vérité de $\mathcal{F}'$

1.2.2 Solveurs

Dans cette section, nous présentons rapidement deux méthodes de résolution pratique dans SAT. Ces méthodes sont divisées en deux classes : dans la première, on retrouve les algorithmes incomplets de type recherche locale et dans la seconde, les algorithmes complets de type DPLL.

1.2.2.1 Algorithme de recherche locale

La méthode recherche locale est une méthode incomplète ne parcourant pas complétement l'arbre de recherche et n'étant pas capable de répondre à l'insatisfaisabilité.
Le schéma de recherche locale est relativement simple. Celui-ci consiste à se déplacer de manière stochastique sur l'ensemble des interprétations complètes de la formule à résoudre. À chaque étape (ou flip), on essaie de réduire le nombre de clauses falsifiées (phase de descente). La prochaine interprétation est choisie parmi l'ensemble des interprétations voisines (c'est-à-dire différant uniquement sur la valeur d'une seule variable) de l'interprétation courante. Lorsqu'aucune interprétation voisine ne permet de diminuer le nombre de clauses insatisfaites, on se trouve dans un minimum local. L'un des points essentiels des méthodes de recherche locale est la technique utilisée pour sortir des minimums locaux.
Dans le cadre de ce TER, nous avons utilisé plusieurs méthodes de recherche locale pour les besoin de nos expérimentations. Ces méthodes sont :

- Adaptnovelty [7] ;
- Adaptg2wsat [9] ;

- Rsaps [8] ;
- Dcrwalk [14] ;
- Irots [13].

Le travail effectué pendant ce TER ne concernant pas la recherche locale, nous ne développerons pas plus en détail ces différents solveurs.

1.2.2.2 Algorithme complet (DPLL)

Les algorithmes complets, tels que DPLL, répondent à la fois à la satisfaisabilité et à l'insatisfaisabilité. On le dit "complet" car si on laisse un temps et un espace illimités, ils sont capables de trouver une solution ou montrer qu'il n'en n'existe pas.
L'approche DPLL est une approche systématique basée sur un arbre permettant ainsi de parcourir l'ensemble des interprétations. Dans le cadre de ce TER nous avons utilisé le solveur Minisat [4].

Chapitre 2

Eternity 2

Sommaire

Nous allons maintenant vous présenter le jeu Eternity II, son origine et son fonctionnement. Nous développerons ensuite les points importants qui font toute la complexité de sa résolution. Nous verrons grâce à cela comment transcrire les contraintes en un format CNF afin de pouvoir le résoudre à l'aide d'un solveur SAT ultérieurement.

2.1 Présentation

Afin de bien comprendre notre TER, nous allons à présent vous expliquer le fonctionnement du jeu Eternity II. Nous commencerons pour cela, par un bref historique avant de présenter le jeu en lui-même.

2.1.1 Historique

Eternity II a été lancé le 28 juillet 2007, dans vingt pays, avec la promesse de deux millions de dollars pour le premier qui le résoudra. Il a été inventé pas Christopher Mockton et commercialisé par Tomy depuis le 28 juillet 2007. Il est constitué de 256 tuiles carrées de même taille dont les arêtes sont colorées. Le but est de disposer toutes les tuiles sur une grille 16×16 de manière à ce que les arêtes communes des tuiles adjacentes aient la même couleur.

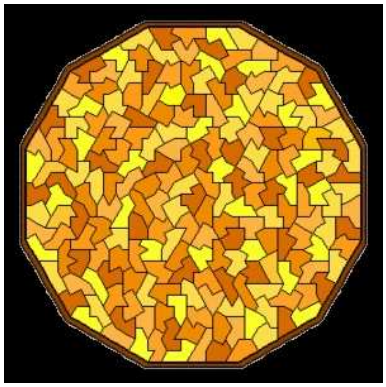


FIGURE 2.1 – solution pour Eternity 1

Ceci n'est pas le premier puzzle du genre, puisqu'il existe une première version d' Eternity sortie en 2000 en Angleterre. Ce puzzle est composé 205 pièces devant former un dodécagone (pour la solution, voir figure 2.1). Comme on peut le voir, le style de puzzle est particulier, mêlant jeu de réflexion et puzzle ludique. Derrière cela, se cache un véritable problème algorithmique qui mérite et demande une étude approfondie.

2.1.2 Présentation du jeu

FIGURE 2.2 – plateau d'Eternity 2

Eternity II est un jeu composé :
- d'un plateau de jeu carré représentant une grille de 16 par 16 (voir figure 2.2) ;
- de 256 pièces carrées toutes différentes ayant chacune un motif sur chaque bord (donc quatre motifs par carré). Étant carrée, il n'y a aucune contrainte pour le sens de placement de la pièce, ce qui donne la possibilité de la disposer de quatre manières différentes :

FIGURE 2.3 – Pièce

Le but est de placer les 256 pièces sur le plateau de manière à ce que chaque motif coïncide avec celui d'à coté :

FIGURE 2.4 – Pièces bien placées

Nous pouvons aussi observer un motif spécifique pour la bordure (gris uni dans les illustrations), ainsi nous pouvons classer les pièces en trois catégories :

1. les coins (figure 2.5)
2. les bords (figure 2.6)
3. les pièces centrales (figure 2.7)

FIGURE 2.5 – Coin

FIGURE 2.6 – Bord

FIGURE 2.7 – Centre

Ainsi, un exemple de solution sur un tableau de quatre par quatre :

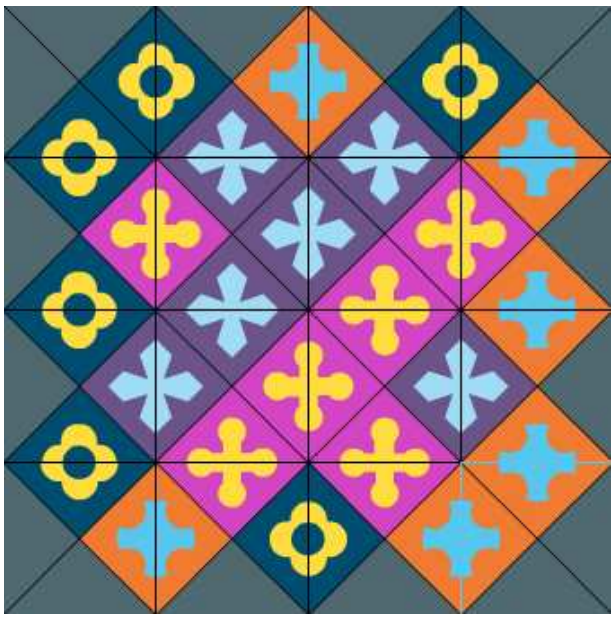


FIGURE 2.8 – jeu rempli

2.2 Codage

Eternity est un problème hautement combinatoire. Par conséquent, plusieurs approches de programmation par contraintes ont déjà été proposées pour résoudre ce problème : par exemple, Schauss et Deville ont proposé une approche incomplète qui utilise une recherche locale stochastique utilisant un voisinage de très grande taille (VLNS)[12], tandis qu'un filtrage spécial à la complexité constante est présenté par Thierry Benoist [2] pour résoudre Eternity. Toutefois, ce problème n'a jamais été abordé en utilisant la technologie SAT. Dans ce document, nous proposons une première codification des puzzles Eternity avec une forme normale conjonctive (CNF), et nous présentons les premières expériences en utilisant des techniques SAT pour tenter de résoudre ce problème.

2.2.1 Définition générique

Eternity est un puzzle composé de n^2 tuiles, chacune ayant ses côtés (top, right, left et bottom) de couleur. Le but de ce problème est de placer les tuiles dans une grille de $n \times n$ bords, de telle sorte que chaque côté de chaque tuile ait la même couleur que le côté adjacent. En outre, une couleur unique, la couleur de bord, est connue pour les tuiles placées dans les angles et les côtés. Ce puzzle a été récemment prouvé problème NP-complet [3].

2.2.1.1 Le challenge

Concentrons-nous maintenant sur l'instance valant 2M$. En effet, celle-ci présente certaines particularités qui doivent être soulignées. Premièrement, c'est un puzzle de 16×16 tuiles, avec chacun des $16 \times 16 \times 4 = 1024$

triangles qui peuvent être colorés de 22 manières. En outre, cinq couleurs n'apparaissent que sur les tuiles de côtés, ce qui fait que le nombre de couleurs possibles pour les tuiles de centre est de 17. Ces couleurs sont presque uniformément réparties : en effet, elles apparaissent 48 ou 50 fois, ce qui contribue à la difficulté de cette instance du puzzle.

Il faut noter également qu'une tuile est placée par le créateur au centre de la grille : elle peut être considérée comme un indice, mais elle empêche aussi certaines symétries du problème. De la même manière, toutes les tuiles ont été conçues pour être différentes, afin de ne pas présenter de solutions symétriques où plusieurs tuiles identiques sont permutées. Le nombre de configurations possibles au sein de l'instance est de $256! \times 4^{256}$, qui est un plus grand nombre que 10^{661}. Si on prend en compte la tuile fixée, on obtient un plus petit nombre de configurations possibles : $4! \times 56! \times 195! \times 4^{195}$, ce qui est encore plus grand que 10^{557}. L'espace de recherche ultérieure est donc absolument énorme : par exemple, le nombre d'atomes dans l'univers est estimé à 10^{80}. Il est affirmé que ce problème présente au moins 20 000 solutions, mais aucune d'entre elles n'a encore été découverte. Dans la section suivante, un encodage CNF est proposé.

2.2.2 Génération CNF d'une instance d'Eternity

Afin de résoudre des instances d'Eternity II, nous allons transcrire les contraintes du jeu en un format CNF. Dans cette section, nous définissons le codage utilisé dans notre application afin de fournir un fichier CNF valide à un solveur SAT.

2.2.2.1 Format du fichier

Pour des raisons de commodité, le format du texte proposé par Thierry Benoist, sur son site web consacré à Eternity, a été adopté. Le format utilisé est présenté dans la figure 2.9, où `size_board` est le nombre de lignes/colonnes du puzzle, `num_colors` est le nombre de couleurs différentes présentes dans le puzzle, et `num_hints` est le nombre de tuiles dont le placement et la position sont donnés.

size_board
num_colors
num_hints
row column id rotation
⋮ } num_hints
row column id rotation
top_color right_color bottom_color left_color
⋮ } $size\_board^2$
top_color right_color bottom_color left_color

FIGURE 2.9 – Format des instances du puzzle

En conséquence, la prochaine `num_hints` ligne représente la position, pour chaque indice, en termes de `row` et `column` (avec $1 \leq$ `row,column` $\leq$ `size_board`), l'`id` de la tuile et de sa `rotation` (voir la section 2.2.2.3 pour des valeurs entières correspondant aux rotations).

Enfin, `size_board`^{2} représente les différents carreaux de la grille, en utilisant des nombres entiers pour représenter les différentes couleurs. Notez que la partie haute est d'abord donnée, puis celle de droite, du bas et enfin de gauche. Bien évidemment, cette liste peut être permutée de manière cyclique.

Notre objectif est donc de produire un équivalent CNF qui code le problème d'Eternity. À cette fin, il faut considérer différents types de contraintes, qui sont :

1. **Contraintes de placement :** Ces contraintes expriment le fait que chaque tuile est placée dans *exactement une* case de la grille et que chaque case contient une tuile différente.
2. **Contraintes d'orientations :** Ces contraintes expriment le fait que chaque tuile est placée dans *exactement une* orientation.

3. **Contraintes sur le plateau :** Ces contraintes symbolisent que chaque triangle du plateau ne peut être que d'une couleur.
4. **Contraintes de jeu :** Ces contraintes indiquent que chaque pièce placé d'une certaine facon entraîne un ensemble de contraintes sur les triangles du plateau.

Ces contraintes sont présentées avec plus de précisions dans les sections suivantes.

2.2.2.2 Contraintes de placement

Tout d'abord, il est préférable d'identifier les cases de la grille dans une seule dimension. En effet, la grille est représentée comme une matrice $size\_board \times size\_board$ dans le format de fichier, les différentes cases étant données par le couple (r, c) (avec $1 \leq r, c \leq size\_board$). Dans notre codage, chaque case peut être identifiée par un entier num_cell donné par :

$$num\_cell = (r-1) * size\_board + c$$

Dans la suite, nous noterons p_{ij} une variable propositionnelle qui est $true$ si, et seulement si, la tuile i est placée dans la case j, $false$ sinon. En conséquence, un ensemble $\mathcal{PV}$ de $size\_board^4$ variables propositionnelles est introduit.

Il est maintenant possible de définir les contraintes de placement des pièces :

1. La première formule assure que toute case contient *au moins* une pièce :

$$\bigwedge_{i=1}^{size\_board^2} \left(\bigvee_{j=1}^{size\_board^2} p_{ji} \right)$$

2. De la même façon, une case doit contenir *au plus* une pièce dans toutes les solutions possibles, qui est codé comme suit :

$$\bigwedge_{i=1}^{size\_board^2} \bigwedge_{j=1}^{size\_board^2-1} \bigwedge_{k=j+1}^{size\_board^2} (\neg p_{ji} \vee \neg p_{ki})$$

   Notons que ces deux premières formules de clauses sont un moyen direct d'imposer que chaque case contient *exactement* une pièce. Ceci peut être facilement remplacé par un des nombreux encodages de contraintes de cardinalité qui ont été proposées ces dernières années [6, 5, 10, 1].

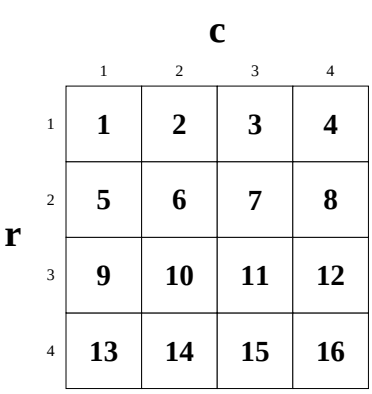
3. Enfin, toutes les cases de la grille doivent contenir des pièces différentes. Dans le contexte CSP, nous pourrions facilement lui indiquer des contraintes *AllDifferent* [11]. Malheureusement, dans le cadre clausale propositionnel, il est nécessaire d'exprimer toutes les exclusions mutuelles. Ceci est codé par :

$$\bigwedge_{i=1}^{size\_board^2-1} \bigwedge_{j=i+1}^{size\_board^2} \bigwedge_{k=1}^{size\_board^2} (\neg p_{ki} \vee \neg p_{kj})$$

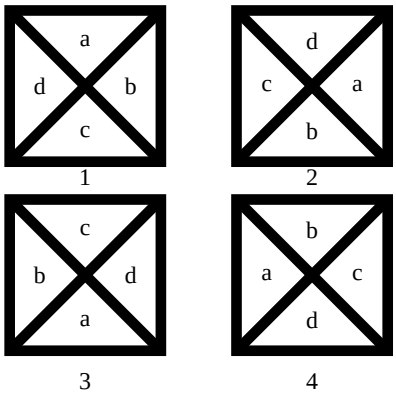
2.2.2.3 Contraintes d'orientations

Une source de la structure hautement combinatoire de ce problème est la possibilité que chaque pièce peut être placée dans n'importe quelle position ***et*** dans tous les sens. Par conséquent, il faut considérer les quatre rotations possibles d'une pièce lorsqu'elle est placée sur la grille. Dans le format texte d'une instance d'Eternity, les pièces sont données par l'ensemble (a, b, c, d), représentant une pièce avec ces quatre couleurs fixées dans le sens horaire.

En conséquence, il existe quatre rotations possibles pour une pièce : nous avons identifié les quatre rotations dans la figure 2.10b. Nous définissons la variable propositionnelle o_{ij} $true$ si, et seulement si, la pièce i est orientée dans le sens j, $false$ sinon. Donc, un ensemble $\mathcal{OV}$ de $4 \times size\_board^2$ variables est introduit. De la même manière que le placement de pièces, il faut s'assurer que chaque pièce est orientée dans *exactement un* sens, en utilisant les contraintes de cardinalité suivantes :



(a) Id des cases pour un problème 4×4



(b) Les 4 orientations possibles d'une pièce

FIGURE 2.10 – choix de représentation pour les orientations des pièces et identification des cases

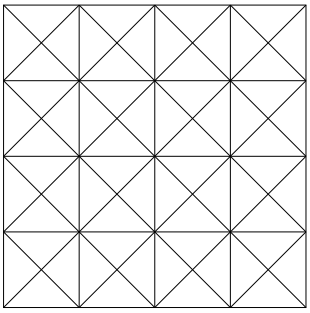


FIGURE 2.11 – Décomposition du plateau en triangles

1. contrainte au-plus-une

$$\bigwedge_{i=1}^{size\_board^2} (o_{i1} \vee o_{i2} \vee o_{i3} \vee o_{i4})$$

2. contrainte au-moins-une

$$\bigwedge_{i=1}^{size\_board^2} \bigwedge_{j=1}^{3} \bigwedge_{k=j+1}^{4} (\neg o_{ij} \vee \neg o_{ik})$$

2.2.2.4 Contraintes sur le plateau

Pour établir les contraintes du jeu nous avons besoin de décomposer le plateau en différents triangles. Chaque case du plateau est découpée en quatre triangles. Ces triangles entrent en correspondance avec les triangles des pièces posées à cet emplacement. La figure 2.11 donne un exemple de la décomposition en triangles d'un plateau 4×4.

Pour travailler avec cette décomposition, il nous a fallu établir une numérotation. Celle-ci consiste à numéroter les triangles ligne par ligne. La figure 2.12 donne la numérotation des triangles d'un plateau 4×4.

Une fois cette numérotation effectuée, posons t_{ij} la variable propositionnelle représentant le fait que le triangle i est de couleur j. Il est à présent possible de définir les contraintes concernant les différents triangles et leurs couleurs. Ses contraintes doivent représenter le fait que chaque triangle possède une unique couleur :

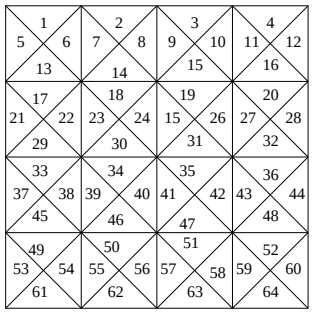


FIGURE 2.12 – Numérotation des triangles

$$\forall i \in [1 \dots 4 * nb\_part], ((\bigvee_{j=1}^{num\_colors} t_{ij}) \wedge (\bigwedge_{j=1}^{num\_colors-1} \bigwedge_{k=j+1}^{num\_colors} \neg t_{ij} \vee \neg t_{ik}))$$

2.2.2.5 Contraintes de jeu

Ensuite, nous définissons les contraintes de jeu. ces contraintes sont de deux types :

– contraintes sur le plateau, c'est-à-dire le fait que les triangles de tour soient de couleur noire et que les triangles contigus aient la même couleur ;
– contraintes fournies par les pièces du jeu, lorsqu'une pièce est placée sur le plateau celle-ci entraîne que les triangles contenues dans cette case aient une certaine couleur.

Tout d'abord, supposons que la couleur du tour ait pour indice 1 (exemple t_{i1} représente la variable propositionnelle codant le fait que le triangle i est de la couleur du tour). Les contraintes sur le plateau sont de deux types, contraintes sur le tour et sur les triangles contigus.

Avant de définir l'ensemble des contraintes il est nécessaire, pour des raisons de lisibilité, de définir certaines fonctions permettant de récupérer plus facilement les cases du tour :

$$\begin{array}{lrcl}
\text{- } \alpha : & [1..size_board] & \rightarrow & [1..4 * size_board^2] \\
& i & \mapsto & (4 * (i-1)) + 1 \\
\text{- } \beta : & [1..size_board] & \rightarrow & [1..4 * size_board^2] \\
& i & \mapsto & (4 * size_board) * (i-1) + 4 \\
\text{- } \chi : & [1..size_board] & \rightarrow & [1..4 * size_board^2] \\
& i & \mapsto & (4 * size_board) * (i-1) - 2 \\
\text{- } \delta : & [1..size_board] & \rightarrow & [1..4 * size_board^2] \\
& i & \mapsto & (4 * size_board) + (size_board - 1) + (i-1) * 4 + 3
\end{array}$$

À présent nous allons représenter le fait que le tour est de couleur noir :

$$(\bigwedge_{i=1}^{size\_board} t_{\alpha(i)1} \wedge t_{\beta(i)1} \wedge t_{\chi(i)1} \wedge t_{\delta(i)1})$$

Nous définissons les contraintes sur les triangles intérieurs au tableau. Dans un premier temps, comme pour le tour, définissons une fonction permettant de récupérer l'indice du triangle contigu en fonction d'un triangle du centre :

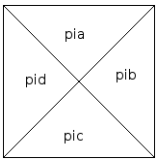


FIGURE 2.13 – La pièce i dans la position 1.

$$\begin{array}{rrl} \text{- } \epsilon : & [1..4 * sz^2] & \rightarrow \quad [1..4 * sz^2] \\ & i & \mapsto \quad \text{si } i \in \{4*(i-1)+1 | i \in [1..sz]\} \text{ alors } i \\ & & \quad \text{sinon si } i \in \{(sz*4)*(i-1)+4 | i \in [1..sz]\} \text{ alors } i \\ & & \quad \text{sinon si } i \in \{(sz*4)*i-2 | i \in [1..sz]\} \text{ alors } i \\ & & \quad \text{sinon si } i \in \{sz*4*(i-1)+4 | i \in [1..sz]\} \text{ alors } i \\ & & \quad \text{sinon si } i \in \{sz*4*(i-1)+4 | i \in [1..sz]\} \text{ alors } i \\ & & \quad \text{sinon si } i\%2 = 0 \text{ alors } \{\text{si } i\%4 \text{ alors } i-6 \text{ sinon } i+6\} \\ & & \quad \text{sinon } \{\text{si } (i+1)\%4 \text{ alors } i-2+(4*sz) \text{ sinon } i+2-(4*sz)\} \end{array}$$

avec $sz = size\_board$.

Les contraintes codant le fait que les triangles contigus sont de même couleurs sont définies par la formule suivante :

$$\left(\bigwedge_{i=1}^{4*size\_board^2} \bigwedge_{k=1}^{num\_color} (\neg t_{ik} \vee t_{\epsilon(i)k}) \wedge (t_{ik} \vee \neg t_{\epsilon(i)k}) \right)$$

Les dernières contraintes représentent le fait qu'une pièce donne ses couleurs aux triangles de la case où elle se trouve. Ses couleurs dépendent de la pièce, de l'orientation et de la case. Notons p_{ia}, p_{ib}, p_{ic} et p_{id} les différentes couleurs de la même manière que sur la figure 2.13 qui est la représentation de la pièce i dans la position 1.

Comme précédemment et pour les mêmes raisons, définissons deux fonctions. La première permet de retourner un indice de couleur en fonction d'une pièce, de son orientation et de la partie que l'on souhaite obtenir :

$$\begin{array}{rrl} \text{- } \eta : & [1..num\_part] \times [1..4]^2 & \rightarrow \quad [1..num\_color] \\ & (l, o, k) & \mapsto \quad \text{plr avec } r =' a' + (k + (4 - o))\%4 \end{array}$$

avec $sz = size\_board$.

La seconde fonction retourne le j^{eme} triangle de la case i :

$$\begin{array}{rrl} \text{- } \gamma : & [1..num\_part] \times [1..4] & \rightarrow \quad [1..size\_board^2] \\ & (i, j) & \mapsto \quad (4*(i-1)) + j \end{array}$$

avec $sz = size\_board$.

Les contraintes données par les pièces sont :

$$\bigwedge_{i=1}^{nbP} \bigwedge_{j=1}^{sz^2} \bigwedge_{m=1}^{4} \bigwedge_{r=1}^{4} ((\neg p_{ij} \vee \neg o_{im} \vee t_{\gamma(i,j)\eta(i,m,r)}))$$

avec $sz = size\_board$ et $num\_part = nbP$.

2.2.2.6 Conclusion

Le codage ainsi défini, nous obtenons un fichier au format CNF représentant parfaitement l'instance d'Eternity II désirée. Cependant, le nombre de variables et de clauses augmente exponentiellement lorsque l'on augmente le nombre de pièces et de couleurs. Pour exemple, une instance de 256 pièces et 17 couleurs possèdent 84 992 variables et 17 846 752 clauses.

Chapitre 3

Developpement

Sommaire

Dans cette partie, nous allons vous décrire en détails le fonctionnement de notre application ainsi que son code. Le codage se trouvant en annexe, nous n'inclurons que les diagrammes de classe dans cette section.

3.1 Analyse

Dans ce chapitre, nous allons expliquer les choix de développement que nous avons pris pour réaliser notre application.

Langage de programmation

Suite à une réunion avec nos tuteurs, nous avons décidé de programmer en $Java$. Ce choix est justifié par le fait que nous devons mettre en place une interface graphique et que nous avions déjà fait des projets avec $JavaSwing$. De plus, nous avions en tête d'utiliser le solveur SAT $SAT4J$ codé par M Le Berre, Maître de Conférence à la faculté des Sciences Jean-Perrin à Lens. Nous nous sommes aussi posé la question du temps, 10 semaines, dont nous disposions. Il nous a paru avantageux d'utiliser un logiciel tel que *Eclipse*, qui est un Environnement de Développement Intégré (IDE) libre et extensible, pour la conception d'un tel projet.

Paradigme de programmation

Pour ce TER, nous avions la contrainte d'utiliser le modèle MVC (Modèle-Vue-Contrôleur).
Nous allons maintenant l'expliquer plus en détails.

3.2 Fonctionnalités

Nous allons à présent lister les fonctionnalités que nous avons implémentées, mais aussi celles que nous n'avons pu faire par manque de temps.

Réalisées

Tout d'abord, nous avons développé une interface graphique permettant de jouer facilement au jeu Eternity II. Celle-ci inclut une grille de jeu (où placer les pièces) et une grille contenant les pièces à placer. Pour le

déplacement, un système de glisser déposer a été mis en place. Une pièce peut soit être placée sur une case libre, soit effectuer une inversion avec une autre pièce. Elle peut également effectuer une rotation suite à un clic droit sur la pièce. À tout moment, le jeu peut être mis en pause puis repris quand bon nous semble.
La fonctionnalité "option couleurs" est utilisée pour changer les couleurs définies par défaut.

Il est possible de générer des parties de la forme carrée, triangle ou losange, possédant un nombre de pièces variant de 4 (ou 5 pour le losange) à 256 (ou 221 pour le losange), et un nombre de couleurs allant de 2 à 21. Il est également possible de charger une instance précise à l'aide d'un fichier.

A chaque instant, il est possible de vérifier si l'état actuel de la partie est capable de nous amener à une solution. Pour cela, il suffit de cliquer sur Résoudre. Nous pouvons également générer totalement la solution. Deux types de résolution ont été mis en place :

- pas à pas : chaque contrainte est traitée en temps réel et l'on affiche l'évolution du solver (Minisat) sur la grille. Pour réaliser cela, il a été nécessaire d'ajouter l'option -pasApas au solver Minisat. Cela entraîne la création d'une socket et une mise en attente d'une connection client. Une fois la connexion établie, la résolution commence, à chaque point de choix un message contenant une variable(positive ou négative) est envoyé. Lors d'un restart ou d'un backtrack, les variables libérées sont envoyées pour être effacées de la grille (envoie de la variable avec une phase négative). Côté client (jeu), un tableau de NBCONTRAINTES booléens est utilisé pour stocker l'état du triangle (s'il est déjà peint ou déjà grisé). Une fois la valeur de la variable récupérée, si celle-ci est négative et est déjà à **false** dans le tableau ou si elle est positive et est déjà à **true** dans le tableau, on ne fait rien, sinon on recherche la pieceIG correspondante à l'indice et on peint le triangle de la bonne couleur (si positive) ou on la grise (si négative) ;
- complète (Bouton Solution) : un solver, choisi parmi une liste proposée, recherche la solution en fonction des contraintes et nous affiche le résultat une fois terminé (s'il existe une solution). Pour ce faire, on génère les contraintes correspondantes à l'instance actuelle, que l'on communique au solver, qui nous retourne la solution trouvée. Une fois le fichier solution généré, on parse celui-ci afin de récupérer les différentes variables de pièces grâce auxquelles on place les pièces à la bonne position et dans la bonne orientation.

Suite à la mise en place de la résolution complète, nous avons constaté que Minisat atteignait rapidement ses limites. En effet, à partir de 36 pièces, la résolution est effective mais prend du temps. Nous avons donc pensé à modifier le solveur afin de restreindre le nombre de variables de choix. Pour cela, nous passons en paramètres un intervalle (par l'option -i=DEBUT-FIN) permettant de ne prendre en compte que les variables concernant les pièces et leur orientation et ainsi gagner en efficacité.

Futures

Nous avons également pensé à d'autres fonctionnalités qui pourraient être développées à l'avenir. L'implémentation du projet a été pensée afin de permettre facilement l'ajout de solveurs.

Du fait de la gestion du chargement d'instances, il serait aussi possible de développer un système de sauvegarde de la partie que l'on pourrait ainsi recharger plus tard.

La mise en place d'un classement en ligne pourrait être effectuée en prenant en compte le temps de résolution des instances.

Enfin, les modèles en place pour le moment sont : carré, triangle et losange. L'ajout d'autre formes est également envisageable.

Il est également envisageable d'ajouter des options (vitesse de résolution du solveur pas à pas...).

3.3 Modèle

Le modèle représente la partie donnée, elle contient le plateau, les pièces ainsi que certaines contraintes, pour la génération du fichier CNF, qui sont fixes quelque soit l'instance courante.

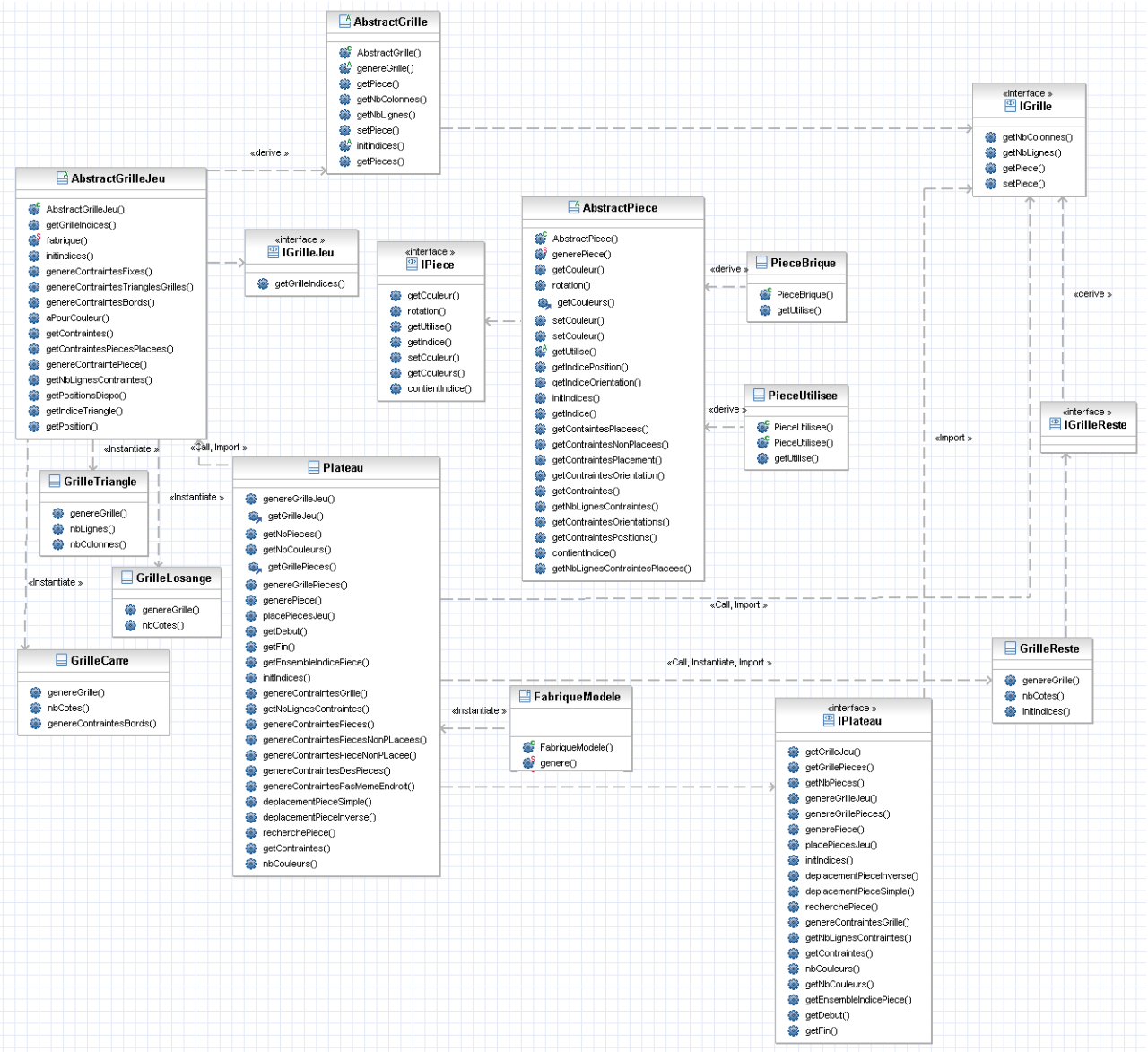


FIGURE 3.1 – partie modèle du MVC

Plateau

La principale fonction du plateau est d'être le lien entre le modèle et le reste de l'application. Il contient deux grilles : l'une représentant le jeu, là où l'on doit résoudre l'instance, et l'autre servant à poser les pièces sur le coté.
Les principales fonctions sont :

- genereGrilleJeu() et genereGrillePieces() permettent de générer les deux grilles du plateau une fois que le nombre de pièces et de couleurs ont été mis à jour ;
- deplacementPieceSimple() et deplacementPieceInverse() gèrent les déplacements des pièces sur le plateau. Les déplacements entre grilles et l'inversion de deux pièces sont ainsi gérés ;

– genereContraintesGrille() calcule et stocke les contraintes fixes du plateau de jeu. Elles ne seront calculées qu'une seule fois de cette façon.

Grilles

La gestion des deux grilles du jeu est menée grâce à l'arborescence ci-dessous : La classe mère $AbstractGrille$ comporte des méthodes générales tels que $getNbLignes()$ et $getNbColonnes()$. Elle contient aussi les références des pièces qu'elles possèdent.
Les deux filles $AbstractGrilleJeu$ et $GrilleReste$ représentent les deux parties du plateau de jeu, c'est-à-dire la grille de jeu et les pièces posées sur le côté. On remarque que $AbstractGrilleJeu$ possède des fonctions de génération de contraintes qui permettent de fournir toutes les contraintes relatives à la grille et aux pièces présentes sur celle-ci.
La dernière branche de l'arborescence correspond aux trois types de grilles disponibles dans notre application. En effet, chaque forme de grille possède des calculs spécifiques.

Pièces

Les pièces sont les éléments centraux du jeu. Elles ne peuvent que tourner sur-elle même mais possèdent leurs propres contraintes. Nous avons donc des fonctions définissant les contraintes de chaque pièce.
La classe $PieceBrique$ est utilisée pour représenter les pièces noires lorsqu'on utilise une grille triangle par exemple.

3.4 Vue

La vue correspond à l'interface avec laquelle l'utilisateur interagit. Sa première tâche est de représenter visuellement les informations contenues dans le modèle. Sa seconde tâche est de recevoir toutes les actions de l'utilisateur (clic de souris, sélection d'une entrée, boutons, etc) et d'envoyer ces évènements au contrôleur.

PieceIG

C'est la représentation graphique d'une $Piece$. Elle contient la pièce qu'elle représente, sa coordonnée dans la grille et une taille. Un système de glisser déposer est associé à ces $PiecesIG$. En effet, chaque $PieceIG$ est capable de recevoir une autre $PieceIG$ (ou de faire l'inversion des deux).

Eternity

Une des principales fonctions d'$Eternity$ est de faire la liaison entre la vue et le reste de l'application. Elle contient :

– les pages de choix permettant de définir type de grilles, nombre de pièces et nombre de couleurs ;
– la page permettant d'aller rechercher dans l'arborescence un fichier à charger ;
– la page des options de couleurs ;
– la page de jeu contenant des piecesGrilleJ, une grille de PieceIG dans laquelle il faut déposer les pièces et piecesGrilleP, une grille de PieceIG qui contient les pièces, un Timer. Les principales fonctions sont afficheResultat(), chargée d'afficher le résultat renvoyé par le solver puis traité par le modèle et changePage(Page page) qui permet de changer les différentes pages du jeu (pages de choix, page de jeu...).

3.5 Contrôleur

Le contrôleur prend en charge la gestion des événements de synchronisation pour mettre à jour la vue ou le modèle et les synchroniser. Il reçoit tous les événements de l'utilisateur et enclenche les actions à effectuer. Si

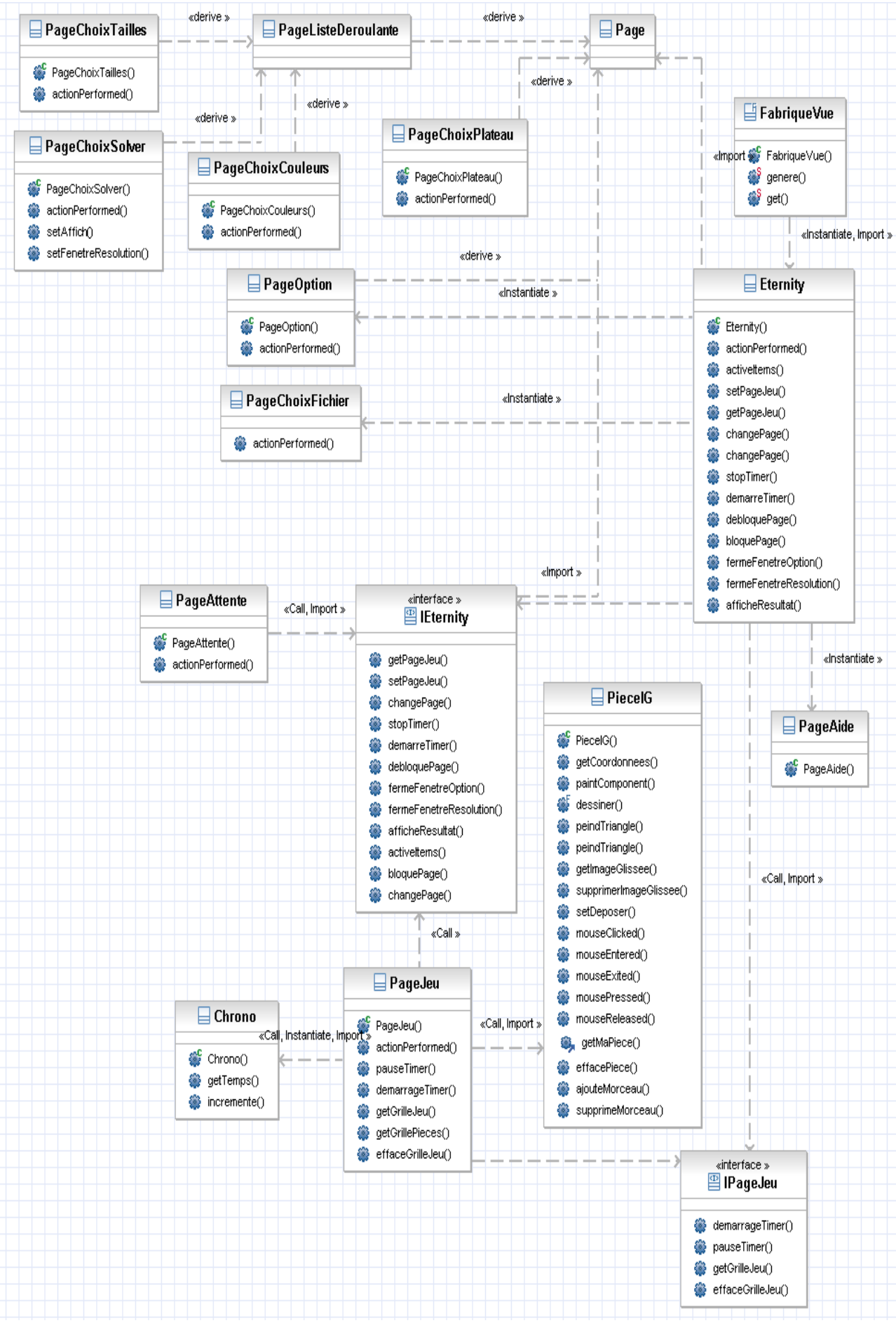


FIGURE 3.2 – partie vue du MVC

une action nécessite un changement des données, le contrôleur demande la modification des données au modèle. Ce dernier avertit la vue que les données ont changés pour qu'elles se mettent à jour. Certains événements de l'utilisateur ne concernent pas les données mais la vue. Dans ce cas, le contrôleur demande à la vue de se modifier. Le contrôleur n'effectue aucun traitement, ne modifie aucune donnée. Il analyse la requête du client et se contente d'appeler le modèle adéquat et de renvoyer la vue correspondant à la demande.

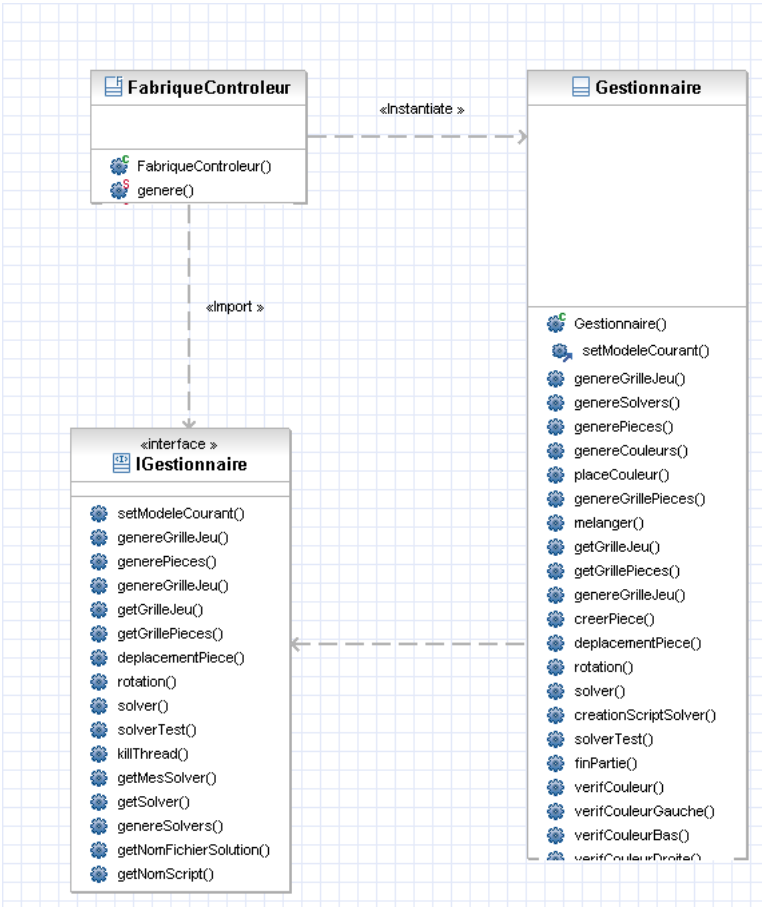


FIGURE 3.3 – partie controleur du MVC

Gestionnaire

Il est chargé de faire la liaison entre la vue et le modèle. C'est lui qui possède les différents solveurs proposés pour la résolution des instances. Les principales fonctions sont :

- genereGrilleJeu(int) et genereGrilleJeu(File) qui permettent de générer les parties soit en fonction du nombre de pièces, soit par fichier ;
- rotation() qui permet de faire pivoter les pièces ;
- solver() qui permet de lancer la résolution de l'instance ;
- deplacementPieces() qui permet de déplacer les pièces d'une case vers une autre (en effectuant une inversion si une pièce est déjà présente sur la case destination.

Chapitre 4

Partie Expérimentation

L'application proposée a la capacité de générer des parties aléatoires selon un nombre de pièces et un nombre de couleurs. Une partie de notre travail a été d'étudier le temps de réponse de différents solveurs SAT afin d'estimer un seuil de difficulté. Dans cette section, nous allons étudier les différents résultats obtenus.

Pour les tableaux, nous aurons le nombre de couleurs (de 2 à 17) et le nombre de pièces (4,9,16,25,36,49,64) comme variables. Chaque case est de forme $tps(nb)$ où nb représente le nombre d'instance résolus dans le temps imparti et tps le temps de résolution totale. La forme (nb) est aussi employée lorsque nb est égal à zéro car il n'y a pas de temps associé. Le nombre d'instances testées, pour chaque couple (nombre couleur, nombre pièce), est de 100. Les résultats expérimentaux reportés dans cette section ont été obtenus sur un quad-core Intel XEON X5550, 2,66GHz, 8Mo cache (32Go DDR3-1333). Le temps CPU est limité à 600 secondes et les résultats sont reportés en secondes.

Minisat

	4	9	16	25	36	49	64
2	.14(100)	.21(100)	.74(100)	3.23(100)	1242.08(100)	4100.53(50)	37.66(7)
3	.14(100)	.22(100)	.77(100)	6.03(100)	263.41(100)	3117.08(85)	1478.43(37)
4	.15(100)	.21(100)	.77(100)	13.45(100)	1068.07(100)	7038.13(90)	4661.40(36)
5		.21(100)	.76(100)	8.05(100)	4084.55(100)	346.31(9)	(0)
6		.20(100)	.76(100)	5.55(100)	8324.54(94)	(0)	(0)
7		.20(100)	.77(100)	4.27(100)	3328.84(100)	(0)	(0)
8		.19(100)	.77(100)	3.68(100)	1019.18(100)	(0)	(0)
9		.20(100)	.78(100)	3.45(100)	344.43(100)	233.56(1)	(0)
10		.18(100)	.78(100)	3.33(100)	198.64(100)	44.07(3)	(0)
11		.21(100)	.77(100)	3.29(100)	104.43(100)	299.49(9)	(0)
12		.20(100)	.79(100)	3.19(100)	108.32(100)	2659.68(28)	(0)
13			.79(100)	3.19(100)	57.13(100)	3169.54(30)	(0)
14			.78(100)	3.17(100)	43.57(99)	8909.11(56)	(0)
15			.80(100)	3.17(100)	41.27(100)	12190.38(70)	(0)
16			.79(100)	3.18(100)	30.79(100)	12973.40(79)	(0)
17			.79(100)	3.17(100)	31.29(100)	10818.77(85)	(0)

TABLE 4.1 – Résultats obtenus par Minisat sur des instances du Jeu Eternity générées de manière aléatoire

On peut voir que pour des instances avec moins de 16 pièces Minisat résoud l'ensemble des instances en moins d'une seconde. Pour les tailles supérieures, on remarque que les performances se dégradent de manière exponentielle (en fonction du nombre de pièces). On observe même un pique de difficulté semblant se situer

aux alentours de $\sqrt{nb\_pieces}$.
Lors du développement, nous avons implémenté une version modifiée de Minisat. Celle-ci accepte en paramètres l'ensemble des variables de choix autorisées. Nous voulions pouvoir indiquer les contraintes des pièces (orientations et positions) afin qu'elles soient prioritaires vis à vis des couleurs des triangles du plateau. Nous avons remarqué une différence notable avec cette version qui décale d'une grandeur le seuil de résolution, arrivant ainsi à résoudre les 49 pièces entièrement.

Adaptnovelty et Adaptwsat

	4	9	16	25	36	49	64
2	0(100)	0(100)	.50(100)	2.67(100)	7.36(100)	135.55(100)	458.81(1)
3	0(100)	0(100)	4.52(100)	508.98(100)	5248.63(100)	649.37(2)	(0)
4	0(100)	0(100)	3.83(100)	5091.54(100)	388.85(1)	(0)	(0)
5		0(100)	3.85(100)	4631.03(100)	(0)	(0)	(0)
6		0(100)	3.53(100)	2570.78(100)	(0)	(0)	(0)
7		0(100)	3.15(100)	1576.80(100)	(0)	(0)	(0)
8		0(100)	3.18(100)	1237.45(100)	(0)	(0)	(0)
9		0(100)	3.03(100)	760.71(100)	(0)	(0)	(0)
10		0(100)	3.25(100)	570.47(100)	(0)	(0)	(0)
11		0(100)	2.70(100)	495.66(100)	(0)	(0)	(0)
12		.01(100)	3.34(100)	411.57(100)	(0)	(0)	(0)
13			2.87(100)	398.69(100)	(0)	(0)	(0)
14			3.39(100)	413.44(100)	104.50(1)	(0)	(0)
15			3.12(100)	401.83(100)	(0)	(0)	(0)
16			2.54(100)	315.10(100)	(0)	(0)	(0)
17			2.96(100)	278.71(100)	(0)	(0)	(0)

TABLE 4.2 – Résultats obtenus par Adaptnovelty sur des instances du Jeu Eternity générées de manière aléatoire

	4	9	16	25	36	49	64
2	0(100)	0(100)	.98(100)	3.48(100)	11.46(100)	50.30(100)	904.24(100)
3	0(100)	0(100)	4.90(100)	187.15(100)	775.01(100)	14580.76(91)	(0)
4	0(100)	0(100)	3.96(100)	3948.46(100)	4940.72(21)	(0)	(0)
5		0(100)	4.79(100)	1978.47(100)	(0)	(0)	(0)
6		0(100)	4.90(100)	1168.63(100)	147.18(1)	(0)	(0)
7		.01(100)	3.96(100)	1088.75(100)	(0)	(0)	(0)
8		0(100)	4.77(100)	672.69(100)	117.19(1)	(0)	(0)
9		0(100)	3.64(100)	585.36(100)	(0)	(0)	(0)
10		0(100)	4.08(100)	514.66(100)	(0)	(0)	(0)
11		0(100)	4.28(100)	495.54(100)	(0)	(0)	(0)
12		0(100)	4.19(100)	586.28(100)	(0)	(0)	(0)
13			3.87(100)	515.66(100)	1291.93(3)	(0)	(0)
14			4.07(100)	434.25(100)	(0)	(0)	(0)
15			4.31(100)	465.90(100)	547.72(2)	(0)	(0)
16			4.20(100)	479.69(100)	354.40(1)	(0)	(0)
17			4.07(100)	480.57(100)	1254.24(3)	(0)	(0)

TABLE 4.3 – Résultats obtenus par Adaptwsat sur des instances du Jeu Eternity générées de manière aléatoire

Les deux solveurs adaptnovelty et adaptwsat répondent très rapidement pour les tailles 16 et moins.
Dès 25 pièces, les réponses se font beaucoup plus longues et on peut remarquer, comme pour Minisat **??**, une impression de seuil de difficulté vers $\sqrt{nb\_pieces}$.
Au delà, on constate que les résultats obtenus par ces solveurs se dégradent fortement... De même, de manière globale, les résultats se dégradent exponentiellement en fonction du nombre de pièces.
Pour l'ensemble des tailles, les instances de deux pièces sont toujours résolues beaucoup plus facilement que les autres.

Rsaps

	4	9	16	25	36	49	64
2	0(100)	0(100)	.41(100)	1.16(100)	2.85(100)	4.60(100)	8.23(100)
3	0(100)	.01(100)	4.32(100)	423.81(100)	1549.94(100)	4221.40(100)	12334.21(99)
4	0(100)	0(100)	5.20(100)	9084.00(97)	747.26(5)	400.08(1)	(0)
5		0(100)	5.80(100)	9868.34(99)	(0)	(0)	(0)
6		0(100)	5.87(100)	7711.59(98)	(0)	(0)	(0)
7		0(100)	4.94(100)	6624.80(100)	(0)	(0)	(0)
8		0(100)	4.84(100)	5613.97(99)	(0)	(0)	(0)
9		0(100)	4.27(100)	6110.32(100)	(0)	(0)	(0)
10		0(100)	4.01(100)	4823.86(100)	(0)	(0)	(0)
11		0(100)	4.62(100)	4423.21(100)	(0)	(0)	(0)
12		0(100)	4.80(100)	3717.22(100)	(0)	(0)	(0)
13			4.40(100)	3788.93(100)	(0)	(0)	(0)
14			4.18(100)	3662.54(100)	(0)	(0)	(0)
15			4.78(100)	3420.84(100)	605.90(1)	(0)	(0)
16			3.61(100)	3743.37(100)	(0)	(0)	(0)
17			3.99(100)	2955.41(100)	(0)	(0)	(0)

TABLE 4.4 – Résultats obtenus par Rsaps sur des instances du Jeu Eternity générées de manière aléatoire

Le solveur Rsaps réagit globalement comme les solveurs Adaptnovelty et Adaptwsat vus précédemment. Néanmoins il nous faut remarquer ses temps beaucoup plus élevés pour les instances de 25 pièces. De plus, il parvient aussi à résoudre les deux couleurs jusqu'aux 64 pièces.

Dcrwalk et Irots

Dcrwalk et irots ont vite été exclus des XP en raison de leurs résultats médiocres par rapport aux autres solveurs testés. Ils n'arrivent plus à résoudre les instances de 25 pièces, et quasiment aucune des 16. Nous avons donc préféré nous concentrer sur les autres solveurs.

Conclusion

Deux points importants sont ressortis des ces statistiques :

1. Il semble avoir un seuil de difficulté tournant autour des instances possédant $\sqrt{nb\_pieces}$ couleurs. Ils serait intéressant de poursuivre des tests dans cette voie.
2. Le solveur Minisat modifié que nous avons implémenté semble permettre les meilleures performances.

	4	9	16
2	.01(100)	1528.95(100)	(0)
3	.14(100)	11029.43(95)	(0)
4	.20(100)	12458.94(81)	(0)
5		11508.57(68)	(0)
6		12022.89(52)	(0)
7		10869.88(40)	(0)
8		10345.21(46)	(0)
9		9046.74(31)	(0)
10		7569.48(29)	(0)
11		5935.50(23)	(0)
12		6163.20(24)	(0)
13			(0)
14			(0)
15			(0)
16			(0)
17			(0)

TABLE 4.5 – Résultats obtenus par Dcrwalk sur des instances du Jeu Eternity générées de manière aléatoire

	4	9	16
2	0(100)	2.07(100)	9506.42(33)
3	0(100)	20.50(100)	(0)
4	0(100)	40.55(100)	(0)
5		67.19(100)	(0)
6		110.14(100)	(0)
7		114.51(100)	(0)
8		121.07(100)	(0)
9		164.16(100)	(0)
10		176.94(100)	(0)
11		197.28(100)	(0)
12		237.23(100)	(0)
13			(0)
14			(0)
15			(0)
16			(0)
17			(0)

TABLE 4.6 – Résultats obtenus par Irots sur des instances du Jeu Eternity générées de manière aléatoire

Conclusion générale

Dans ce rapport, nous avons proposé une application $Java$ d'Esternity II. Celle-ci permet de faire appel à différents solveurs afin de vérifier si l'instance courante du jeu est satisfaisable ou non. Une résolution dite "pas à pas" est aussi disponible offrant ainsi une représentation visuelle de la résolution d'un solveur en temps réel.

Nous avons donc construit un codage CNF permettant de transcrire chaque instance du jeu en un fichier CNF. Il peut donc être utilisé avec n'importe quel solveur SAT. Nos expérimentations nous ont démontré l'efficacité de certains solveurs par rapport à d'autres. Il serait néanmoins intéressant d'élargir les recherches sur d'autres solveurs SAT.

Pour finir, nous sommes ravis d'avoir eu l'occasion de travailler sur SAT. Nous comprenons mieux l'importance des recherches du CRIL dans ce domaine et espérons avoir pu aider à notre manière avec nos travaux.

Bibliographie

[1] Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. A translation of pseudo boolean constraints to SAT. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2 :191–200, 2006.

[2] Thierry Benoist and Éric Bourreau. La programmation par contraintes à l'attaque d'eternity II. In *Proceedings of the Journées Francaises de Programmation par Contraintes (JPFC'08)*, 2008. (in French).

[3] Erik D. Demaine and Martin L. Demaine. Jigsaw puzzles, edge matching, and polyomino packing : Connections and complexity. *Graphs and Combinatorics*, 23(1) :195–208, 2007.

[4] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[5] I Gent and P Nightingale. A new encoding of alldifferent into SAT. In *3rd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, Toronto (Canada), 2004.

[6] Ian P. Gent and Patrick Prosser. An empirical study of the stable marriage problem with ties and incomplete lists. In *Proceedings of the 15th Eureopean Conference on Artificial Intelligence (ECAI'02)*, pages 141–145, 2002.

[7] Holger H. Hoos. An adaptive noise mechanism for walksat. In *AAAI/IAAI*, pages 655–660, 2002.

[8] Frank Hutter, Dave A. D. Tompkins, and Holger H. Hoos. Reactive scaling and probabilistic smoothing : Efficient dynamic local search for sat. In Pascal Van Hentenryck, editor, *CP*, volume 2470 of *Lecture Notes in Computer Science*, pages 233–248. Springer, 2002.

[9] Chu Min Li, Wanxia Wei, and Harry Zhang. Combining adaptive noise and look-ahead in local search for sat. In João Marques-Silva and Karem A. Sakallah, editors, *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2007.

[10] João P. Marques Silva and Inês Lynce. Towards robust CNF encodings of cardinality constraints. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP'07)*, pages 483–497, 2007.

[11] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, volume 1, pages 362–367, Seattle (United States), 1994.

[12] Pierre Schauss and Yves Deville. Hybridization of CP and VLNS for eternity II. In *Proceedings of the Journées Francaises de Programmation par Contraintes (JPFC'08)*, 2008.

[13] Kevin Smyth, Holger H. Hoos, and Thomas Stützle. Iterated robust tabu search for max-sat. In Yang Xiang and Brahim Chaib-draa, editors, *Canadian Conference on AI*, volume 2671 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2003.

[14] Dave A. D. Tompkins and Holger H. Hoos. On the quality and quantity of random decisions in stochastic local search for sat. In Luc Lamontagne and Mario Marchand, editors, *Canadian Conference on AI*, volume 4013 of *Lecture Notes in Computer Science*, pages 146–158. Springer, 2006.