CentraleSupelélec

Fondements logiques

Jeu du Binero exprimé sous forme d'un problème de satisfaction de contraintes booléennes

Professeur :
Pascal Le Gall

Auteurs :
Stepan Barrau, Marcello Vaccarino

26 mai 2018

Table des matières

1 Introduction **1**

2 Mode d'emploi **2**

3 Modélisation du problème **2**
3.1 Variables 2
3.2 Modélisation de les règles du jeu 3

4 Solveur **7**

1 Introduction

Dans cette projet nous avons modélisé, et en suite résolu le jeu du Binero comme un problème de satisfaction des contraintes. D'aborde nous avons formalisé les règle du jeux en logique propositionnelle pour trover une solution avec l'aide d'un solveur SMT. Dans en deuxième temps nous avons exprimé les contraintes sous forme normale conjonctive pour le résoudre avec un solveur SAT. Cependant nous ne sont pas arrivé à résoudre des jeux avec cette deuxième méthode pour des raisons que vont entre expliqué. Pourtant une explication détaillé et justifié de l'approche adopté est donné.

2 Mode d'emploi

Voici quelque instruction pour reproduire les résultats présenté dans la section 3. Le contenu est divisé en trois dossiers :

— dataIn : contient de jeu à résoudre en format binero.
— output : contient le ficher Dimacs et le jeu résolus.
— source : contient le code source. Le point d'accès est le ficher "$Client\backslash app.py$".

Pour résoudre un jeux faut mettre le ficher de input, par exemple "monJeu", dans le dossier dataIn. En suite changer le nom du ficher prise en entré par le "$Client\backslash app.py$", c'est à dire changer le nom du ficher a la ligne 8 du code en "monJeu". Finalement suffit de exécuter "$Client\backslash app.py$" sous python3 pour avoir en sortie, dans le dossier output, un ficher dimacs avec le condition sous forme normale conjonctive. De plus la solution est trouvé avec un solveur SMT et écrite en output.

3 Modélisation du problème

3.1 Variables

Un jeu de Binero consiste en une matrice carrée G. Chaque élément de la matrice peut être soit 0 soit 1. Pour cette raison une variable $x_{i,j}$ a été définie pour chaque case du jeu.

Pour la mise sou format DIMACS, chaque variable doit être numérotée par un entier unique. Par convention la variable $x_{i,j}$ **est numérotée** $ni + j + 1$ (le $+1$ est utile pour ne pas avoir la variable 0, qui en DIMACS est réservé pour indiquer la clause vide).

Pour la mise sous forme normale, on créé aussi des variables annexes qui nous permettront d'exprimer plus facilement les conditions que doit respecter une grille de Binero.
Ansi, pour chaque ligne i (colonne j) a été définie une matrice n*n de variables $y_{i,i',j'}$ ($y_{j,i',j'}$).
Pour le format DIMACS, ces variables seront numérotées ainsi :
$y_{i,i',j'} : (i+1)n^2 + ni' + j' + 1$
$y_{i,i',j'} : (n+j+1)n^2 + ni' + j' + 1$

3.2 Modélisation de les règles du jeu

Cette section traite la formalisation des réglés du jeu en logique propositionnelle et leur mise sous forme normale conjonctive.

Première règle

La première règle dit qu'il doit y avoir sur chaque ligne et colonne autant de 0 que de 1. Dans la suite on dira qu'une ligne ou une colonne est paire si satisfait cette règle.

première approche .
Notre première approche a été de créer un tableau P (comme possible) avec toutes les permutation de l'ensemble contenant $\frac{n}{2}$ fois 0 et $\frac{n}{2}$ fois 1, c-à-d qui un tableau dont les lignes ont autant de 0 que de 1.

Exemple pour n = 4 : $\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$

Grâce à ce tableau de taille à peu près égale à $(n, 2^n)$, il est possible d'écrire la formule qui exprime la parité d'une ligne ou d'une colonne. Sous forme normale **disjonctive**, ce tableau traduit directement la formule :
Pour chaque ligne i, il faut, qu'elle soit égale à l'une des lignes de notre tableau P des lignes possibles. On comprends donc bien que soit chaque variable de la ligne i soit égale à celle qui lui correspond dans la première ligne du tableau P, **ou** dans la deuxième, **ou** dans la troisième, etc. Ceci donne donc bien une forme normale **disjonctive**.

$$\begin{aligned}(x_{i,1} == p[0][1] \wedge x_{i,2} == p[0][2] \wedge ... \wedge x_{i,n} == p[0][n]) \\ \vee (x_{i,1} == p[1][1] \wedge x_{i,2} == p[1][2] \wedge ... \wedge x_{i,n} == p[1][n]) \\ etc.\end{aligned}$$

Pour transformer cette condition en forme normale **conjonctive**, il faut donc développer cette forme disjonctive (par la distributivité des *et* et des *ou*)

Malheureusement cette approche n'est valable que si on dispose d'un solveur SMT parce que la mise sous forme normale conjonctive (le développement) multiplie le nombre de clauses : on se retrouve avec environ n^{2^n} clauses. Par exemple, pour n = 6, c'est 6^{20}

deuxième approche .
Nous avons donc tenté une deuxième façon de faire, qui ne nécessite pas d'énumérer toutes les lignes possibles.

Cette approche (qui créé des lignes en complexité polynomiale vis-à-vis de n) se base sur des séries de transformations imposées aux lignes. Le but est de mettre tous les 1 à droite et les 0 à gauche au sein d'une ligne, ainsi il sera facile de dire qui elle contient le bon nombre de chaque.

C'est pour cela qu'on créé des variables annexes, qui contiendront les étapes de transformation de chaque ligne/colonne. Notre transformation prends au plus n étapes, donc on a créé n lignes de n variables par ligne et colonne de grille du Binero. Cela fait donc $2 * n^3$

variables en plus, ce sont les variables $y_{i/j,i',j'}$, que l'on numérote comme expliqué dans la section **Variables**.

Voici un exemple de transformation, pour n = 4, en partant de la ligne 1,0,1,0 :

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

La transformation se base sur une condition sur 4 bits : **ordre**

Cette condition (qui s'écrit en 16 clauses conjonctives) force ceci :
pour 4 littéraux $x1, x2, y1, y2$, les conditions $ordre(x_1, x_2, y_1, y_2)$ imposent que :

$$x_1 = 0, x_2 = 0 \Leftrightarrow y_1 = 0, y_2 = 0$$
$$x_1 = 1, x_2 = 1 \Leftrightarrow y_1 = 1, y_2 = 1$$

$$(x_1 = 0, x_2 = 1) ou (x_1 = 1, x_2 = 0) \Leftrightarrow y_1 = 0, y_2 = 1$$

On voit que cette opération fait en effet ce qu'il faut, sur des lignes de taille 2. La transformation de la ligne (x_1, x_2) en la ligne (y_1, y_2) suit bien les règles qu'il faut.
Important : le nombre de 0 et de 1 est conservé, ce qui garantit qu'on ne fera pas d'erreurs par la suite.

Pour faire passer tous les 1 à droite sur des lignes plus longues, on répète plusieurs fois cette étape.

On va créer une succession de lignes qui correspondent aux étapes de la transformation.
A chaque transformation, on applique ordre à toutes les paires disjointes d'éléments. Ainsi pour la première transformation, on applique :

$ordre(x_1, x_2, y_{1,1}, y_{1,2}), ordre(x_3, x_4, y_{1,3}, y_{1,4}), ..., ordre(x_{n-1}, x_n, y_{1,n-1}, y_{1,n})$

Ensuite, on recommence mais cette fois-ci en prenant les autres paires, pour avoir ceci :

$ordre(y_{1,2}, y_{1,3}, y_{2,2}, y_{2,3}), ordre(y_{1,4}, y_{1,5}, y_{2,4}, y_{2,5}), ..., ordre(y_{1,n-2}, y_{1,n-1}, y_{2,n-2}, y_{2,n-1})$

Et on continue en alternant les paires (soit celles du type [1,2], [3,4], [5,6], soit celles du type [2,3], [4,5], [6,7]) et en descendant dans les lignes de variables y.

Ainsi, petit à petit, dans chaque ligne, les 1 se retrouvent un peu plus à droite et les 0 un peu plus à gauche, mais **le nombre est conservé**. au bout de n itérations, on est sûr que tous les 1 sont à droite.

Autre exemple, avec la ligne $(1, 1, 1, 0, 0, 0)$. Les parenthèses représentent les paires qu'on ordonne d'une ligne à la suivante, et chaque ligne est doublée pour voir clairement le résultat :

$$\left[\begin{array}{cccccc}
(1 & 1) & (1 & 0) & (0 & 0) \\
1 & 1 & 0 & 1 & 0 & 0 \\
 & & & & & \\
- & - & - & - & - & - \\
1 & (1 & 0) & (1 & 0) & 0 \\
1 & 0 & 1 & 0 & 1 & 0 \\
 & & & & & \\
(1 & 0) & (1 & 0) & (1 & 0) \\
0 & 1 & 0 & 1 & 0 & 1 \\
 & & & & & \\
0 & (1 & 0) & (1 & 0) & 1 \\
0 & 0 & 1 & 0 & 1 & 1 \\
 & & & & & \\
(0 & 0) & (1 & 0) & (1 & 1) \\
0 & 0 & 0 & 1 & 1 & 1
\end{array}\right]$$

Comme on le voit, le fait l'alterner les paires sur lesquelles on applique **ordre** permet de faire avancer les 1 vers la droite d'un cran à chaque itération.
Si tous les 1 sont déjà à droite, répéter l'opération ordre ne change rien.

Reste pour finir, à ajouter la condition finale, qui est que la dernière ligne ainsi obtenue doit être exactement du type 0,0,0,1,1,1. Ceci se fait en n clauses de 1 littéral.

Tout ceci résulte en un nombre polynomial de clauses, et on arrive donc à faire tourner cet algorithme avec n = 30.

Deuxième règle

La deuxième règle dit que, sur chaque colonne ou ligne de la grille, il ne peut y avoir plus de deux 0 ou deux 1 consécutifs.
Ceci peut être traduit, en logique du premier ordre, en imposant que si deux cases adjacentes sont égales, la troisième doit être différente. Cette condition est écrite formellement pour le ligne dans l'équation 1. Les quantificateurs ne sont qu'un raccourci pour noter de façon générale les $(n-2)^2$ formules qui expriment la conditions (n-2 par ligne et par colonne).

$$\forall i \forall j (x_{i,j} \Leftrightarrow x_{i,j+1}) \Rightarrow (x_{i,j+2} \not\Leftrightarrow x_{i,j}) \tag{1}$$

La formule 2 exprime en forme normale conjonctive cette même condition. Intuitivement la formule exprime le fait que il ne peut pas y avoir trois 1 de suite ou trois 0 de suite.

$$\forall i \forall j (x_{i,j} \vee x_{i,j+1} \vee x_{i,j+2}) \wedge (\neg x_{i,j} \vee \neg x_{i,j+1} \vee \neg x_{i,j+2}) \tag{2}$$

En termes de complexité, cette règle créé un nombre polynomial de clauses.

Troisième règle

La troisième règle dit qu'il n'y a pas deux lignes remplies identiquement ou deux colonnes remplies identiquement. En notant i et j les indices des deux lignes et k celui de la colonne,

la 3 impose que dans deux lignes il y ai au moine un élément différent.

$$\forall_{i \neq j} i, j \neg(x_{i,1} \Leftrightarrow x_{i,1}) \vee \neg(x_{i,2} \Leftrightarrow x_{i,2}) \vee ... \vee \neg(x_{i,n} \Leftrightarrow x_{i,n}) \tag{3}$$

Cette formule est donc sous forme normale **disjonctive**, et pour qu'elle soit conjonctive, il faut appliquer la distributivité.

Application de la distributivité
Pour écrire cette distribution en forme normale conjonctive un tableau p de combinaisons a été généré. Dans le cas n = 4 :

$$\mathrm{p} = \begin{bmatrix} -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & 1 \\ -1 & -1 & 1 & -1 \\ -1 & 1 & -1 & -1 \\ -1 & 1 & -1 & 1 \\ -1 & 1 & 1 & -1 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Ainsi la condition 3 peut être écrite comme dans la 4 grâce a des samples transformations.

$$p[k][1]x_{i,1} \vee p[k][1]x_{j,1} \vee p[k][2]x_{i,2} \vee p[k][2]x_{j,2} ... p[k][n]x_{i,n} \vee p[k][n]x_{j,n} \tag{4}$$

fin de l'application de la distributivité

Cette condition créé un nombre de clauses de l'ordre de 2^n, on peut donc calculer les conditions pour n = 14, au-delà ça prends trop de temps.

4 Solveur

Pour réaliser cette projet deux solveurs ont été considéré. D'abord Z3, un prouver modulo des théories qui accept des formules de la logique du premier ordre avec égalité et qui dispose de API python très confortables. Cette solveur arrive très bien à résoudre le cas de test "$petit\backslash binero$".
MiniSat a été utilisé comme solveur de formules de la logique propositionnelle en forme normale conjonctive. Cette outil à été utilisé pour conduire des test non satisfaisants sur la mise en forme normale. En effet, bien que l'approche soit très prometteur, le clause relative a la condition 1 décrite en section 3.2 ne sont pas satisfiables pour de raisons que, à aujourd'hui nous ne comprenons pas. Le dimacs relative à trois exemples sont contenus dans le ficher output.