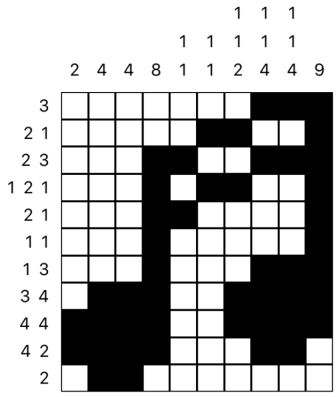
Compte Rendu INF402

Sofiane DJERBI et Salem HAFTARI
https://github.com/Kugge/Nonogram-Solver

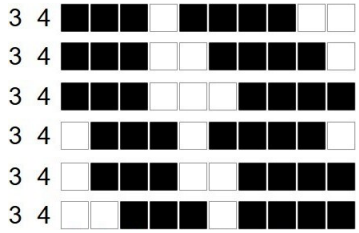
Mars 2021

1 Présentation du projet

Le but du projet est de modéliser et résoudre un *nonogramme* (ou *picross*/*logigraphe*) à l'aide d'un *solveur SAT*. Un nonogramme est un *jeu logique* présenté sous forme d'une grille munie de deux ensembles de données correspondant aux lignes et colonnes de la grille.



Sur une ligne, chaque nombre indique le nombre de pixel à colorier qui se suivent. S'il y a plusieurs nombres alors les groupes de pixels correspondant doivent être séparés d'au moins une case blanche. Idem avec les colonnes. Par exemple, voilà toutes les configurations possibles pour les coefficient $3, 4$:



2 Modélisation du problème

Soient $(N, M) \in \mathbb{N}^2$ et $[\![1, N]\!] \times [\![1, M]\!]$ représentant une grille de taille $N \times M$. Soient $i \in [\![1, N]\!]$ une ligne quelconque (resp. $j \in [\![1, M]\!]$ une colonne quelconque) et $X_i = (a_{i,1} \dots a_{i,k_i})$ les coefficients de la ligne i (resp. Y_j les coefficients de la colonne j) avec pour restriction:

$$0 \leq k_i + \sum_{\mu=1}^{k_i} a_{i,\mu} \leq M + 1$$

Enfin, soit $C_{i,j}$ la valeur de la case (i, j) (i.e 0 ou 1/ Blanche ou Noire), l'indexation est la même que pour les matrices.

Alors, $\forall p \in [\![1, k_i]\!], \exists\, l_{i,p} \in \mathbb{N}^*$ tel que
$C_{i,l_{i,p}} = C_{i,l_{i,p}+1} = \dots = C_{i,l_{i,p}+a_{i,p}-1} = 1$ et $C_{i,l_{i,p}+a_{i,p}} = 0$ avec les contraintes suivantes:

$$\begin{cases} l_{i,p} \;\geq\; l_{i,p-1} + a_{i,p-1} + 1 & (1) \\ l_{i,k_i} \leq\; M - a_{i,k_i} + 1 & (2) \end{cases}$$

Pour les colonnes, le raisonnement est analogue à celui des lignes, il suffit de remplacer les a_i par b_j, les k_i par t_j et M par N.

La ligne (1) fait en sorte que des groupes de cases noires ne se suivent pas et ne se chevauchent pas.
La ligne (2) il s'agit juste d'une condition pour que l'indexation soit cohérente. En effet, ce serait embêtant de colorier des cases inexistantes. De plus, si la dernière case doit être noire, la condition $C_{i,l_{i,p}+a_{i,p}} = 0$ est omise (puisque cette case n'existe pas).

3 Modélisation en FNC

3.1 Difficultés

Dans un premier temps, nous avons essayé de développer "bêtement" les FND que l'on obtenait à partir du nonogramme mais la complexité étant beaucoup trop élevée, nous avons donc rapidement abandonné cette idée. Une autre idée était de modéliser directement le nonogramme en FNC sans avoir à passer par le développement d'une FND.

3.2 Nonogramme en FNC

Soit $Config_i$ un booléen correspondant au choix d'une configuration particulière de la ligne i.
Alors chaque case $C_{i,j}$ de la ligne doit obéir à cette configuration, par exemple avec une configuration "Vrai, Faux, Faux":

$$\begin{gathered} Config_i \Rightarrow C_{i,0} \\ Config_i \Rightarrow \neg C_{i,1} \\ Config_i \Rightarrow \neg C_{i,2} \end{gathered}$$

Et on en déduit la forme normale disjonctive suivante:

$$\begin{gathered} \neg(Config_i \vee C_{i,0}) \equiv (\neg Config_i \vee C_{i,0}) \\ \neg(Config_i \vee \neg C_{i,1}) \equiv (\neg Config_i \vee \neg C_{i,1}) \\ \neg(Config_i \vee \neg C_{i,2}) \equiv (\neg Config_i \vee \neg C_{i,2}) \end{gathered}$$

La conjonction de ces clauses nous donnera bien la forme normale conjonctive recherchée.

Sachant qu'il y a plusieurs configurations possibles par ligne, nous notons $Config_{i,k}$ la k^{eme} configuration de la i^{eme} ligne.
Seul une configuration par ligne est possible, nous avons donc la FND suivante si il y a n configurations possibles pour la ligne i:

$$Config_{i,1} \vee \cdots \vee Config_{i,n}$$

Nous avons donc au final, pour une ligne de taille n avec k configurations possibles, et $K_{i,j,x}$ la valeur de la case $C_{i,j}$ décidée par la x^{eme} configuration.
Le problème final est donc représenté par la FNC suivante:

$$\begin{gathered} (\neg Config_{i,0} \vee K_{i,0,0}) \wedge \cdots \wedge (\neg Config_i \vee K_{i,n,0}) \wedge \\ \cdots \\ (\neg Config_{i,k} \vee K_{i,0,k}) \wedge \cdots \wedge (\neg Config_i \vee K_{i,n,k}) \wedge \\ (Config_{i,0} \vee \cdots \vee Config_{i,k}) \end{gathered}$$

Le raisonnement est identique pour les colonnes.

3.3 Implémentation

L'algorithme générant ces FNC est un générateur Python créant un arbre des différentes configurations possibles en fonction des espaces séparant deux rangées coloriées.
L'algorithme retourne le chemin feuille-racine de chaque feuille. Ce chemin feuille-racine correspond à une configuration particulière.

3.4 Complexité

Soit n le nombre de noeuds de l'arbre généré par notre algorithme.
Chaque noeud est visité une seule fois, comme il s'agit d'un générateur, la complexité est de $O(n)$.
Cependant, il est plus compliqué d'estimer la complexité en fonction du nombre de cases ou du nombre de coefficients.
En fixant une ligne avec comme coefficient $1, 1$ il y a, par récurrence, t_{n-2} configurations possibles, avec n le nombre de cases de la ligne et t_i le i^{eme} nombre triangulaire.
De même, avec comme coefficient $1, 1, 1$ il y a, par récurrence, T_{n-3} configurations possibles, avec n le nombre de cases de la ligne et T_i le i^{eme} nombre tétraédrique.
En supposant qu'avec k coefficients 1 on obtient une généralisation des nombres triangulaires en dimension k, et à l'aide de la formule de Faulhaber, nous obtenons donc, pour une simple ligne munie de k coefficients une complexitée de $O(n^k)$ avec k le nombre de case de la ligne.

Nous pouvons nous ramener à cette situation en posant $k = n + k - \sum_{i=1}^{k} c_k$ avec n le nombre de case sur la ligne, k le nombre de coefficients et c_i le i^{eme} coefficient, chaque coefficient peut être considéré comme un "1".

4 Limitations

4.1 Limitations de la modélisation en problème SAT

La modélisation (*complète*) en problème SAT implique l'analyse de plusieurs millions de possibilités, générant rapidement des fichiers de clause de plusieurs gigas, ce que certains solveurs ne supportent pas. Malgré cela, s'il est possible de charger toutes les clauses dans la RAM, les solveurs parviennent à trouver un modèle en un temps raisonnable.

4.2 Optimisations possibles

Il est donc envisageable de créer un solveur SAT ne manipulant pas les clauses dans la RAM de l'ordinateur mais directement dans un fichier DIMACS, effectuant la réduction unitaire, etc, directement au sein de ce fichier.
Il est également possible de trouver des stratégies, de programmer "par contrainte" afin de réduire le nombre de clauses.
Sachant également qu'il existe des languages compilés plus rapide que Python, il est envisageable de reproduire ce programme en C/C++ ou en Rust.
Certaines librairies d'optimisation existent pour Python, notamment Numba.