Modélisation et Résolution par SAT et CSP :
Le Problème de Schur

Vendredi 2 février 2024

GHANNAY Nesrine

M1 Informatique parcours IAAA

TABLE DES MATIERES

Dans ce rapport, nous explorons la modélisation et la résolution informatique du problème de Schur, qui implique la répartition de n balles étiquetées de 1 à n dans k boîtes (avec k ≥ 3). L'objectif est de garantir que pour toute combinaison de trois balles numérotées x, y, z ; si x + y = z, alors ces trois balles ne peuvent pas être contenues dans une même boîte. Pour aborder cette problématique, nous étudierons deux approches de résolution : SAT (Satisfiability) et CSP (Constraint Satisfaction Problem).

I. MODELISATION ET RESOLUTION PAR SAT

1. MODELISATION

Le problème de satisfaisabilité propositionnelle (SAT), permet étant donné une formule de la logique propositionnelle en CNF (forme normale conjonctive), de déterminer s'il existe une interprétation de ses variables qui permet de satisfaire toutes ses clauses.

Notre projet vise donc à mettre en œuvre premièrement une réduction du problème de Schur, en une formule de la logique propositionnelle SAT (sous format DIMACS CNF), permettant ainsi d'automatiser la vérification de la solvabilité de cette modélisation.

La première étape afin de modéliser ce problème est d'en exprimer les règles par un ensemble de contraintes à satisfaire. Dans une instance du problème de Schur, pour un nombre k de boîtes et n de balles, nous devons respecter les règles suivantes :

- Pour chaque balles $i, j, l \in \{1, \dots n\}$, si $i + j = l$, alors les balles ne doivent pas toutes être dans la même boîte.
- Chaque balle doit être dans une boîte exactement. Ce qui peut se décomposer en deux contraintes :
    - Chaque balle doit être dans au moins une boîte.
    - Chaque balle est au plus dans une boîte.

Ces règles seront alors les contraintes que notre formule SAT devra satisfaire. Par conséquent, nous devons transformer ces contraintes en ensemble de clauses sous format CNF. Pour cela on introduit les variables suivantes :

$$X_{i,j}$$

Où :

- $i \in [\![1, n]\!]$, spécifie le numéro de la balle;
- j $\in [\![1, k]\!]$, spécifie le numéro de la boîte dans laquelle elle se trouve ;

Donc si $X_{i,j}$ est évaluée à vrai, cela signifie que la balle i est dans la boite j.

On peut maintenant écrire notre formule SAT qui est composée de la disjonction de chaque contrainte précédente sous forme CNF :

$$\boldsymbol{\varphi := \varphi_1 \wedge \varphi_2 \wedge \varphi_3}\textbf{ , où :}$$

- $\boldsymbol{\varphi_1}$: Pour chaque balles $i, j, l \in \{1, \dots n\}$, si $i + j = l$, alors les balles ne doivent pas toutes être dans la même boîte.

$$\varphi_1 := \bigwedge_{i,j,l \in [\![1,n]\!];\ i+j=l} \ \bigwedge_{b \in [\![1,k]\!]} \overline{X_{i,b}} \vee \overline{X_{j,b}} \vee \overline{X_{l,b}}$$

- $\boldsymbol{\varphi_2}$: Chaque balle doit être au moins dans une boîte

$$\varphi_2 := \bigwedge_{i \in [\![1,n]\!]} \left(\bigvee_{b \in [\![1,k]\!]} X_{i,b} \right)$$

- $\boldsymbol{\varphi_3}$: Chaque balle doit être au moins dans une boîte

$$\varphi_3 := \bigwedge_{i \in [\![1,n]\!]} \ \bigwedge_{b,b' \in [\![1,k]\!];\ b \neq b'} \overline{X_{i,b}} \vee \overline{X_{i,b'}}$$

2. PROGRAMMATION

Dans la suite du projet, il faut implémenter cette réduction afin de pouvoir utiliser le solver **MiniSat**, qui utilise en entrée un fichier en format DIMACS CNF. Et il indiquera si oui ou non, la formule est satisfaisable et donc si l'instance du problème de schur est solvable.

Nous avons tout d'abord commencé à définir certaines structures de données qui nous permettent par la suite d'implémenter correctement la réduction.

L'ENSEMBLE DES LITTERAUX

Lors de l'implémentation, un littéral sera définit comme un tuple [i, j]. Cela nous permet de conserver la totalité des informations que la variable X_{ij} nous donne. Chaque clause sera alors un tuple[i,j], où les éléments du tableau seront tous positifs si la variable représente un littéral positif, et tous négatifs sinon.

L'ENSEMBLE DES CLAUSES

Pour générer l'ensemble des clauses présentées ci-dessus, on fait appel à la fonction ***`generate_schur_sat_instance()`,*** permettant d'implémenter les trois contraintes vues précédemment.

MISE SOUS FORMAT DIMACS CNF

Cependant, bien que la forme sous laquelle chaque littéral est représenté ([i, j]), nous permet de conserver la totalité des informations, elle ne respecte pas le format DIMACS CNF. En effet, le solveur MiniSat s'attend à recevoir un fichier `.cnf`, définit de manière suivante :

```
p cnf 60 350
-1 -4 -7 0
-2 -5 -8 0
-3 -6 -9 0
-1 -7 -10 0
  ...
```

(Exemple de début de fichier .cnf produit, pour l'instance du problème de schur avec k = 3, et n = 20)

Où la première ligne apporte des informations sur le nombre de littéraux utilisé et le nombre de clauses générées. Puis, chaque ligne doit représenter une clause, et chaque littéral est représenté par un entier (positif ou négatif indiquant respectivement un littéral ou son opposé).

On doit donc trouver une façon de transformer notre triplet [i, j, k], en un unique entier. La façon plus intuitive de transformer ce tuple est de trouver une fonction bijective. Voici la fonction utilisée :

$$f(i,j) = \begin{cases} -((k * i - 1) + j), & i,j < 0 \\ ((k * i - 1) + j), & i,j > 0 \end{cases}$$

Où, k représente le nombre total de boîte pour une instance.

Après, s'être assurée que le format est bien respecté on peut soumettre notre formule au solveur MiniSat, afin de savoir si notre instance de schur est solvable ou non.

VISUALISATION DE LA SOLUTION :

Enfin, si le solveur MiniSat nous indique que l'instance du problème est bien solvable, il nous retourne un fichier .out de la forme suivante :

```
SAT
1 -2 -3 -4 5 -6 7 -8 -9 -10 11 -12 -13 14 -15 16 -17 -18 -19 -20 21 22 -23 -24 -25 -26 27 -
28 29 -30 -31 -32 33 -34 -35 36 37 -38 -39 -40 -41 42 -43 -44 45 -46 47 -48 -49 -50 51
52 -53 -54 -55 56 -57 58 -59 -60 0
```

(Exemple de fichier .out produit par Minisat, pour l'instance du problème de schur avec k = 3, et n = 20)

Il serait intéressant de pouvoir visualiser comment MiniSat résout le problème de Schur (dans le cas où cette modélisation a bien une solution et qu'elle a été trouvée en un temps raisonnable $\leq 600\ s$.). Pour cela, on créer une méthode d'affichage, qui récupère toutes les littéraux positif du fichier produit par MiniSat.

Par la suite, il faut trouver un moyen d'appliquer la transformation inverse à celle que nous avons appliqué précédemment afin de produire à partir d'un littéral retourné x , un littéral représenté par le tuple (i, j).

Cela revient dans notre cas à faire la réciproque de la transformation précédente et de l'appliquer à un entier x afin de retrouver i, j (le numéro de la balle et de la boîte dans laquelle elle se trouve):

$$i = \big((x-1)//k\big) + 1$$

$$j = ((x - 1)\ \%\ k) + 1$$

Où, k représente le nombre total de boîte pour une instance.

Après avoir appliquer cette transformation à chaque littéral positif, on pourra afficher la solution de la manière suivante :

- Chaque ligne i représente le contenu de la boîte i ;
- Le contenu d'une boite est représenté par une suite de numéro des balles se trouvant dans la boîte (tous séparés par un espace).

Par exemple, pour le fichier précédant, on obtient la solution suivante :

```
1 3 6 8 13 18 20
2 4 5 10 16 19
7 9 11 12 14 15 17
```

(Exemple d'affichage produit, pour l'instance du problème de schur avec k = 3, et n = 20)

II. MODELISATION ET RESOLUTION PAR CSP

1. MODELISATION

Le Problème de Satisfaction de Contraintes (CSP) constitue une autre approche puissante pour la résolution de problèmes informatique complexes. Pour comprendre cette méthodologie, considérons le problème CSP noté P = (X, D, C), où chaque composant est défini de la manière suivante :

- $X = \{x_1, \dots, x_n\}$ représentant un ensemble de n variables,
- $D = \{d_{x1}, \dots, d_{xn}\}$ formant un ensemble de domaines définis, chaque domaine d_{xi} ayant une taille maximale de d,
- C constitué d'un ensemble de m contraintes, notées c = (S(c), R(c)), où :
  - $S(c) \subseteq X$: détermine la portée de la contrainte,
  - $R(c) \subseteq \Pi x \in S(c)\ dx$: représente la relation associée à la contrainte, définie sur le produit cartésien des domaines correspondants.

Dans le contexte de la modélisation du Problème de Schur, nous considérons k comme le nombre de boîtes et n comme le nombre de balles. De plus, on définit une variable par balle i, b_i : numéro de la boîte associée à la balle i
Ainsi, l'instance P = (X, D, C) est définie comme suit :

- **$X = \{b_1, \dots, b_n\}$** : il s'agit de l'ensemble des variables représentant les balles du problème de Schur, chaque variable b_i représente une balle spécifique du problème.
- **$D = \{d_{b1}, \dots, d_{bn}\}$ avec** $\forall\ \boldsymbol{i} \in \{\mathbf{1}, \dots, \boldsymbol{n}\},\ \boldsymbol{d_{bi}} = \{\mathbf{1}, \dots, \boldsymbol{k}\}$: chaque variable p_i a son domaine d_{bi}, qui est constitué des valeurs possibles que peut prendre la variable p_i. Dans notre cas, c'est-à-dire l'ensemble des boîtes dans lesquelles la balle i peut se trouver.
- **C** $= \{\ \boldsymbol{c_{ijl}} = \boldsymbol{i + j = l}\}$: permet de définir les règles du problème de Schur, et concerne donc les balles numéro i, j, et l, de telles sorte que i + j = k. Avec :
  - **$S(c_{ijl}) = \{ b_i , b_j, b_l\}$:** La portée de la contrainte indique les variables impliquées, c'est à dire, les variables associées aux balles bi, bj, et bl
  - **$R(c_{ijl}) = \{ (v_i , v_j, v_l) \mid v_i \in d_{bi}, v_j \in d_{bj}, v_l \in d_{bl},\ v_i = v_j \implies v_l \neq v_i\}$:** Cela garantit que si les valeurs associées aux balles i et j (vi et vj) sont égales, alors la valeur associée à la balle l (vl) doit être différente de celles de i et j.

Cette instance CSP, pourra par la suite être utilisé par différents solveur CSP, tels que Choco ou Ace.

2. PROGRAMMATION

Afin d'implémenter cette modélisation, de sorte à pouvoir la donner aux solveurs cités précédemment, on utilise la librairie pycsp3.

DECLARATION DES VARIABLES

Initialement, nous définissons la variable b, représentant la boîte à laquelle chaque balle doit être assignée. Pour cela, on déclare b structuré sous la forme d'un tableau (VarArray) de taille n (nombre de balles), qui permet d'indiquer pour chaque balle i, la boîte dans laquelle elle se trouve, dont le numéro peut aller de 1 au nombre de boîte k ($dom = range(1, k+1)$).

FORMULATION DES CONTRAINTES

Ensuite on définit à l'aide de trois boucle for, pour chaque balles i, j, l qui respectent le fait que $i + j = l$, la contrainte qu'elles ne peuvent pas être toutes dans la même boîte :

$$((b[i] != b[j]) \mid (b[i] != b[l]) \mid (b[j] != b[l]))$$

RECOURS AUX SOLVEURS

Ainsi, on pourra vérifier si l'instance générée a une solution, en appelant la méthode `solve()` de la bibliothèque Pycsp3 en précisant le solveur que l'on souhaite utiliser : ACE ou CHOCO.

AFFICHAGE DES RESULTATS

Enfin, si une solution est trouvée dans un délai raisonnable (≤ 600 s), nous affichons cette solution, de la même manière que SAT (en parcourant la solution donnée sous forme de tableau d'entier). Ceci conclut le processus de résolution du Problème de Schur à l'aide de la modélisation CSP

Dans la partie suivante, nous essaierons d'analyser les résultats obtenus selon les différentes modélisations faite (SAT ou CSP) et les différents solveur utilisés (MiniSat pour SAT, et ACE ou CHOCO pour CSP)

III. RESULTATS OBTENUS

(k, n)	Résultat (avec MiniSat pour la modélisation SAT)	Temps de calcul (SAT)	Résultat (avec CSP, solveur = ACE)	Temps de calcul (CSP)	Résultat (avec CSP, solveur = ACE)	Temps de calcul (CSP)
(3, 20)	SATISFIABLE	0.002119 s	SATISIFIABLE	0.801 s	SATISIFIABLE	0.62 s
(3, 23)	SATISFIABLE	0.003192 s	SATISIFIABLE	0.86 s	SATISIFIABLE	0.9225
(3, 24)	UNSATISFIABLE	0.022463 s	UNSATISIFIABLE	1.286 s	UNSATISIFIABLE	0.987644
(3, 43)	UNSATISFIABLE	0.043326 s	UNSATISIFIABLE	2.035 s	UNSATISIFIABLE	1.77740359 s
(3, 60)	UNSATISFIABLE	0.048144 s	UNSATISIFIABLE	3.175 s	UNSATISIFIABLE	2.3197979 s
(3, 100)	UNSATISFIABLE	0.05152 s	UNSATISIFIABLE	5.933 s	UNSATISIFIABLE	5.0538 s
(4, 60)	SATISFIABLE	8.16977 s	SATISIFIABLE	9.838 s	SATISIFIABLE	1.0770614 s
(4, 66)	SATISFIABLE	574.176 s	TIME OUT		TIME OUT	
(4, 67)	TIME OUT		TIME OUT		TIME OUT	
(4, 100)	SATISFIABLE	8.12223 s	TIME OUT		TIME OUT	
(5, 140)	TIME OUT		TIME OUT		TIME OUT	
(5, 150)	TIME OUT		TIME OUT		TIME OUT	
(5, 160)	TIME OUT		TIME OUT		TIME OUT	
(5, 170)	TIME OUT		TIME OUT		TIME OUT	
(5, 171)	TIME OUT		TIME OUT		TIME OUT	

ANALYSE ET COMMENTAIRES :

Les résultats montrent une certaine robustesse de la part du modèle MiniSat, lors de sa résolution de problème SATISFAISABLE et UNSATISFAISABLE. En effet, pour différentes valeurs pour (k, n), il permettait en général que la résolution se fasse avec un temps de calcul relativement bas.

Par ailleurs, le solveur CSP, présentent tout aussi de bonnes performances, bien que les temps de calculs restent moins bon qu'avec MiniSat.

De manière générale, pour des instances avec des valeurs plus élevée de n et k, nous avons une augmentation significative de la complexité, conduisant à des temps de calcul plus long et donc dans notre cas (avec une limite 600 s.) à des échecs de résolutions.

IV. SYNTHESE : COMPARAISON ENTRE SAT ET CSP

En conclusion, plusieurs remarques émergent de la comparaison entre la modélisation et la résolution du problème de schur à l'aide de la logique propositionnelle (SAT) et de CSP.

MODELISATION : RAPIDITE ET SIMPLICITE

En termes de modélisation du problème de Schur, que ce soit à travers l'écriture ou la programmation, CSP se démarque par sa rapidité et sa simplicité. En effet, il suffit seulement de comparer la taille des textes décrivant la modélisation des problèmes ou encore le code associé pour voir que modéliser un problème en SAT demande plus de réflexion et de temps qu'en CSP.

RESOLUTION : AVANTAGE EN FAVEUR DE SAT

En ce qui concerne la résolution des problèmes, SAT démontre une supériorité globale. Les résultats obtenus dans ce projet indiquent que MiniSat, utilisé pour la modélisation SAT, affiche des performances de résolution rapides et efficaces, même pour des problèmes complexes.

POTENTIEL D'AMELIORATION AVEC CSP

Cependant, il est important de souligner que CSP offre un potentiel d'amélioration significatif dans la modélisation, par ses autres possibilités pour exprimer des contraintes de manière plus efficace. Malgré le fait que je n'ai pas su comment exploiter les contraintes globales vu en cours, je pense qu'avec une meilleure compréhension et une utilisation plus avancée des méthodes de CSP, la modélisation pourrait être optimisée et donc permettre d'obtenir de meilleurs résultats.

CONSIDERATIONS SUR LE MATERIEL ET LES CONDITIONS

Enfin pour les expérimentations, la configuration matérielle repose sur l'utilisation de machines virtuelles de la faculté. La totalité de la partie consacrée au CSP a été développée sur ces machines en raison d'un problème spécifique lié à la bibliothèque Pycsp3. De plus, concernant la résolution du problème avec MiniSat, j'ai privilégié le faire sur un système d'exploitation Linux en utilisant également les VM de la fac. Dans le souci de maintenir la cohérence des résultats, toutes les expériences ont été exécutées le même jour, sur les ordinateurs de la faculté, en veillant à ne pas exécuter d'autres tâches susceptibles d'influencer les performances.

Ainsi, SAT se révèle être la meilleure option pour la résolution du problème de Schur, cependant, CSP présentent tout de même des opportunités d'améliorations dans la modélisation qui pourraient améliorer les performances lors de la résolution.