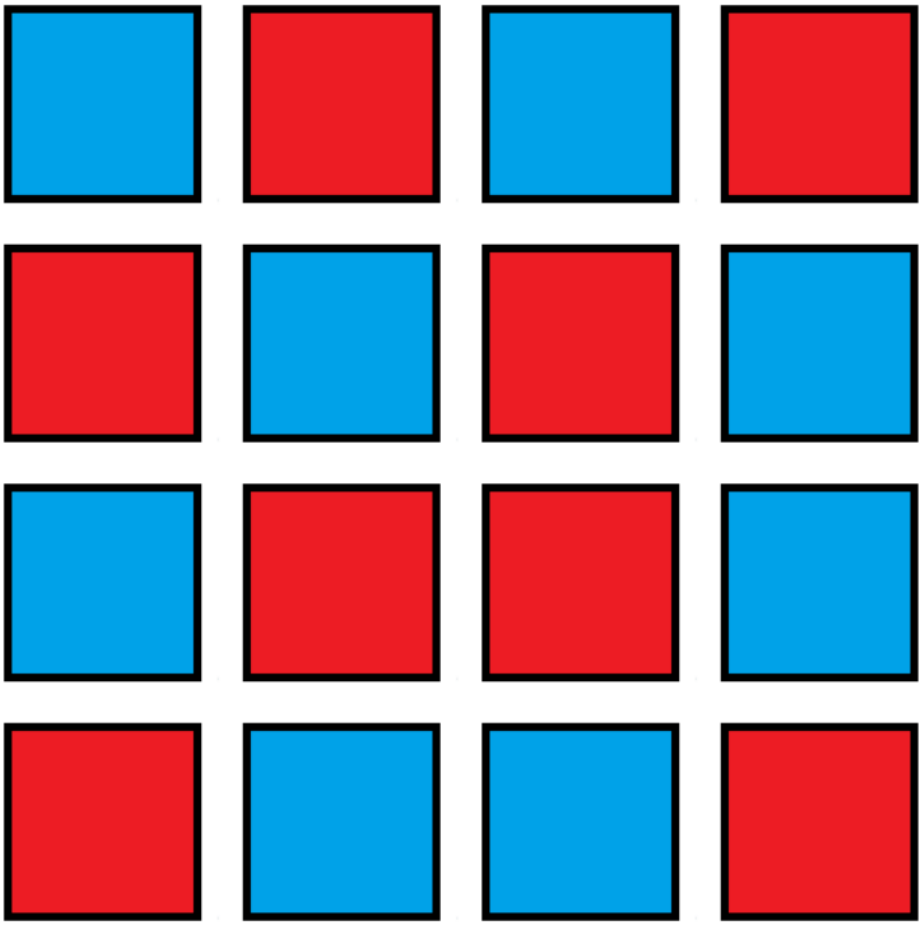
Rapport Binairo

Formalisation et présentation du problème.



Chambe Vivien, Marion Judicaël

07/03/2022

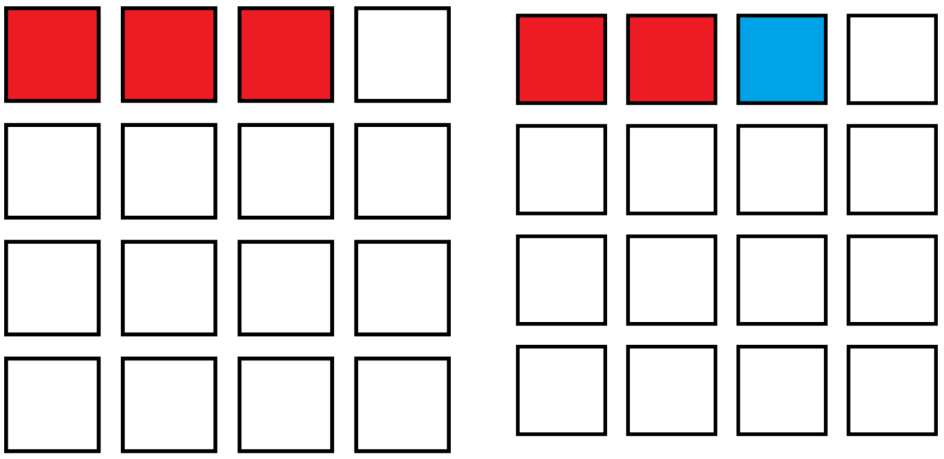
INF 402 – L2 MIN-3

Introduction

Présentation du problème : Le binairo se déroule dans une grille de taille n avec n pair.

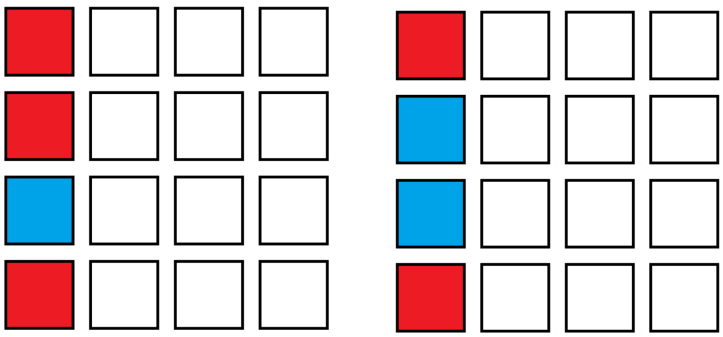
Le jeu commence avec une disposition de cases bleues et rouges dans la grille. Le but est de remplir la grille pour la remplir entièrement de cases de couleur en respectant les règles suivantes:

1. Il ne peut pas y avoir 3 cases consécutives de la même couleur horizontalement ou verticalement:



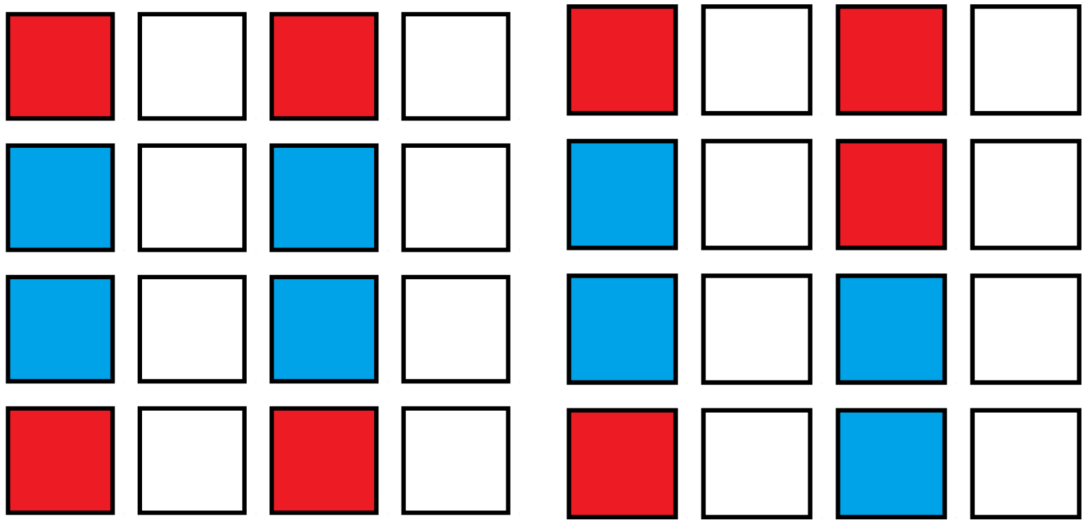
❌ ✅

2. Il doit y avoir le même nombre de cases bleues et rouges sur chaque ligne et chaque colonne:



❌ ✅

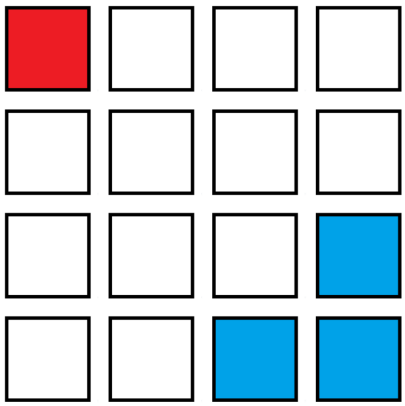
3. Deux colonnes (respectivement deux lignes doivent être différentes)



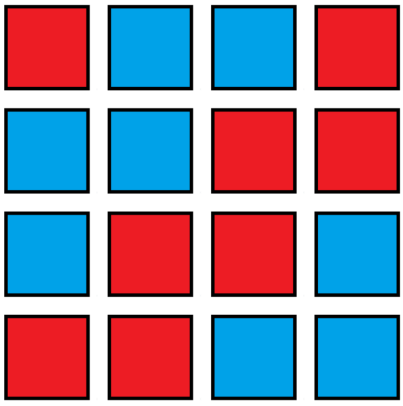
❌ ✅

Et évidemment la grille doit être pleine.

Exemple d'une grille 4x4:



Configuration de base



Grille complétée

Formalisation des règles :

Nous avons décidé de nous débarrasser des relations que l'on avait posées lors du pré rapport car celles-ci nous rendaient la tâche plus difficile que nous le pensions. Nous avons aussi adopté une autre approche pour notre formalisation.

Nos grilles seront dorénavant la concaténation de nos lignes.

Exemple simple : une grille X:

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Deviendra "A B C D E F G H I J K L M N O P" .

"A" voudra dire que la case d'indice A est bleue, tandis que "!A" voudra dire qu'elle est rouge.

<u>3 carrés côtes à côtes ne peuvent pas être de la même couleur:</u>

En reprenant la notation et la grille X énoncée précédemment, on peut écrire:

Pour la première ligne:

AB=>!C (si on a A et B en bleu, C est forcément rouge) = (!A+!B+!C)

!A!B=>C (si on a A et B en rouge, C est forcément bleu) = (A+B+C)

BC=>!D (si on a B et C en bleu, D est forcément rouge)

Pour une grille de largeur n, on aura pour la première ligne :

C1,i C1,i+1 => !C1,i+2 et ce $\forall$i <=n-2 (autrement dit, si une case et la suivante sont de la couleur bleue, alors la suivante de celle-ci est rouge) = !C1,i + !C1,i+1 + !C1,i+2

!C1,i !Ci+1 => Ci+2 et ce ∀i<=n-2 (si une case et sa suivante sont rouge, alors la suivante de celle-ci est bleu) = C1,i + C1,i+1 + C1,i+2

En FNC cela donne : (avec j et i l'indice des lignes et des colonnes respectivement)

∀(j <=n-2),∀(i<=n) (!Cj,i + !Cj,i+1 + !Cj,i+2) & (Cj,i + Cj,i+1 + Cj,i+2)

Avec un raisonnement similaire sur les colonnes, on en déduit que :

Pour une grille de largeur n, on aura:

∀(j <=n),∀(i<=n-2) (!Cj,i + !Cj+1,i + !Cj+2,i) & (Cj,i + Cj+1,i + Cj+2,i)

Donc pour toute la grille on a :

$$\bigwedge_{x=0}^{n-1}\bigwedge_{y=0}^{n-3}(P_{x,y}+P_{x,y+1}+P_{x,y+2})(\overline{P_{x,y}}+\overline{P_{x,y+1}}+\overline{P_{x,y+2}})\bigwedge_{y=0}^{n-1}\bigwedge_{x=0}^{n-3}(P_{x,y}+P_{x+1,y}+P_{x+2,y})(\overline{P_{x,y}}+\overline{P_{x+1,y}}+\overline{P_{x+2,y}})$$

<u>Il doit y avoir le même nombre de cases rouges et de cases bleues sur chaque ligne et colonne:</u>

En reprenant la notation et la grille X énoncée précédemment :

Pour la première ligne on a, par disjonction de cas :

AB => !C!D (Si A et B sont bleu, alors C et D sont rouges = !A + !B + !C!D

!A!B => CD (Si A et B sont rouge, alors C et D sont bleu) = A + B + CD

A!B + !AB => (!CD) + (C!D) (Si, A est bleu et B est rouge ou que A est rouge et B est bleu, alors C est rouge et D est bleu ou C est bleu et D est rouge) = !A!B + AB + !CD +C!D

Même raisonnement pour les colonnes.

$$moitié\_plus\_un = \left\{ \left(x_1, x_2, \cdots, x_{\frac{n}{2}+1}\right) \in N^* \mid \forall ij, x_{i/} = x_j, x_i \leq n \right\}$$

$$\bigwedge_{l=1}^{n} \bigwedge_{X \in moitié\_plus\_un} \left(\bigvee_{x \in X} x \wedge \bigvee_{x \in X} \overline{x} \right)$$

Deux lignes ne peuvent être identiques :

En reprenant la notation et la grille X énoncée précédemment :

ABCD & !(EFGH) & !(IJKL) & !(MNOP) (Si A, B, C et D sont bleu, alors E, F, G, H ne peuvent pas l'être simultanément, idem pour IJKL et MNOP) = A & B & C & D & (!E+!F+!G+!H) & (!I+!J+!K+!L) & (!M+!N+!O+!P)

Même raisonnement pour les colonnes.

Sur une grille de taille n, on a alors:

C1,i & C2,i & … & Cn,i & (!C1,j, + !C2,j + … + ! Cn,j) ∀ i != j

et pour les colonnes :

Ci,1 & Ci,2 & … & Ci,n & (!Ci,1 + !Ci,2 + … + ! Cj,n) ∀ i != j

et donc on obtient la FNC:

∀ i != j

C1,i & C2,i & … & Cn,i & (!C1,j, + !C2,j + … + ! Cn,j) & Ci,1 & Ci,2 & … & Ci,n & (!Ci,1 + !Ci,2 + … + ! Cj,n)

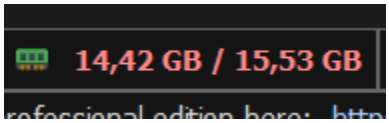
$$\forall i \neq j \left(\prod_{k=1}^{n} Ck, i\, Ci, k \sum_{k=1}^{n} \neg Ck, j \sum_{k=1}^{n} \neg Cj, k \right)$$

Création de fichier dimacs:

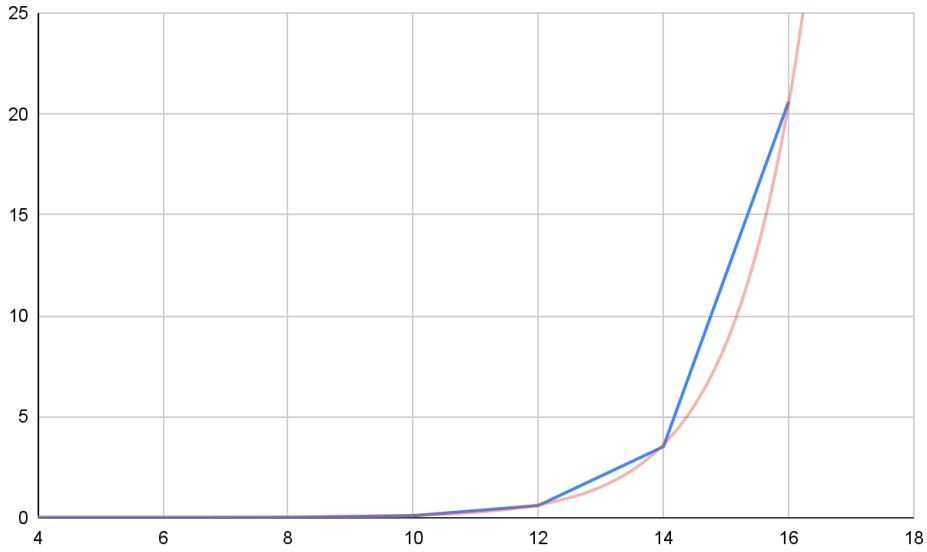
Dans le répertoire only_dimacs nous avons les fonctions où sont implémentées nos règles et leur utilisation pour la création d'un fichier Dimacs. Pour utiliser créer un fichier dimacs pour une grille de taille n il suffit de lancer main.py avec en argument n. On peut également utiliser make tester_dimacs pour produire les .cnf pour des grilles de 4 à 14.

En termes de performances nous pouvons calculer uniquement les fichiers pour des tailles de grille inférieure ou égale à 16 car au-delà nous saturons la mémoire et notre serveur plante. Le fichier dimacs_16.cnf contient un nombre de clauses monstrueux,

3 821 856 exactement ce qui est est déjà 4.81 fois plus que pour n = 14.



Quand bien même nous pourrions avoir plus de ram on peut observer que le temps de calcul augmente très rapidement.



On peut voir qu'entre 14 et 16 le temps est multiplié par 5.8, on peut donc s'attendre à minimum 2 minutes d'exécution (en réalité bien plus car c'est exponentiel et pas linéaire comme le montre la courbe de tendance en rouge.)

Utilisation du sat solver:

Maintenant que nous avons nos fichiers .cnf nous pouvons les donner au solver. Nous avons choisi Picosat à l'origine pour le fait d'avoir une solution de secours si nous n'arrivions pas à faire le dimacs grâce au module python qui est pycosat et qui fait tout de façon interne. L'avantage serait de pouvoir faire nos résolutions sous windows en local ce qui serait bien plus rapide côté interface qu'actuellement en ssh mais cette fluidité visuelle se paierait sur le calcul des solutions en python et donc beaucoup plus lentement que picosat qui est compilé en C .

On peut essayer de chercher une solution unique, ce qui va assez vite pour ne pas dire

instantané. Pour la recherche de toutes les solutions possibles avec l'option [-n] pour n'avoir que le nombre de solutions et [--all] pour les avoir toutes , le calcul prend vite beaucoup de temps. Nous n'avons pas réussi à trouver le nombre de solutions pour n>6 qui a déjà 4140 configurations possibles. Pour palier à ce problème nous pouvons supposer certaines cases déjà remplies et donc réduire le nombre de possibilités.

Nos différents tests nous ont montré que plus le nombre de cases pré remplies tend vers 20% de 64 (donc 13 cases) plus le résultat se rapproche de l'instantané. car il y a forcément moins de configurations possibles.

Cependant pour des grilles plus grandes (10, 12) on estime qu'il faut 25% minimum et dispersées dans la grille pour avoir cette solution instantanée.

Interface graphique:

Maintenant que nous pouvons obtenir une solution, nous voulons l'afficher proprement et lisiblement. Pour cela nous devons transformer une solution de cette forme:

```
s SATISFIABLE
v -1 -2 3 4 -5 6 7 -8 9 10 -11 -12 13 -14 -15 16 0
```

Nous avons donc toutes les configurations possibles dans un fichier resultat.txt au format ci-dessus. Nous utilisons la fonction extract_solution pour récupérer une solution possible puis la convertir pour pouvoir l'utiliser et remplacer les cases vides par la couleur qui correspond.

Pour l'interface graphique nous sommes donc partis sur l'utilisation de Pygame une librairie utilisée pour faire des jeux vidéo en python. Pour lancer le programme il suffit d'utiliser make run_interface et le jeu se lance.

Pour l'utilisation du sat solver nous sommes passé sur une machine sous linux en ssh ce qui rends l'interface graphique assez lente. Nous pourrons montrer la différence lors de la soutenance.

Pour tester une grille il faut utiliser le menu puis choisir son nombre de cases c'est à ce moment là que le fichiers dimacs est créé puisqu'il ne dépend de rien d'autre que n. Pour cela nous appelons main.py vu précédemment.

Nous pouvons alors appuyer sur **S** pour chercher toutes les solutions et prendre la

première qu'il trouve et l'afficher à l'écran. On fait calculer toutes les solutions possibles car notre objectif est de pouvoir tirer une solution différente à chaque fois.

Si l'utilisateur rentre des cases alors la solution proposée sera celle avec des contraintes.

La deuxième partie du programme est la partie jouer qui prend des exemples de grille et qui permet au joueur d'essayer de chercher une solution et d'appuyer sur **V** pour vérifier sa validité dans la console. Mais cette partie relève du bonus car non demandé alors elle est moins développée.

Conclusion:

Nous avons pu réaliser au travers de ce projet la complexité de formaliser un problème en apparence simple. Nous avons aussi vu que même si nous trouvions un algorithme pour représenter une règle il pouvait être bien plus dur de la formaliser en forme logique et de la généraliser que de l'importer dans notre programme. Le fait d'être parti dans la mauvaise direction nous a fait perdre beaucoup de temps et nous a amené à faire des concessions et à revoir notre façon de s'organiser.