Projet INF402 : NoriNori

HEULS Amandine

SELME Kelyan

TERESE Niels

1. Introduction

Ce document présente le projet réalisé dans le cadre de l'INF402. L'objectif est de modéliser un problème type jeu de casse-tête en clause puis de résoudre ces clauses à l'aide d'un SAT-solveur. Nous avons choisi le jeu du Norinori

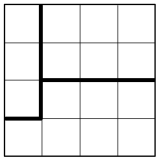
1.1 Rappel des règles

Le jeu du Nornori consiste à remplir une gille carré avec des zones de forme libre en coloriant (ou non) les cases avec les règles suivantes :

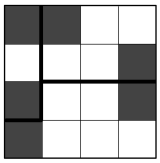
1. Chaque case coloriée doit former des blocs de 2x1 ou 1x2.
2. Deux blocs coloriés ne peuvent pas se toucher entre eux par les côtes (mais ils peuvent par les angles)
3. Chaque zone doit contenir exactement deux cases coloriées noires.

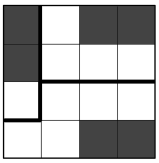
1.2 Exemple

Prenons l'exemple d'une grille de 4x4.



Nous pouvons la résoudre de plusieurs façons, voici deux exemples :





L’exemple est volontairement très simple mais respecte les 3 règles. La difficulté peut être modulée en modifiant la taille de la grille ou le nombre de zones.

2. Modélisation sous forme logique

Afin de trouver une solution au problème, nous devons transformer les règles du Norinori en clauses.

- Soit une grille de taille $n * n$ comportant k zones
- $\forall i \in [1,n], \forall j \in [1,n], \exists k \in [1; \lfloor \frac{n^2}{2} \rfloor] \text{ tel que } x_{i,j,k}$ désigne la case de ligne i et de colonne j et de zone k
- Chaque variable $x_{i,j,k}$ possède une valeur de vérité :
  - 1 : la case $x_{i,j,k}$ est coloriée
  - 0 : la case $x_{i,j,k}$ n’est pas coloriée

Le k désigne la zone à laquelle appartient la case. Plusieurs cases peuvent donc avoir le même indice k. La numérotation des zones k doivent se suivre, il ne peut pas y avoir un k supérieur au nombre de zones. Chaque zone k doit contenir au moins 2 cases.

2.1 Les règles 1 et 2

2.1.1 Logique

Dans cette partie, nous ignorerons volontairement l’indice k car il n’a pas d’utilité.

D’un point de vue logique, ces deux règles sont proches donc peuvent être combinées en une seule et traduites de la manière suivante : “si une case est coloriée alors cette case doit avoir exactement 1 case voisine coloriée qui doit se trouver sur la même ligne ou la même colonne”.

Cette règle impose donc un placement en bloc de 2x1 ou 1x2 et empêche de poser deux blocs qui se touchent par un côté.

Pour chaque case $x_{i,j}$, on obtient alors :

$$x_{i,j} \implies \big((x_{i-1,j} \land \neg x_{i,j+1} \land \neg x_{i+1,j} \land \neg x_{i,j-1}) \lor (\neg x_{i-1,j} \land x_{i,j+1} \land \neg x_{i+1,j} \land \neg x_{i,j-1}) \lor (\neg x_{i-1,j} \land \neg x_{i,j+1} \land x_{i+1,j} \land \neg x_{i,j-1}) \lor (\neg x_{i-1,j} \land \neg x_{i,j+1} \land \neg x_{i+1,j} \land x_{i,j-1})\big)$$

On peut ensuite convertir en forme normale conjonctive :

$$(\neg x_{i,j} \lor \neg x_{i-1,j} \lor \neg x_{i,j-1}) \land (\neg x_{i,j} \lor \neg x_{i-1,j} \lor \neg x_{i,j+1}) \land (\neg x_{i,j} \lor \neg x_{i-1,j} \lor \neg x_{i+1,j}) \land (\neg x_{i,j} \lor x_{i-1,j} \lor x_{i,j-1} \lor x_{i,j+1} \lor x_{i+1,j}) \land (\neg x_{i,j} \lor \neg x_{i,j-1} \lor \neg x_{i,j+1}) \land (\neg x_{i,j} \lor \neg x_{i,j-1} \lor \neg x_{i+1,j}) \land (\neg x_{i,j} \lor \neg x_{i,j+1} \lor \neg x_{i+1,j})$$

2.1.2 Algorithme

```
clauses = []
pour i allant de 1 à n:
  pour j allant de 1 à n:
    #on fait au cas par cas pour gérer les cases qui "dépassent" du plateau
    si i = 1 && j = 1: #case en haut a gauche x_{i-1,j} et x_{i,j-1} n'existent pas
      clauses = clauses + (!x_{i,j}  || x_{i,j+1} || x_{i+1,j}) && (!x_{i,j} || !x_{i,j+1} || !x_{i+1,j})
    sinon si i = 1 && j = n: #case en haut à droite x_{i-1,j} et x_{i,j+1} n'existent pas
      clauses = clauses + (!x_{i,j}  || x_{i,j-1} || x_{i+1,j}) && (!x_{i,j} || !x_{i,j-1} || !x_{i+1,j})
    sinon si i = n && j.= 1: #case en bas a gauche x_{i+1,j} et x_{i,j-1} n'existent pas
      clauses = clauses + (!x_{i,j}  || x_{i,j+1} || x_{i-1,j}) && (!x_{i,j} || !x_{i,j+1} || !x_{i-1,j})
    sinon si i = n && j = n: #case en bas à droite x_{i+1,j} et x_{i,j+1} n'existent pas
      clauses = clauses + (!x_{i,j}  || x_{i,j-1} || x_{i-1,j}) && (!x_{i,j} || !x_{i,j-1} || !x_{i-1,j})
    sinon si i = 1 #cases sur la premiere ligne x_{i-1,j} n'existe pas
      clauses = clauses + (!x_{i,j}  || x_{i,j-1} || x_{i,j+1} || x_{i+1,j}) && (!x_{i,j} || !x_{i,j-1} || !x_{i,j+1}) && (!x_{i,j} || !x_{
    sinon si i = n #cases sur la derniere ligne x_{i+1,j} n'existe pas
```

```
      clauses = clauses + (!x_{i,j}  || x_{i,j-1} || x_{i,j+1} || x_{i-1,j}) && (!x_{i,j} || !x_{i,j-1} || !x_{i,j+1}) && (!x_{i,j} || !x_{
    sinon si j = 1 #cases sur la premiere colonne x_{i, j_1} n'existe pas
      clauses = clauses + (!x_{i,j}  || x_{i+1,j} || x_{i,j+1} || x_{i-1,j}) && (!x_{i,j} || !x_{i+1,j} || !x_{i,j+1}) && (!x_{i,j} || !x_{
    sinon si j = n #cases sur la dernière colonne x_{i, j+1} n'existe pas
      clauses = clauses + (!x_{i,j}  || x_{i+1,j} || x_{i,j-1} || x_{i-1,j}) && (!x_{i,j} || !x_{i+1,j} || !x_{i,j-1}) && (!x_{i,j} || !x_{
    sinon #toutes les autres cases
      clauses = clauses + clause trouvée en 2.1.1
retourner clauses
```

2.2 La règle 3

2.2.1 Logique

Afin de vérifier qu'il existe seulement 2 cases coloriées par zone, nous pouvons prendre toutes les cases d'une zone et vérifier que seulement 2 ont une valeur de vérité "vraie".

D'un point de vue logique, on obtient alors : $(\forall a, \forall b (a \wedge b)) \wedge \neg(a \wedge b \wedge c \wedge d ... \text{ toutes les cases d'une même zone}))$ où $a, b, c, d...$ sont des cases appartenant à la même zone et $a \neq b \neq c \neq d ...$

Exemple pour une zone k comportant 3 cases :

$((a \wedge b) \vee (a \wedge c) \vee (b \wedge c)) \wedge \neg(a \wedge b \wedge c)$ avec a, b et c les 3 cases de la zone k

Exemple pour une zone k comportant 4 cases :

$((a \wedge b) \vee (a \wedge c) \vee (a \wedge d) \vee (b \wedge c) \vee (b \wedge d) \vee (c \wedge d)) \wedge \neg(a \wedge b \wedge c) \wedge \neg(a \wedge b \wedge d) \wedge \neg(a \wedge c \wedge d) \wedge \neg(b \wedge c \wedge d)$

On peut ensuite convertir sous forme normale conjonctive :

$(\neg a \vee \neg b \vee \neg c) \wedge (\neg a \vee \neg b \vee \neg d) \wedge (\neg a \vee \neg c \vee \neg d) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (a \vee b \vee c) \wedge (a \vee b \vee d) \wedge (a \vee c \vee d) \wedge (b \vee c \vee d)$

Et ainsi de suite….

On effectue ensuite la conjonction de toutes les clauses générées pour chaque zone pour $k \in [\![1; \lfloor \frac{n^2}{2} \rfloor]\!]$. Nous avons donc désormais toutes les clauses de la règle 3.

2.2.2 Algorithme

Cet algorithme génère directement les clauses sous formes normales conjonctives :

```
        zones = nombre de zones totales
        clauses = []
        pour chaque zone dans zones :
            casesInZone = les cases dans la zone actuelle
            nb_zone = nombre de cases dans la zone actuelle
            combs = combinaisons(range(1, nb_zone  + 1), nb_zone  - 1)
            for comb in combs:
                temp = []
                for j in comb:
                    ajouter casesInZone[j - 1] dans temp
                ajouter tableau temp dans clauses

            combs = combinaisons(range(1, nb_zone  + 1), 3)
            for comb in combs:
                temp = []
                for j in comb:
                    ajouter !casesInZone[j - 1] dans temp
                ajouter tableau temp dans clauses
        retourner clauses
```

combinaisons(x, y) renvoie les combinaisons distinctes comportantes de y nombres parmi x

Exemple : combinaisons($[0, 1, 2], 2$) renvoie : $(0, 1), (0, 2), (1, 2)$

3. Modélisation informatique

3.1 Génération de fichier DIMACS et résolution

Pour modéliser informatiquement notre problème, nous avons décidé d'utiliser Python.

L'implémentation est constituée de deux modules :

- grid.py

  Permet l'initialisation d'une grille, ainsi que diverses méthodes :

  - **setCellValueZone(**i**,** j**,** k**)** : met la case $x_{i,j}$ dans la zone k
  - **setCellValueColor(**i**,** j**,** k**)** : colorie la case $x_{i,j}$

- rule.py

  Permet l'initialisation du modèle logique selon une certaine grille, ainsi que diverses methodes :

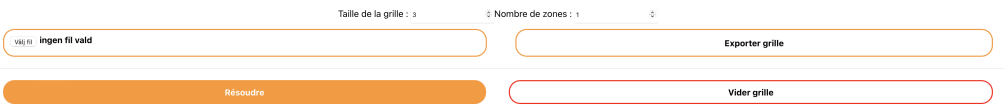
  - **resolve()** : génère les clauses des 3 règles, et les convertit au format 3-SAT
  - **generateDimacs(**nom**)** : génère un fichier dimacs nommé nom contenant toutes les clauses de notre problème

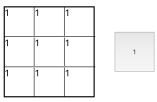
Nous utilisons le SAT-solver Glucose3

3.2 Interface graphique

3.2.1 Composition visuelle

Pour faciliter la saisie des caractéristiques d'une grille ainsi que pour avoir une représentation visuelle du résultat, nous avons construit une interface graphique.





L'interface se compose de 5 éléments principaux :

1. Choix des caractéristiques de la grille : sa taille et son nombre de zones.

   Il est possible d'augmenter ou de diminuer la taille de la grille, le chiffre entré représentant n le nombre de cases sur une largeur/longueur de la grille

   Il est possible d'augmenter ou de diminuer le nombre de zones k.

2. Importer ou exporter un fichier contenant une grille (et éventuellement sa solution)

   L'export d'une grille exporte la grille telle qu'elle est affichée sur l'écran : avec les zones définies si définies, ainsi que la solution si la grille à été résolue

   L'import d'une grille importe la grille telle qu'elle a été exportée

3. Vider ou résoudre la grille

   La résolution d'une grande grille peut prendre quelques secondes. Avant de cliquer sur résoudre, les conditions suivantes doivent être réunies :

   - Le serveur Flask est actif
   - Chaque zone doit avoir au minimum 2 cases

4. La grille

   Après résolution, les cases à colorier pour résoudre le problème se colorient en violet.

5. Le choix d'une zone d'une case

   Après un clic sur une case, on peut choisir dans quelle zone se situe la case à l'aide des boutons à droite. Le contour de la zone se dessine alors.

   Lorsque qu'une case est sélectionnée pour lui affecter une zone, elle est coloriée en bleu clair. Un appui sur "Esc" la désélectionne.

3.2.2 Composition technique

L'interface graphique est réalisée en React JS.

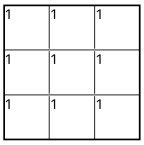
Comme notre modèle logique est modélisé en Python, nous utilisons Flask pour créer une communication entre l'interface graphique et notre module Python.

Il crée un mini serveur web/api. Par exemple, pour résoudre la grille, l'interface graphique envoie via le serveur web la grille dessinée et le module Python renvoie la solution, qui s'affiche alors sur l'interface graphique

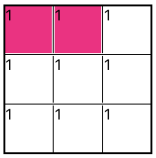
L'interface graphique s'ouvre via *gui>index.html* et requièrt le lancement de *server.py*

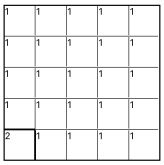
4. Exemples

Divers exemples qui illustrent le bon fonctionnement de notre modèle logique et son implémentation

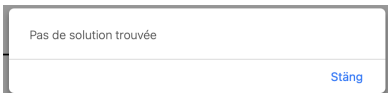


→ Clic sur "Résoudre" →

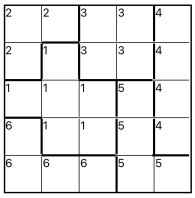




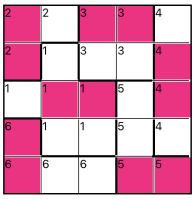
→ Clic sur "Résoudre" →



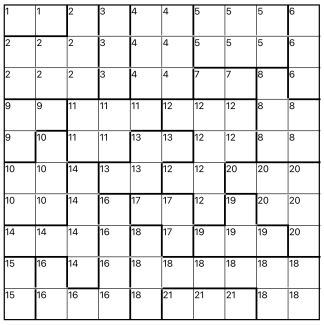
Un pop-up s'affiche : pas de solution trouvée car il y a une zone avec seulement une case

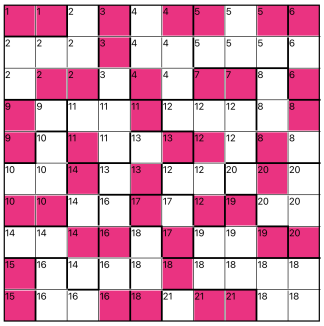


→ Clic sur "Résoudre" →



→ Clic sur "Résoudre" →





Tous ces exemples peuvent être retrouvés dans l’archive pour être importés et testés via l’interface graphique.

5. Conclusion

Nous obtenons une solution valide sur différents exemples. Nous en concluons que notre modèle logique était correct, ainsi que son implémentation informatique.