Rapport projet

Pascal Isak & Givre Zolan & Weber Loïc

Table of Content

Information et dates.

Dates :

1. pré-rapport décrivant la partie modélisation : 10 mars 2023
2. Rapport final + code source 28 avril 2023

Installation

Premièrement, il faut installer **opam, dune, ocaml et minisat**.
https://opam.ocaml.org/doc/Install.html
https://dune.build/install

Deuxièmement, placez vous dans le dossier *norinori/bin/* et assurez-vous que les sous-dossiers *INSTANCES/*, *DIMACS/* et *RESULTATS/* existent.

Troisièmement, exécutez `'dune exec norinori test'`
vous pourrez remplacer "test" par le nom d'un fichier qui se trouve dans le dossier *INSTANCES/*

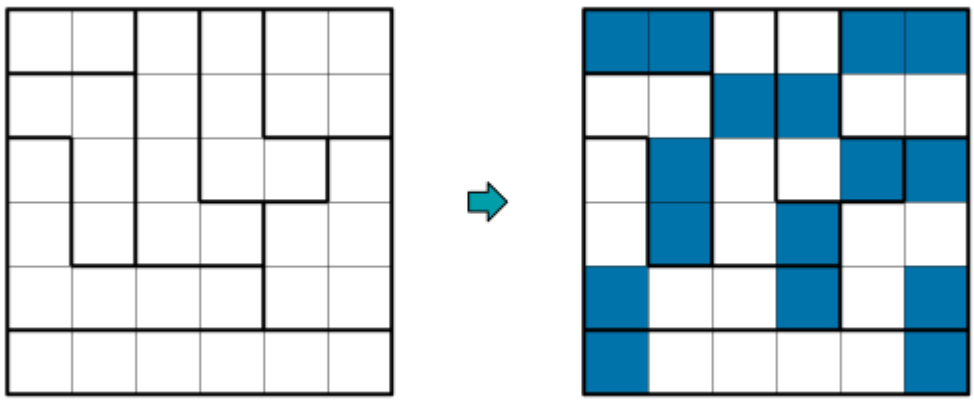
Ainsi vous pourrez trouver :

- Dans INSTANCES : Les grilles dans le format .liz (format que nous avons créé).
- Dans DIMACS : Les fichiers en .dimacs : la traduction des fichiers .liz dans le format DIMACS.
- Dans RESULTATS :
  - Les fichiers .res : la solution trouvée par minisat.
  - Les fichiers .visu : la grille avec la solution.

Rapport.

Presentation du jeu.

Le jeu que nous avons choisi est NoriNori.



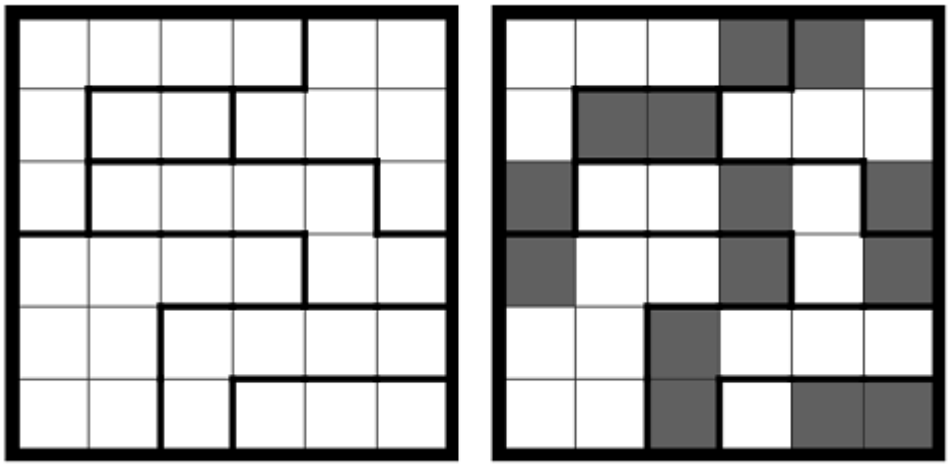
Le jeu commence avec une grille de X cases dans laquelle est préalablement défini un certain nombre de zones distinctes.

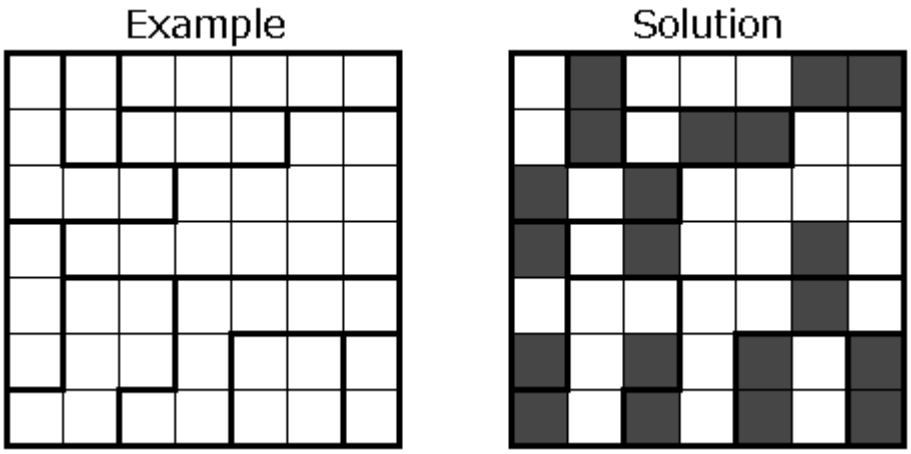
Règles.

Le but du jeu est de compléter la grille en suivant ces 3 règles:

1. Chaque zone doit avoir un total de 2 cases coloriées.
2. Les cases coloriées sont toujours par pair de 2 indépendament des limites des zones.
3. On ne peux pas coller cote a cote deux pairs (mais on peut diagonalement).

Exemples.





Contraintes & Remarques.

Il y a une unique solution si la grille est générée correctement.

Nous avons remarquée plusieurs cas qui permette d'avancer dans la résolution de la grille :

- Les zones constituées d'exactement deux cases sont toujours a coloriées elles feront toujours partie de la solution finale.
- Toutes les cases cote à cote aux cases coloriées deviennent donc inutilisable car ne respectent plus la règle N°3
- Toutes les cases seules et entourées par des cases inutilisable deviennent aussi inutilisable.
- Les zones dans lequel il ne reste plus que deux cases peuvent être coloriées.
- Ensuite on peut compléter les cases coloriées qui ne sont pas par pair et qui n'ont plus que une seule solution.

Et après on peut réutiliser les étapes précédentes jusqu'à ce que notre grille soit complétée.
(Attention ces étapes ne sont pas exhaustive, il existe très probablement des grilles dans lequel appliquer ces étapes ne suffisent pas.)

Traduction du problème en logique.

Premièrement nous avons traduit simplement le fait de dire qu'une case est coloriée ou non avec n étant la taille de la grille.

$$(i,j) : \text{la case de coordonées (i,j) est coloriée}$$
$$(i,j) \in [1,n]^2$$

$$\neg(i,j) : \text{la case de coordonées (i,j) n'est pas coloriée}$$
$$(i,j) \in [1,n]^2$$

Formalisation des Règles 2) et 3)

Rappel :
2) Les cases coloriées sont toujours par paire de 2 indépendamment des limites des zones.
3) On ne peux pas coller côte a côte deux paires (mais on peut diagonalement).

Pour modéliser le problème, on commence par fusionner les règles 2) et 3).
Si une case est coloriée, alors elle a une et une seule case coloriée à coté d'elle.

On va simplifier "alors elle a une et une seule case coloriée à coté d'elle" en deux règles.

1. Elle a au moins une case coloriée a coté d'elle (qui se traduit simplement avec une disjonction)
2. Elle a au plus une case coloriée a coté d'elle (qui se traduit en logique comme: "elle n'a aucune paire de case adjacente coloriée")

$$(1)$$

$$(i+1,j) \vee (i-1,j) \vee (i,j+1) \vee (i,j-1)$$

$$(2)$$

$$\begin{aligned}&\neg((i+1,j) \wedge (i-1,j))\\ \wedge&\neg((i+1,j) \wedge (i,j+1))\\ \wedge&\neg((i+1,j) \wedge (i,j-1))\\ \wedge&\neg((i-1,j) \wedge (i,j+1))\\ \wedge&\neg((i-1,j) \wedge (i,j-1))\end{aligned}$$

$$\wedge\neg((i,j+1)\wedge(i,j-1))$$

$$\equiv$$

$$\begin{array}{c}(\neg(i+1,j)\vee\neg(i-1,j))\\ \wedge(\neg(i+1,j)\vee\neg(i,j+1))\\ \wedge(\neg(i+1,j)\vee\neg(i,j-1))\\ \wedge(\neg(i-1,j)\vee\neg(i,j+1))\\ \wedge(\neg(i-1,j)\vee\neg(i,j-1))\\ \wedge(\neg(i,j+1)\vee\neg(i,j-1))\end{array}$$

la règle est donc:

$$(i,j)\Rightarrow(1)\wedge(2)$$

$$\equiv$$

$$\begin{array}{c}(i,j)\Rightarrow((i+1,j)\vee(i-1,j)\vee(i,j+1)\vee(i,j-1))\\ \wedge(\neg(i+1,j)\vee\neg(i-1,j))\\ \wedge(\neg(i+1,j)\vee\neg(i,j+1))\\ \wedge(\neg(i+1,j)\vee\neg(i,j-1))\\ \wedge(\neg(i-1,j)\vee\neg(i,j+1))\\ \wedge(\neg(i-1,j)\vee\neg(i,j-1))\\ \wedge(\neg(i,j+1)\vee\neg(i,j-1))\end{array}$$

on applique $A\Rightarrow B\equiv\neg A\vee B$

$$\begin{array}{c}\equiv\neg(i,j)\vee\big[\\ ((i+1,j)\vee(i-1,j)\vee(i,j+1)\vee(i,j-1))\\ \wedge(\neg(i+1,j)\vee\neg(i-1,j))\\ \wedge(\neg(i+1,j)\vee\neg(i,j+1))\\ \wedge(\neg(i+1,j)\vee\neg(i,j-1))\\ \wedge(\neg(i-1,j)\vee\neg(i,j+1))\\ \wedge(\neg(i-1,j)\vee\neg(i,j-1))\\ \wedge(\neg(i,j+1)\vee\neg(i,j-1))\\ \big]\end{array}$$

Donc finalement :

$$\begin{array}{c}[\neg(i,j)\vee(i+1,j)\vee(i-1,j)\vee(i,j+1)\vee(i,j-1)]\\ \wedge[\neg(i,j)\vee\neg(i+1,j)\vee\neg(i-1,j)]\\ \wedge[\neg(i,j)\vee\neg(i+1,j)\vee\neg(i,j+1)]\\ \wedge[\neg(i,j)\vee\neg(i+1,j)\vee\neg(i,j-1)]\\ \wedge[\neg(i,j)\vee\neg(i-1,j)\vee\neg(i,j+1)]\\ \wedge[\neg(i,j)\vee\neg(i-1,j)\vee\neg(i,j-1)]\\ \wedge[\neg(i,j)\vee\neg(i,j+1)\vee\neg(i,j-1)]\end{array}$$

Il faut donc appliquer cela pour chaque case.
N'oublions pas le cas spécial des cases au bord de la grille. Deux approches possibles:

1. Modifier la règle des cases du bord pour que la case colorié adjacente soit sur le terrain.
2. Garder la même règle pour les cases au bord et rajouter une ligne de cases autour du terrain.

Nous avons choisis de partir sur la première options, en ajoutant quelques cas particulier dans le code.

Formalisation Règle 1)

note: $\{C_j \mid 0 \leq j < n\}$ *est l'ensemble des cases de la zone z de n éléments*

Exemple pour comprendre règle 1)

Soit z une zone de 5 éléments {A,B,C,D,E}.

On sait qu'il doit y avoir exactement 2 cases coloriées dans la zone :
Une stratégie pour modéliser cette règle est de `divide et impera` :

On sépare la règle "exactement 2" en : Au moins 2 $\wedge$ Au plus 2.
(On fait ça, car on a essayé de directement faire une règle "exactement 2" mais elle était beaucoup trop longue)

Au moins 2 :

$$(A \vee B \vee C \vee D) \wedge (A \vee B \vee C \vee E) \wedge (A \vee B \vee D \vee E) \wedge (A \vee C \vee D \vee E) \wedge (B \vee C \vee D \vee E)$$

Au plus 2 :

$$\begin{gathered}\neg(A \wedge B \wedge C) \wedge \neg(A \wedge B \wedge D) \wedge \neg(A \wedge B \wedge E) \\ \wedge\neg(A \wedge C \wedge D) \wedge \neg(A \wedge C \wedge E) \wedge \neg(A \wedge D \wedge E) \\ \wedge\neg(B \wedge C \wedge D) \wedge \neg(B \wedge C \wedge E) \wedge \neg(B \wedge D \wedge E) \\ \wedge\neg(C \wedge D \wedge E)\end{gathered}$$

$$\equiv$$

$$\begin{gathered}(\neg A \vee \neg B \vee \neg C) \wedge (\neg A \vee \neg B \vee \neg D) \wedge (\neg A \vee \neg B \vee \neg E) \\ \wedge(\neg A \vee \neg C \vee \neg D) \wedge (\neg A \vee \neg C \vee \neg E) \wedge (\neg A \vee \neg D \vee \neg E) \\ \wedge(\neg C \vee \neg D \vee \neg E)\end{gathered}$$

Cas général Règle 1):

Nous devons maintenant généraliser la règle et nous proposons de l'écrire comme suit:

Au moins 2

$$\bigwedge_{i=1}^{n}\Big(\bigvee_{\substack{j=1\\ j\neq i}}^{n}(C_j)\Big)$$

Preuve que cette formule au moins 2 est correct (facultatif)

Preuve que la formule fonctionne :
Supposons qu'il existe C_a et C_b vrai. on a :

$$\bigwedge_{i=1}^{n}\Big(\bigvee_{\substack{j=1\\ j\neq i}}^{n}(C_j)\Big) \equiv \Big(\bigvee_{\substack{j=1\\ j\neq a}}^{n}(C_j)\Big) \wedge \bigwedge_{\substack{i=1\\ i\neq a}}^{n}\Big(\bigvee_{\substack{j=1\\ j\neq i}}^{n}(C_j)\Big)$$
$$\equiv \Big(\Big(\bigvee_{\substack{j=1\\ j\neq a,b}}^{n} C_j\Big) \vee \underbrace{C_b}_{\text{vrai}}\Big) \wedge \bigwedge_{\substack{i=1\\ i\neq a}}^{n}\Big(\bigvee_{\substack{j=1\\ j\neq i,a}}^{n}(C_j) \vee \underbrace{C_a}_{\text{vrai}}\Big) \equiv \top$$

Au plus 2

$$\bigwedge_{\substack{i,j,k=1\\ i\neq j\\ i\neq k\\ j\neq k}}^{n}\neg\big(C_i \wedge C_j \wedge C_k\big)$$
$$\equiv$$
$$\bigwedge_{\substack{i,j,k=1\\ i\neq j\\ i\neq k\\ j\neq k}}^{n}\big(\neg C_i \vee \neg C_j \vee \neg C_k\big)$$

Modelisation sous forme normale conjonctive.

Donc au final, quand on regroupe toutes nos clauses, on a :

$$\begin{gathered}[\neg(i,j)\vee(i+1,j)\vee(i-1,j)\vee(i,j+1)\vee(i,j-1)]\\ \wedge[\neg(i,j)\vee\neg(i+1,j)\vee\neg(i-1,j)]\\ \wedge[\neg(i,j)\vee\neg(i+1,j)\vee\neg(i,j+1)]\\ \wedge[\neg(i,j)\vee\neg(i+1,j)\vee\neg(i,j-1)]\\ \wedge[\neg(i,j)\vee\neg(i-1,j)\vee\neg(i,j+1)]\\ \wedge[\neg(i,j)\vee\neg(i-1,j)\vee\neg(i,j-1)]\\ \wedge[\neg(i,j)\vee\neg(i,j+1)\vee\neg(i,j-1)]\\ \wedge\\ \bigwedge_{i=1}^{n}\Big(\bigvee_{\substack{j=1\\ j\neq i}}^{n}(C_j)\Big)\\ \wedge\end{gathered}$$

$$\bigwedge_{\substack{i,j,k=1 \\ i \neq j \\ i \neq k \\ j \neq k}}^{n} \left(\neg C_i \vee \neg C_j \vee \neg C_k \right)$$

Les programmes.

Formalisation d'un problème dans un fichier.

Nous avons choisis la structure de fichier suivant pour modéliser nos grilles :

fichier.liz

```
hauteur(nombre entier positif);
largeur(nombre entier positif);
nb_zone(nombre entier positif non nulle);
numero_zone_case_1_1;.............;numero_zone_case_L_1;
..................................................;
.................numero_zone_case_i_j.................;
..................................................;
numero_zone_case_1_H;.............;numero_zone_case_L_H;

(Attention les \n ne sont pas obligatoire pour le format du fichier donc on peut tout écrire sur u
```

Exemple :

ma_grille_nori_nori.liz

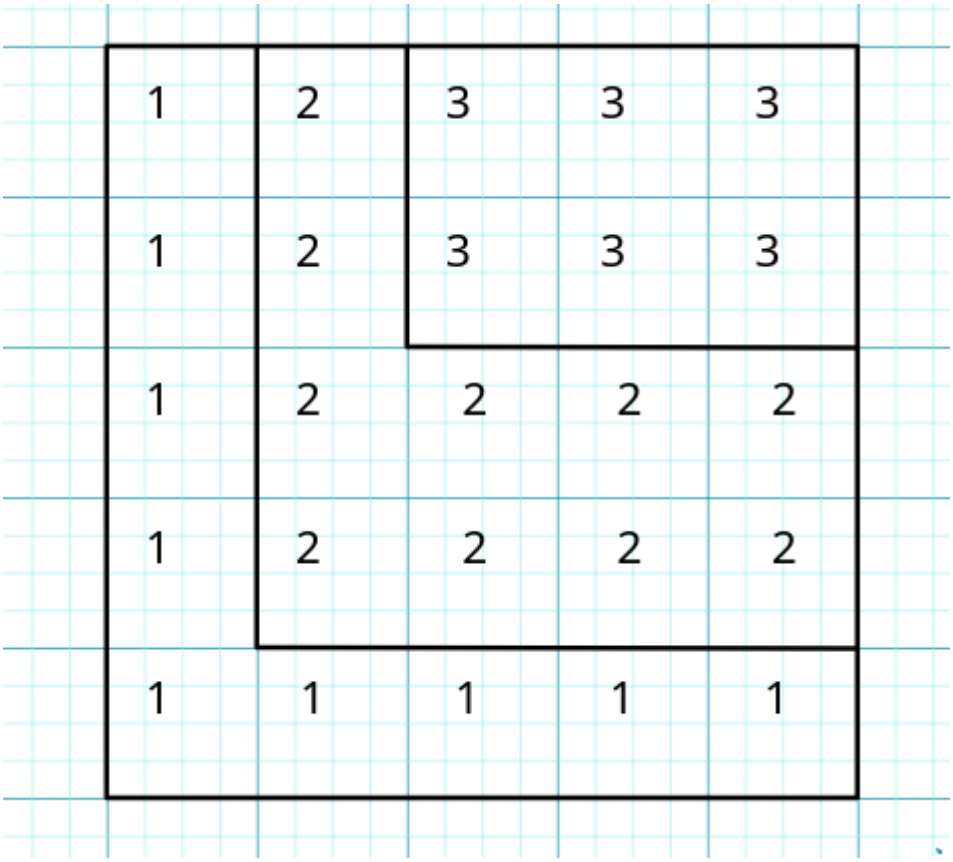
```
5;
5;
3;
1;2;3;3;3;
1;2;3;3;3;
1;2;2;2;2;
1;2;2;2;2;
1;1;1;1;1;
```

ou bien :

ma_grille_nori_nori2.liz

```
5;5;3;1;2;3;3;3;1;2;3;3;3;1;2;2;2;2;1;2;2;2;2;1;1;1;1;1;
```

Ce qui donne la grille suivante :



Parsing du fichier en Python :

parsing_fichier_liz.py

```
import sys #Pour recuperer le nom du fichier a parse.

def parsing_fichier(fichier):
  # On ouvre le fichier
  f = open(fichier)
  lines_in_list = f.readlines()
  f.close()

  line = ""
  # On concatene toutes les lignes pour n'en faire que une.
  for elem in lines_in_list:
      line += elem

  c = 0
  buffer = ""
  info_grille = []
  nbr_separator_seen = 0

  # Parsing des info de la grille info_grille = [largeur, hauteur, nb_zone]
  while nbr_separator_seen < 3:
      if line[c] == ';':
          nbr_separator_seen += 1
          info_grille.append(int(buffer))
          buffer = ""
      else:
          buffer += line[c]

      c += 1
```

```python
    # On recupere les information de la grille dans des variables
    largeur = info_grille[0]
    hauteur = info_grille[1]
    nb_zones = info_grille[2]
    #print(info_grille)

    # On crée une matrice vide de la taille de notre grille
    matrix = [[0 for y in range(hauteur)] for x in range(largeur)]

    # On parse le fichier dans la matrice
    i, j = 0, 0
    for c in range(c, len(line)-1):

        if j == largeur:
            j = 0
            i += 1

        if line[c] == ';':
            matrix[i][j] = int(buffer)
            buffer = ""
            j += 1

        else:
            buffer += line[c]

    #print(matrix)
    return info_grille,matrix

if __name__ == "__main__":
    info_grille, matrix = parsing_fichier(sys.argv[1])
    print(info_grille)
    print(matrix)
```

On peut l'éxecuter en faisant : `python parsing_fichier_liz.py ma_grille.liz`

Parsing en Ocaml :

parsing_fichier_liz.ml

```
type largeur = int;;
type hauteur = int;;
type nb_zones = int;;

type case = int*int;;
type list_zone = case list list;;
type jeu = largeur*hauteur*nb_zones*list_zone;;

type etat_automate = Largeur|Hauteur|Zone|Fin|CS of (int*int);; (*CS((x,y),liste_de_collones,colo
type mem_automate = jeu*int*etat_automate;;

let print_jeu ((larg, haut, zone, l): jeu) =
    (*
    | SPÉCIFICATION
    | print_jeu : affiche dans le terminal le jeu.
    | - Profil : append_n_eme : element -> int -> 'a list list -> 'a list list
```

```
    | - Sémantique : (append_n_eme element int ('a list list) ('a list list)) :
    | - Exemple et propriétés :
    |   (a) automate (5) (2) ([[0;1];[3;4];[]]):  [[0;1];[3;4];[5]]
    |   (b) automate ('u') (0) ([couco]):  [[coucou]]
    | - Implémentation :
*)
    let p_iteri (index: int) (l: case list) =
        let () = Printf.printf "    %d: [ " (index+1) in
        let () = (List.iter (fun (x,y) -> Printf.printf "(%d,%d); " x y) l ) in
        let () = Printf.printf "]\n" in
        ()
    in
    let () = Printf.printf "largeur: %d\nhauteur: %d\nnombre de zones: %d\ndictionnaire de zone:\
    let () = List.iteri p_iteri l in
    ()
;;

let rec append_n_eme (element: 'a) (n: int) (pr::fin: 'a list list): 'a list list =
    (*

    | SPÉCIFICATION
    | append_n_eme : rajoute un élément à la n eme liste de la liste de liste donné en argument.
    |                /!\ premier élément d'indice 0.
    |                /!\ la liste de liste donné doir avoir au moins n+1 listes (pour pouvoir in
    | - Profil : append_n_eme : element -> int -> 'a list list -> 'a list list
    | - Sémantique : (append_n_eme element int ('a list list) ('a list list)) :
    | - Exemple et propriétés :
    |   (a) automate (5) (2) ([[0;1];[3;4];[]]):  [[0;1];[3;4];[5]]
    |   (b) automate ('u') (0) ([couco]):  [[coucou]]
    | - Implémentation :
*)
    match n with
    | 0 -> (element::pr)::fin
    | n -> pr::(append_n_eme element (n-1) fin)
    [@@warning "-8"]
;;

let automate (((larg, haut, zone, gril), nb, etat):mem_automate) (carac:char): mem_automate=
(*
    | SPÉCIFICATION
    | automate : a utiliser dans un String.fold_left, parse le fichier.
    | - Profil : automate : mem_automate -> char -> mem_automate
    | - Sémantique : (automate  mem_automate  char  mem_automate) : la fonction modifie mem_automa
    | - Exemple et propriétés :
    |   (a) automate ((0, 0, 0, []), 15, Largeur) (';'): ((15, 0, 0, []), 0, Hauteur)
    |   (b) automate ((15, 0, 0, []), 2, Hauteur) ('4'): ((15, 0, 0, []), 24, Hauteur)
    | - Implémentation :
*)
    match carac,etat with
    | ';',Largeur ->    ((nb(*<- enregistre la largeur ici*), haut, zone, gril), 0, Hauteur)
    | ';',Hauteur ->    ((larg, nb, zone, gril), 0, Zone)
    | ';',Zone ->       ((larg, haut, nb, List.init nb (fun x -> [])), 0, CS(1,1))
    |  _ ,Fin ->        ((larg, haut, zone, gril), 0, Fin)
    | ';',CS((x,y)) -> if (x == larg)
                       then if (y == haut)
                            then ((larg, haut, zone, (append_n_eme (x,y) (nb-1) gril)), 0, Fin)
                            else ((larg, haut, zone, (append_n_eme (x,y) (nb-1) gril)), 0, CS(1,y
                       else ((larg, haut, zone, (append_n_eme (x,y) (nb-1) gril)), 0, CS(x+1,y))
    | a,etat -> (*etat!=Fin*) if (((Char.code a) >= 48) && ((Char.code a) <= 57))
       then ((larg, haut, zone, gril), nb*10 + (Char.code a) - 48 (*ascii(48) = '0'*) , etat)
        else ((larg, haut, zone, gril), nb, etat) (* si le carac n'est pas reconus en tant que ch
```

```
;;

let lire_fichier (file: string) : jeu =
    (*
    | SPÉCIFICATION
    | automate : permet douvrir un fichier, de le parser et retourner le jeu corespondant.
    | - Profil : automate : string -> jeu
    | - Sémantique : (lire_fichier string) : récupère toutes les lignes puis applique un String.fo
    | - Implémentation :
*)
    let lines_array = Array.to_list (Arg.read_arg file) in (*on recupere les lignes*)
    let lines =  String.concat "" lines_array in (*on concatene toutes les lignes*)
    let (mon_jeu, _, _) = (String.fold_left (automate) ((((0,0,0,[]): jeu), 0, Largeur):mem_autor
    mon_jeu (*return*)
;;

let file:string = ( if (Array.length Sys.argv == 2)
then Sys.argv.(1) (*le fichier est donné en argument*)
else "test.liz");; (*nom de fichier pas defaut*)

let mon_jeu = lire_fichier file;; (* on parse le fichier *)

print_jeu mon_jeu;; (*affichage de jeu*)
```

Pour la suite du projet, nous avons choisis de continuer en Ocaml.

Formalisation d'une entrée en logique propositionnelle.

Maintenant que nous avons une matrice en Python ou bien un dictionnaire en Ocaml représentant notre instance, nous devons la transformée en forme de logique propositionnelle.

Règle 1)

Nous avons réalisé une fonction récursive "dimacs_case_de_colonnes" qui s'occupe de parcourir une colonnes et d'appliquer la règle 1 sur chaque case de la colonne.
Elle est appelée par "dimacs_colonnes"

Règle 2) & 3)

Pour la règle 2 on appelle la fonction `au_plus_2` et la fonction qui utilise l'itérateur.

Conversion d'une instance au format DIMACS.

Pour convertir dans le fichier DIMACS, on fait simplement des Printf.fprintf dans le fichier .dimacs en même temps que notre formalisation.

Afficheur de solution.

Notre afficheur de solution se trouve dans le fichier `visuel_resultat.ml` il prend un fichier .res donnée par minisat et va afficher un tableau dans lequel il va mettre des `.` pour des cases vide et des `#` pour des cases coloriées.

Programme principale qui utilise les autre progs.

Notre programme principale est main.ml il commence par récupérer les arguments puis crée les futures noms des fichiers .res, .dimacs et .visu ensuite il fait la formalisation et la transformation en DIMACS par la même occasion, après ça, il lance minisat avec le fichier .dimacs et enfin il utilise la solution de minisat pour créer le fichier .visu.

Les instances tests.

instance basique problème simplifiée.

grille_simple.liz

```
6;6;8;
1;1;2;2;2;3;
1;1;1;1;2;3;
4;1;1;1;5;5;
4;1;1;6;6;5;
1;1;6;6;6;6;
8;8;6;7;7;7;
```

Représente une grille basique de 6 par 6 avec 8 zones.

instance normal avec tout les cas de figure.

semi_grosse_grille.liz

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

Représente une grille plus complexe avec tous les cas de figures que nous avons observé.

instances problématique avec cas de figure critiques.

grille_1_case.liz

```
1;1;1;1
```

Représente une grille de une seule case, qui est censé ne pas avoir de solutions.

grille_2_cases.liz

```
2;1;1;1;1
```

Une grille avec deux cases, soit une seule solution unique.

instance avec un nombre de clause énorme.

grille_immense_mais_simple.liz

```
25;25;1;
1;1;1;1;.........;1;
.                  .
.                  .
.                  .
1;1;1;1;.........;1;
```

On essaye avec une très grosse grille pour voir si notre système est capable de résoudre malgré un nombre de clause énorme (40 millions).

Soutenance.

A faire ...

Présentation du jeu.

Explication transformation des règles en logique.

Explication des algorithmes.

Démonstration des algorithmes.

Conclusion.