

# Rapport de projet Picross

CONNES Victor      PRYSIAZHNIUK Anastasiia  
BENAIS-HUGOT Charles      RIVIERE Cecilie

26 avril 2015

## Table des matières

<b>1</b>	<b>Description du problème</b>	<b>3</b>
<b>2</b>	<b>Description de la méthode de résolution utilisée</b>	<b>4</b>
2.1	Résolution d'une ligne . . . . .	4
2.1.1	La méthode SLG . . . . .	5
2.1.2	Les méthodes SLPG et SLPD . . . . .	5
2.1.3	La méthode Fusion . . . . .	5
2.1.4	La méthode remplirCasesSureBl . . . . .	6
2.2	Résolution du Picross . . . . .	6
2.2.1	la méthode de Backtracking . . . . .	6
<b>3</b>	<b>Description de l'implémentation</b>	<b>7</b>
3.1	Les besoins du programme : . . . . .	7
3.1.1	Pour les indices de chaque ligne/colonne : . . . . .	7
3.1.2	Pour représenter la grille du Picross : . . . . .	7
3.1.3	Pour l'ensemble des indices de lignes/colonnes : . . . . .	7
3.1.4	Pour ligmodif/colmodif . . . . .	7
3.2	Les classes . . . . .	7
3.2.1	La classe Cell : . . . . .	7
3.2.2	La classe Liste : . . . . .	8
3.2.3	La classe Tabliste : . . . . .	8
3.2.4	La classe Matrice : . . . . .	8
3.2.5	La classe Picross : . . . . .	8
3.3	Algorithme remarquable : . . . . .	9
3.3.1	SLG : . . . . .	9
3.4	L'interface graphique : . . . . .	10
<b>4</b>	<b>Améliorations/Evolution du programme :</b>	<b>12</b>

<b>5</b>	<b>Annexes :</b>	<b>13</b>
5.1	Algorithme de backtracking :	13
5.2	Schéma de l'implémentation :	13
5.3	Construire une grille de Picross :	15

# 1 Description du problème

Le Picross est un casse-tête qui consiste à retrouver une figure depuis les indices. La figure à découvrir est une grille dans laquelle chaque case est de couleur noire ou blanche. Pour chacune des lignes et colonnes on dispose d'un indice qui est une séquence de nombres représentant les longueurs des blocs de cases noires contiguës de la ligne/colonne. Les blocs de cases noires sont séparés par au moins une case blanche.

		<b>3</b>	<b>4</b>	<b>4</b>	<b>4</b>	<b>3</b>
<b>2</b>	<b>2</b>					
	<b>5</b>					
	<b>5</b>					
	<b>3</b>					
	<b>1</b>					

TABLE 1 – Un exemple de Picross et sa solution.

Certaines grilles peuvent ne pas avoir de solution, d'autres en avoir plusieurs.

		<b>2</b>	<b>2</b>	<b>2</b>	<b>2</b>
<b>2</b>					
<b>2</b>					
<b>2</b>					
<b>2</b>					

		<b>2</b>	<b>2</b>	<b>2</b>	<b>2</b>
<b>2</b>					
<b>2</b>					
<b>2</b>					
<b>2</b>					

TABLE 2 – Un exemple de Picross qui a deux solutions.

		1			1
		1	1	1	1
1	1				
	3				
1	1				

		1			1
		1	1	1	1
1	1	■	□	□	■
	3	■	■	■	■
1	1	■	□	□	■

TABLE 3 – Un exemple de Picross qui n'a pas de solution.

Le but du projet est d'implémenter un solveur de Picross en C++ qui, étant donnée une matrice et des indices liés à ses lignes/colonnes, calcule, puis affiche la solution. On ne considère que des Picross qui ont (au moins) une solution. La résolution d'une grille de Picross est un problème NP-complet (en temps Non-déterministe Polynomial). Pour le problème du Picross cela signifie qu'il ne faut pas espérer un algorithme qui résout n'importe quelle grille de Picross avec une complexité polynomiale, trouver une solution peut être très coûteux.

## 2 Description de la méthode de résolution utilisée

Afin de résoudre une grille de Picross donnée, nous traitons ses lignes indépendamment de ses colonnes. Le traitement des lignes et des colonnes se faisant de la même manière; nous n'évoquerons par la suite que le traitement des lignes. Une première méthode de résolution consisterait à générer toutes les solutions d'une ligne, partiellement remplie ou non; et de déterminer les cases noires ou blanches communes à chacune des solutions. Une telle méthode permettrait de déterminer un plus grand nombre de cases. En revanche, elle serait très coûteuse. C'est pourquoi, au lieu de générer toutes les solutions, nous avons choisi d'en générer deux; la plus à gauche et la plus à droite. Cette méthode ne permet pas de déterminer autant de cases que la précédente mais est nettement moins coûteuse.

### 2.1 Résolution d'une ligne

La résolution d'une ligne passe par l'appel de deux méthodes donnant respectivement la solution la plus à gauche et la solution la plus à droite de la ligne. On fait ensuite la fusion de ces deux solutions afin de déterminer les cases nécessairement noires ou blanches.

### 2.1.1 La méthode SLG

La méthode SLG a pour but de générer la solution la plus à gauche d'une ligne contenant ou non des cases déjà déterminées. Une fois un bloc placé, elle cherchera à placer le suivant, ainsi tous les blocs de la liste d'indices de la ligne. Afin d'explicitier son fonctionnement, son algorithme est explicité en en page 9.

### 2.1.2 Les méthodes SLPG et SLPD

Les méthodes SLPG et SLPD sont les méthodes qui génèrent les deux solutions nécessaires à la résolution d'une ligne. Elles font toutes deux appel à SLG. Pour la solution la plus à droite, SLG est appelé sur une ligne et une liste d'indices préalablement inversés.

### 2.1.3 La méthode Fusion

Les appels à SLPG et SLPD nous donnent deux solutions possibles pour une ligne et, pour parvenir à sa résolution, il nous faut déterminer les cases n'ayant qu'une solution possible.

Pour ce faire, on fait appel à Fusion qui placera les cases nécessairement noires. Cette dernière nécessite que nos blocs de cases noires soient numérotés, on fait donc un appel préalable à une méthode Numeroter sur nos solutions SLPG-SLPD.

Fusion sera ainsi en mesure de noircir les cases qui sont noires dans SLPG et SLPD et qui ont le même numéro de bloc. En effet, le status de case noire sûre ne peut être défini que si une case noire commune aux deux solutions appartient à un même et unique bloc.

L'exemple ci-dessous illustre l'importance de cette spécification.

*Considérons la ligne suivante :*

**1 1**

--	--	--	--	--

*SLPG renvoi :*

**1 1**

1		2		
---	--	---	--	--

*SLPD renvoi :*

**1 1**

		1		2
--	--	---	--	---

Une case commune aux deux solutions mais n'appartenant pas au même bloc.

Cette case ne peut pas être déterminée car une autre solution serait :

	1		2	
--	---	--	---	--

#### 2.1.4 La méthode remplirCasesSureBl

Cette méthode est appelée après Fusion dans le but de placer les cases blanches sûres. Elle est en mesure de les déterminer grâce aux solutions SLPG-SLPD en calculant les envergures des blocs de cases noires. Dès lors, les espaces situés entre les intervalles trouvés sont des cases blanches sûres.

Elle commence par générer une ligne de cases blanche de la même taille que celles prises en paramètres ; à savoir, le résultat de Fusion et les solutions SLPG-SLPD ; puis “noirci” les cases correspondantes à l’intervalle qu’occupe chacun des blocs dans l’une et l’autre des solutions SLPG-SLPD.

A la fin de ce procédé, elle copie les cases blanches restantes dans le résultat de Fusion.

## 2.2 Résolution du Picross

Pour résoudre un Picross, un appel initial des méthodes précédentes sur toutes les lignes et colonnes est effectué (cf. TINY\_SOL), cela pour but d’initialiser nos listes d’indices de lignes/colonnes modifiées afin de rappeler les méthodes sur les lignes/colonnes correspondantes, et ce tant que les listes contiennent des cases modifiées (cf. FAT\_SOL).

### 2.2.1 la méthode de Backtracking

A l’aide de nos méthodes précédentes, il peut arriver que des cases ne puissent être déterminées ;

Nous avons choisi d’implémenter une méthode de backtracking ‘simple’ qui place une case indéterminée (arbitrairement, la première qu’il trouve en parcourant les lignes de la matrice) à noire et étudie les conséquences de cette coloration ; L’algorithme est explicité en annexe.

## 3 Description de l'implémentation

### 3.1 Les besoins du programme :

#### 3.1.1 Pour les indices de chaque ligne/colonne :

- Imperatif à satisfaire :
  - structure de taille dynamique (nombre variable d'indices par lignes)
  - parcours en  $\theta(n)$  (plusieurs méthodes recourant à un parcours total ou partiel de la structure)
  - relation d'ordre (le parcours ne sont réalisés que dans l'ordre de l'indice le plus en haut vers celui le plus en bas respectivement gauche-droite pour les lignes)
- Choix : la liste simplement chaînée, car elle correspond parfaitement aux spécifications.

#### 3.1.2 Pour représenter la grille du Picross :

- Imperatif à satisfaire :
  - structure de taille fixe
  - accès en temps constant a chaque case
  - facilitée de copie de la structure ( $\theta(n)$ )
- Choix : La matrice (Un tableau de tableaux)

#### 3.1.3 Pour l'ensemble des indices de lignes/colonnes :

- Imperatif à satisfaire :
  - accès en temps constant a chaque liste
  - indicage cohérent avec la matrice
- Choix : Un tableau de listes

#### 3.1.4 Pour ligmodif/colmodif

- Imperatif à satisfaire :
  - ajout d'un élément en temps constant
  - retrait d'un élément en temps constant
  - taille dynamique
- Choix : Le choix naturel serait l'ensemble mais nous avons choisi la liste simplement chaînée car déjà implémentée.

### 3.2 Les classes

#### 3.2.1 La classe Cell :

- Attributs :
  - Val : contient la valeur d'un bloc de case ou un indice
  - Suiv : pointeur sur la cellule suivante de la liste

### 3.2.2 La classe Liste :

- Attributs :
  - Longueur : contient le nombre de cellules de la liste.
  - Fini : indique si la ligne/colonne à laquelle est rattachée la liste est entièrement rempli (booléen à true) ou non (booléen à false), cet attribut est notamment très utile pour vérifier la condition d'arrêt de notre résolution.
  - Tête : Pointeur vers le premier élément de la liste.
- Méthode remarquable :
  - Surcharge de l'opérateur() : Nous l'utilisons notamment pour faciliter l'accès à un élément de la liste.

### 3.2.3 La classe Tabliste :

- Attributs :
  - Tab : le tableau de listes
  - Taille : la taille du tableau
- Méthode remarquable :
  - Surcharge de l'opérateur[] : Nous permet l'accès aux listes en temps constant comme pour un tableau.

### 3.2.4 La classe Matrice :

- Attributs :
  - Mat : la matrice d'entiers
  - Nbl : Nombre de ligne
  - Nbc : Nombre de colonnes

### 3.2.5 La classe Picross :

La classe Picross est la classe dont l'instance représente le Picross ; S'est dans cette classe que sont implémentées nos méthodes de résolution.

- Attributs :
  - Une Matrice (-1 : blanc, 0 : indéterminé, 1 : noir)
  - Deux TabListes pour nos listes de blocs
  - Deux listes d'indices ligModif et colModif pour limiter nos récursions



### 3.3 Algorithme remarquable :

#### 3.3.1 SLG :

**Spécification :** void SLG(int\*, short int, Cell\*, short int, bool&);

- Paramètres :
  - Un tableau d'entier : (Donnée/résultat) il représente une ligne/colonne de notre matrice, initialement il peut contenir des cellules à 1,-1,0
  - Un short int : représentant la taille du tableau
  - Une cellule de liste : il représente le prochain indice à placer dans le tableau, initialement la première cellule associée à la dite ligne/colonne
  - Un short int : représentant l'indice dans le tableau auquel on souhaite placer le bloc, initialement 0
  - Un booléen passé par référence : représentant si il est possible ou non de placer la fin de la liste d'indices indice courant compté à partir de l'indice courant dans le tableau.
- Retour de la fonction.

Le retour se fait par l'intermédiaire du tableau et du booléen. Si le booléen est à false c'est que le couple (ligne/colonne, liste d'indices) n'a pas de solution. Dans le cas contraire, la solution la plus à gauche nous est donnée dans le tableau.

**SLG** (L : ligne ; n : taille de la ligne ; Lind : liste d'indices ; i : entier ; possible : boolean)

**Debut**

- Si Lind est vide alors
  - Si L ne contient pas de cases noires après l'indice i alors
    - on blanchit toutes les cases de L d'indices supérieurs à i ;
  - Sinon
    - possible == faux ;
- Sinon on essaie de placer le 1er bloc de Lind à l'indice i
  - Si on y arrive pas alors
    - si L[i] n'est pas noir alors
      - SLG(L, n, Lind, i+1, possible) ;
    - sinon
      - possible == faux ;
  - Sinon
    - SLG(L, n, queue(L), i+tête(L)+1, possible) ;
- Si non possible alors
  - Si L[i] n'est pas noir alors
    - SLG(L, n, Lind, i+1, possible) ;
  - Sinon
    - possible == faux ;

**Fin algorithme**

### Complexité pire des cas : $\theta(n^2) < c \leq \theta(2^n)$

Notons :

- $i$  : l'indice dans notre tableau
- $n$  : la taille du dit tableau
- $T$  : le dit tableau
- $l$  : la taille de la liste d'indices (le nombre d'indices)
- $m_i$  : l'envergure d'un indice  $i$

Dans le pire des cas de notre algorithme, on peut imaginer que chaque hypothèse faite par notre algorithme soit fausse. Pour cela, il nous faut imaginer une liste d'indices de petite taille (ici on choisira  $\forall i \in [0, l[ \ m_i=1$ ). En effet, plus les indices sont grands moins ils ont de positions possibles pour une même taille de tableau, et donc moins d'hypothèses.

Enfin, on voit bien que la complexité va dépendre de la taille de la liste ; en effet, chaque cellule appelle la fonction au moins une fois pour chaque maillons placé après dans liste. Au plus, pour un indice  $j$  de la liste SLG sera appelée  $\sum_{i=l-j}^l (i)$  fois c'est à dire  $l-i$  fois  $\forall i \in [j, l[$ .

En outre, on sait que le coût de SLG sans les appels récursifs revient à  $n$  puisque dans tous les cas où la liste n'est pas vide, on recopie le tableau. Sinon, si la liste est vide, on parcourt le tableau de l'indice  $i$  à  $n$ . Ici, on considérera cette valeur  $cst=n$ , bien que ce cas soit défavorable. Il n'est pas réaliste en effet, dans le cas d'une liste non-vide (ce qui est notre cas) cette valeur est nécessairement inférieure et décroît tout au long des appels de SLG. Soit une complexité inférieure à  $2^n$ .

Avec ce modèle défavorable on obtient l'équation de récursion suivante :

$$SLG(i) = \sum_{k=l-i}^l (SLG(k) * n) + cst$$

Ainsi, en notant  $c$  la complexité dans ce pire des cas :  $\theta(n^2) < c \leq \theta(2^n)$  ; pour chaque case on parcourt au minimum une fois le tableau (la recopie) pour chaque cellule après ici cela fait déjà  $\theta(n^2)$  et on doit le faire pour chaque indice.

### 3.4 L'interface graphique :

Pour implémenter une interface utilisateur graphique et afficher notre Picross on utilise la bibliothèque Gtkmm, qui est une surcouche de GTK pour le langage C++. Dans une fenêtre du programme on a quatre boutons :

- “Description” contient une brève description du programme ;
- “Ouvrir” pour ouvrir un fichier texte (nos grilles Picross), qui permet l'initialisation d'une grille vide avec ses indices ;
- “Résolution” permet le déclenchement de nos méthodes de résolution et affiche la grille ainsi résolue ;
- “Quitter” appel d'une fonction GTK permettant la fermeture de la fenêtre.

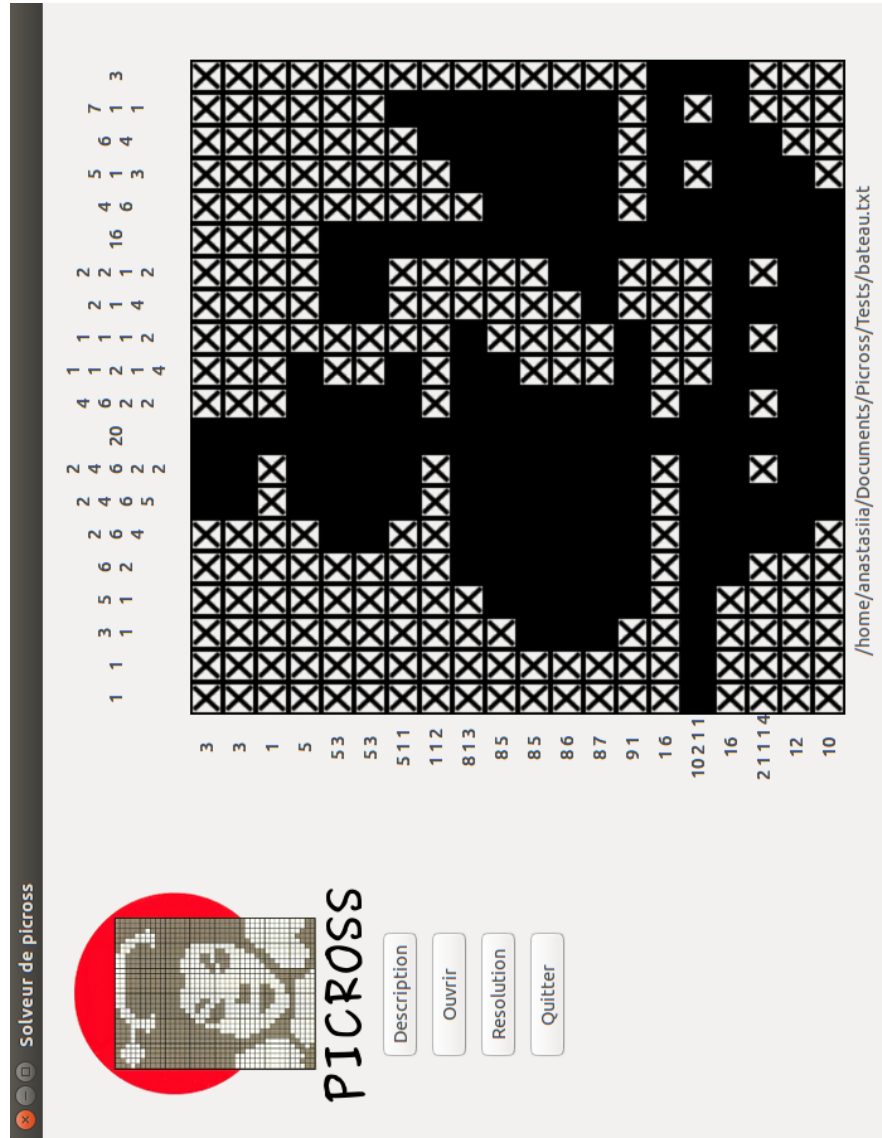
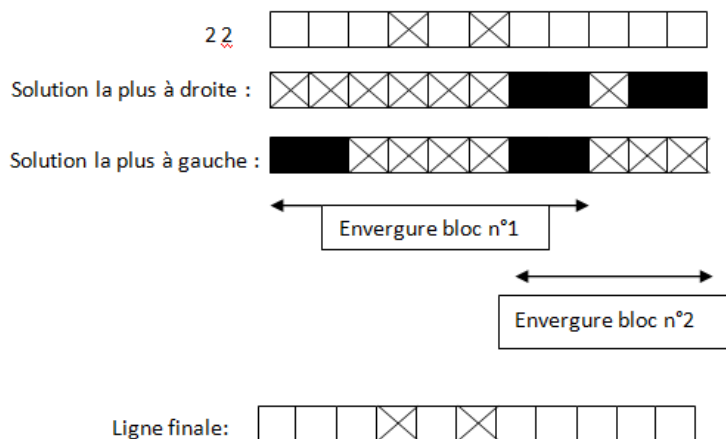


FIGURE 1 – Un exemple de solution Picross.

## 4 Améliorations/Evolution du programme :

- Dans le cas où le picross nécessite un backtrack pour sa résolution, les cases restantes peuvent se voir attribuer un degré de liberté, et la case ainsi déterminée la plus probable aurait alors été celle choisi pour lancer le backtracking.
- Dans le souci d'améliorer la complexité de la résolution d'une grille en moyenne, nous aurions pû, plutôt que de faire des copies/recopies de matrices entre les environnements de backtrack, ne maintenir que les cases modifiées entre les environnements de backtracks; Cela dit, dans le pire des cas (1ere hypothese fausse), la complexité reste équivalente ( $\theta(n^2)$ ) car l'ensemble des cases se verraient modifiées après le premier environnement de backtrack. (Exemple BT 1 sol)
- L'implémentation de méthodes permettant de rentrer des grilles de picross graphiquement (au clic).
- L'implémentation de méthodes permettant de résoudre des grilles de picross graphiquement (au clic).
- Dans certains cas la méthode pour trouver les cases blanches sûres que nous avons choisies ne trouve pas toutes les cases blanches que nous aurions pu déterminer. Par exemple :



En effet, il n'y a aucun chevauchement de noir appartenant au même bloc, et aucun blanc qui ne soit dans l'envergure des blocs. Pourtant on peut aisément voir que la case d'indice 4 (indice débutant à 0) est blanche puisqu'elle est coincée entre deux blanches et que la taille minimum des blocs est de deux. On pourrait donc imaginer un algorithme qui teste pour chaque blocs d'indéterminés coincés entre deux blancs si sa taille est inférieure au plus petit indice à rentrer, dans ce cas on le blanchit.

## 5 Annexes :

### 5.1 Algorithme de backtracking :

```
void Picross::backtrack(bool &poss)
{
    FAT_SOL(ligModif.getLongueur()); //méthode de résolution
    if(!isPicrossFini())
    {
        int** SAVE=copieMat(); //Copie de la matrice
        bool* TL=new bool [getNbLignes()];
        bool* TC=new bool [getNbColonnes()];
        copieBool(TL,TC); //Copie de nos booléens lignes|colonnes finies
        sint i=0,j=0; //Indices utilisés pour le placement de la case noire
        Placer1noir(poss,i,j); //poss à true ssi la matrice contient un indéterminé(0)
                                //on place un noir(1) et on initialise (i,j)

        if(poss)
        {
            backtrack(poss); //On recommence
//A ce niveau nous sommes dans le dernier environnement où on à Placer1noir(i,j)
//Si le picross n'est pas correctement rempli qu'on à pas pu Placer1noir poss=false
            if(!poss)
            {
                recopieMat(SAVE); //Recopie de la matrice sauvegardée avant le placement du noir
                recopieBool(TL,TC); //Recopie de nos booléens fini qui auraient pu être modifiés
                Placer1blanc(i,j); //On place la case (i,j) à blanc
                backtrack(poss); //On recommence
            }
        }
    }
}
```

### 5.2 Schéma de l'implémentation :

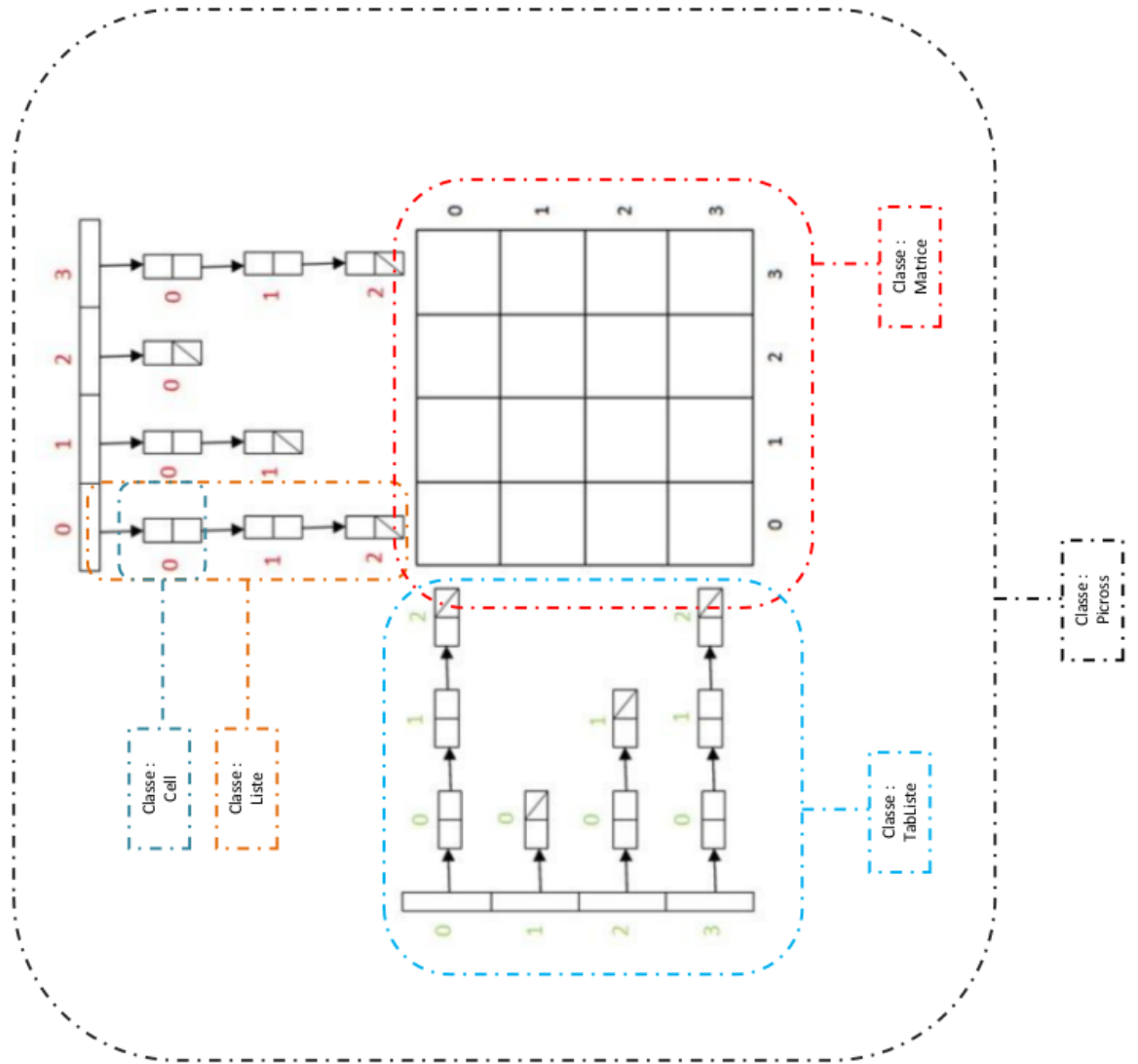


Schéma classe/structure projet Picross

### 5.3 Construire une grille de Picross :

5 5  $\Rightarrow$  nbl nbc  
 5  
 2 2  
 1 1 1  
 2 2  
 5  
 5  
 2 2  
 1 1 1  
 2 2  
 5

} *indices des lignes*  
 }  
 } *indices des colonnes*

FIGURE 2 – Un exemple de fichier.