# Rapport de projet Picross

CONNES Victor PRYSIAZHNIUK Anastasiia
BENAIS-HUGOT Charles RIVIERE Cecilie

April 26, 2015

## Contents

# 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 contiges de la ligne/colonne. Les blocs de cases noires sont séparés par au moins une case blanche.

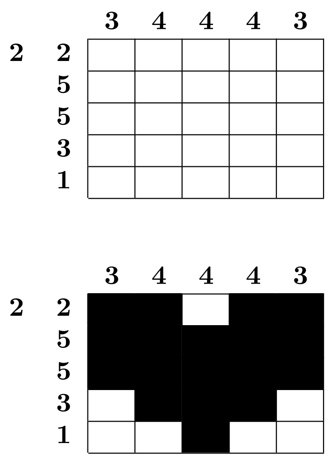
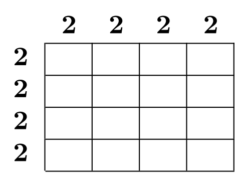
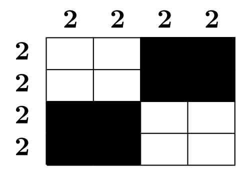



Table 1: Un exemple de Picross et sa solution.

Certaines grilles peuvent ne pas avoir de solution ou en avoir plusieurs.





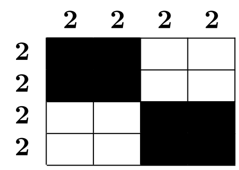
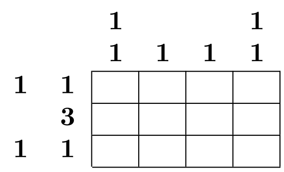




Table 2: Un exemple de Picross qui a deux solutions.



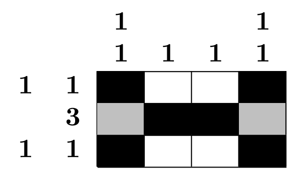


Table 3: Un exemple de Picross qui n'a pas de solution.

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 grand nombre de cases. En revanche, c'est une méthode 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 elle 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 et celles nécessairement blanches.

### 2.1.1 La méthode SLG

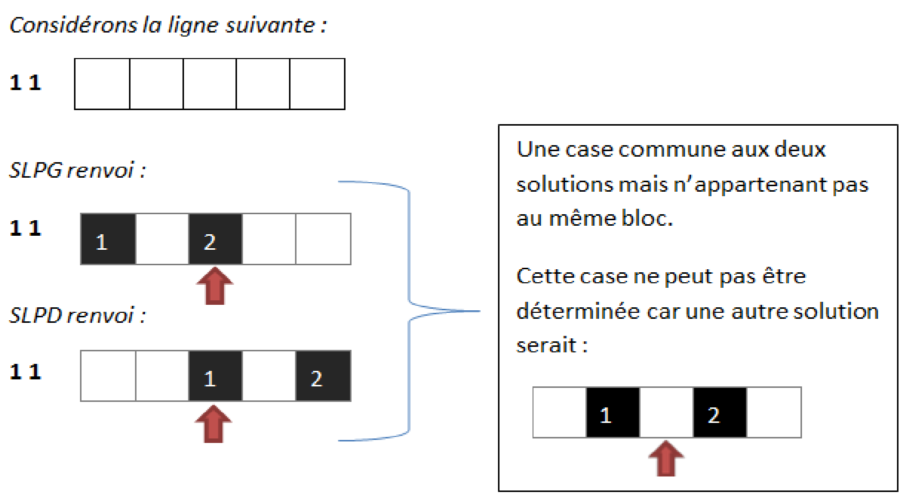
La méthode SLG a pour but de générer une solution à une ligne contenant ou non des cases déjà déterminées. Elle cherchera à placer chaque blocs de la liste d'indices dans la ligne. Une fois un bloc placé, elle cherchera à placer le suivant. Si elle n'y parvient pas, elle recommence en i+1. Afin d'expliciter son fonctionnement, son algorithme est détaillé en annexe.

### 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 avec i=0.
Pour la solution droite, SLG est appelé sur une ligne et une liste d'indices préalablement inversées.

### 2.1.3 La méthode Fusion

A la suite des appels à SLPG et SLPD, nous aurons deux solutions possibles pour une ligne; or, pour parvenir à la résolution de notre Picross, il nous faut déterminer les cases n'ayant qu'une solution.
Pour ce faire, on fait appel à Fusion qui placera les cases nécessairement noires. Cette dernière nécessite que nos blocs de cases noires soient numérotées. Pour cela, on fait donc appel à une méthode Numeroter qui consiste à numéroter avec les numéros de blocs, les cases noires de 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.




### 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 paramétres; à savoir, le résultat de Fusion et les solutions SLPG-

SLPD; puis "noirci" les cases correspantes 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 effectuer (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).
La méthode de résolution du Picross se résume donc ainsi :

- traiter ligne;
- liste C == colonnes à traiter;
- tant que la liste n'est pas vide
  - liste L == vide;
  - pour chaque i de C faire
    - L == L + traiter ligne(i);
  - fin pour;
  - C == vide;
  - pour chaque i de L faire
    - C == C + traiter colonne(i);
  - fin pour;
- fin tant que;

## 2.3 Algorithmes remarquables

### 2.3.1 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 des cases choisies arbitrairement à noir et étudie les conséquences de cette coloration :

- Recherche la 1ere case indeterminée de la matrice (parcours par les lignes)
- On place cette case à noir
  - On étudie la conséquence de cette coloration arbitraire :
    - Si cela mène à un picross résolu, on sort

        * Si cela mène à un picross non rempli, on réitère sur une nouvelle case que l'on backtrack
        * Si cela mène à un picross rempli mais non résolu, on sort de nos environnements successifs avec un booléen à false, on effectue alors une recopie de notre matrice qui avait été sauvegardée, et on réitère, cette fois avec la case backtrackée à blanc

# 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 chainés, car elle correspond parfaitement au spécification et parait finalement très proche de la réalité

### 3.1.2 Pour représenter la grille du picross :

- Imperatif à satisfaire:
  - structure de taille fixe
  - accèes en temps constant a chaque case
  - faciliter de copie de la structure ($\theta$(n))
- Choix: La matrice car très proche de la réalité

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

- Imperatif à satisfaire:
  - accéder en temps constant a chaque liste
  - garder un indicage cohérent avec la matrice
- Choix :Un tableau

### 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 on choisit la liste simplement chainés car déjà implémentés pour les indices de chaque ligne/colonne. Son comportement est similaire dans le cas d'un ajout et d'un retrait en début de liste .

## 3.2 Les classes

L'ensemble de nos classes disposent d'une surcharge de l'operateur¡¡ afin de faciliter l'affichage dans le terminal. De plus, il existe d'autres classes dans notre programme qui permettent d'améliorer l'affichage en utilisant une interface graphique. Nous ne distinguerons ici seulement les classes permettant la resolution du picross.

### 3.2.1 La classe Cell :

Elle représente un indice logique

- Attributs :
  - Val : qui représente la valeur de cette indice logique (cette valeur étant strictement entière positive et a priori non borné nous avons choisi de la représenter par un short int)
  - Suiv : qui est pointeur sur la cellule suivante de la liste

### 3.2.2 La classe Liste :

- Attributs :
  - Longueur : qui représente la longueur de la liste. Cette attribut et incrémenter ou décrémenter automatiquement dès qu'il y a variations de la taille de la liste.
  - Fini : Nous indique si la ligne/colonne a laquelle est rattaché la liste est entièrement remplie (booléen à true) ou non (booléen à false), cette attribut est notamment très utile pour vérifier la condition d'arrêt de notre résolution (cf. main)
  - Tête : Pointeur vers le premier élément de la liste.
- Méthode remarquable :
  - Surcharge de l'opérateur() : Nous avons utiliser l'operateur() comme un operateur d'indexage d'une liste cela nous permet notamment de faciliter l'accès à un element de la liste.

### 3.2.3 La classe Tabliste :

- Attributs :
  - Tab : le tableau de liste
  - Taille : la taille du tableau
- Méthode remarquable :
  - Surcharge de l'opérateur[]: Nous permet d'utiliser comme si elle était simplement un tableau.

### 3.2.4 La classe Matrice :

- Attributs:
  - Tab : la matrice d'entier (-1 : blanc 0 : indéterminé 1 : noir)
  - Nbl : Nombre de ligne
  - Nbc : Nombre de colonnes

## 3.3 La classe Picross :

La classe picross est la classe dont les instances représentent les picross; S'est dans cette classe que sont implémenté nos méthodes de résolution.

## 3.4 Les algorithmes remarquables :

### 3.4.1 Backtrack :

**Spécification :** void backtrack(bool &poss);

- Paramètre:
  - Un booléen signalant la possibilité de placer une case dans un picross non fini

  Dans le cas ou cela est possible, on "backtrack" une case Dans le cas ou cela n'est pas possible, on revient à l'environnement de la dernière case backtrackée et on l'inverse

- Retour de la fonction :

La fonction ne retourne rien, on travaille directement sur notre Matrice.
Cela à pour conséquence une complexité élevée ($\theta(n^2)$) lors de la recopie de celle-ci.

### 3.4.2 Fusion :

**Spécification :** Fusion(int* Merge,int* TG,int* TD,sint taille);

- Paramètre:
  - Un tableau d'entier Merge: (Resultat) il représente la ligne/colonne de la matrice,après la resolution de notre fonction, initialisé avec la ligne/colonne existante
  - Un tableau d'entier TG: (Donnée) il représente la solution la plus à gauche qui résout notre ligne/colonne en fonction de sa liste d'indices et des informations à notre disposition
  - Un tableau d'entier TD: (Donnée) idem pour la solution la plus à droite

- Un short int taille: représentant la taille du tableau

- Retour de la fonction:

La fonction retourne par l'intermédiaire du tableau Merge notre ligne/colonne de depart avec en plus d'éventuelles cases noires "sûres".

**Comlexité : $\theta$(n)**
Nous parcourons chaque case du tableau pour vérifier si nous avons des informations supplémentaires. Remarque: on aurait pu gérer une liste des indices qui était encore indéterminé pour chaque ligne/colonne ce qui aurait améliorer notre complexité moyenne. Néanmoins, nous avons juger cela peut indispensable au vue du caractère negligeble de cette complexité par rapport aux autres fonctions du programme.

#### 3.4.3 remplirCasesSureBl :

**Spécification :** remplirCasesSureBl(int* Merge, int* TG,int* TD, sint taille, Liste &L);

- Paramètres:
  - Un tableau d'entier Merge: (Resultat) il représente la ligne/colonne de la matrice,après la résolution de notre fonction, initialisé avec le retour de fusion.
  - Un tableau d'entier TG: (Donnée) il représente la solution la plus à gauche qui résout notre ligne/colonne en fonction de sa liste d'indices et des informations à notre disposition
  - Un tableau d'entier TD: (Donnée) idem pour la solution la plus à droite
  - Un short int taille: représentant la taille du tableau
  - Une liste d'indices L: La liste d'indices associés a la ligne/colonne
- Retour de la fonction:

La fonction retourne par l'intermédiaire du tableau Merge notre ligne/colonne de depart avec en plus d'éventuelles case blanches "sures" en se basant sur l'envergure des blocs.

**Comlexité :** $\theta(n)$
On doit pour chaque indice de la liste inscrire son envergure dans un tableau intermediaire, pour cela il nous faut parcourir SLG puis SLD mais pas en totalité, en effet, on parcours SLG jusqu'a trouver un indice i1 tel que TG[i1]=0. On continu notre parcours jusqu'a trouver un indice i2 tel que TG[i2]=0, c'est alors qu'on parcours SLD à partir de i2-1 jusqu'a trouver un indice i3 tel que TG[i3]=0. On a alors l'envergure du premier bloc. Pour le deuxiéme on peut commencer notre parcours dans SLG à partir de i2 car il est forcément à droite

du premier.
On trouve alors un nouveau i1 puis un i2 puis un i3 et on recommence alors à partir du nouveau i2 et ainsi de suite. Au final, on aura parcouru entre n et 2n case du tableau.

#### 3.4.4 SLG :

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

- Paramètres:
  - Un tableau d'entier: (Donnée/resultat) 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 repésente le prochain indice a placer dans le tableau, initialement la première cellule associé 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 a partir de l'indice courant dans le tableau.
- Retour de la fonction.

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

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

**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'indice (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'indice de petite taille (ici on choisira ∀i ∈[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 appeé $\sum_{i=l-j}^{l}(i)$ fois c'est à dire l-i fois ∀i ∈[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 parcours 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 decroit tout au long des appels de SLG. Soit une complexité inférieure à $2^n$.
Avec ce modèle défavorable on obtient l'équation de recursion suivante:

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

Ainsi, en notant c la complexité dans ce pire des cas: $\theta(n^2) <$c$\leq \theta(2^n)$; pour chaque case on parcours au minimun 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.

**Complexité meilleur des cas : $\theta(\mathbf{n})$**

Dans le meilleur des cas la liste est vide, on doit pacourir une fois le tableau pour pouvoir vérifier.

### 3.5 L'interface graphique:

Pour implémenter une interface utilisateur graphique et afficher une grille Picross résolue on utilise une 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(d'où on a mit la taille et les indices d'une grille 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 une fonction GTK permettant la fermeture d'une fenêtre.

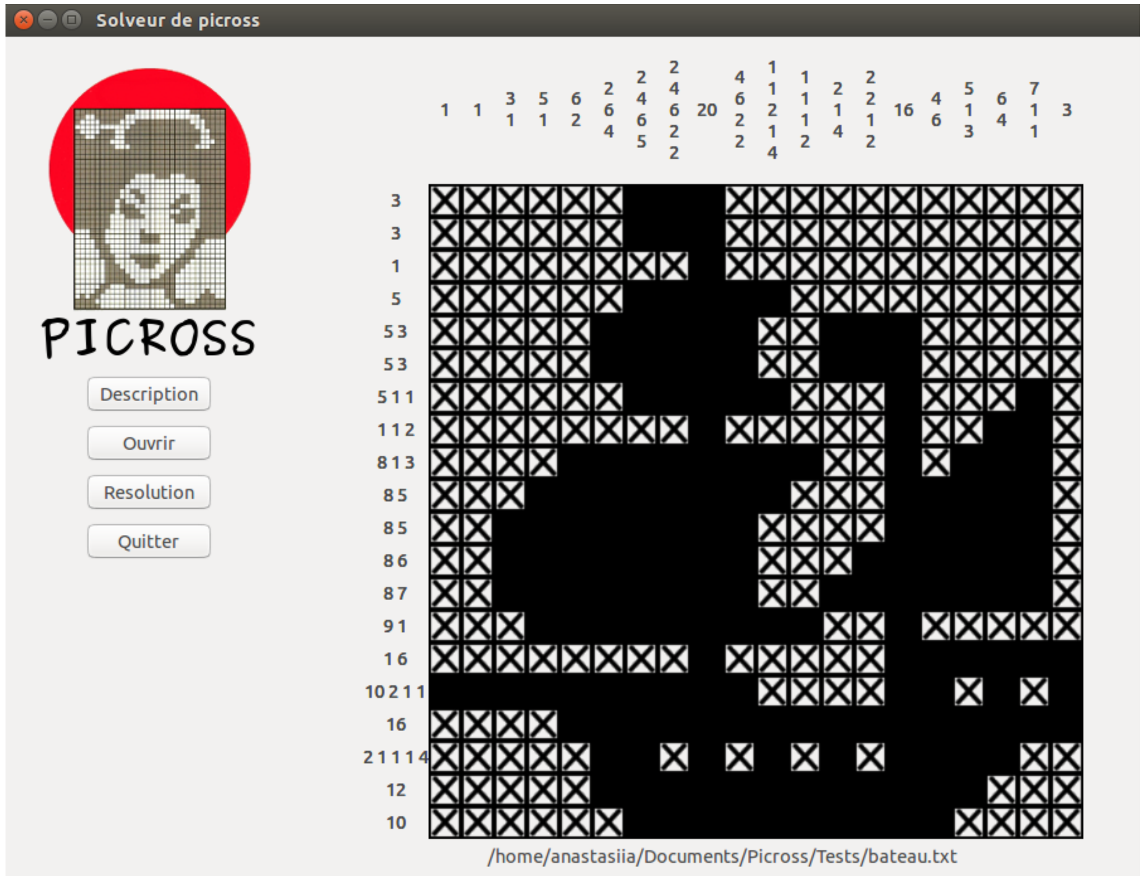



Figure 1: Un exemple de solution Picross.

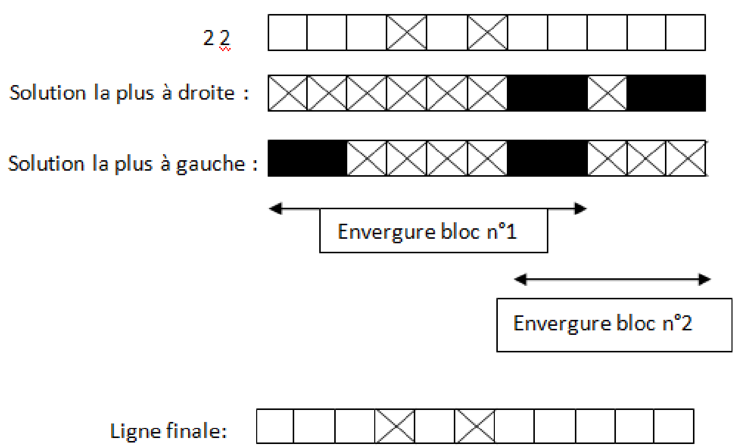
# 4 Ameliorations/Evolution du programme:

- Dans le cas ou le picross nécessite un backtrack pour sa résolution, les blocs restants peuvent se voir attribuer un degré de liberté, et la case ainsi determinée la plus probable aurait alors été celle choisie pour lancer le backtracking.

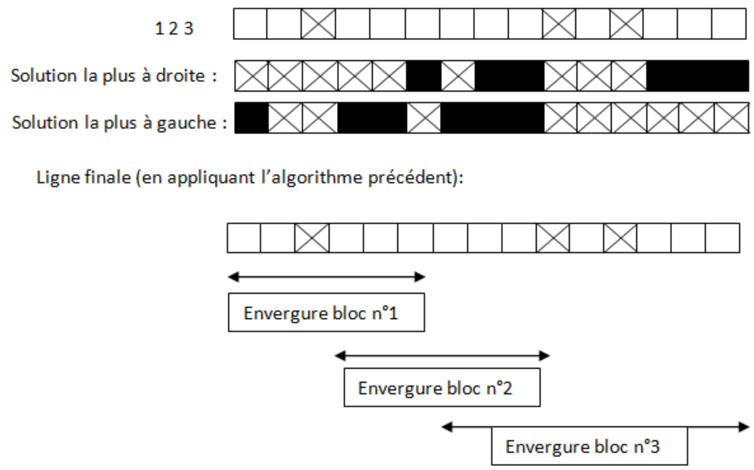
- Dans le souci d'améliorer la compléxitée de la résolution d'une grille en moyenne, nous aurions pû, plutôt que de faire des copies/recopies de matrices entre les environnements de backtrack, ne maintenir que les cases modifiées entre les environnements de backtracks; Cela dit, dans le pire des cas (aucune case n'est determinable), 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)

- Nous aurions pû implémenter des methodes permettant de rentrer 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 bloc d'indéterminé coincé entre deux blancs si sa taille est inferieure au plus petit indice à rentrer, dans ce cas on le blanchit. Néanmoins, on pourrait imaginer un algorithme plus fin. Regardons cet autre exemple :




Ici on voit bien que la cellule 11 devrait être blanche puisque qu'elle n'est pas dans l'envergure du seul bloc qui pourrait convenir. Ainsi, en sachant que la solution la plus à gauche générale et la solution la plus à droite générale définissent aussi respectivement la solution la plus à droite et la solution la plus à gauche de chaque bloc, on aurait pu définir un algorithme qui, en fonction de ces deux solution et de la liste d'indices, vérifie pour chaque bloc d'indéterminé coincé entre deux blanc (ou un bord) s'il est dans l'envergure des indices qui sont inférieurs à sa taille.

# 5 Remerciements :

- Nous tenions à remercier M.Janssen pour son implication dans le projet, ainsi qu'à l'équipe pédagogique de l'Université.

# 6 Annexes :

## 6.1 Schéma de l'implémentation :

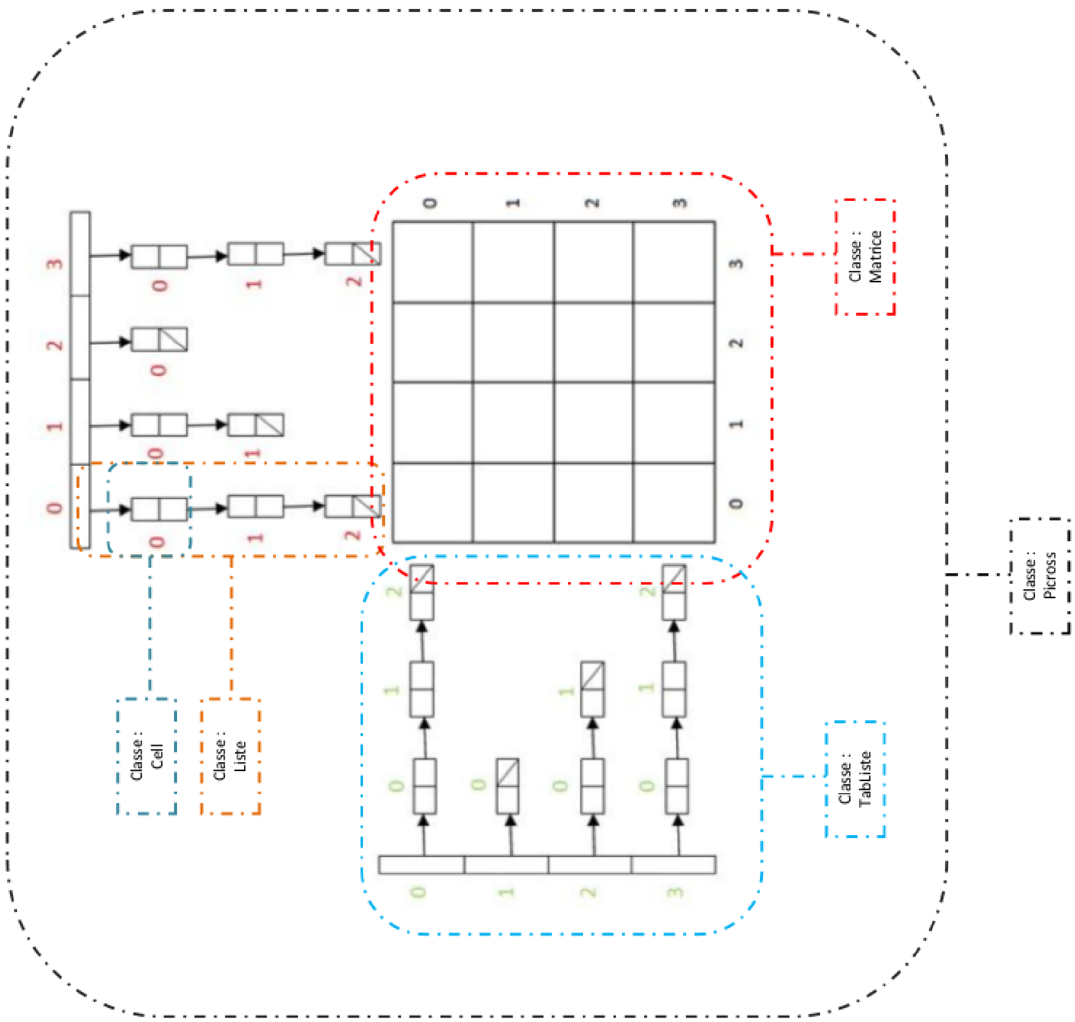



Schéma classe/structure projet Picross

### 6.2 Lecture d'un fichier pour construire une grille:

Dans le but de construire une grille Picross nous avons choisi du structurer un fichier texte de la manière suivante :
Les deux premiers nombres séparés d'un espaces représentent nos Lignes — Colonnes. La méthode passe ensuite sur toutes les lignes du fichier, ou chaque saut de ligne correspond à une nouvelle liste de tabListe.

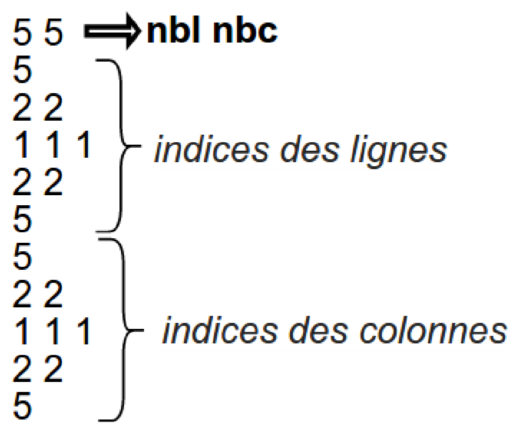



Figure 2: Un exemple de fichier.

### 6.3 Algorithme SLG :

```
void Picross::SLG(int* Tab, size_t n, Cell* P, size_t i, bool &poss)
{
  if(P && i<n)
  {
    int* Tab2=new int [n];
    for(size_t j=0;j<n;j++)
    {
      Tab2[j]=Tab[j];
    }
    PlacerBloc(Tab,n,P->getVal(),i,poss);
    if(poss)//si j'ai placer mon bloc
    {
      SLG(Tab,n,P->getSuiv(),i+P->getVal()+1,poss);//bloc suivant
      if(!poss)
      {
        for(size_t j=0;j<n;j++)
        {
          Tab[j]=Tab2[j];
        }
      if(Tab[i]==1){poss=false;}
        else{SLG(Tab,n,P,i+1,poss);}
      }
    }
    else
    {
      if(Tab[i]==1){poss=false;}
      else{SLG(Tab,n,P,i+1,poss);}
    }
    delete [] Tab2;
  }
  else
  {
    poss=Verification(Tab,i,n);
    if(poss)
    {
    for(size_t i=0; i<n; i++){if(Tab[i]==0){Tab[i]=-1;}}
    }
  }
}
```

### 6.4 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 indeterminé(0) qu'on à p
    if(poss)
    {
      backtrack(poss); //On recommence
//Si, a la suite des appels recursifs, le picross n'est pas correctement rempli et on ne peu
//A ce niveau nous sommes dans le dernier environnement où on à Placer1noir en (i,j)
      if(!poss)
      {
        recopieMat(SAVE); //Recopie de la matrice sauvegardée avant le placement du noir
        recopieBool(TL,TC); //Recopie de nos booléens de lignes|colonnes finies qui aurait p
        Placer1blanc(i,j); //On place la case (i,j) à blanc, car on est sûr qu'elle n'est pa
        backtrack(poss); //On recommence
      }
    }
  }
}
```