Rapport de projet Picross

CONNES Victor PRYSIAZHNIUK Anastasiia
BENAIS-HUGOT Charles RIVIERE Cecilie

24 avril 2015

Table des matières

1 Description du problème

Le Picross est un casse-tête qui consiste à retrouver une figure depuis les indices. La figure à d'ecouvrir 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 contigues de la ligne/colonne. Les blocs de cases noires sont séparées par au moins une case blanche.

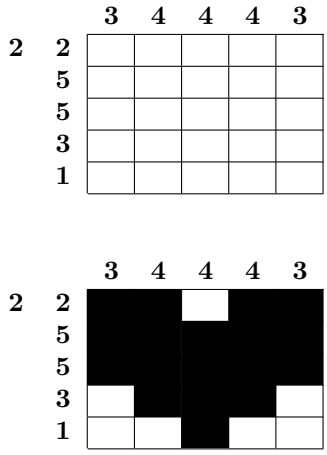
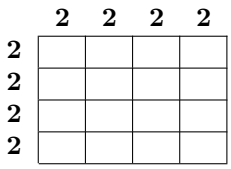


TABLE 1 – Un exemple de Picross et sa solution.

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



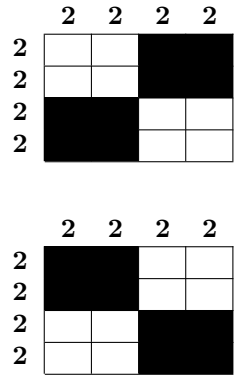
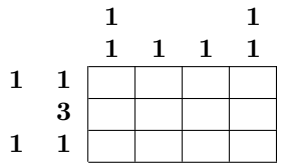


TABLE 2 – Un exemple de Picross qui a deux solutions.



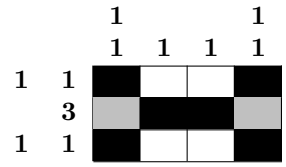


TABLE 3 – Un exemple de Picross qui n'a pas de solution.

La résolution du Picross est un problème NP-complet(NP-Complet est un problème difficile à résoudre, par exemple, le problème SAT). 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 (en fonction de la taille de la grille).

Le but du projet s'agit d'implémenter un solveur de Picross en C++ qui lit les données d'un problème Picross, calcule puis affiche sa solution. On ne considère que des Picross qui ont une solution unique (contrainte donnée dans le cahier des charges).

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 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 chacun des blocs de la liste d'indices à partir d'un indice i 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 détaillée ci-dessous.

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 alors
  - — possible == vrai ;
- Si possible alors
  - — on blanchit toutes les cases de L ;

Sinon Si l'espace entre i et n est insuffisant pour placer le bloc ou qu'on ne peut pas placer de case blanche après le bloc ou que l'espace entre i à la taille du bloc contient une case blanche alors
- — possible == faux ;
- — On rappelle SLG avec i = i+1 ;

Si possible alors
- — On place le bloc à partir de i ;
- — On rappelle SLG pour l'indice de bloc suivant avec i = i+valeur d'indice précédent+1 ;

Fin algorithme

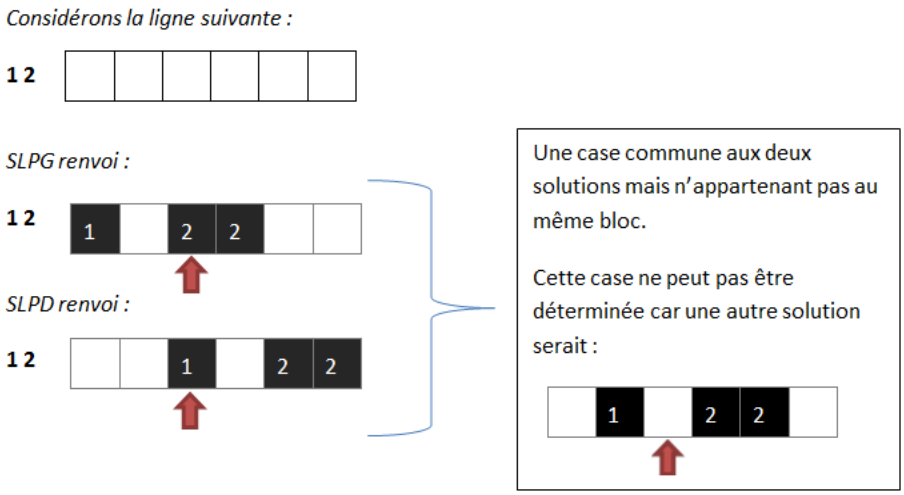
2.2 Résolution d'une ligne

2.2.1 Les méthodes SLPG et SLPD

Les méthodes SLPG et SLPD génèrent respectivement pour une ligne, sa solution la plus à gauche et sa solution la plus à droite. Elles font toutes deux appel à SLG avec i=0 et possible = faux.
Pour la solution droite, SLG est appelé sur une ligne et une liste d'indices préalablement inversées.

2.2.2 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 ne gardera que les cases noires communes aux deux solutions. Cette dernière nécessite que nos blocs de cases noires soient numérotées (appel à la méthode Numeroter).
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 une unique bloc.
L'exemple ci-dessous illustre l'importance de cette spécification.



2.2.3 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.

remplirCasesSureBl commence par générer une ligne de cases blanche de la même taille que celles prises en paramétre ; à savoir, le résultat de Fusion et les solutions SLPG-SLPD ; puis "noircit" 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.

2.2.4 Ajout de la ligne à la grille de Picross

Une fois les cases sûres noires et blanches ajoutées à notre ligne, on vérifie par la méthode isFini si elle complète ou non, afin de ne plus la traiter si tel est le cas.
Ensuite, on ajoute les indices des cases qui ont été modifiées à une liste appelées colModif. Celle-ci permettra de traiter par la suite les colonnes où une case à été ajoutée.
Pour finir, on recopie notre ligne ainsi complétée dans la grille de Picross.

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 θ(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
  - accès en temps constant a chaque case
  - faciliter de copie de la structure (θ(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 .

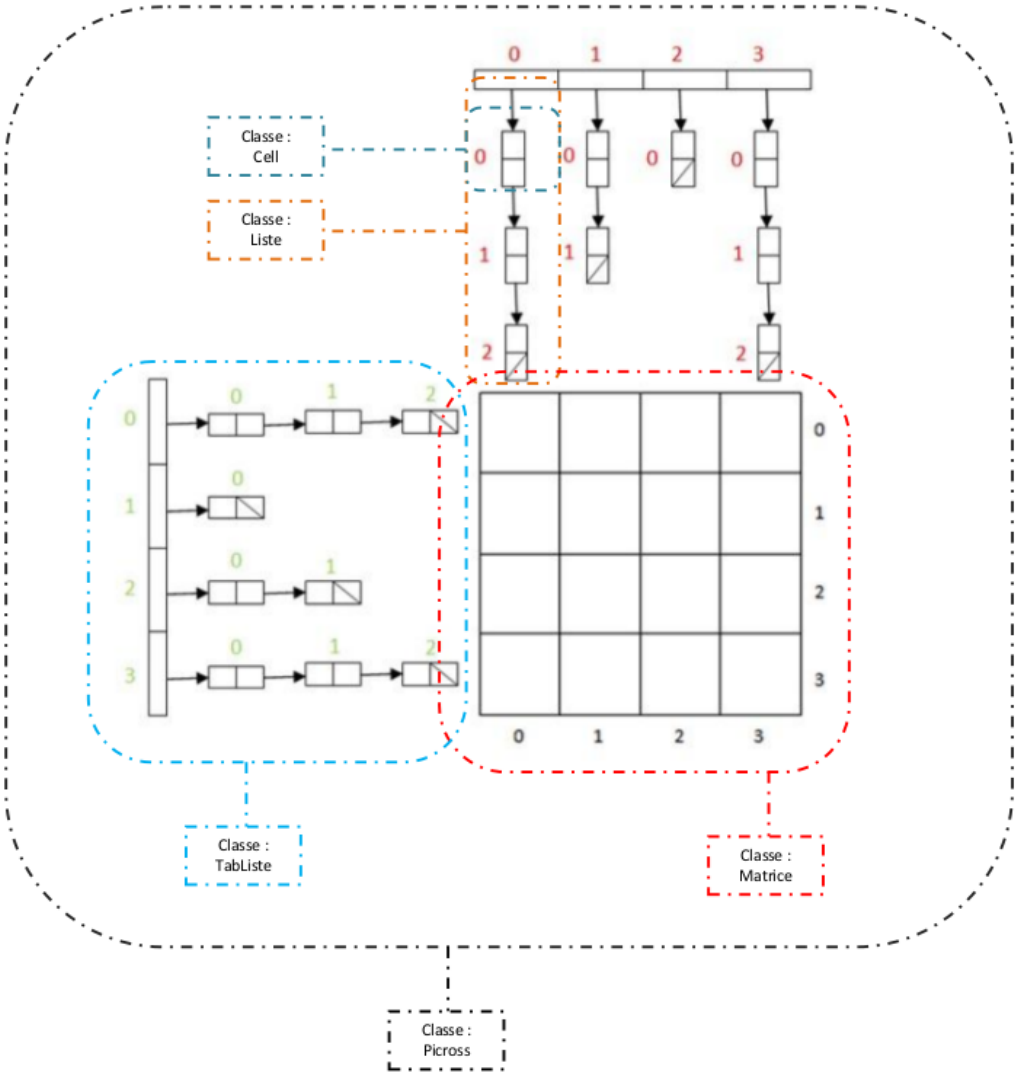


Schéma classe/structure projet Picross

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'arrt de notre résolution (cf. main)
  - Tte : 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 SLG :

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

— Paramètre :
  — Un tableau d'entier : (Donée/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 lintermédiaire 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 donée dans le tableau.

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

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 hypothéses faites par notre algorithme soit fausse notre fonction : Pour cela, il nous faut imaginer une liste d'indice de petite taille (ici on choisira $\forall$i $\in$[0,l[m_i=1) en effet,plus les indices sont grand moins il ont de position possible pour une même taille de tableau, et donc moins d'hypothèse.
Enfin,on voit bien que la complexité vas dépendre de taille de la liste ,en effet,chaque cellule appelle la fonction au moins une fois pour chaque maillons placer 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 $\forall$i $\in$[j,l[.
En outre, on sait que le coût de SLG sans les appels récursif reviens a n puisque dans tous les cas où la liste n'est pas vide ont recopie le tableau.Sinon, si la

liste est vide on parcours le tableau de l'indice i à n, ici, on considerera cette valeur cst=n, néanmoins bien que ce cas soit defavorable il n'est pas réaliste en effet, dans le cas d'une liste non-vide (ce qui notre cas) alors cette valeur est necessairement inférieure et decroit tout au long des appels de SLG. Soit une complexité inférieure à 2^n.
Avec ce modèle défavorable on obtient l'equation 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$\leq \theta(2^n)$,en effet, pour chaque case on parcours au minimun un 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 : θ(n)
Dans le meilleur des cas la liste est vide on doit pacourir une fois le tableau pour pouvoir vérifié .

3.4.2 Backtrack :

Spécification : void backtrack(bool &poss) ;
- Paramètre :
  - Un booléen signalant la possibilitée 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 (θ(n)) lors de la recopie de celle-ci.

3.4.3 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 information 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'enventuelle case noire "sures".

Comlexité : θ(n)

Nous parcourons chaque case du tableau pour vérifier si nous avons des informations suplementaires. 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 au autre fonctions du programme.

3.4.4 remplirCasesSureBl :

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

- Paramètre :
  - Un tableau d'entier Merge : (Resultat) il représente la ligne/colonne de la matrice,après la resolution 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 information 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'enventuelle case blanche "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 á 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 á partir du nouveau i2 et ainsi de suite.Au final, on aura parcouru entre n et 2n case du tableau.

3.4.5 remplirCaseBlcoince :

Spécification : CaseBlcoince(int* Merge, sint taille, Liste &L) ;

- Paramètre :
  - Un tableau d'entier Merge : (Resultat) il représente la ligne/colonne de la matrice.

- 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'éventuelle case blanche "sures" en se basant sur les tailles des blocs.

Comlexité : $\theta(n)$