Rapport projet Picture

22 décembre 2013

Table des matières

Chapitre 1

Compte-rendu algorithmique

1.1 Définitions

Les mots que nous allons utiliser pour décrire un objet ou un concept précis mis en œuvre dans la résolution du problème sont des mots pouvant donner lieu à de nombreuses lectures et approches différentes qui peuvent fausser les démonstrations qui s'appuient dessus. Pour fixer les idées, nous nous devons d'imposer des bases, seul appui pour nos raisonnements ultérieurs.

Une *case* c sera considérée comme un élément pouvant être associé aux états : `WHITE`, `BLACK`, ou `UNDEFINED`, représentée par :

```
EMPTY: |?|    WHITE: | |    BLACK: |B|
```

Un *bloc* est un ensemble de cases adjacentes ayant toutes le même état `BLACK`. Il est caractérisé par sa taille ou longueur correspondant au nombre de cases qu'il contient, ainsi que la position de sa première case dans le tableau.

Par exemple, le tableau suivant contient 1 bloc : de taille 4, contenant donc 4 cases `BLACK`, à la position 2.

```
| |B|B|B|B| | | | | | | |
```

Une *grille* $\mathcal{G}$ est une structure, dont les éléments sont une *matrice*, un ensemble de séquences d'entiers qu'on appellera *contraintes*, associées aux lignes et aux colonnes de la matrice, et une fonction $\eta_{\mathcal{G}}$ sur l'ensemble $C_{\mathcal{G}}$ des cases de la matrice, à valeurs dans l'ensemble $\Omega = \{\text{White, Black, Undefined}\}$, et qui associe à chaque case un état.

$$\forall c \in C, \exists \omega \in \Omega : \eta_{\mathcal{G}}(c) = \omega$$

Par exemple la grille 4×3 cases suivante :

```
     6 5 2
       4
    -------
3   | | | |
    -------
5 1 |B|B| |
    -------
3 2 | | | |
    -------
1 2 | | | |
    -------
```

Une ligne ou une colonne T est considérée comme *remplie* si toutes ses cases ont pour image par la fonction $\eta_{\mathcal{G}}$ un état défini ("White" ou `BLACK`).

$$\forall c \in T, \exists \omega \in \Omega \backslash \{\text{Undefined}\} : \eta_{\mathcal{G}}(c) = \omega$$

A chaque ligne ou colonne remplie, on peut associer une *contrainte induite*. C'est la séquence d'entiers qui représentent les tailles des blocs de la ligne ou de la colonne considérée.

Étant donné la ligne remplie suivante :

```
|B|B| |B| |
```

on déduit la contrainte induite suivante :

```
[|2, 1|]
```

Une grille est *remplie* si toutes ses lignes et colonnes sont remplies.

Une grille $\mathcal{H}$ est dite *remplissage* d'une autre grille $\mathcal{G}$ si elles ont même matrice, mêmes contraintes, et si leurs fonctions états vérifient la propriété suivante : quelque soit une case de la matrice, si l'état de cette case par la fonction état de $\eta_{\mathcal{G}}$ est `WHITE` ou `BLACK`, alors l'image de cette case par la fonction état de $\eta_{\mathcal{H}}$ est la même.

$$\forall c \in C_{\mathcal{H}}, \begin{cases} \eta_{\mathcal{G}}(c) = \text{White} \Rightarrow \eta_{\mathcal{H}}(c) = \text{White} \\ \eta_{\mathcal{G}}(c) = \text{Black} \Rightarrow \eta_{\mathcal{H}}(c) = \text{Black} \end{cases}$$

Une grille remplie $\mathcal{G}$ est dite *valide* si les contraintes induites, associées à sa matrice, sont identiques aux contraintes de la grille.

Une grille $\mathcal{H}$ est dite *solution* d'une grille $\mathcal{G}$ si $\mathcal{H}$ est un remplissage de $\mathcal{G}$ et si la grille $\mathcal{H}$ est *valide*.

1.2 Problèmes

1.2.1 Présentation du jeu

Picture est un jeu de casse-tête. Étant donnée une grille, il nous faut choisir quelles cases sont noires, et quelles cases sont blanches. Le but est de former des blocs de cases noires. Par exemple, si la contrainte d'une ligne est $(1, 2, 5)$, alors on devra retrouver dans cette ligne trois blocs, de tailles respectivement une case, deux cases et cinq cases, dans cet ordre, et séparés par au moins une case blanche. Le jeu est fini quand la grille est valide.

Pour nous, une grille de Picture peut avoir une, plusieurs ou aucune solutions. Le premier algorithme de la résolution est donc de déterminer s'il existe une solution à une grille donnée, que l'on étendra pour obtenir cette solution si elle existe. On pourra ensuite s'intéresser à l'unicité, évaluer la difficulté de cette grille, et même générer un grand nombre de grilles.

1.2.2 Spécification des problèmes

Le premier problème intitulé `resolve`, est la question d'existence d'une solution :

```
Problème resolve:
    Entrée: Une grille A
    Sortie: Une grille B et un booléen b.
        Si b = VRAI, la grille B est une solution de A.
        Si b = FAUX, il n'existe pas de solution de A.
```

Notons que la résolution de ce problème peut seulement nous assurer l'existence ou non d'une solution, mais en aucun cas l'unicité.

D'où le problème suivant :

```
Problème unicite:
    Entrée: Une grille A admettant au moins une solution.
    Sortie: Un booléen a.
        Si a = VRAI, la grille A admet une unique solution.
        Si a = FAUX, la grille A admet plus d'une solution.
```

Un problème d'évaluation de la difficulté d'une grille :

```
Problème difficulte:
    Entrée: Une grille A admettant au moins une solution.
    Sortie: Un entier naturel a inférieur à 3.
        Si a = 0, la grille A est considérée comme facile.
        Si a = 1, la grille A est considérée de difficulté moyenne.
        Si a = 2, la grille A est considérée comme difficile.
```

Et enfin, un problème de génération de grilles :

```
Problème generation:
    Entrée: Deux entiers pour la taille de la grille souhaitée.
    Sortie: Une grille A incomplète admettant une solution.
```

1.3 Algorithmes

1.3.1 Des types

Il nous faut manipuler des représentations des objets définis précédemment. Ces objets sont en particulier : l'`Etat`, le `Bloc` et la `Grille`.

L'`Etat` est un élément admettant 3 valeurs possibles (`BLACK, WHITE, UNDEFINED`).

Le `Bloc` est un type définit comme suit, selon les explications données précédemment :

```
Bloc:
    - i_début : Entier naturel
    - longueur : Entier strictement positif
```

La `Contrainte` :

```
Contrainte:
    - contrainte : tableau d'entiers naturels
    > nombreDeContraintes : Contrainte -> Entier
        E: Une contrainte C
        S: Nombre d'éléments du tableau C.contrainte
        Ex: (1, 2, 5) -> 3
```

Enfin la `Grille` est définie par :

```
Grille:
    - case : tableau d'Etats
    - contrainteLigne : tableau de Contrainte
    - contrainteColonne : tableau de Contrainte
    - nombreDeLignes : entier strictement positif
    - nombreDeColonnes : entier strictement positif
    > ligne : Entier x Grille -> Tableau d'Etats
        E: Grille A et entier i strictement positif et inférieur à A.nombreDeLignes
        S: Tableau d'Etats de longueur nombreDeLignes représentant la ligne d'indice i
        de la Grille A
    > colonne : Entier x Grille -> Tableau d'Etats
        E: Grille A et entier strictement positif et inférieur à A.nombreDeColonnes
        S: Tableau d'Etats de longueur nombreDeColonnes représentant la colonne d'indice i
        de la Grille A
```

```
> estRemplie : Grille -> Booléen
    E: Grille A
    S: Booléen : VRAI si A est remplie, FAUX sinon
> estValide : Grille -> Booléen
    E: Grille A remplie
    S: Booléen : VRAI si A est valide, FAUX sinon
> copie : Grille -> Grille
    E: Grille A
    S: Grille B copie de la grille A
> definirCase : Entier x Grille x Etat -> Grille
    E: Grille A, entier i identifiant une case indéfinie de la grille A
    (1 <= i <= A.nombreDeLignes * A.nombreDeColonnes) et un Etat w = BLACK ou WHITE
    S: Grille B remplissage de A dont la fonction état ne diffère de celle de A qu'en
    la case identifiée par l'indice i, et dont l'image par cette fonction est w.
> sauverLigne : Tableau d'Etats x Entier x Grille -> Grille
    E: Une grille A, un tableau d'Etats t de longueur la longueur des lignes dans A et
    un indice valide d'une ligne i de A
    S: Une grille B identique à A mais dans laquelle la ligne i est égale au tableau t.
> sauverColonne : Tableau d'Etats x Entier x Grille -> Grille
    E: Une grille A, un tableau d'Etats t de longueur la longueur des colonnes dans A et
    un indice valide d'une colonne i de A
    S: Une grille B identique à A mais dans laquelle la colonne i est égale au tableau t.
```

1.3.2 Réponse aux problèmes

resolve

Avant de présenter l'algorithme général, nous devons préciser le rôle d'une fonction qui intervient dès le début en désignant le problème qu'elle résout. La fonction est `resolveCase`, qui résout le problème suivant :

```
Problème resolveCase:
    Entrée: Une grille A
    Sortie: Une grille B remplissage de A telle que si A admet une solution, B aussi.
```

On utilise dans son implémentation l'effet de bord sur la grille passée en entrée. D'autre part, on élargira cette fonction au renvoi d'un entier indiquant le nombre de cases définies en plus lors de l'appel courant de cette fonction, ainsi qu'une valeur spéciale que nous allons détailler immédiatement :

```
> resolveCase : Grille -> Grille x Entier
    E: Grille A
    S: Grille B remplissage de A telle que si A admet une solution, alors B aussi.
       Entier naturel a : * a = -1 si B est remplie
                          * a est égal au nombre de cases définies en plus dans B par rapport à A
                          dans l'exécution courante.
```

Nous considérons par la suite une autre fonction `choisirCase : Grille -> Entier` résolvant le problème suivant :

```
Problème choisirCase
    Entrée: Une grille A
    Sortie: Un entier identifiant une case à l'état UNDEFINED de la grille A.
```

Nous en venons donc à l'algorithme récursif général suivant :

```
fonction resolve(A : Grille) -> Grille * Booléen :
    res <- resolveCase(A);
```

```
Si (res = -1) alors
    Retourner (A, estValide(A));
Sinon si (res > 0) alors
    Retourner resolve(A);
Sinon si (res = 0)
    case <- choisirCase(A);
    B <- copie(A);
    definirCase(case, B, BLACK);
    (G, b) <- resolve(B);
    Si (b != VRAI) alors
        definirCase(case, A, WHITE);
        (G, b) <- resolve(A);
    Retourner (G, b);
```

Le cas de base `res = -1` est le cas où la grille en entrée est remplie, il nous reste alors à déterminer sa validité. Le cas suivant est celui où l'on vient de définir des cases dans la grille, mais il en reste qui sont indéfinies, alors on rappelle la fonction avec la grille pour continuer à résoudre. Enfin le dernier cas est un peu plus complexe : il correspond au moment où notre fonction `resolveCase` n'a pas réussi à définir de nouvelles cases. Auquel cas nous n'avons pas le choix, il nous faut "deviner" l'état d'une case indéfinie. On choisi arbitrairement de commencer par définir cette case à l'état `WHITE`, puis on regarde si l'on trouve une solution avec cette hypothèse. Si c'est le cas, nous avons notre retour, sinon, on essaye avec cette case à l'état `BLACK`. On retourne alors en fonction du résultat de cette dernière hypothèse.

Intéressons-nous maintenant de plus près à l'algorithme de résolution sur les lignes et les colonnes `resolveCase`. L'idée est de considérer une case, et **uniquement avec les contraintes de sa ligne et de sa colonne**, essayer de la définir logiquement. Cet algorithme s'appuie sur deux sous-fonctions fondamentales pour notre problème : `alignerGauche` et `alignerDroite` dont nous allons détailler immédiatement les problèmes résolus ainsi que les spécifications.

Expliquons de manière intuitive l'idée de l'algorithme. Il se place dans l'algorithme de résolution comme un extracteur des informations d'une ligne et d'une colonne ainsi que de leurs contraintes respectives. En effet, on peut déterminer grâce à une heuristique simple la couleur de certaines cases d'une ligne ou d'une colonne.

Par exemple, considérons la ligne suivante :

```
3 2 |?|?|?|?|?|?|?|
```

On détermine ensuite quels sont les arrangements où toutes les cases noires sont le plus à droite, puis le plus à gauche. Suivant les règles du jeu, la solution est triviale :

```
3 2 |B|B|B| |B|B| |
3 2 | |B|B|B| |B|B|
```

En considérant maintenant les blocs correspondants dans ces deux arrangements, on constate qu'ils se recouvrent sur certaines cases. Or, étant dans leurs positions les plus extrêmes et éloignées l'une de l'autre, il s'agit de cases qui seront toujours `BLACK`. On en déduit par conséquent la ligne mise à jour suivante :

```
3 2 |?|B|B|?|?|B|?|
```

Cette ligne pourra se retrouver complétée à la fin de notre algorithme dans une grille solution si elle existe.

On comprend alors qu'en itérant ce principe sur les lignes et les colonnes, on va pouvoir **compléter successivement des cases et même dans certains cas simples, résoudre la grille** uniquement via cette heuristique. Dans ce cas, la grille obtenue est alors **certifiée comme n'ayant qu'une solution**, puisqu'on émet aucune hypothèse supplémentaires sur l'état des cases que celles fournies par les contraintes.

On détermine également des cases blanches. C'est en effet le cas de cases entre des blocs qui se recouvrent entièrement ou entre ces blocs et un des bords de la ligne.

Là où l'algorithme se complique, c'est lorsque l'on a une ligne ou colonne partiellement résolu. Non content d'obtenir la possibilité où les blocs sont le plus à gauche/droite, nous devons en plus la faire correspondre avec la ligne/colonne déjà partiellement résolue, ce qui justifie l'emploi du mot "possibilité". Le problème résolu par `alignerGauche` est ainsi le suivant :

```
Problème alignerGauche
    Entrée: Ligne ou colonne partiellement résolue ainsi que sa contrainte
    Sortie: Ligne ou colonne remplie représentant la possibilité où les blocs sont placés
    le plus à gauche.

Problème alignerDroite
    Entrée: Ligne ou colonne partiellement résolue ainsi que sa contrainte
    Sortie: Ligne ou colonne remplie représentant la possibilité où les blocs sont placés
    le plus à droite.
```

Nous avons enfin parlé de l'intersection bloc à bloc des possibilités, réalisée par la fonction `blocEt(bufferGauche, bufferDroite : Tableaux de Blocs, buffer : tableau d'Etats)` que nous ne détaillerons pas, mais qui résout le problème suivant :

```
Problème blocEt :
    Entrée: Deux tableau de blocs pour l'alignement à gauche et à droite
    Sortie: Leur intersection blocs à blocs
```

Par soucis de simplicité, nous allons regrouper la fonctionnalité d'alignement à gauche/droite et d'intersection au sein d'une fonction (resp. pour les lignes et les colonnes), qui résout un problème plus général :

```
bufferLigne : Entier x Grille -> Tableau de blocs x Tableau de blocs x Entier
    Entrée : i : indice d'une case, une grille A
    Sortie : Une entier a et G, D : tableaux de blocs représentants respectivement les alignements à
    gauche et droite compatibles avec la ligne de la case i de la grille A si cela est possible,
    auquel cas a est leur nombre de blocs.
    Sinon s'il y a incompatibilité (contradiction), a = -1.
```

Comme il y a effet de bord, on considérera que l'on fourni à la fonction les tableaux de blocs `G` et `D`, et qu'elle ne renvoie que l'entier `a`.

Avec ce nouveau bagage, on peut maintenant donner l'algorithme de la fonction fondamentale `resolveCase`, définie comme suit :

```
fonction resolveCase(G : Grille) -> Statut :
    case <- choisirCase(G);
    Si (case = -1) alors
        Retourner case;
    ajoute <- 0;
    sauver <- case;
    bufferGauche, bufferDroite : tableau de blocs;
    buffer : tableau d'Etats;
    Tant que (ajoute = 0) faire
        bloc <- bufferLigne(bufferGauche, bufferDroite, case, G);
        Si (bloc = -1) alors
            Retourner bloc;
        blocEt(bufferGauche, bufferDroite, buffer, G);
        ajoute += sauverLigne(buffer, case, G);
        bloc <- bufferColonne(bufferGauche, bufferDroite, case, G);
        Si (bloc = -1) alors
            Retourner bloc;
        ajoute += sauverColonne(buffer, case, G);
        case++;
        Si (case = save) alors
            Retourner ajoute;
```

Nous avons donc introduit un certain nombre de fonctions auxiliaires qu'il nous faut maintenant éclaircir. Cette fonction suit la méthode usuelle que nous avons décrite sur l'exemple : les fonctions

`chargerLigneGauche(ligne : Entier, G : Grille) -> Tableau de blocs` et `chargerLigneDroite` (symétrique) se contentent de charger la possibilité la plus à droite et la plus à gauche. Commençons par `bufferLigne` :

```
fonction bufferLigne(g, d : Tableaux de blocs, case : Entier, G : Grille) -> entier :
    ligne <- getLigne(case, G);
    g <- chargerLigneGauche(ligne, G);
    d <- chargerLigneDroite(ligne, G);
    resG <- alignerGauche(g, ligne, G);
    Si (resG = -1) alors
        Retourner resG;
    resD <- alignerDroite(d, ligne, G);
    Si (resD = -1) alors
        Retourner resD;
    Retourner nombreDeContraintes(G.contrainteLigne[i]);
```

La nouvelle méthode réellement intéressante est la méthode d'alignement (`alignerGauche` et `alignerDroite`). Pour en comprendre l'algorithme, il faut le faire fonctionner sur un exemple :

```
3 2 |?|?|B|?|?|?|?|B|?|?|
```

Considérons dans une grille quelconque la ligne précédente, partiellement déterminée. L'objectif de l'alignement est de faire correspondre les blocs avec cette ligne (si possible). Nous allons donc décaler successivement les blocs jusqu'à obtenir un alignement favorable pour chacun d'eux. Voyons cela en action :

```
|?|?|B|?|?|?|?|B|?|?| : SOURCE

|B|B|B| |B|B| | | | | : le premier bloc est compatible, on s'intéresse au second.

|B|B|B| |B|B| | | | | : le second est compatible, on s'intéresse à la ligne entière
|B|B|B| |B|B| | | | | : il y a incompatibilité sur la ligne, on décale le dernier bloc déplacé
|B|B|B| | |B|B| | | | : incompatibilité sur la ligne, on décale.
|B|B|B| | | |B|B| | | : premier compatible, second compatible : on a l'alignement !
```

De la même façon, on aligne à droite :

```
|?|?|B|?|?|?|?|B|?|?| : SOURCE

| | | | |B|B|B| |B|B| : incompatibilité sur la ligne
| | | |B|B|B| | |B|B|
| | |B|B|B| | | |B|B|
| |B|B|B| | | | |B|B|
|B|B|B| | | | | |B|B| : on arrive au bout, on recolle les blocs au premier, et on décale
| | | | |B|B|B| |B|B|
| | | |B|B|B| |B|B| | : incompatible
| | |B|B|B| | |B|B| | : la ligne est compatible, on a l'alignement à droite
```

Ces deux algorithmes sont les plus complexes. Il restent néanmoins similaires, bien que l'implémentation en pratique nécessite de la rigueur pour les cas limites.

Expliquons avant le rôle des méthodes `estCompatible` et `deplacerGauche` que nous allons utiliser, sans trop détailler l'algorithme qui reste assez intuitif.

On a :

```
function deplacerGauche(t : Tableau de blocs, deplacements : entier) -> entier
    0 si le décalage de deplacement blocs est impossible,
    1 sinon
```

qui va décaler `deplacements` blocs d'un cran vers la gauche si possible (en laissant les autres en place, comme on l'a vu dans l'exemple), et

```
function estCompatible(source : tableau d'Etats, t : tableau de Blocs, n : Entier) -> Entier
    0 si il y a compatibilité sur l'intervalle [1,n] entre la source et t
    1 sinon.
```

qui va vérifier la compatibilité comme nous l'avons pratiqué dans l'exemple entre la ligne possible en cours et la source.

```
fonction alignerGauche(g : Tableau de blocs, ligne : Entier, G : Grille) -> Entier :
    nombreDeBlocs <- deplacements <- longueur(g);
    source <- ligne(ligne);
    valide <- 0;
    Tant que (valide != 1) faire
        valide = 0;
        Si (deplacements = 0) alors
            Si (estCompatible(source, g, G.nombreDeColonnes)) alors
                valide = 1;
            Sinon
                deplacements++;
                Si (deplacerGauche(g, deplacements) = 0) alors
                    Retourner -1;
        Sinon si (estCompatible(source, g, g[nombreDeBlocs - deplacements].i_begin
                                        + g[nombreDeBlocs - deplacements].longueur)) alors
            deplacements--;
        Sinon si (deplacerGauche(g, deplacements) == 0) alors
            Pour i = (nombreDeBlocs - deplacements) à nombreDeBlocs faire
                g[i].i_begin = g[i-1].i_begin + g[i-1].longueur + 1;
            Si (deplacements != nombreDeBlocs) alors
                deplacements++;
                Si (deplacerGauche(g, deplacements) == 0) alors
                    Retourner -1;
            Sinon
                Retourner -1;
    Retourner 0;
```

La difficulté d'écriture de cette fonction vient du fait que **l'on cherche à y détecter les erreurs** (qui peuvent être nombreuses), ce qui explique le nombre de conditions a priori effrayant.

Nous avons couvert les méthodes importantes de notre algorithme. Les autres, plus utilitaires, que nous n'avons pas détaillées ont étés implantées en C. Cependant, il nous reste une part très importante à traiter et qui concerne **ce qui se passe quand notre algorithme de résolution ligne par ligne échoue**. À ce moment précis, il nous faut "deviner" l'état d'une case, c'est ce que nous avons présenté dans le premier algorithme. On choisit arbitrairement un état pour une case, puis on relance l'algorithme ligne par ligne, et ainsi de suite, jusqu'à trouver une contradiction.

C'est précisément **la recherche d'une contradiction qui est la clé de cet algorithme**, car dès lors que l'on en trouve une, on peut modifier le dernier choix effectué, et continuer, jusqu'à éventuellement retrouver une contradiction ou trouver une solution.

L'algorithme présenté ici est un algorithme qui fait un **parcours en profondeur dans un arbre binaire équilibré**, ce qui est fortement inefficace. Si **on ne peut éviter la recherche dans cette arbre**, c'est à dire la complexité exponentielle, on peut au moins **orienter** l'algorithme pour retrouver une solution rapidement. Comme nous l'avons souligné, ceci ne peut se faire que par la découverte rapide de contradiction, il faut donc considérer des cases qui peuvent amener rapidement des contradictions. Mais d'autre part, relancer souvent l'algorithme de résolution ligne par ligne, qui on le rappelle, termine en passant en revue toute les lignes et colonnes, est fortement contre-productif. Il faut donc faire des sélections de cases judicieuses.

La recherche s'organise par niveaux : dès que l'algorithme de résolution ligne par ligne s'arrête, on consulte les différents niveaux à la recherche de cases qui peuvent être définies par contradiction. Le premier niveau concerne **les cases voisines de la dernière case considérée par l'algorithme ligne par ligne**. Le second niveau sélectionne **les cases voisines des dernières cases ajoutées dans la grille**. Le troisième niveau sélectionne **les cases qui ont au moins deux cases voisines définies**. Le quatrième niveau sélectionne **les cases qui en ont au moins une**. Et enfin, si on ne peux décider d'aucune case dans chacun de ces niveaux, on prendra pour couleur de la case celle qui nous permet de définir le plus de cases possibles en lançant l'algorithme ligne par ligne avec

cette hypothèse (nous n'avons implémenté que cette dernière). L'idée est constante : on se rapproche de cases déjà connues pour essayer de créer des contradictions, qui sont aussi importantes que de trouver un bon état puisqu'il en découle.

Cependant, même avec ces précautions, dès lors que l'on ne peux se baser directement sur les contradictions et que l'on fait un choix arbitraire, **il faut conserver une trace des opérations qui ont été faîtes avant** pour les inverser successivement si besoin. Ceci est permis par la récursivité de notre algorithme, qui permet de lancer le calcul dans une autre direction si le premier n'admet pas le retour attendu.

unicite

Les problèmes d'unicité sont souvent des problèmes bien **plus complexes que la simple existence**, car ils recquierent de passer en revue **toutes** les possibilités pour l'affirmer. Compte-tenu de la nature du problème, on se doute que la preuve d'unicité pourra être très longue à obtenir s'il y a beaucoup de niveaux de récursion qui soulèvent peu de contradictions ou qui les soulèvent à la toute fin. Auquel cas, remonter dans l'arbre sera très long. Un exemple est la grille de test "9-Doom", de taille seulement 19×19, mais qui soulève des contradiction que lorsque l'algorithme en vient à placer les dernières cases...

L'algorithme est simplement un assouplissement de la fonction `resolve`. Au lieu de s'arrêter à la première solution trouvée, on va continuer à chercher une autre solution en ayant incrémenté un compteur. Si on en trouve une de plus, on pourra conclure immédiatement à la non-unicité. Si l'algorithme se termine sans soulever d'autre solution, on aura notre preuve d'unicité.

Remarquons pour conclure sur cette partie que **la démonstration de l'unicité est triviale dès lors que nous n'avons pas eu besoin de deviner la couleur d'une case**. En effet, si la grille est solvable par l'algorithme ligne par ligne seul, qui ne rajoute aucune hypothèse supplémentaire sur l'état des cases, alors la solution est forcément unique.

difficulte

La mesure de la difficulté d'une grille est à considérer du point de vue du lecteur plutôt que de celui de l'algorithme. Nous avons passé peu de temps à étudier cette partie, cependant, les paramètres à retenir dans l'évaluation semblent être les suivants : taille de la grille, nombre d'hypothèses supplémentaires sur les états à émettre dans la recherche d'une solution, unicité ou non des solutions, et enfin proportion de cases noires dans une grille solution.

generation

La génération d'un grand nombre de grilles peut se faire par un processus aléatoire de choix de couleur des cases (ce qui ne donnera évidemment presque aucune grille figurative). Les algorithme précédents permettent alors d'évaluer les différents paramètres de la grille : difficulté, solutions, etc.

1.3.3 Remarque pour choisirCase

Nous avons eu l'idée pour l'implémentation de la fonction `choisirCase` de calculer un taux de criticité, dans un premier temps pour une ligne et une colonne, défini comme le rapport entre le nombre de cases "actuellement" définies à `BLACK` dans la ligne par le nombre de case à définir à `BLACK` pour avoir une ligne valide (la somme des entiers de la séquence de contrainte). Pour une case, on fait la somme de ces taux, ce qui nous donne un moyen de comparer la criticité d'une case, c'est à dire si l'information sur son état va potentiellement plus ou moins (respectivement criticité élevée ou faible) nous permettre de résoudre une ligne ou une colonne entière. Ceci donne d'assez bon résultats, et est toujours mieux que d'itérer sur les cases une par unes.

1.3.4 Complexité

Évaluation dans le meilleur et pire des cas

Commençons par le meilleur des cas. L'algorithme de résolution ne passe jamais par la recherche récursive de la couleur d'une case. On reste sur la méthode ligne par ligne. À chaque itération, on suppose que l'on est capable de déterminer exactement la ligne et la colonne d'une case (en pratique ce sera seulement certains éléments de ces lignes et colonnes). On va considérer pour simplifier une grille de taille $n \times n$. L'algorithme de choix d'une case

non-définie, appelé à chaque itération, calcule pour chacune de ces cases la criticité et choisit la case non-définie de criticité maximale. Cet algorithme s'exécute en $\Theta(n^2)$ en nombre de comparaisons (il faut parcourir toute la grille).

S'en suit un algorithme permettant de générer une possibilité où les blocs sont le plus alignés à gauche, et une autre où ils le sont à droite. Ces deux algorithmes ne consiste qu'en un simple parcours d'une ligne et s'exécutent donc en $\Theta(n)$. Il sont absorbés par la complexité de la fonction précédente. La solution est unique, et donc l'algorithme d'alignement n'appelle que la vérification de la compatibilité, qui s'exécute également en $\Theta(n)$. On a ainsi déterminé la ligne et la colonne de la case considérée.

Finalement, il nous suffit de considérer que n cases pour parvenir à la solution. On en déduit donc que la complexité globale de l'algorithme dans ce cas extrêmement favorable est $\mathcal{O}(n^3)$...ce qui n'est pas formidable pour un cas aussi simple.

Plaçons nous maintenant dans un cas moins favorable, mais toujours sans double appel récursif. On suppose qu'à chaque itération, on ne peut déterminer exactement que la couleur d'une case. A ce moment là, il faut considérer les n^2 cases de la grille, et notre complexité augmente d'une facteur n supplémentaire, donc en $\mathcal{O}(n^4)$.

Profitons de ce calcul pour rappeler que dans le cas général, on peut être amené lorsque notre algorithme de décision ligne par ligne ne trouve rien, à passer en revue la grille entière (puisque cette heuristique de résolution ligne par ligne n'est pas complète) à la recherche d'une case à marquer. Cela arrive souvent en pratique (à commencer lorsque l'on ne trouve vraiment plus rien). On peut donc affirmer sans mal que dans le cas général, l'algorithme de résolution ligne par ligne utilisé avec une seule hypothèse est de complexité au moins $\Theta(n^4)$, ce qui devient fortement inefficace.

Mais remarquons que nous n'avons même pas pris en compte l'algorithme d'alignement complet ! Pour une ligne à n éléments et k blocs, on peut être amené au maximum à effectuer n déplacements. Pour chacun de ces déplacement, on a effectué une vérification de compatibilité sur un intervalle de taille n/k (pour simplifier), amenant donc une complexité pour le décalage en $\Theta(\frac{n^2}{k})$. Ce qui nous donne la complexité totale pour l'alignement en $\mathcal{O}(n^2)$. Ceci change peu de choses compte-tenu de la complexité de notre algorithme de choix d'une case.

Passons au cas où l'on explore l'arbre en entier (pour l'unicité par exemple dans des cas complexes). Considérons notre algorithme naïf consistant à choisir arbitrairement la couleur de la case amenant à un parcours en profondeur de l'arbre. La complexité s'exprime par la formule : $C(n^2) = 2C(n^2 - 1) + n^4$. Ceci se résout immédiatement en $\mathcal{O}(e^{n^2 \ln(2)})$. C'est très inefficace... On comprend tout l'intérêt de rapidement trouver des contradictions pour s'orienter efficacement dans l'arbre.

1.4 Exemple de résolution

Nous allons résoudre pas à pas la grille suivante :

```
   1 1 1 1 1
  -----------
1 |?|?|?|?|?|
  -----------
1 |?|?|?|?|?|
  -----------
1 |?|?|?|?|?|
  -----------
1 |?|?|?|?|?|
  -----------
1 |?|?|?|?|?|
  -----------
```

Notre algorithme va débuter avec la fonction `resolveCase`. Cette fonction fait appel à la méthode `choisirCase` qui dans notre implémentation, av renvoyer l'identifiant de la première case à l'état `UNDEFINED`, ici 1.

La fonction va ensuite faire appel à la résolution de la ligne et de la colonne. on commence donc pour la ligne par charger les buffers droit et gauche, puis on les aligne (ce qui ici est immédiat) :

```
|B| | | | |
| | | | |B|
```

On procède ensuite à l'intersection bloc à bloc, qui ne donne ici aucune information supplémentaire. La fonction `resolveCase` va donc passer en revue toute les cases de la grille, ce qui ne va évidemment apporter aucune information supplémentaire. Elle va donc renvoyer la valeur 0, correspondant aux cases qui ont été déterminées durant cette itération.

On rentre alors dans la procédure d'hypothèse. On va donc copier la grille, puis choisir arbitrairement la case 1 à la valeur `BLACK`.

La nouvelle grille à résoudre est alors la suivante, envoyée par le premier sous-appel récursif :

```
   1 1 1 1 1
  -----------
1 |B|?|?|?|?|
  -----------
1 |?|?|?|?|?|
  -----------
1 |?|?|?|?|?|
  -----------
1 |?|?|?|?|?|
  -----------
1 |?|?|?|?|?|
  -----------
```

On en revient au traitement par la fonction `resolveCase`, qui en choisissant la case 2, va être a même de concilier les buffers gauche et droit, et ainsi de résoudre la première ligne :

```
   1 1 1 1 1
  -----------
1 |B| | | | |
  -----------
1 |?|?|?|?|?|
  -----------
1 |?|?|?|?|?|
  -----------
1 |?|?|?|?|?|
  -----------
1 |?|?|?|?|?|
  -----------
```

Par un nouvel appel récursif, on en vient au traitement par la fonction `resolveCase`, qui en choisissant la case 6, va être a même de concilier les buffers gauche et droit, et ainsi de résoudre la première colonne :

```
   1 1 1 1 1
  -----------
1 |B| | | | |
  -----------
1 | |?|?|?|?|
  -----------
1 | |?|?|?|?|
  -----------
1 | |?|?|?|?|
  -----------
1 | |?|?|?|?|
  -----------
```

Le schéma d'exécution depuis l'hypothèse jusqu'à une ligne et une colonne résolue est le même pour la sous-grille composée des colonnes 2 à 5 et des lignes 2 à 5.

On aboutit à la grille suivante, au bout du 12ème appel récursif (et de la 4ème hypothèse) :

```
   1 1 1 1 1
  -----------
```

```
1 |B| | | | |
  -----------
1 | |B| | | |
  -----------
1 | | |B| | |
  -----------
1 | | | |B| |
  -----------
1 | | | | |?|
  -----------
```

A ce moment, la fonction `resolveCase` est à même de résoudre la dernière ligne et la dernière colonne de façon logique, pour aboutir à la grille G suivante :

```
   1 1 1 1 1
  -----------
1 |B| | | | |
  -----------
1 | |B| | | |
  -----------
1 | | |B| | |
  -----------
1 | | | |B| |
  -----------
1 | | | | |B|
  -----------
```

Elle renvoie alors la valeur `(G,estValide(G))=(G,VRAI)` puisque les contraintes induites sont égales aux contraintes de la grille. On n'effectue jamais de second appel récursif puisque le booléen renvoyé est vrai, et on notifie ainsi la fonction appelante de cette solution qui peut donc l'afficher.

1.5 Idées d'amélioration

La première idée d'amélioration serait l'implémentation du système de niveaux présentés précédemment, qui permettraient de réduire considérablement le nombre d'hypothèses à considérer et de se rapprocher d'une solution rapidement. Si l'on arrive à composer avec la complexité exponentielle, il n'en reste pas moins que notre algorithme de résolution ligne par ligne a une complexité assez élevée par rapport au résultat qu'il fournit. Une solution implémentée sur des solveurs "de compétition" est par exemple une table de hashage, qui permet de contourner ces calculs coûteux et d'appliquer le principe de la programmation dynamique. Enfin, ce dernier algorithme est incomplet, on peut voir qu'en pratique, on ne tire pas le maximum des informations disponibles sur une ligne ou une colonne avec notre simple algorithme de superposition gauche/droite. Une solution serait d'implémenter et d'utiliser quand nécessaire un algorithme complet, plus dur à implémenter et complexe, mais qui évite néanmoins d'émettre potentiellement une hypothèse inutile et d'augmenter inutilement la hauteur de l'arbre...ce qui augmente de façon exponentielle le nombre de nœuds à vérifier !

Chapitre 2

Travail de programmation

2.1 Base du programme

2.1.1 Fichier source

Le travail du programme va se faire à partir d'un fichier texte représentant la grille, ayant une mise en forme particulière.

- Sur la première ligne se trouvent deux entiers séparés par un espace. Ils représentent les dimensions de la matrice : le nombre de lignes, et le nombre de colonnes.
- Sur la seconde ligne se trouvent les séquences de contraintes associées aux lignes de la matrice, séparées par des points virgules.
- Sur la troisième ligne se trouvent les séquences de contraintes associées aux colonnes de la matrice, séparées elles aussi par des points virgules.
- Les lignes suivantes contiennent des séquences de 0, 1 et de 3 représentant des informations considérées comme acquis sur la grille : 1 : case noire, 0 : case blanche, 3 : case indéfinie.

2.1.2 Lecture

Les fonctions assurant la lecture du fichier d'entrée font quelques vérifications de sa syntaxe, et retourneront simplement une erreur dans le cas où la grille récupérée est invalide ! Heureusement, nous fournissons un bon nombre de grilles dont la syntaxe a été testée et vérifiée. Avec plus de temps, nous aurions aimé mettre en pratique l'enseignement sur les Automates finis pour traiter ce type de fichiers.

2.2 Structures en C

Les structures ont été créées de façon identique aux types présentés dans la partie algorithmique. La difficulté supplémentaire par rapport à notre représentation idéaliste d'un tableau par exemple est qu'il nous faut également stocker, et donc envoyer comme argument la taille d'un tableau dès lors que l'on veut s'en servir.

Nous avons d'autre part mis en œuvre l'allocation dynamique étudiée à la toute fin de ce semestre, ce qui exige plus de rigueur pour des fonction telles que `decalageGauche`.

Enfin, nous pouvons profiter de l'utilisation d'énumérations, qui permettent de simplifier les retours entiers de certaines fonctions.

2.3 Résultats obtenus en pratique sur des grilles destinées au solveurs informatiques

Les algorithmes autour des nonogrames instaurent une compétition au sein des développeurs, sur le terrain de certaines grilles difficilement résolubles. Nous nous sommes contentés par manque de temps d'implémenter une méthode basique qui lorsque vient le moment de faire un choix, choisit la solution qui complète le plus la grille. Ceci nous a permis de résoudre une grille qui nous était alors inaccessible avec la méthode DFS de base. Cette petite amélioration, aussi faible soit-elle permet de composer avec la complexité exponentielle sur des grilles pas

trop exigeantes. Malheureusement, cette implémentation (qui représente le cas désespéré dans les niveaux que nous avons présentés précédemment), se révèle futile sur certaines grilles où les contradictions ne se trouvent qu'avec beaucoup d'itérations.

Voici dessous quelques statistiques obtenues sur des grilles que nous avons fournies avec notre programme.

Grille	Taille	Hypothèses	Temps
simple1	5x5	0	1 ms
simple2	10x10	0	3 ms
simple3	25x25	0	220 ms
complex1	5x5	4	3 ms
complex2	6x6	1	2 ms
complex3	30x30	3	1,8 s
complex4	15x15	0	12 ms
Dancer	10x5	0	2 ms
Cat	20x20	0	63 ms
Skid	25x14	0	40 ms
Bucks	23x27	2	550 ms
Edge	11x10	15	32 ms
Smoke	20x20	8	360 ms
Knot	34x34	0	2,8 s
Swing	45x45	0	17,7 s

2.4 Quelques considérations algorithmique

2.4.1 La complexité spatiale

Dès lors que l'on veut sauvegarder l'état d'une grille pour une fonction récursive, afin de pouvoir bifurquer par la suite si le premier appel récursif ne donne pas le retour attendu, il faut dupliquer la grille ce qui a de lourdes conséquences quant à l'espace utilisé. Nous stockons une grille supplémentaire à chaque bifurcation, ce qui nous amène à une complexité spatiale exponentielle ! Ceci d'ailleurs sur des grilles que notre solveur n'est pas capable de résoudre, se traduit par une occupation progressive très importante de mémoire, puisque nous devons composer avec la mémoire finie de nos ordinateurs.

2.4.2 Améliorations dans l'implémentation

Nous n'avons malheureusement pas eu le temps de développer ces améliorations, cependant, l'implémentation des algorithmes permettant de s'orienter rapidement dans l'arbre aurait été plus que bénéfique pour notre programme. Ensuite, nous aurions pu diminuer la complexité spatiale en évitant de stocker une grille entière mais seulement les modifications apportées, dans une séquence particulière par exemple.

De la même façon, notre algorithme de résolution ligne par ligne aurait pu bénéficier, au détriment de la complexité spatiale mais au profit de la complexité temporelle, d'un fichier de stockage des solutions nous permettant de rapidement les retrouver lors de l'exécution de l'algorithme mais aussi, d'avoir un algorithme qui "apprend".

Conclusion

Ce projet et son encadrement nous a mis à l'épreuve d'un point de vue rigueur algorithmique, en nous rappelant de ne pas glisser trop tôt vers l'implémentation en C et le codage pur, mais de plutôt prendre le temps pour une réflexion antérieure, qui au final nous permet d'implémenter l'algorithme plus rapidement. L'autre aspect est la rigueur dans la définition des problèmes eux-mêmes, qui s'il elle n'est pas effectuée adéquatement, peut aboutir à une mauvaise identification du problème posé et à un algorithme faux. Nous avons vu les étapes de la réflexion nécessaire à la résolution d'un problème de ce type, ce qui est enrichissant pour nos études.

Enfin l'objet du problème lui-même, d'apparence didactique, se révèle finalement d'un certain degré de complexité et nous a permis d'avoir un aperçu de ce que peut être l'optimisation, par le manque d'optimisation justement de notre implémentation. Lors de nos recherches, nous avons découverts différentes solutions proposées, qui améliorent sensiblement la résolution pour certaines grilles, et fait même l'objet d'un certain esprit de compétition.