Projet INF441 - Les liens dansants

Sujet proposé par Jean-Christophe Filliâtre

Bruna Halila Morrone et Felipe Vieira Frujeri - X2013

June 2, 2015

1 Introduction

Comme déjà défini dans le sujet du projet, son but a été de mettre en oeuvre l'algorithme des liens dansants (*dancing links*) proposé par Knuth en 2000, pour résoudre des problèmes de pavage.

Le problème que l'on s'attaque est celui de la couverture exacte de matrice (EMC). L'intérêt de le résoudre repose sur le fait qu'il est très facile d'y encoder des nombreux problèmes. Comme par exemple le problème du pavage (traité dans ce projet) et le problème des N-Reines (bonus pour le projet). Mais aussi bien d'autres problèmes qui utilisent la technique du *backtracking*, comme le sudoku.

1.1 Les liens dansants

L'algorithme des liens dansants (DLX) s'utilise d'une structure de donnés qui est un réseau de listes circulaires doublement chaînées horizantelement et verticalement (en topologie toroïdale).

Si on prend comme exemple la matrice indiqué dans l'article de Knuth:

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

On obtient la répresentation comme dans la Figure 1

L'ingéniosité de l'algorithme est due notamment à la possibilité de réinsérer un élément qui vient d'être supprimé de la structure (pour le *backtracking* sans utiliser d'autre information que celle qui se trouve déjà dans les deux pointeurs vers ses anciens voisins.

1.2 Problème du pavage

Comme bien détaillé dans l'article, le problème de Scott, qui s'agit de paver un échiquier évidé comme dans la Figure 2 peut être réduit à l'EMC. Pour construire la matrice, on modélise chaque case à paver

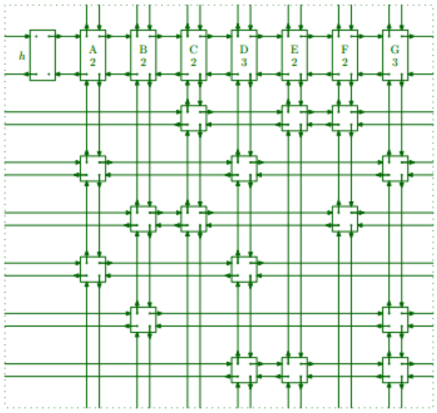

Figure 1: Structure en grille doublement chaînée

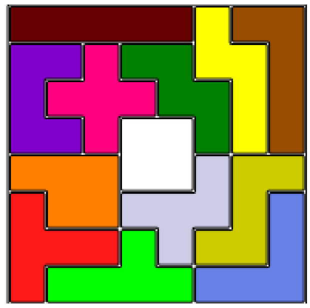

Figure 2: Problème de Scott *Pentominoes*

du échiquier comme une colonne et on ajoute de plus une colonne pour chaque pièce. Ainsi, chaque ligne correspond à une façon de poser un pentamino (ou plus généralement un n-omino) sur l'échiquier, et on peu utiliser l'algorithme DLX. On peut aussi considérer (éliminer) des symétries du problème, en considérant qu'une seule orientation possible pour l'une de pièces complètement asymétriques.

2 Organisation du code

On a pris le soin de bien modulariser le code. Pour cela on a créé une classe pour chaque tâche à rendre, en séparant l'algorithme DLX dans les classes *ExactCoverProblem* et *TilingProblem* pour les tâches obligatoires et *NQueensProblem* pour la tache bonus. Le diagramme UML (Unified Modeling Language) du projet est présenté dans la figure 3.

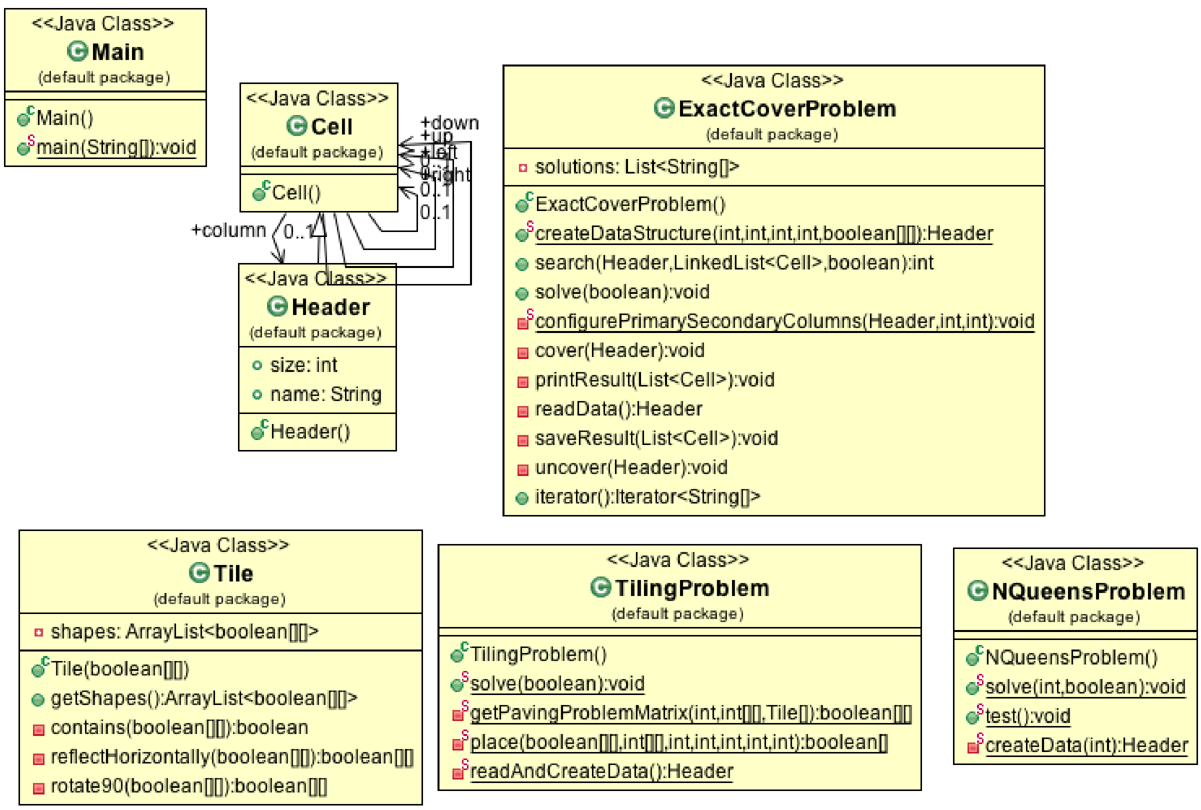

Figure 3: Diagramme UML du projet avec les relations entre les classes.

2.1 Couverture exacte de matrice

```
public class ExactCoverProblem implements Iterable<String[]>{
    private List<String[]> solutions;

    public static Header createDataStructure(int numCols, int numRows,
            int numPriCols, int numSecCols, boolean[][] matrix);
    public int search(Header h, LinkedList<Cell> output, boolean printResults){
    public void solve(boolean printSolutions){
    private static final void configurePrimarySecondaryColumns(Header h,
            int numPriCols, int numSecCols);
    private void cover(Header column);
    private void printResult(List<Cell> output);
    private Header readData();
    private void saveResult(List<Cell> output);
    private void uncover(Header column);
}
```

2.1.1 Structure de données et lecture de l'entrée

Comme indiqué dans l'article, pour implémenter chaque case de la structure doublement chaînée, on a créé un *data object* appelé *Cell*:

```
public class Cell {
    public Cell left, right, up, down;
    public Header column;
}
```

Cette classe possède quatre champs en pointant vers chacun des ses voisin (*left*, *right*, *up*, *down*) et un champ qui pointe vers l'entête (*Header*) de la colonne correspondante (C). Cette entête est un autre type d'objet crée (cité dans l'article comme *column object*. Cette classe hérite de *Cell* et en ajoute deux champs, pour indiquer le nombre de cases dans la colonne et une étiquette *name* pour bien gérer le résultat de l'algorithme.

```
public class Header extends Cell {
    public int size;
    public String name;
}
```

Pour faire la lecture de l'entrée, on utilise la méthode *readData()*, qui à son tour s'utilise de la méthode *createDataStructure()* pour créer la grille et renvoie la racine des entête h, appelé *root* dans l'article.

2.1.2 Choix de la prochaine colonne à couvrir

Le prochain point important de l'implémentation a été le choix de la prochaine colonne à être couverte dans le *backtracking*. On a implémenté, basé dans l'article, deux façons de le faire: soit en prenant la colonne non couverte la plus à gauche, soit en utilisant l'heuristique s, en choisissant la colonne avec le plus petit nombre de $1s$. Cette dernière permet de minimiser le facteur de branchement de l'arbre de possibilités.

2.1.3 Problème de la couverture et Découverture d'une colonne

Une fois que l'on a choisit la colonne a couvrir, on appelle la méthode *cover(Header c)* qui, enlève c de la liste de l'entête et enlève toutes les lignes dans la propre liste de c des autres colonnes. Pour supprimer chaque *Cell* x on utilise l'opération base de l'article:

$$L[R[x]] \leftarrow L[x], R[L[x]] \leftarrow R[x]$$

Et l'opération de découverture se produit en ordre reverse de l'opération de coverture, en découvrant les lignes dans le sens *bottom-top*. L'operation dans caque *Cell* est l'inverse de la dernière:

$$L[R[x]] \leftarrow x, R[L[x]] \leftarrow x$$

2.1.4 L'algorithme DLX - *backtracking* et Solutions

Ainsi, avec toutes le modules partiel déjà prêts, la méthode récursive *search(Header h, LinkedList< Cell > output, boolean printResults)* rassemble toutes et exécuté le *backtracking* jusqu'à le voisin $R[h] = h$, quand on peut renvoyer les solutions accumulées dans la list *output*.

Comme remarque, pour encapsuler la récursion de *search(Header h, LinkedList< Cell > output, boolean printResults)* on a créé la méthode *public static void solve(boolean printSolutions)* qui reçoit l'argument *printSolutions* pour définir si elle imprime ou pas les résultats. Les résultats sont imprimés comme suggérée dans l'article, c'est-à-dire les lignes couvrants et dans chaque ligne les colonnes étiquetés par des lettre (on peut étiquetés avec des numéros dans le cas plus grands).

Pour satisfaire les spécifications d'avoir un *iterator* la classe *ExactCoverProblem* implémente l'interface *Iterable< String[] >* et les solutions sont estoqués par la méthode *saveResult(List< Cell > output)*.

2.1.5 Différentiation entre colonnes primaires et secondaires

Comme dernière variante de l'algorithme on a ajouté la possibilité de données des importances relatives aux colonnes (ce qui a été utilisé dans le problème des N-Reines). Pour l'implémenter, on a choisit de laisser les *numPriCols* premières colonnes comme primaires et les *numSecCols* dernières colonnes comme secondaires. La mise en oeuvre se fait à travers de la méthode *configurePrimarySecondaryColumns(Header h, int numPriCols, int numSecCols)*, qui change le réferences R et L des entêtes de colonnes secondaires pour elles même.

2.2 Problème du pavage

2.2.1 Lecture de l'entrée et réduction au problème EMC

La lecture de l'entrée est très simple: on lit l'échiquier et les différentes pièces (n-ominos) à poser. La structure de données utilisée pour encoder les pièces est décrite dans 2.2.2

Ensuite vient la réduction au problème EMC. On considère, pour chaque pièce, toutes les positions possibles de la poser, ainsi que les rotations et réflexions qui donnent une pièce différente. À chaque position et orientation possible d'une pièce, correspond une ligne de la matrice. Cette matrice contient toutes les cellules à être remplies dans le tableau, ainsi qu'une colonne correspondante à chaque pièce, pour garantir que celle-ci soit posé une et une seule fois.

La classe responsable pour résoudre le problème du pavage s'appelle *TilingProblem*, et ses méthodes se trouvent listés ci-dessous:

```
public class TilingProblem {
    public static void solve(boolean printSolutions);
    private static boolean[][] getPavingProblemMatrix(
        int numAvailableSpaces, int[][] positionsMatrix, Tile[] tiles);
    private static boolean[] place(boolean[][] shape, int[][] positionsMatrix,
            int row, int column, int nRows, int nCols, int totalNumColumns);
    private static Header readAndCreateData();
}
```

2.2.2 La classe *Tile*

La classe *Tile* a été crée pour répresenter des pièces à poser dans l'échiquier. Il faut noter qu'une pièce peut être positionnée de plusieurs façons (soit par rotation ou réflexion). La classe *Tile* prend en compte ce fait via l'attribut *shapes*, un *ArrayList¡boolean[][]¿* qui contient toutes les variations distinctes pour une pièce. Cet attribut est construit à l'aide des méthodes privés de la classe:

```
public class Tile {
    private ArrayList<boolean[][]> shapes;

    public Tile(boolean[][] shape);
    public ArrayList<boolean[][]> getShapes();

    private boolean contains(boolean[][] shape);
    private boolean[][] reflectHorizontally(boolean[][] shape);
```

```java
    private boolean[][] rotate90(boolean[][] shape);

}
```

2.3 Problème des N reines

Le très connu problème des N reines est un bon exemple de problème qui peut être réduit au problème EMC. Pour y encoder, il suffit de construire une matrice en considérant les lignes et colonnes de l'échiquier comme des colonnes primaires, i.e., qui doivent obligatoirement être remplies et, ensuite, les diagonales et antidiagonales comme des colonnes secondaires. Ce dernier pas est nécessaire pour garantir qu'il n'y ait pas de reines placées dans les mêmes diagonales.

Finalement, une possible optimisation est d'ignorer les 2 diagonales et 2 antidiagonales dans les extremités de l'échiquier pour construire la matrice, une fois qu'il y a au maximum une seule reine qui occupe chacun d'elles. Il y a aussi des optimisations qui considèrent un ordre différent des colonnes de la matrice, afin de réduire les branchements de la récursion.