LES LIENS DANSANTS

Projet INF441 2#

Diego de Souza | Henrique Gasparini

SOMMAIRE

1 LA COUVERTURE EXACTE DE MATRICE

Le problème de la couverture exacte de matrice (EMC) est un problème d'optimisation combinatoire NP-complet. Étant donnée une matrice contenant uniquement des 0 et des 1, il s'agit de déterminer un sous-ensemble de ses lignes contenant un 1 et un seul par colonne. Pour exemplifier, les lignes $0, 4$ est une solution possible du problème EMC suivant.

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

Une variante de ce problème est d'avoir des colonnes primaires, qui doivent nécessairement être couvertes, et des colonnes secondaires, qui peuvent ne pas être couvertes.

De nombreux problèmes peuvent être réduits au problème d'EMC. Dans ce projet nous avons montré deux applications : le problème de pavage et la solution d'un jeu Sudoku quelconque.

2 LES LIENS DANSANTS

Les liens dansants sont une structure de données utilisée dans l'algorithme X, développé par Donald Knuth, qui permet de résoudre le problème de la couverture exacte de matrice.

Dans l'approche de notre projet pour ce problème, nous avons créé une classe "DLXTable" qui représente cette structure et qui est munie d'une méthode Solve() récursive qui implémente l'algorithme X. Pour ce faire, on a dû aussi implémenter d'autres méthodes auxiliaires, comme les méthodes pour lire l'entrée, pour couvrir/découvrir une colonne, sauvegarder et imprimer les sorties de l'algorithme.

Dans le code que nous vous avons fourni, on imprime la quantité totale de solutions ainsi que la première solution. Si besoin, notre code peut être très facilement changé pour imprimer toutes les solutions.

En ce qui concerne les structures auxiliaires, nous avons utilisé deux classes "Cell" et "Columns" (qui hérite de Cell) pour représenter les cellules avec des 1's et les cellules correspondant aux colonnes dans la structure des liens dansants.

En ce qui concerne l'algorithme, nous avons utilisé l'optimisation suggérée par l'article de Donald Knuth, selon laquelle il fallait toujours couvrir la colonne avec le moins d'éléments restants avant de faire l'appel récursif. Cela permet d'avoir moins de branches dans l'arbre de récursion.

3 APPLICATION AU PROBLÈME DU PAVAGE

Le problème du pavage peut être résolu utilisant l'algorithme X. Pour cela, on doit interpréter l'entrée donnée (matrice de caractères et pièces). Chaque élément significatif de la

matrice est représenté par une colonne ainsi comme chaque pièce. Chaque ligne représente une position possible d'une pièce. Pour exemplifier cela, considère la matrice suivante

$$\begin{bmatrix} * & * \\ * & * \end{bmatrix}$$

et les deux pièces

$$\begin{bmatrix} * & * \\ & * \end{bmatrix} \qquad \begin{bmatrix} * \end{bmatrix}$$

Alors, la matrice que représente cette entrée a 6 colonnes. Les quatre premières pour représenter chaque élément de la matrice et les deux dernières pour les deux pièces. Les configurations

$$\begin{bmatrix} p1 & p1 \\ * & p1 \end{bmatrix} \qquad \begin{bmatrix} * & p2 \\ * & * \end{bmatrix}$$

où $p1$ représente la pièce 1 et $p2$ la pièce 2, peuvent être représentés pour les lignes suivantes, respectivement

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Ainsi, après construire la matrice avec toutes positions possibles pour chaque pièce (considérant les rotation, réflections, etc) trouver une couverture exacte est le même que trouver une solution pour le problème du pavage.

En ce qui concerne le code, la classe abstraite *Element* a des attributs nombre de lignes, nombre de colonnes et une matrice de caractères, et représente un élément du type *Board* ou du type *Piece*. La classe *Board* hérite de la classe *Element* ainsi comme la classe *Piece*. Cette dernière a des méthodes pour donner les 8 symétries possibles, et une méthode qui donne la matrice correspondant à toutes les positions possibles de la pièce. La classe *Pavage* contient un *Board* et une liste des pièces du type *Piece*. Elle a des méthodes pour lire l'entrée, pour visualiser une solution en utilisant l'interface graphique, et une méthode qui donne la matrice qui représente le problème du pavage.

4 APPLICATION AU PROBLÈME SUDOKU

Nous avons voulu profiter de la structure des liens dansant pour résoudre un deuxième problème qui nous intéressait : il s'agit de résoudre le Sudoku 16x16. En effet, alors que le Sudoku 9x9 peut être résolu avec un Bactracking assez simple, le Sudoku 16x16 ne peut être rapidement résolu que avec des algorithmes plus complexes, comme l'algorithme X.

Pour ce faire, nous avons implémenté deux nouvelles classes : une classe SudokuTable (qui hérite de la classe DLXTable) et une classe Sudoku.

La classe SudokuTable redéfini le constructeur pour qu'il prenne seulement une matrice de booléennes comme argument (donc l'attribut SecondaryColumns vaut toujours 0). Elle redéfinit aussi la méthode printSolutions() (*overriding*) pour qu'elle n'imprime plus une solution du problème de la couverture exacte, mais une solution du problème du Sudoku. Son principal intérêt est de pouvoir imprimer toutes les solutions possibles. En effet, cela nous premettrait d'imprimer chaque solution dès qu'on la trouve en utilisant la méthode Solve() de la classe parent. Ainsi, on n'aurait pas besoin de stocker toutes les solutions dans une Collection pour pouvoir les imprimer après que la méthode Solve() termine. Pourtant nous avons finalement décidé d'imprimer une seule solution pour le problème du Sudoku et donc nous n'avons pas exploité cet avantage de notre implémentation.

La classe Sudoku contient un attribut "dancingTable" qui est un objet de la classe SudokuTable. Elle est responsable de lire l'entrée et générer la matrice correspondant au problème de couverture exacte dans le champs "dancingTable". Elle contient des méthodes Solve() et printSudokuSolution() qui appellent des méthodes de la classe SudokuTable sur le champs "dancingTable" pour résoudre le problème.

La construction de la SudokuTable à partir de l'entrée (qui est un tableau de Sudoku), qui est une implémentation de la structure de liens dansants, se fait à partir de la matrice de booléennes du problème de couverture exacte correspondant. Cette matrice est une matrice de taille N^3 x $3N^3$, dans laquelle à chaque cellule du tableau sont associées N lignes correspondant aux N choix possibles pour le chiffre qu'on dans cette cellule, à chaque apparition d'un chiffre dans une ligne du tableau de Sudoku on associe une colonne, à chaque apparition d'un chiffre dans une colonne du tableau de Sudoku on associe une colonne et à chaque apparition d'un chiffre dans un sous-tableau on associe une colonne. Ainsi, choisir un ensemble de lignes correspond à choisir les chiffres qu'on écrira dans chaque cellule du tableau de Sudoku. Si le tableau est déjà partiellement rempli, le choix de certains chiffres est déjà fixé, donc il faut enlever certaines colonnes avant d'appeler la méthode Solve() (qui implémente l'algorithme X).

5 ORGANISATION DU CODE

Nous avons utilisé un total de 13 classes dans le cadre de notre projet, en faisant attention à des concepts travaillés dans le cours d'INF441, comme l'abstraction, l'organisation et la clarté de notre code.

Nous avons décidé de schématiser cette organisation dans le diagramme UML ci-dessous en illustrant les rapports entre chacune des classes implémentées.

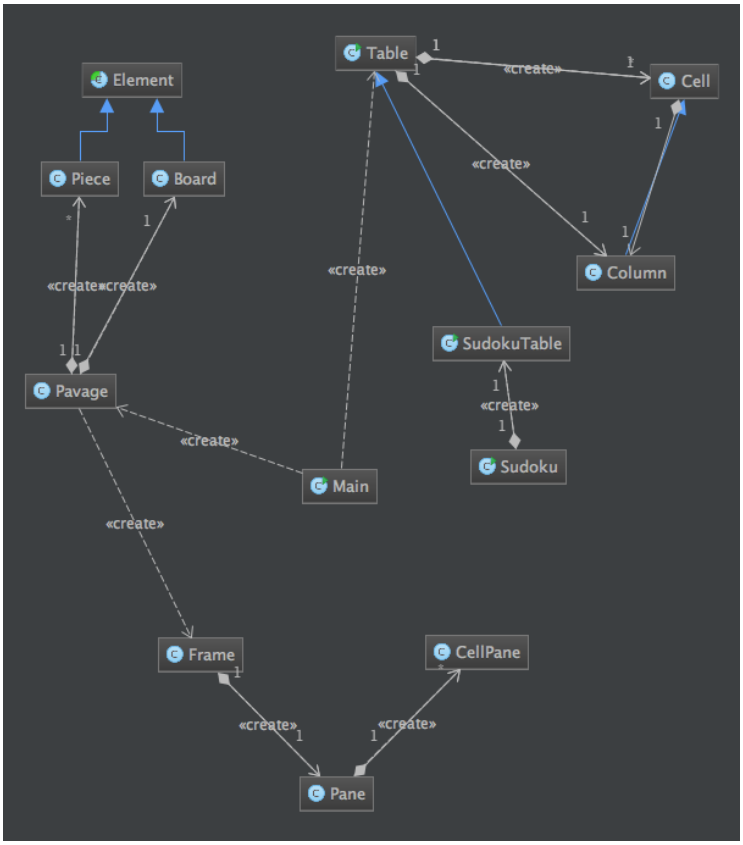

FIGURE 1 – Diagramme UML

6 RÉSULTATS

Les algorithmes implémentés ont tous donné des bons résultats sur les entrées fournies dans la page du cours d'INF441.

Pour le problème du pavage, nous avons affiché les solutions dans une fenêtre graphique et obtenu les figures 2, 3, 4, 5 et 6. Nous avons aussi compté le nombre de solutions, dont la valeur trouvée pour l'entrée *scott.txt* a été 520. Cette valeur ne considère pas toutes les symétries du problème. On pourrait même calculer

$$\frac{520}{8} = 65$$

pour trouver le bon résultat, mais ce calcul n'est pas censé de donner le bon résultat dans tous les cas. Dans cet exemple, il fonctionne parce qu'il correspond à fixer la position d'une pièce qui n'a aucune symétrie, ce qui fait qu'une rotation d'une configuration ne puisse pas donner une autre, par exemple. S'il n'y avait pourtant pas une telle pièce, le calcul effectué serait différent. Bien que l'analyse de ces cas ne soit pas très difficile, nous avons choisi de ne pas la faire vu que nous avions déjà obtenu des résultats très satisfaisants.

Pour le problème du Sudoku, nous avons aussi obtenu des bons résultats dans plusieurs entrées. Pour faire les testes, nous avons adapté notre code pour le soumettre sur un juge en ligne (plus précisément, le lien pour le problème est http ://www.spoj.com/problems/SUDOKU/). On l'a écrit dans un fichier unique "Sudoku.java", qui est dans le répertoire "SPOJ". Bien que la solution ait été trop lente pour le juge en ligne (où le temps limite était de moins de 3 secondes), nous l'avons testée dans nos ordinateurs sur des entrées que nous avons obtenues sur le site du modal algorithmes et elle a donné de bons résultats pour toutes les entrées dans moins de 10 secondes.

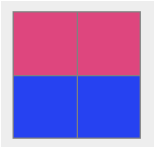

FIGURE 2 – Chessboard 2x2

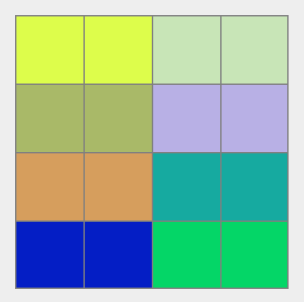

FIGURE 3 – Chessboard 4x4

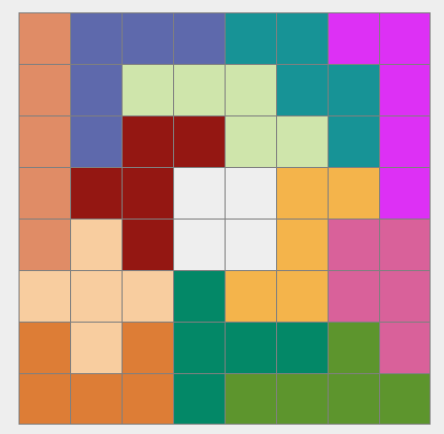

FIGURE 4 – Scott

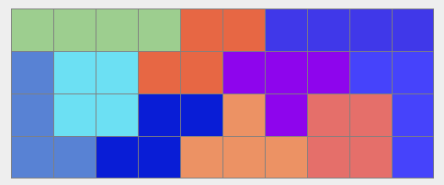

FIGURE 5 – Tetris 4x10

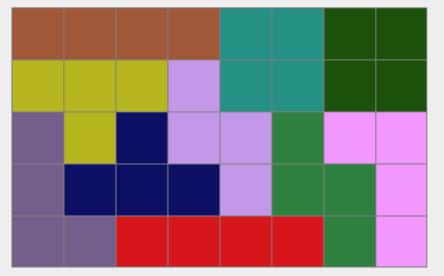

FIGURE 6 – Tetris 5x8

RÉFÉRENCES

[1] Donald E. Knuth. Dancing links. Millenial Perspectives in Computer Science, 2000. http://arxiv.org/abs/cs/0011047.

[2] http://www.enseignement.polytechnique.fr/informatique/INF441/projets/dlx/sujet.pdf

[3] www.enseignement.polytechnique.fr/informatique/INF474A/TD1/index.html