POLYTECH LYON

Thomas RICHARD
Pierre BINAULD
2012-2013

GRAPHES ET APPLICATIONS

PROJET : ROTATION

I. Objectifs

Le but de ce projet est de développer une application permettant la résolution du problème suivant :

Partant d'une matrice 3×3 ne contenant qu'une seule fois chaque chiffre de 1 à 9 dans le désordre, et en effectuant uniquement des rotations des 4 sous-matrices 2×2, réordonner la matrice.

Il faut également répondre aux questions suivantes :

- Combien y a-t-il de matrices non ordonnables ?
- Quel est le nombre de rotations minimal suffisant pour ordonner toute matrice ordonnable ?

II. Réalisations

Nous n'avons pas jugé utile d'intégrer du code étranger à notre programme. Ce projet est le résultat d'une réflexion et d'un travail personnels.

1. Modélisation du problème

Pour résoudre le problème, nous construisons un arbre tel que :

- Chaque nœud est une matrice.
- La racine est la matrice ordonnée.
- Chaque nœud a pour fils les matrices à une rotation de distance qui ne sont pas déjà dans l'arbre.
- Le parcours est effectué en profondeur pour traiter toutes les matrices d'un niveau avant de passer au suivant.

Cela correspond à l'algortihme suivant :

```
1  // queue est la file permettant le parcours en profondeur
2  QQueue<Matrix *> queue;
3  // _tree est la matrice ordonnée (la racine de l'arbre)
4  queue.enqueue(_tree);
5  // matrix contiendra la matrice en cours de traitement
6  Matrix * matrix = NULL;
7  // child contiendra le prochain fils à ajouter
8  Matrix * child = NULL;
9
10 // Tant que la file n'est pas vide
```

```
11  while(!queue.isEmpty())
12  {
13    // On récupère son premier élément dans matrix
14    matrix = queue.dequeue();
15    // Si la matrice n'est pas déjà dans l'arbre
16    if(!_explored.contains(matrix->getHash()))
17    {
18      // On marque la matrice comme étant dans l'arbre
19      _explored.insert(matrix->getHash(), matrix);
20      // Pour chaque sous-matrice 2x2
21      for(qint8 i=0; i<3; ++i)
22      {
23        for(qint8 j=0; j<3; ++j)
24        {
25          // On définit la rotation dans le sens horaire
26          Cell topLeftCell = j + i * matrix->getDimension();
27          Rotation rotation(topLeftCell, CW);
28          // On crée le noeud fils dans child
29          child = new Matrix(matrix, rotation);
30          // On fait pointer child vers son parent
31          matrix->addChild(rotation, child);
32          // On rajoute child dans la file
33          queue.enqueue(child);
34
35          // Idem avec la rotation dans le sens anti-horaire
36          rotation = Rotation(topLeftCell, CCW);
37          child = new Matrix(matrix, rotation);
38          matrix->addChild(rotation, child);
39          queue.enqueue(child);
40        }
41      }
42    }
43  }
```

Le code précédent est une version simplifiée du code réel ne tenant compte ni des opérations nécessaires à la gestion des threads ni de celles liées aux calculs des statistiques du graphe.

2. Résolution du problème

Une fois l'arbre généré, il suffit de partir du nœud correspondant à la matrice à résoudre et de remonter la liste des parents jusqu'à la racine pour trouver la plus courte séquence de rotations permettant le réordonnancement. En effet, avec le parcours en profondeur on s'assure que tous les nœuds de niveau `N` soient visités avant de visiter les nœuds de niveau `N+1`.

3. Statistiques du graphe

Pendant la génération de l'arbre, notre programme effectue aussi quelques opérations afin de déterminer la hauteur `H` de l'arbre et le nombre `N` de nœuds (nombre de matrices ordonnables). Cela nous permet de répondre aux deux questions posées :

- Le nombre de coups maximal pour réordonner une matrice vaut `H-1`.
- Le nombre de matrices non ordonnables vaut N_{TOTAL}`-N`. (N_{TOTAL} `= 9!` : nombre de matrices 3×3 existantes).

4. Génération aléatoire

Afin de faciliter les tests, nous avons ajouté la possibilité de générer une matrice aléatoirement, ce qui rend moins fastidieux la saisie d'une matrice valide.

III. Choix

1. C++ / Qt

Nous avons choisi d'utiliser le C++ avec le framework Qt :

- pour avoir la possibilité de gérer la mémoire de manière plus précise qu'en Java.
- pour profiter des meilleures performances de calcul.
- pour la portabilité (Qt est multi-plateforme).
- pour la simplicité de création d'une interface graphique.

2. Clé de hachage

Chaque matrice peut être identifiée de manière unique par une clé de hachage. Cette clé correspond à la concaténation des 8 premiers chiffres de la matrice auxquels on soustrait 1 et est calculée à la création de la matrice et stockée. En terme de complexité, la comparaison de deux matrices est bien moins coûteuse : il suffit de comparer un entier plutôt que huit.

Cette clé permet également de retrouver rapidement la matrice à résoudre depuis la table de hachage `_explored` (`QMap`) contenant un pointeur vers chaque matrice visitée. En calculant la clé de la matrice on a accès à son nœud et on peut ainsi la résoudre.

3. Matrice 4×4

La clé de hachage est codée sur 8 octets (`quint64`). La valeur maximale prise par la clé est obtenue pour la matrice ordonnée ($76\,861\,433\,640\,456\,465 < 2^{64}$). On peut donc théoriquement utiliser le programme pour générer l'arbre de résolution d'une matrice 4×4.

IV. Problèmes rencontrés

1. Matrice 4×4

En pratique on réalise rapidement que les $20\,922\,789\,888\,000$ matrices à prendre en considérations saturent la mémoire avant d'avoir atteint le premier pourcent.