Projet de programmation

Riyad Kettaf,Antony Albergne

Mars 2024

Table des matières

1 Is sorted

(Le parcours de G est effectué ligne par ligne).

2 Méthode naïve

2.1 pseudocode

Voici le pseudo-code de la méthode naïve choisie :
Le "placement" est explicité de manière concrète dans le code.

Données : grille G de taille $m \times n$
Résultat : Vrai si G est triée, faux sinon
compte←1;
pour $x \in G$ **faire**
  si $x \neq$ *compte* **alors**
    renvoyer Faux;
  fin
fin
Renvoyer Vrai;

Données : grille G de taille $m \times n$
pour *chaque élément x de [1,nm]* **faire**
  placer x sur la bonne colonne;
  placer x sur la bonne lignet;
fin

2.2 Complexité

Cet algorithme est en $O(nm(n+m))$ pire cas car pour chaque nombre de $[1, nm]$, la recherche coûte au pire des cas nm si l'on parcout la grille entièrement, et une fois trouvé, on fait au pire $n+m$ échanges car cela correspond au nombre d'échange pour aller de la case d'indice $(0,0)$ a la case d'indice $(m-1, n-1)$.

2.3 Preuve de correction

Démontrons que cet algorithme est correct de manière inductive : c'est-a-dire, supposons que l'algorithme est correct jusqu'a k valeurs et montrons que l'on puisse bien placer la $k+1$-ème valeur de sorte a ce que la grille soit rangée correctement jusqu'a $k+1$. (Il n'y a pas de cas de base (0 valeurs bien placées initialement)).

Soit G une grille de taille m, n ayant les k premières valeurs bien rangées.Soit (i_0, j_0) les coordonées de la valeur k.Soit (i, j) les coordonées de la valeur $k+1$. Comme G est déja bien rangée jusqu'a k, nécessairement, $i >= i_0$. On a maintenant 2 cas de figure :

— Soit $i = i_0$, et dans ce cas, on a nécessairement $j >= j_0$ car sinon on aurait $k+1$ derrière k, ce qui contredit l'hypothèse de départ.Et dans ce cas, l'algorithme place $k+1$ sur la colonne $j_0 + 1$ par la droite donc ne change pas l'ordre des k premières valeurs.

— Soit $i > i_0$, et dans ce cas, l'algorithme place d'abord $k+1$ sur la bonne colonne et $k+1$ est maintenant de coordonées $i, j_0 + 1$ (sans problème car la colonne i n'est pas encore concernée par l'hypothèse d'induction). Ensuite l'algorithme remonte $k+1$ jusqu'a la ligne i_0 et donc k est finalement bien placée.

D'ou $k+1$ bien placée ce qui finalise la preuve.

2.4 Optimalité

Cet algorithme est également optimal en le nombre de swaps : En effet, pour chaque nombre, il faut au moins le placer sur la bonne ligne et la bonne colonne, et le faire de manière "diagonale" n'est pas mieux que le placer d'abord par rapport a la colonne puis par rapport a la ligne.

3 BFS

3.1 Création du graphe des grilles

Voici la façon de le graphe des grilles a été généré.

Données : grille G de taille $m \times n$
Résultat : graphe Gr
gr←$(chaine(G),\emptyset)=(V,E)$;
$O \leftarrow [chaine(G)]$ (File de priorité) ;
$F \leftarrow [chaine(G)]$;
tant que $O \neq []$ **faire**
  s←O.defile();
  g← $chaine^{-}1$(s);
  pour $(i,j) \in [0,m-1] \times [0,n-1]$ **faire**
    g1,g2,g3,g4←grilles issus de swaps entre (i,j) et ses voisins ;
    pour $v \in chaine(g1,g2,g3,g4), v \notin F$ **faire**
      ajouter v a gr.V;
      ajouter (s,v) a gr.E;
      O.enfile(v);
      F.enfile(v);
    fin
  fin
fin
renvoyer gr;

L'idée de cet algorithme est donc d'effectuer un parcours en largeur du graphe (d'ou la file de priorité) et d'ajouter les noeuds que l'on découvre dans le graphe et les arrêtes par lesquelles on l'a trouvé.On garde une liste F des sommets déja parcourus car O ne peut pas garder en mémoire cette information.
Il est a noter que cet algorithme est suffisant car le graphe des grilles est nécessairement connexe (la méthode naïve le montre)
Ici, la fonction *chaine* correspond a la transformation d'une grille en chaine de caractères afin de s'assurer d'avoir des noeuds de type hashable. $chaine^{-}1$ correspond donc a la fonction qui renvoie la grille associée a la chaine de caractère en entrée. Le fonctionnement de *chaine* consiste en la création d'une chaîne vide, a laquelle on concatene les valeurs de la grille en lisant ligne par ligne. Pour $chaine^{-}1$, on prend en paramètre les dimensions de la grille (et une

chaîne de caractère), et on lit les m premiers nombres dans la chaîne pour avoir la première ligne, puis on en lit m autres pour la deuxième ligne...

L'algorithme de création de graphe est de complexité élevée : En effet, en complexité temporelle, si l'on suppose que l'opération d'enfilement est en temps constant (possible avec certaines structures de données), alors on a quelque chose de l'ordre de $O(|V| + |E|)$ avec V l'ensemble des noeuds et E l'ensemble des arrêtes. On a toutes les permutations de $[1, nm]$ comme noeuds donc $(nm)!$ noeuds, et pour chaque noeud (pour des grosses valeurs de n, m les effets de bord sont négligés), on peut effectuer $n(m-1)$ swaps horizontaux distincts ou $m(n-1)$ swaps verticaux.Ainsi, on a une complexité de $O((nm)!(1 + \frac{(n(m-1)+m(n-1))}{2})$.

Enfin, dans l'implémentation pratique de la création de graphe, il est a noter que pour effectuer un swap tout en préservant une copie de la grille d'origine, on doit faire appel a la méthode *deepcopy* de la librairie *copy*

3.2 Application de BFS aux grilles

En utilisant cet outil de création de graphe combiné a la méthode BFS, on arrive a trouver le plus court chemin entre une grille donnée et la grille de même dimensions, ordonnée.Cependant, on peut dire que les performances sont très mauvaises. En effet, dès qu'on s'attaque au grilles dont le produit des dimensions est supérieur a 8, l'algorithme met beaucoup de temps a terminer.

3.3 Amélioration de BFS

Lorsque l'on calcule le graphe des grilles, on ajoute beaucoup de noeuds et d'arrêtes qu'on n'explorera jamais ainsi, on peut considérer une amélioration de BFS qui consiste comme avant a construire le graphe des grilles jusqu'a ce que l'on tombe sur la grille triée. Ensuite comme sur l'algorithme générique BFS, on "backtrack" vers la grille de départ.

Même si théoriquement cette amélioration réduit en effet la taille du graphe, comme on effectue toujours un parcours en largeur "sans se poser de question", l'éxecution de l'algorithme prend quand même beacoup de temps pour des petites dimensions.

4 A*

4.1 heuristiques

Afin d'implémenter l'algorithme A*, on a besoin de choisir une heuristique, pour cela on a plusieurs choix : Soit $(m, n) \in \mathbb{N}^2, (G_{i,j})_{(i,j)\in[0,m-1]\times[0,n-1]} \in M_{m,n}(\mathbb{R})$

— Manhattan :Soit $f_1 : G \to \sum_{(i,j)\in[0,m-1]\times[0,n-1]} |x_i - (x^*)_i| + |x_j - (x^*)_j|$ avec (x_i, x_j) les coordonées du nombre x dans G et $((x^*)_i, (x^*)_j)$ ses coordonées dans la matrice ordonnée de mêmes dimensions que G.
— Euclidienne : Soit $f_2 : G \to \sum_{(i,j)\in[0,m-1]\times[0,n-1]} \sqrt{((x_i - (x^*)_i)^2 + (x_j - (x^*)_j)^2)}$
— Maximum : Soit $f_3 : G \to max(|x_i - (x^*)_i| + |x_j - (x^*)_j|, (i,j) \in [0, m-1] \times [0, n-1])$

On aurait pu également définir une heuristique en calculant la longueur de la solution sur des sous-grilles de notre grille (par exemple, pour une matrice 4x4, on peut divisier notre grille en 4 grilles 2x2 dont la résolution est relativement facile).

On peut déja écarter une heursitique, celle du maximum car elle est aussi facile a calculer que celle de Manhattan mais moins précise, en effet, si on échange 2 cellules dont une réalise le maximum, alors peut être que la distance maximale va réduire mais au coût d'avoir éloigné l'autre cellule de sa position voulue.

De plus, on peut dire que la distance de Manhattan est plus précise que celle d'Euclide : En effet, si pour une cellule donnée de coordonées (i, j), la position voulue est sur la même ligne mais 2 cases au dessus, la distance de manhattan et d'euclide valent 2, cependant, si la position voulue est en diagonale (écart de 1 en colonne et en ligne), alors la distance d'euclide l'évaluera a $\sqrt{2}$, et pourtant, que la position soit en diagonale ou dans le même axe ne change pas la quantité de swaps a faire.

De plus, f_1 est "presque" admissible : en effet, pour chaque nombre, (de distance de manhattan d a sa position voulue), il faut au moins d swaps pour le placer, cependant, pour des grilles de petites tailles, comme on a des phénomènes de "1 pierre, 2 coups", c'est-a-dire, qu'on arrive a placer 2 nombres en un swaps, alors que la distance de manhattan estimait qu'il fallait au moins 2 swaps pour les placer, d'ou le "presque".

Ainsi, l'heuristique utilisée pour A* est de la forme $h = d + f_1$ avec d la distance déja calculée pour la grille.

4.2 Implémentation de A*

A* ressemble beaucoup au dernier algorithme utilisé dans BFS. Il y 2 différences majeures : au lieu d'utiliser une file de priorité on utlise un tas min qui nous permet d'avoir la valeur mimimale en heuristique des grilles a traiter,les opérations d'insertion et d'extraction coûte au pire cas $O(log((nm)!)) = o(mnlog(mn))$, mais on un coût moyen bien meilleur comparée a une liste classique ou l'on devrait trier a chaque fois, ce qui couterait également $O(mnlog(mn))$ dans le pire des cas mais aurait un coût moyen plus élevé. Et on modifie la forme des noeuds, ils sont maintenant de la forme $(h(grille), chaine(grille))$ qui est aussi hashable en tant que couple de types hashables.

4.3 Performances de A*

En complexité pire cas, A* est pire que les 2 versions de BFS, en effet, on a les opérations dans le tas et les calculs d'heuristiques qui alourdissent la complexité, cependant, en pratique A* performe bien mieux que les algorithmes BFS, en effet, l'algorithme termine assez rapidement sur des grilles dont le produit des dimensions dépasse 16.

5 Compléments

On ne le détaillera pas dans cette section mais on a également rajouté les implémentations par rapport a celles demandées :

- Ajout de barrières (argument $forb$ dans swap)
- Visualisation de l'algorithme A* (A* vizu)
- Version interactive du jeu
- outil (primitif) permettant de mesurer la performance des différents algorithmes.

5.1 Difficulté contrôlée

Voici un algorithme permettant d'obtenir un niveau de difficulté donné

Données : [a,b] :intervalle de coûts souhaité,m,n
Résultat : Grille dont la résolution de longueur $\in$ [a,b]
G$\leftarrow$ Grille generee aleatoirement;
tant que *longueur solution de G* $\notin$ *[a,b]* **faire**
  repeter;
fin
retourner G

La terminaison de l'algorithme repose sur le fait de choisir des paramètres cohérents (en effet, trouver une solution de longueur 50 pour une grille de taille 2x2 risque d'être compliqué...).

5.2 Cas particuliers

Les algorithmes suivants n'ont pas été implémentés.

Considérons une grille de taille $1 \times n, n \in \mathbb{N}^*$ Une façon intelligente de trouver une solution efficace au problème de la grille appliqué a ce cas particulier serait de faire un tri a bulles. En effet, c'est un mauvais algorithme de tri, mais c'est le seul qui procède par échanges de nombres adjacents. De plus, c'est un algorithme en $O(n^2)$, ce qui surpasse les algorithmes précédemments vus (sauf la méthode naïve).

Considérons maintenant une grille de taille $k \times n, k \in \mathbb{N}^*$ fixé et $n \to \infty$ Considérons

maintenant l'algorithme suivant :

Données : Grille G de taille $k \times n$
Trier les n colonnes ;
Interclasser les colonnes avec leurs voisins ;
Trier les k lignes ;

Trier : Tris a bulles.

Le principe de cet algorithme est de d'abord trier les n colonnes, puis, on regarde les nombres sur chaque colonne, on regarde pour chaque ligne, on y trouve un nombre qui est supposé être sur cette ligne. Si on trouve des lignes pour lesquelles le nombre n'est pas supposé être sur cette ligne, alors on regarde chez les colonnes voisines si on peut trouver un nombre qui corresponderait a cette ligne et on remplace (opération d'interclassement). Enfin, on trie les k lignes. La complexité de cet algorithme est $O(nk^2+nk+kn^2) = O(n^2)$. Cette algorithme a une très bonne complexité, mais c'est un algorithme qui n'est pas exact. En effet, l'opération d'interclassement ne garantit pas d'avoir une association parfaite ligne-nombre pour chaque colonne. On peut avoir quelques nombres qui ne sont pas au bon endroit a la fin de l'algorithme, des "résidus".Cependant, si l'on considère $f(n)$ le nombre moyen de "résidus" pour un n donné, alors on pourrait établir que $f = o(n)$), et donc, on aurait une matrice très largement triée avec quelques résidus.