TP #24 — Un super algorithme de graphe pour toutes paires de sommets

Contexte

Le but de ce sujet est d'étudier une famille d'algorithmes de graphes qui présentent une grande similarité avec le produit de matrices. Les versions les plus efficaces de ces algorithmes suivent une approche par *programmation dynamique*, que nous détaillerons prochainement en cours.
Dans tout le sujet, on considérera des graphes **orientés** de sommets $[\![n]\!]$ représentés par matrices d'adjacences ; en fonction du cas d'application, les entrées d'une matrice A appartiendront à des ensembles différents et modéliseront des quantités différentes :

— Premier cas : l'ensemble est $\{0, 1\}$ (ou $\{\top, \bot\}$), et $A_{i,j}$ vaut 1 ss.'il existe un arc (orienté, non pondéré) entre le sommet (représenté par l'entier) i et le sommet (représenté par l'entier) j.
— Second cas : l'ensemble est $\mathbb{N} \cup \{\infty\}$, et $A_{i,j}$ vaut la longueur entière positive d'un arc (orienté, pondéré) entre les sommets i et j, ou ∞ s'il n'existe pas d'arc entre i et j. On prendra la convention que pour tout i, $A_{i,i} = 0$.
— Troisième cas : l'ensemble est $\mathbb{N} \cup \{\infty\}$, et $A_{i,j}$ vaut la *capacité* pouvant transiter par un arc (orienté, pondéré) entre les sommets i et j, ou 0 s'il n'existe pas d'arc entre i et j. On prendra la convention que pour tout i, $A_{i,i} = \infty$.

Les matrices d'adjacence seront représentées en C par des objets de type `double**` : la matrice d'adjacence A d'un graphe à n sommets sera représentée par une variable `a` d'un tel type telle que pour $0 \leq i, j < n$ l'entrée `a[i][j]` contient la valeur de $A_{i,j}$. L'utilisation d'un type `double` pour les entrées permettra de représenter la valeur ∞ au sein du type (par une valeur « `inf` »), mais hormis ce cas les entrées de `a` **seront bien des entiers**. On rappelle que le format double-précision IEEE754 permet (entre autres) de représenter exactement tous les entiers $[\![0, 2^{53} - 1]\!]$, ce qui sera largement suffisant pour nous.
L'écriture de littéral « `inf` » pourra se faire en utilisant la *macro* `HUGE_VAL` définie dans le fichier d'en-tête `math.h`.

Consignes relatives à l'allocation mémoire. Pour alléger la programmation, vous pourrez supposer que tous les appels à `malloc` réussissent (c'est à dire qu'ils ne renvoient jamais de valeur de pointeur nulle). En revanche les fuites mémoire doivent être considérées comme des erreurs de programmation et être corrigées

si détectées ; l'utilisation d'un mécanisme de détection des fuites (*valgrind* ou *sanitizer*) est **obligatoire**.

Fonctions préliminaires

1. Écrivez une fonction C de signature :
   ```
   double **make_a(size_t n)
   ```
   qui alloue « sur le tas » et renvoie (l'adresse d') un « tableau à deux dimensions $n \times n$ » dont les entrées sont initialisées à `0.`.
2. Écrivez une fonction C de signature :
   ```
   void free_a(size_t n, double **a)
   ```
   telle que pour un argument `a` renvoyé par un appel `make_a(n)` libère **toute** la mémoire occupée par celui-ci.
3. Écrivez une fonction C de signature :
   ```
   double **copy_a(size_t n, double **a)
   ```
   qui renvoie une copie « profonde » (sans aucun *aliasing*) du « tableau à deux dimensions $n \times n$ » `a`.

Fermeture réflexive-transitive

Dans toute cette partie, on considère uniquement des graphes orientés non pondérés représentés par des matrices d'adjacence à coefficients dans $\{0, 1\}$.
On définit la *fermeture* (ou *clôture*) *réflexive-transitive* d'un graphe orienté G est le graphe non pondéré G' tel qu'il existe un arc reliant les sommets i et j de G' si et seulement si j est accessible depuis i dans G, c'est à dire ssi. $i = j$ ou il existe un chemin (orienté) reliant i à j dans G.
N.B. Par abus de langage, on appellera ci-après « fermeture transitive » cette fermeture réflexive-transitive. La *vraie* fermeture transitive est définie de façon similaire en omettant la condition « $i = j$ » ci-dessus.

On souhaite trouver un algorithme efficace pour calculer la fermeture transitive d'un graphe représenté par matrice d'adjacence. On commence par définir pour cela un produit $\odot$ entre matrices d'adjacence de graphes de même nombre de sommets comme :

$$(A \odot B)_{i,j} := \bigvee_{k=0}^{n-1} A_{i,k} \wedge B_{k,j}$$

où $0 \vee 1 = 1 \vee 0 = 1 \vee 1 = 1$, $0 \vee 0 = 0$; $0 \wedge 0 = 0 \wedge 1 = 1 \wedge 0 = 0$, $1 \wedge 1 = 1$ (formellement, ceci correspond au produit de matrices sur le *semi-anneau* $(\{0, 1\}, \vee, \wedge)$.)

4. Soit A la matrice d'adjacence d'un graphe G, on a par définition que les entrées de A indiquent la présence ou non d'un chemin de longueur un (arc) entre

chaque paire de sommet. De façon similaire, quelle information est donnée par les entrées de la matrice $A \odot A^*$?

5. Montrez que s'il existe un chemin $v \rightsquigarrow w$ dans un graphe à n sommets, alors il en existe un de longueur au plus $n-1$ arcs.
6. Déduisez des deux questions précédentes un algorithme permettant de calculer la fermeture transitive d'un graphe de n sommets en *au plus* $O(n^4)$ opérations.

Pour simplifier l'implémentation de cet algorithme, on introduit une représentation légèrement enrichie de G par une matrice d'adjacence A^+ où pour tout i on a ${A^+}_{i,i} = 1$: on introduit (éventuellement) un arc virtuel de tout sommet à lui-même afin de traduire le fait qu'un sommet est toujours accessible depuis lui-même.

7. Montrez que la puissance $n-1^{\text{ème}}$ $(A^+)^{n-1}$ de A^+ pour le produit $\odot$ coïncide avec la matrice d'adjacence de la fermeture transitive de G.
8. Déduisez-en une variante de l'algorithme précédent permettant de calculer la fermeture transitive d'un graphe de n sommets en au plus $O(n^3 \log n)$ opérations.
9. Écrivez une fonction C de signature

   ```
   double **mul_and_or(size_t n, double **a, double **b)
   ```

   qui **ne modifie pas** les entrées de ses arguments `a` et `b` et renvoie un *nouveau* « tableau » représentant le produit $\odot$ de deux matrices de même dimension n.
10. Écrivez une fonction C de signature :

    ```
    double **tc_pown(size_t n, double **a)
    ```

    qui **ne modifie pas** les entrées de son argument `a` représentant la matrice d'adjacence d'un graphe à `n` sommets et qui utilise l'algorithme trouvé aux questions précédentes pour calculer et renvoyer la matrice d'adjacence de la fermeture transitive de celui-ci.
    Indication. Cette fonction s'écrit aisément de façon itérative, notamment en exploitant l'existence d'un point fixe.
11. Testez votre fonction `tc_pown` sur des exemples bien choisis.

On va maintenant encore diminuer le coût du calcul de la fermeture transitive en adoptant une approche de type *programmation dynamique*. On introduit pour cela la suite de graphes $G^{(k)}$ définie par $G^{(-1)} = G$, et $G^{(k \geq 0)}$ le graphe qui contient un arc entre les sommets i et j si et seulement si : $i = j$; ou il existe un arc entre i et j dans G ; ou il existe un chemin entre i et j dans G *qui (hormis i et j)* **ne passe que par les sommets d'indices $\leq k$**.

12. Donnez une expression de la fermeture transitive de G en fonction des graphes $G^{(k)}$.

∗. On rappelle qu'il n'existe pas nécessairement d'arc entre un sommet et lui-même dans G, et donc que $A_{i,i}$ ne vaut pas nécessairement *1*.

13. En utilisant (par abus de notation) $G^{(k)}_{i,j}$ pour désigner le coefficient *(i,j)* de la matrice d'adjacence de $G^{(k)}$, montrez que pour $k \geq 0$ l'on a :

$$G^{(k)}_{i,j} = G^{(k-1)}_{i,j} \vee (G^{(k-1)}_{i,k} \wedge G^{(k-1)}_{k,j})$$

14. Déduisez de ce qui précède un algorithme permettant de calculer la fermeture transitive d'un graphe de n sommets en au plus $O(n^3)$ opérations. Quel est le coût mémoire de votre algorithme ?

N.B. Cet algorithme se modifie aisément (comment ?) si l'on souhaite calculer la « vraie » fermeture transitive (et non la fermeture réflexive-transitive) du graphe.

15. Montrez que s'il modifie son entrée pour la remplacer par la fermeture transitive calculée, l'algorithme précédent peut être implémenté *en place*, c'est à dire sans **aucun** stockage supplémentaire (à part un nombre constant de variables temporaires, par exemple pour servir d'indices de boucles).

16. Écrivez une fonction C de signature :

```
double **tcfw(size_t n, double **a)
```

qui **ne modifie pas** les entrées de son argument `a` représentant la matrice d'adjacence d'un graphe à `n` sommets et qui utilise l'algorithme trouvé aux questions précédentes pour calculer et renvoyer la matrice d'adjacence de la fermeture transitive de celui-ci.

L'algorithme implémenté à la question précédente ne fait que calculer la fermeture transitive, et ne construit pas de chemin explicite reliant toute paire de sommets (quand cela est possible). Cependant, comme habituellement en programmation dynamique, il est possible d'enrichir l'algorithme pour qu'il calcule des donnée supplémentaires qui pourront ensuite être utilisées dans un algorithme de reconstruction fournissant cette information. Ici il suffit (comme souvent) de retenir la valeur de l'indice ayant permis de constater l'existence d'un chemin (s'il y en a un).

17. Écrivez une variante `tcfwp` de la fonction précédente ainsi qu'une fonction `path` telles que `tcfwp` « renvoie » (en plus de la matrice d'adjacence de la fermeture transitive) une seconde matrice `p` telle qu'un appel à `path` « renvoie » sous forme de tableau les sommets d'un chemin entre `i` et `j` si un tel chemin existe (on pourra par exemple renvoyer un tableau d'`int` de longueur toujours égale au nombre de sommets, dont les entrées non utilisées sont à `-1` ; un tableau constant à `-1` indique alors l'inexistence d'un chemin).
N.B. Vous pouvez choisir le format qui vous arrange le plus pour représenter exactement les chemins ; une implémentation récursive pour `path` peut être une bonne approche.

18. Les chemins renvoyés par votre fonction ci-dessus sont-ils toujours les plus court possibles (en nombre d'arcs) ?

Plus-court-chemins entre toutes paires de sommets

On s'intéresse maintenant au problème suivant : soit un graphe orienté pondéré G représenté par une matrice d'adjacence à coefficients dans $\mathbb{N} \cup \{\infty\}$, on souhaite calculer la matrice des distances D^G telle que $D^G_{i,j}$ contient 0 si $i = j$, et sinon la distance d'un plus-court-chemin entre les sommets i et j si un tel chemin existe, et sinon ∞.

19. Donnez la matrice des distances du graphe dont la matrice d'adjacence (en pseudo-code) est :

```
[ [0,   1, 3, 7  ],
  [inf, 0, 1, inf],
  [inf, inf, 0, 4],
  [2, inf, inf, 0] ]
```

20. Esquissez un algorithme pour résoudre ce problème en utilisant l'algorithme « de Dijkstra ». Quel est le coût de l'algorithme résultant ?

21. Montrez que la matrice D^G peut être calculée par le *même* algorithme de programmation dynamique qu'à la question 14, à condition de remplacer l'opération $\vee$ (resp. $\wedge$) par min (resp. $+$). Comparez son coût avec l'algorithme obtenu à la question précédente quand le nombre d'arcs est un $\Omega(S^2)$ « au moins S^2, à facteurs et termes négligeables près » ; un $O(S)$ « au plus S, à facteurs et termes négligeables près » .

22. Écrivez une fonction C de signature :

```
double **apspfw(size_t n, double **a)
```

qui implémente l'algorithme de la question précédente.

23. Testez votre fonction, notamment sur le graphe de la question 19.

Chemin de capacité maximale entre toutes paires de sommets

On s'intéresse maintenant au problème suivant : soit un graphe orienté pondéré G représenté par une matrice d'adjacence à coefficients dans $\mathbb{N} \cup \{\infty\}$, on souhaite calculer la matrice des capacités max C^G telle que $C^G_{i,j}$ contient ∞ si $i = j$, et sinon la *capacité maximale* d'un chemin (le *widest path*) entre les sommets i et j si un tel chemin existe, et sinon 0.
La capacité d'un chemin est elle-même définie comme le minimum du poids des arcs constituant le chemin.

24. Donnez la matrice des capacités max du graphe dont la matrice d'adjacence (en pseudo-code) est :

```
[ [inf, 1, 3, 7],
  [0, inf, 1, 0],
  [0, 0, inf, 4],
  [2, 0, 0, inf] ]
```

25. Montrez que la matrice C^G peut être calculée par le *même* algorithme de programmation dynamique qu'à la question 14, à condition de remplacer l'opération $\vee$ (resp. $\wedge$) par max (resp. min).

26. Écrivez une fonction C de signature :

```
double **apwpfw(size_t n, double **a)
```

qui implémente l'algorithme de la question précédente.

27. Testez votre fonction, notamment sur le graphe de la question 24.

Commentaires

L'algorithme de programmation dynamique étudié dans ce sujet est généralement connu sous le nom d'*algorithme de Floyd-Warshall*. Il est au programme et à connaître.

Bien que celui-ci ressemble *très fortement* à l'algorithme naïf de calcul d'un produit de matrice, le changement d'ordre des indices fait qu'il n'en est pas un.

Cependant, dans les trois cas d'application étudiés, il est possible de *réduire* le problème au calcul d'*un* (et non pas $\log n$) « vrai » produit de matrices de dimensions $O(n)$ sur le *semi-anneau* sous-jacent ($(\{0,1\}, \vee, \wedge)$ dans le premier cas, $(\mathbb{N}, \min, +)$ dans le second, $(\mathbb{N}, \max, \min)$ dans le dernier). Dans le premier cas on sait résoudre ce problème en un coût « réellement sous-cubique », c'est à dire en $O(n^{3-\varepsilon})$ avec $\varepsilon > 0$: par exemple, la famille des algorithmes « de Strassen-Winograd » (initialement développée pour le produit de matrice sur un corps) a une complexité en $O(n^{\log_2(7)}) \approx O(n^{2.81})$ et est concrètement plus efficace que l'algorithme cubique naïf pour des dimensions dès l'ordre de la centaine. Dans le second, bien que l'on connaisse des algorithmes de coût inférieur à $O(n^3)$, on n'en connaît actuellement aucun qui soit réellement sous-cubique. Dans le dernier, on connaît des algorithmes sous-cubiques légèrement moins efficaces que dans le premier cas : on peut déduire un algorithme de coût $O(n^{\frac{3+\omega}{2}})$ de tout algorithme de coût $O(n^\omega)$ pour le produit de matrices de dimension n sur un corps.

De façon générale, la conception d'algorithmes de multiplication de matrices et leur application à des problèmes de théorie des graphes est un domaine de recherche très actif, cf. par exemple :

— New Graph Decompositions and Combinatorial Boolean Matrix Multiplication Algorithms (STOC 2024),
— Subcubic Equivalences Between Path, Matrix, and Triangle Problems (JACM 2018)

pour des résultats récents à ce sujet.