TP #2 — Un super algorithme de graphe pour toutes paires de sommets

Contexte

Le but de cet exercice 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 l'exercice, on considérera des graphes orientés 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, $A_{i,i} = \infty$.

Rappels : copies profondes & compréhension

Les copies de `list` sont par défaut *superficielles*, ce qui peut entraîner des comportements non désirés :

```
>>> a = [0, 1, 2]
>>> a
[0, 1, 2]
>>> b = a
>>> b[0] = 1
>>> a
[1, 1, 2]
>>> m = [[0]*2]*2
>>> m
[[0, 0], [0, 0]]
>>> m[0][0] = 1
```

```
>>> m
[[1, 0], [1, 0]]
```

Une façon pratique de réaliser des copies *profondes* (ou des créations de `list` de `list`) est d'utiliser la *compréhension* de liste :

```
>>> a = [0, 1, 2]
>>> b = [a[i] for i in range(len(a))]
>>> b[0] = 1
>>> a
[0, 1, 2]
>>> m = [[0 for _ in range(2)] for _ in range(2)]
>>> m
[[0, 0], [0, 0]]
>>> m[0][0] = 1
>>> m
[[1, 0], [0, 0]]
>>> m2 = [[m[i][j] for j in range(2)] for i in range(2)]
>>> m2
[[1, 0], [0, 0]]
>>> m2[0][0] = 0
>>> m2
[[0, 0], [0, 0]]
>>> m
[[1, 0], [0, 0]]
```

Fonction préliminaire

1. Écrivez une fonction Python `genChain(n:int)` qui renvoie la matrice d'adjacence à valeurs dans `0` et `1` d'un graphe orienté de n sommets dont les seuls arcs sont entre les sommets i et $i+1$ pour tout $0 \leq i < n-1$.
   On représentera cette matrice d'adjacence par une `list` de n `list` de n `int`, où `G[i][j]` vaut le coefficient $G_{i,j}$ de la matrice d'adjacence G. Par exemple, pour $n = 3$, `genChain` doit renvoyer la valeur :
   ```
   [[0,1,0], [0,0,1], [0,0,0]]
   ```

Fermeture 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*) *transitive* d'un graphe orienté $\mathcal{G}$ comme le graphe non pondéré $\mathcal{G}'$ tel qu'il existe un arc reliant les sommets i et j de $\mathcal{G}'$ si et seulement s'il existe un chemin reliant ces sommets dans $\mathcal{G}$.

Pour le calcul de $\mathcal{G}'$, on considérera qu'un sommet est toujours accessible depuis lui-même. Autrement dit, pour tout i il doit y avoir un arc entre i et lui-même dans $\mathcal{G}'$. Cependant, on fera attention au fait que ceci n'est pas nécessairement vrai dans le graphe initial $\mathcal{G}$.

2. Dans le cas d'un graphe non orienté, quelle notion proche pourrait on utiliser pour représenter de façon « compressée » le graphe de la fermeture transitive (en espace $O(S)$, avec S le nombre de sommets du graphe) ? Comment pourrait-on calculer cette information efficacement ?

3. Soit $\mathcal{G}_4$ le graphe généré par `genChain(4)`, donnez la matrice d'adjacence de sa fermeture transitive.

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 (indexés par $0, \dots n-1$) 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 $\mathcal{G}_A$ la matrice d'adjacence d'un graphe $\mathcal{G}$, on a par définition que les entrées de $\mathcal{G}_A$ 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 $\mathcal{G}_A \odot \mathcal{G}_A$ [*] ?

5. Montrez que s'il existe un chemin entre deux sommets d'un graphe comportant 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 $\mathcal{G}$ par une matrice d'adjacence $\mathcal{G}_A^+$ où pour tout i on a $(\mathcal{G}_A^+)_{i,i} = 1$: tout sommet i est désormais définit comme étant accessible depuis lui-même (par un arc éventuellement virtuel).

7. Montrez que la puissance $n-1^{\text{ème}}$ $(\mathcal{G}_A^+)^{n-1}$ de $\mathcal{G}_A^+$ pour le produit $\odot$ coïncide avec la matrice d'adjacence de la fermeture transitive de $\mathcal{G}$.

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

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 Python :

   `mulANDOR(A:list[list[int]], B:list[list[int]])`

   qui *ne modifie pas ses entrées* et calcule et renvoie le produit $\odot$ de deux matrices de même dimension n. Vous pouvez exploiter le fait qu'en Python `or` (resp. `and`) implémente l'opération $\vee$ (resp. $\wedge$).
10. Écrivez une fonction Python `TCpowN(A:list[list[int]])` qui *ne modifie pas son entrée* représentant la matrice d'adjacence d'un graphe et qui utilise l'algorithme trouvé aux questions précédentes pour calculer et renvoyer la matrice d'adjacence de la fermeture transitive de celui-ci.
11. Testez votre fonction `TCpowN`, notamment sur des exemples générés par `genChain`.

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 $\mathcal{G}^{(k)}$ définie par $\mathcal{G}^{(-1)} = \mathcal{G}$, $\mathcal{G}^{(k\geq 0)}$ 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 $\mathcal{G}$; ou il existe un chemin entre i et j dans $\mathcal{G}$ *qui (hormis i et j) ne passe que par les sommets d'indices* $\leq k$.

12. Donnez une expression de la fermeture transitive de $\mathcal{G}$ en fonction des graphes $\mathcal{G}^{(k)}$.
13. En utilisant (par abus de notation) $\mathcal{G}^{(k)}_{i,j}$ pour désigner le coefficient *(i,j)* de la matrice d'adjacence de $\mathcal{G}^{(k)}$, montrez que pour $k \geq 0$, $\mathcal{G}^{(k)}_{i,j} = \mathcal{G}^{(k-1)}_{i,j} \vee (\mathcal{G}^{(k-1)}_{i,k} \wedge \mathcal{G}^{(k-1)}_{k,j})$.
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 ?
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 Python `TCFW(A:list[list[int]])` qui *ne modifie pas son entrée* représentant la matrice d'adjacence d'un graphe 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 liste les sommets d'un chemin entre `i` et `j` si un tel chemin existe, et la liste vide sinon.

18. Pensez-vous que les chemins renvoyés par votre fonction ci-dessus sont 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 $\mathcal{G}$ représenté par une matrice d'adjacence à coefficients dans $\mathbb{N} \cup \{\infty\}$, on souhaite calculer la matrice des distances $\mathcal{D}^{\mathcal{G}}$ telle que $\mathcal{D}^{\mathcal{G}}_{i,j}$ 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 est :

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

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

21. Montrez que la matrice $\mathcal{D}^{\mathcal{G}}$ 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 Python `APSPFW(A:list[list[int]])` 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 $\mathcal{G}$ représenté par une matrice d'adjacence à coefficients dans $\mathbb{N} \cup \{\infty\}$, on souhaite calculer la matrice des capacités max $\mathcal{C}^{\mathcal{G}}$ telle que $\mathcal{C}^{\mathcal{G}}_{i,j}$ contient ∞ si $i = j$, et sinon

la *capacité maximale* de tout chemin 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 est :

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

25. Montrez que la matrice $\mathcal{C}^{\mathcal{G}}$ 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 Python `APWPFW(A:list[list[int]])` 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*.
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 dits 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.