TD #17 — Graphes #2 (avec solutions)

Exercice 1. *Détection de triangles de poids négatifs*

Un *triangle* d'un graphe orienté $G = S, A$ est tout triplet d'arcs $(u, v), (v, w), (w, u) \in A$. Dans le cas d'un graphe dont les arcs sont pondérés (par des poids $\in \mathbb{Z} \setminus \{0\}$), le *poids* d'un triangle est la somme des poids de ses arcs.

1. Écrivez une fonction C de signature :
   `bool has_negative_triangle(size_t n, int g[n][n])`
   qui décide si le graphe pondéré de `n` sommets représenté par la matrice d'adjacence `g` contient un triangle de poids négatif. (On suppose le graphe sans boucle : `g[i][i]` vaut zéro pour tout `i`.)

   On peut simplement procéder par recherche exhaustive des sommets de tous les triangles possibles, sans même chercher à éviter de tester deux fois un même sommet. Ceci donne par exemple :

```c
bool has_negative_triangle(size_t n, int g[n][n])
{
    for (size_t i = 0; i < n; i++)
    {
        for (size_t j = 0; j < n; j++)
        {
            for (size_t k = 0; k < n; k++)
            {
                int e1 = g[i][j];
                int e2 = g[j][k];
                int e3 = g[k][i];
                bool r = (e1 != 0) && (e2 != 0) && (e3 != 0) && (e1 + e2 + e3 < 0);
                if (r)
                {
                    return true;
                }
            }
        }
    }

    return false;
}
```

2. Analysez le coût pire-cas de votre fonction en fonction de `n`.

   Le coût de la fonction est dominé par celui de l'exécution du corps de la boucle interne, qui est de coût constant et répété au plus n^3 fois. C'est donc un $O(n^3)$.

3. Montrez que l'existence d'un algorithme de détection de triangle de poids négatif dans un graphe de n sommets de coût $O(n^{2.99})$ (dans un modèle RAM transdichotomique) implique celle d'un algorithme calculant le produit de deux matrices carrées de dimension n dans le semi-anneau $(\min, +)$ en temps $O(n^{3-\delta})$ pour un certain $\delta > 0$.

   C'est une conséquence assez directe du théorème 1.1 de (Vassilevska Williams & Williams, 2018).

4. Proposez un algorithme de détection de triangles de poids négatifs etc. de coût $O(n^{2.99})$.

   Si vous en trouvez un cela fera sans-doute le meilleur résultat de TIPE du pays, parce qu'à ma connaissance c'est toujours un problème ouvert, conjecturé comme n'ayant pas de solution.

Exercice 2. *Composantes fortement connexes*

Dans cet exercice, on considère un graphe orienté $G = S, A$ quelconque représenté par tableau de listes d'adjacence. Pour $v, w \in S$, on note $v \rightsquigarrow w$ la relation binaire indiquant qu'il existe un chemin de v à w dans G,

et R la relation binaire définie par : $v\,R\,w$ ssi. $(v = w) \vee (v \rightsquigarrow w \wedge w \rightsquigarrow v)$. On rappelle que G est dit fortement connexe ssi. $\forall v, w \in S.\ v\ R\ w$.
On définit les *composantes fortement connexes* de G comme ses sous-graphes induits fortement connexes maximaux pour l'inclusion, ou alternativement comme les sous-graphes induits par les classes d'équivalences de R.

1. Montrez que R est une relation d'équivalence.

On doit montrer sa réflexivité, symétrie & transitivité.

- **Réflexivité**. Par définition.
- **Symétrie.** Par définition.
- **Transitivité.** Soit u, v, w deux-à-deux distincts t.q. $u\,R\,v$ et $v\,R\,w$, par définition l'on a $u \rightsquigarrow v$, $v \rightsquigarrow u$, $v \rightsquigarrow w$, $w \rightsquigarrow v$, et il existe donc un chemin de u à w et vice-versa obtenu en concaténant ceux de u à v et v à w & w à v et v à u.
  Les cas où certains de ces sommets sont égaux se traitent de façon analogue.

2. Montrez l'équivalence des deux caractérisations ci-dessus des composantes fortement connexes d'un graphe.

Soit S_1 maximal pour l'inclusion dont le sous-graphe induit est fortement connexe., alors pour toute paire de sommets $v, w \in S_1$ l'on a bien $v\,R\,w$. De plus pour tout $v \in S_1$ il n'y a pas de $u \in S \setminus S_1$ t.q. $u\,R\,v$ car sinon le graphe induit par $S_v \cup \{u\}$ serait fortement connexe (on aurait $u \rightsquigarrow v$, $v \rightsquigarrow u$, et de même pour tout $w \neq S_v$ par $u \rightsquigarrow v \rightsquigarrow w$ et symétrique), ce qui contredirait la maximalité de S_1 ; il s'ensuit que S_1 est une classe d'équivalence pour R.
Le sens réciproque est analogue.

3. Soit $v \in S$, esquissez un algorithme pour calculer $S_v := \{w \in S \mid v \rightsquigarrow w\}$. Celui-ci devra être de coût linéaire en la taille de la représentation de G.

S_v correspond exactement à l'ensemble des sommets visités après un parcours (de type quelconque) d'origine v, qui fournit donc l'algorithme désiré. Bien implémenté, celui-ci sera de coût $O(\#A + \#S)$.

4. Soit $v \in S$ et S_v, esquissez un algorithme calculant l'ensemble des sommets de la composante fortement connexe de v. Celui-ci devra être de coût $O(\#S_v \times (\#A + \#S))$.

On a que $w \in S_v$ ssi. $v \rightsquigarrow w$. On a donc que w est dans la même composante fortement connexe que v ssi. $w = v \vee w \rightsquigarrow v$. La première condition se teste trivialement, et la seconde en temps $O(\#A + \#S)$ par un parcours d'origine w. Ceci suggère immédiatement un algorithme atteignant le coût cible (*viz.* on effectue le test précédent pour tous les sommets de S_v et l'on rajoute v à la fin).

Pour améliorer le coût de l'algorithme impliqué par les deux questions précédentes, on introduit le *graphe transposé* $G^{\leftarrow}$ de G comme le graphe obtenu depuis G en inversant l'orientation de chaque arc $u \to v$ en $v \to u$.

5. Esquissez un algorithme de coût $O(\#S^2)$ (resp. $O(\#A + \#S)$) calculant une représentation de $G^{\leftarrow}$ depuis une représentation de G par matrice (resp. tableau de listes) d'adjacence.

La matrice d'adjacence de $G^{\leftarrow}$ est donnée par la transposée de la matrice d'adjacence de G, qui peut bien se calculer en temps c'est à dire $O(\#S^2)$.
En cas de représentation par tableau de listes d'adjacence, il suffit pour chaque arc (v, w) de G d'ajouter l'arc (w, v) dans $G^{\leftarrow}$. Ceci peut se faire en temps $O(\#A + \#S)$ en parcourant les $\#S$ listes d'adjacence de G.

6. Montrez que les composantes fortement connexes de G et $G^{\leftarrow}$ sont les mêmes.

C'est une conséquence immédiate du fait que l'appartenance à une composante fortement connexe est une relation d'équivalence symétrique pour l'orientation des chemins. Avec à peine plus de détails, si $v, w \neq v$ sont dans la même composante fortement connexe dans G, alors il existe des chemins $v \rightsquigarrow w$ et $w \rightsquigarrow v$ dans G (resp. $G^{\leftarrow}$) qui deviennent des chemins $w \rightsquigarrow v$ et $v \rightsquigarrow w$ dans $G^{\leftarrow}$ (resp. G), ce qui permet de conclure.

7. Esquissez un algorithme qui calcule l'ensemble $S_v^{\leftarrow}$ des sommets de G **depuis** lesquels v est accessible (c'est à dire des sommets w t.q. $w \rightsquigarrow v$) en temps $O(\#A + \#S)$ (pour un graphe représenté par listes d'adjacence).

Par construction de $G^{\leftarrow}$, $w \rightsquigarrow v$ (dans G) ssi. $v \rightsquigarrow w$ (dans $G^{\leftarrow}$). On peut donc calculer $S_v^{\leftarrow}$ comme l'ensemble des sommets visités par un parcours d'origine v dans $G^{\leftarrow}$, ce qui coûte un $O(\#A + \#S)$ à la fois pour la construction de $G^{\leftarrow}$ et le parcours, et donc au total.

8. De tout ce qui précède, déduisez un algorithme qui calcule la partition des sommets de G correspondant à ses composantes fortement connexes en temps $O(\#S(\#A + \#S))$ (et en fait $O(\#S(\#A + 1))$).

La composante fortement connexe contenant le sommet v est donnée par l'intersection de S_v et $S_v^{\leftarrow}$; chacun de ces ensembles peut se calculer en temps $O(\#A + \#S)$, et l'intersection en $O(\#S)$ en utilisant une structure de donnée appropriée (si $S = [\![n]\!]$ pour un certain n, un simple tableau de booléens suffit). L'ensemble des composantes fortement connexes peut donc se calculer en répétant ce processus pour chaque sommet dont la composante fortement connexe n'a pas encore été déterminée, soit au plus $\#S$ fois. L'optimisation en $O(\#S(\#A + 1))$ vient simplement du fait que si $\#A < \#S$, il est inutile d'effectuer un parcours pour les sommets de degré sortant zéro et l'on effectue donc au plus $\#A$ parcours.

L'algorithme précédent (et les résultats intermédiaires y ayant mené) peuvent se résumer de façon plus élégante (?) en utilisant la *fermeture transitive* G^C de G. Celle-ci est définie comme le graphe S, A^C t.q. $(v, w) \in A^C$ ssi. dans G l'on a $v = w \vee (v \rightsquigarrow w)$ Autrement dit, c'est le graphe représentant la fermeture réflexive-transitive de la relation $\rightsquigarrow$ (qui n'est autre que la relation R).
On admettra le résultat suivant (démontré après les vacances) : si G est représenté par matrice d'adjacence, la représentation par matrice d'adjacence de sa fermeture transitive peut être calculée en temps $O(\#S^3)$.

9. Esquissez un algorithme qui étant donnée une matrice d'adjacence représentant G^C calcule les composantes fortement connexes de G en temps $O(\#S^2)$.

On note $\boldsymbol{C}$ la matrice d'adjacence (à valeur dans $\{0, 1\}$) de la fermeture transitive de G. Par définition de cette dernière, les indices à 1 de la ligne $\boldsymbol{C}_{v,\cdot}$ (resp. colonne $\boldsymbol{C}_{\cdot,v}$) indiquent l'ensemble S_v des sommets accessibles depuis v (resp. l'ensemble $S_v^{\leftarrow}$ des sommets depuis lesquels v est accessible) dans G. La composante fortement connexe contenant v est donnée par l'intersection de ces deux ensembles, qui peut être calculée en temps $O(\#S)$ par un « ET » terme à terme entre cette ligne et cette colonne. En répétant ce processus pour chaque sommet dont la composante fortement connexe n'a pas encore été déterminée, on peut calculer l'ensemble des composantes fortement connexes à partir de $\boldsymbol{C}$ en temps $O(\#S^2)$.

Exercice 3. *Fermeture transitive d'un graphe non-orienté*

On considère à nouveau la *fermeture transitive* d'un graphe $G = S, A$ quelconque, définie à la fin de l'exercice précédent.
On s'intéresse ici uniquement au cas de graphes **non-orientés** représentés par tableau de listes d'adjacence.

1. Esquissez un algorithme qui calcule une « information équivalente » au graphe de la fermeture transitive G^C en temps linéaire en la taille de la représentation de G.

Dans le cas d'un graphe non orienté, cette « information équivalente » est donnée par la partition des sommets selon leurs composantes connexes, et cette dernière peut être calculée par un unique parcours de tout le graphe (ce qui ici se fait bien en temps linéaire en la taille de sa représentation).

2. Pouvez-vous esquisser une modification de votre algorithme pour qu'il renvoie une représentation de G^C de type OCaml `int list array`, toujours en temps linéaire ?

Une fois déterminée la partition des sommets en composantes connexes, il suffit de construire **une fois** pour chaque composante connexe la `int list` de ses sommets. Le coût total de ces constructions est un $O(\#S)$, et chaque sommet d'une même composante peut alors partager la même `int list` (on exploite ici le fait que dans notre définition de G^C chaque sommet possède une boucle). On peut donc en fait construire G^C en temps $O(\#S)$, qui est trivialement aussi un $O(\#A + \#S)$.

Exercice 4. *Handshake lemma*

1. Montrez le *handshake lemma* : un graphe non orienté possède un nombre pair de sommets de degré impair.

Soit G un graphe non-orienté possédant au moins un sommet de degré impair, on va prouver le résultat en construisant une suite finie de sous-graphes de G qui termine quand G ne possède plus aucun sommet de degré impair et dont le nombre de tels sommets décroît de 0 ou 2 entre chaque terme (ce qui prouvera le résultat).
On initialise la suite à $G_0 = G$ et l'on procède de la façon informelle suivante pour calculer G_{i+1} à partir de G_i : on choisit un sommet de degré impair v arbitraire de $G_i = S, A_i$, et l'on définit G_{i+1} comme $S, A_i \setminus \{(v, w)\}$ pour un w arbitraire (A_i contient au moins un arc de cette forme puisque d_v est impair donc strictement positif). Par construction, v est de degré pair dans G_{i+1}. On distingue alors deux cas pour w : s'il était de degré impair dans G_i il devient de degré pair dans G_{i+1}, et dans ce cas le nombre de sommets de degré impair dans G_{i+1} a diminué de deux par rapport à celui dans G_i ; sinon (il était de degré pair et) il devient de degré impair, et dans ce cas le nombre de sommets de degré impair dans G_{i+1} est le même que pour G_i.
Enfin l'algorithme de construction de la suite termine puisque le nombre d'arête de chaque terme de la suite décroît strictement et est minoré par 0 (c'est donc un variant).