DM #5 — Exercices de graphe des vacances de printemps (avec solutions)

Exercice 1. *Transposé d'un DAG*

On rappelle qu'un DAG est un graphe orienté sans cycle.

1. Montrez que le graphe transposé d'un DAG est un DAG.

> On montre le résultat équivalent que si un graphe G possède un cycle, alors il en va de même de son transposé. Soit $v_1 \to \cdots \to v_n \to v_1$ un cycle de G (un chemin simple élémentaire fermé qui n'est pas une boucle), alors $v_1 \to v_n \to \cdots \to v_1$ est un chemin simple élémentaire fermé qui n'est pas une boucle dans le transposé $G^{\leftarrow}$, qui contient donc un cycle.

Exercice 2. *Acyclicité du graphe des composantes fortement connexes*

On définit (un peu informellement) le *graphe des composantes fortement connexes* (GCFC) d'un graphe $G = S, A$ comme le graphe orienté $H = S', A'$ dont les sommets « représentent » les composantes fortement connexes de G (notamment $\#S'$ est égal au nombre de c.f.c. de G) et soit $v, w \in S'$ et C_v, C_w les c.f.c associées de G, $(v, w) \in A'$ ssi. $\exists v' \in C_v, \exists w' \in C_w.\ (v', w') \in A$.

1. Soit G un graphe orienté quelconque, montrez que le graphe de ses composantes fortement connexes H est un DAG (ne possède pas de cycle).

> On suppose par l'absurde la présence d'un cycle $v_1 \to v_2 \to \cdots \to v_1$ dans H :(Par définition de H, il existe un sommet w_1 (resp. w_2) de la c.f.c V_1 représentée par v_1 (resp. V_2 représentée par v_2) tel qu'il existe un arc $(w_1, w_2) \in A$. Par définition des c.f.c, on a donc que $\forall w'_1 \in V_1, w'_2 \in V_2.\ \exists w'_1 \rightsquigarrow w'_2$ (on a la garantie d'existence des chemins $w'_1 \rightsquigarrow w_1$ & $w_2 \rightsquigarrow w'_2$ et $w'_1 \rightsquigarrow w_1 \to w_2 \rightsquigarrow w'_2$ est donc un tel chemin).
> Si le cycle $v_1 \to v_2 \to v_1$ est de longueur deux, on a de même que $\forall w'_2 \in V_2, w'_1 \in V_1.\ \exists w'_2 \rightsquigarrow w'_1$. Le sous-graphe de G induit par $V_1 \cup V_2$ est donc fortement connexe, ce qui contredit la maximalité des composantes fortement connexes.
> Si le cycle est de longueur strictement supérieure à deux, on a de la même façon l'existence d'un chemin entre tout sommet de V_n et V_1, et une récurrence immédiate sur la longueur du chemin permet de montrer l'existence d'un chemin entre tout sommet de V_1 et de V_i, ce qui permet de conclure pour $i = n$.

Exercice 3. *Coupe particulière dans un DAG particulier*

(Adapté d'un exercice de Jeff Erickson)

Dans cet exercice, on considère un DAG connexe $G = S, A$ représenté par tableau de listes d'adjacence et possédant un unique sommet s de degré entrant nul (c'est à dire tel qu'il n'y a aucun arc orienté $* \to s$), appelé *source*, et un unique sommet t de degré sortant nul (c'est à dire tel qu'il n'y a aucun arc orienté $t \to *$), appelé *puits*. On définit la (s, t)-*coupe* de G comme l'ensemble maximal de sommets de G distincts de s et t tel que pour tout sommet v appartenant à cette coupe, tout chemin $s \rightsquigarrow t$ passe par v (ou de façon équivalente, le sous-graphe de G induit par $S \setminus \{v\}$ ne possède pas de chemin $s \rightsquigarrow t$). *Cf.* Fig. 1.

L'objectif de cet exercice est de trouver un algorithme efficace (en temps linéaire en la taille de la représentation) pour calculer la (s, t)-coupe de G.

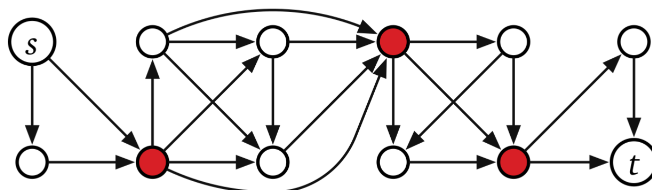

FIGURE 1 – Un exemple de *(s,t)*-coupe (en rouge). Crédit : Jeff Erickson

Deux options. Vous pouvez chercher à résoudre l'exercice sans aucune indication (c'est ainsi qu'il a été posé par son auteur original), ou suivre les étapes proposées ci-dessous.

1. Montrez qu'il existe un ordre topologique pour G tel que s et t sont respectivement minimaux et maximaux pour cet ordre. (Ceci est en fait vrai pour tous les ordres topologiques possibles, comme le montre implicitement la question suivante.)

   Puisqu'il n'y a aucun arc $* \to s$ (resp. $t \to *$) dans G, on peut définir que $s < v$ (resp. $v < t$) pour tout sommet v sans violer la contrainte d'un ordre topologique.

2. Montrez que pour tout sommet v quelconque de G, on a que v est accessible depuis s ($v = s$ ou il existe un chemin $s \rightsquigarrow v$) et t est accessible depuis v.

   On prouve le résultat en montrant la correction totale d'un algorithme qui construit explicitement un tel chemin (si nécessaire). On esquisse un tel algorithme pour le cas $s \rightsquigarrow v$ (le cas $v \rightsquigarrow s$ se traite de façon identique *mutatis mutandis*), qui procède récursivement.
   Si $v = s$ il n'y a rien à faire (on peut par exemple renvoyer un chemin vide).
   Sinon l'on considère l'ensemble des sommets immédiatement « prédécesseurs » de v, c'est à dire $\{v' \in S \mid (v', v) \in A\}$. Celui-ci est non vide puisque $v \neq s$ l'unique source de G. On choisit alors un v' quelconque dans cet ensemble et procède récursivement pour trouver un chemin $s \rightsquigarrow v'$, ce qui donne un chemin $s \rightsquigarrow v' \to v$ de s à v. Cet algorithme est partiellement correct, et il termine car G est sans cycle (on peut prouver cela rigoureusement par exemple en ajoutant une entrée donnant l'ensemble des sommets déjà visités, dont la taille est un variant).

Pour tout sommet $v \neq s, t$, on note P_v (resp. S_v) l'ensemble des sommets de G plus petits (resp. plus grands) que v pour un ordre topologique quelconque satisfaisant la condition de la première question.

3. Montrez que v appartient à la coupe ssi. il n'existe d'arc $a \to b$ pour aucun $a \in P_v$, $b \in S_v$.

   Dans le cas où il n'existe aucun tel arc, alors tout éventuel chemin $a \rightsquigarrow b$ (et en particulier $s \rightsquigarrow t$) doit passer par v (les rangs dans l'ordre topologique des sommets sont strictement croissants, et soit a' (resp. b') le sommet de P_v (resp. S_v) de rang le plus grand (resp. petit) dans le chemin, les seuls cas possibles pour le chemin sont $\cdots \to a' \to v \to b' \to \cdots$ ou $\cdots \to a' \to b' \to \cdots$, mais ce dernier est impossible par hypothèse) ; supprimer ce sommet rend donc t inaccessible depuis s, et v appartient donc bien à la coupe.
   Dans le cas où il existe un tel arc, alors par la question précédente il existe un chemin $s \rightsquigarrow a \to b \rightsquigarrow t$ qui ne passe pas par v, et donc supprimer v ne rend pas t inaccessible depuis s, et donc v n'est pas dans la coupe.

4. Déduisez de ce qui précède un algorithme de coût $O(\#S\#A)$ permettant de calculer la (s, t)-coupe de G.

   On commence par calculer en temps $O(\#A + \#S)$ un tri topologique pour G avec s (resp. t) comme élément minimal (resp. maximal). On vérifie ensuite la caractérisation de la question précédente pour chaque sommet, ce qui coûte un $O(\#A)$ par sommet (il faut pour cela possiblement inspecter tous les arcs du graphe, mais déterminer pour un arc donné si les extrémités appartiennent ou non aux ensembles P_* et S_* peut se faire en temps (espéré) constant en utilisant une structure de données appropriée (par ex. une implémentation par table de hachage d'un tableau associatif donnant pour chaque sommet son rang dans l'ordre topologique), et donc $O(\#S\#A)$ au total.

5. Améliorez (au besoin) votre algorithme précédent pour qu'il s'exécute en temps $O(\#A \log \#S + \#S)$, ou même $O(\#A + \#S)$.
   Indication. Une structure de donnée auxiliaire comme une file de priorité peut être utile, mais elle n'est pas nécessaire (et est à éviter pour atteindre le meilleur coût possible).

   Un algorithme (assez) efficace possible est le suivant : on calcule un tri topologique comme à la question précédente, on crée une file contenant initialement le puits s, puis tant qu'il y a des sommets à traiter, on retire le premier élément de la file et l'on ajoute à la file tous les sommets qui lui sont adjacents et qui ne sont pas déjà dedans par ordre topologique croissant (c'est à dire que l'on insère (et l'on retirera) d'abord les éléments les plus petits). Les sommets appartenant à la coupe sont exactement les sommets v (hormis s et t) tels que la file est vide après qu'on les en a retiré. En effet, si ce n'est pas le cas, cela veut dire qu'un sommet w plus petit que v pour l'ordre topologique avait ajouté (et donc est adjacent à) un sommet plus grand que lui, et par la question précédente v n'est pas dans la coupe ; sinon cela veut dire qu'aucun tel sommet w n'existe, et v est donc dans la coupe.
   Le coût de cet algorithme est celui du calcul d'un tri topologique (soit $O(\#A + \#S)$) plus $O(\#A)$ parcours d'arcs et $O(\#S)$ opérations de file et de tests (notamment de tests d'appartenance à la file). Dans le cas d'une file FIFO, les insertions pourront se faire en coût constant si l'on a au préalable trié les listes d'adjacence par ordre topologique ; ceci peut se faire pour un coût total en $O(\#A \log \#S)$ en utilisant un tri par comparaison optimal ($\log \#A = O(\log \#S)$). (On remarque cependant que si $\#A$ est un $O(\#S^2)$ il serait plus efficace d'utiliser par exemple un tri par dénombrement dont le coût serait ici un $O(N + \#S)$, avec N le nombre de données à trier.) Les extractions se font en coût constant quoi qu'il arrive.

Si l'on utilise à la place une file de priorité (min) implémentée par un tas binaire pour insérer les sommets avec une priorité correspondant à leur rang dans l'ordre topologique, les $\#S$ opérations se feront pour un coût total $O(\#S \log \#S)$ (sans nécessiter de tri préalable).
Dans les deux cas, maintenir une structure de données donnant les sommets présents dans la file peut se faire en coût (espéré) constant ou $O(\log \#S)$ par opération en utilisant par exemple une table de hachage ou un ABR équilibré.
Au final, le coût total est bien un $O(\#A \log \#S + \#S)$.
On donne une description plus formelle ci-dessous sous la forme d'une fonction OCaml, pour une représentation usuelle du graphe. On suppose implémentées par ailleurs les fonctions de tri topologique et (ici) de file de priorité. Le test de présence de la file se fait avec un simple tableau de booléens, qui exploite également le fait qu'un sommet ne peut plus être ajouté une fois qu'il a été extrait (ceci contredirait l'ordre topologique). Enfin, on inclut ici s et t dans la coupe, ce qui simplifie (légèrement) l'écriture.

```
let st_cut g s =
    let torder = topological_sort g in
    let pq = pq_create () in
    let vis = Array.make (Array.length g) false in
    let push_cond v =
        if not vis.(v) then (vis.(v) <- true ; pq_push pq (v, torder.(v)))
    in
    let rec loop cut
        = function
        | None -> cut
        | Some v -> let cut' = if pq_is_empty pq then v::cut else cut in
                    let () = List.iter push_cond g.(v) in
                    loop cut' (pq_pop_opt pq)
    in
    loop [] (Some s)
```

Une approche un peu différente permet d'encore améliorer le coût en $O(\#A+\#S)$: pour chaque sommet v on calcule le rang maximum m_v dans l'ordre topologique des sommets qui lui sont adjacents (pour un coût $O(\#A+\#S)$) ; on parcourt ensuite les sommets par ordre topologique croissant, et un sommet appartient à la coupe ssi. son rang est supérieur ou égal aux maxima m_v des sommets qui le précèdent. Ce dernier parcours (et tous les tests nécessaires) peut être effectué pour un coût linéaire en $\#S$, ce qui donne un coût total en $O(\#A+\#S)$.
(Merci à Jean-Baptiste Bianquis pour avoir initialement proposé cette meilleure solution.)

Exercice 4. *Trouver tous les isthmes* ⋆

Dans tout cet exercice, on considère un graphe non-orienté $G = S,A$ connexe (le propos se généralise immédiatement à un graphe non connexe). On appelle *isthme* de G toute arête $a \in A$ t.q. le graphe $G' := S,A \backslash \{a\}$ n'est pas connexe. On rappelle également qu'un *arbre couvrant* de G en est un sous-graphe $G'' = S,A''$ connexe.

1. Montrez que si une arête est un isthme alors elle appartient à tout arbre couvrant de G.

Si une arête a n'apparaît pas dans un arbre couvrant S,A'' de G, alors $A'' \subseteq A \setminus \{a\}$ et $S,A \setminus \{a\}$ est connexe. Cette arête n'est donc pas un isthme.

2. Donnez un encadrement du nombre possible ι d'isthmes pour un graphe connexe quelconque de N sommets, et un exemple de graphe atteignant chacune des bornes.

Par la question précédente l'on a $0 \leq \iota \leq N-1$ (puisqu'un arbre couvrant de G est acyclique et possède donc $N-1$ arêtes). La borne 0 s'atteint par exemple pour le graphe cycle C_N, et la borne $N-1$ pour tout graphe acyclique.

3. Montrez qu'une arête est un isthme ssi. elle n'appartient à aucun (éventuel) cycle de G.

On commence par montrer que si une arête (v_1, v_2) n'est pas un isthme alors elle appartient à un cycle. Par hypothèse, le graphe obtenu en supprimant cette arête est connexe. En particulier il possède un chemin $v_2 \rightsquigarrow v_1$, et notamment un chemin simple élémentaire (que l'on peut par exemple déduire de l'arborescence d'un parcours d'origine v_2 dans le graphe où l'arête a été supprimée). Ajouter l'arête (v_1, v_2) à ce chemin donne alors un cycle.
On utilise la même approche pour le sens réciproque. Soit (v_1, v_2) une arête qui n'appartient à aucun cycle, alors il n'existe aucun chemin reliant v_2 à v_1 dans le graphe obtenu en la retirant, et celle-ci est donc un isthme. En

effet, supposons par l'absurde :(l'existence d'un tel chemin (que l'on suppose simple et élémentaire sans perte de généralité), alors ajouter (v_1, v_2) à celui-ci donne un cycle dans le graphe original, ce qui est une contradiction.

4. Déduisez de ce qui précède un algorithme trouvant tous les isthmes de G (représenté par tableau de listes d'adjacence) en temps linéaire en la taille de sa représentation.

Conseils. Commencez par écrire l'algorithme immédiatement suggéré par les questions précédentes sans vous soucier du coût, puis essayez de l'améliorer pour atteindre le coût cible. Pour cela, il peut être intéressant de d'abord faire le parcours, **puis** de faire ce qu'il faut par ordre de pré-visite croissant en s'arrêtant dès que possible.

Par les questions précédentes il suffit de calculer un arbre couvrant (*a priori* quelconque) et d'en retirer toutes les arêtes appartenant à (au moins) un cycle. Une façon efficace de réaliser cette seconde opération consiste à utiliser le fait que tout cycle d'un graphe contient un arc (ou ici une arête) arrière dans tout parcours en profondeur.
Ceci donne l'algorithme suivant : on réalise un parcours en profondeur depuis un sommet quelconque v de G, et l'on maintient une structure de données d'ensemble C stockant des arêtes de A qui ne peuvent pas être des isthmes. Au cours du parcours, si une arête arrière $w \to w'$ est détectée, soit $w' \rightsquigarrow w$ le chemin (simple élémentaire) reliant w' à w dans l'arborescence du parcours alors $w' \rightsquigarrow w \to w'$ est un cycle et toute arête composant ce chemin n'est pas un isthme, et l'on insère alors celles-ci dans C. Puisque tout cycle doit emprunter une arête arrière dans l'arborescence d'un parcours en profondeur, C contient par construction *toutes* les arêtes de l'arborescence qui apparaissent dans (au moins) un cycle. Il s'ensuit qu'à l'issue du parcours, les isthmes sont exactement les arêtes de l'arborescence qui ne sont pas contenues dans C.

Même en supposant un coût constant pour toute opération portant sur C, le coût de cet algorithme n'est pas nécessairement linéaire en la taille de la représentation de G. En effet, si le parcours a bien un coût linéaire, on pourrait être amené à vouloir ajouter la même arête dans C de nombreuses fois, ce qui pourrait faire augmenter le coût asymptotique.
On pourrait éviter cela en utilisant une structure de données efficace pour C (en représentant des *ensembles* d'arêtes qui peuvent être fusionnés), mais il est plus élégant d'adapter l'algorithme afin de considérer les arêtes arrières du haut de l'arborescence vers le bas : ceci va permettre d'interrompre l'ajout d'arête dans C dès que l'on rencontre une arête qui y est déjà présente (ce que l'on ne peut pas faire dans l'algorithme décrit ci-dessus).
Avant de décrire cette nouvelle variante avec plus de détails, on explicite une façon efficace de représenter C : puisqu'il s'agit uniquement de représenter des arêtes d'une arborescence de parcours, on peut identifier l'unique arête (i, j) telle que i est parent de j dans l'arborescence avec le sommet j. Si l'on a $S = [\![n]\!]$, un simple tableau de booléens donne alors une représentation de C dont toutes les opérations élémentaires sont en temps constant.
Maintenant, on effectue un premier parcours en profondeur depuis un sommet v quelconque, puis l'on énumère les sommets *dans l'ordre de pré-visite croissant*, et pour chaque sommet v_i possédant une arête arrière $v_j \to v_i$ pour le parcours, on marque (au besoin) v_j à vrai dans le tableau représentant C (l'arête reliant v_j à son parent dans le parcours n'est pas un isthme), et l'on « remonte » l'arborescence jusqu'à v_i **ou** jusqu'au premier sommet que l'on rencontre qui est déjà marqué, en marquant les sommets que l'on rencontre (sauf v_i).
Le coût de cet algorithme est linéaire, puisque l'on considère chaque arête du parcours (pour l'ajouter) au plus une fois.
Il est aussi clair que toutes les arêtes marquées ainsi appartiennent à des cycles et ne sont donc pas des isthmes, mais il n'est pas évident que la réciproque tient. On peut prouver ce second sens par récurrence sur le nombre d'arêtes arrière qui ont été traitées. Le cas de base est évident puisque la remontée ne peut s'arrêter qu'à la destination de la première arête arrière, et donc toutes les arêtes de l'arborescence contenues dans le cycle $v_j \rightsquigarrow v_i \to v_j$ ont été ajoutées à C. On distingue deux cas pour la conservation : celle-ci est évidente si la remontée s'arrête à nouveau à la destination de l'arête arrière ; sinon soit v_j (resp. v'_j) le sommet d'où l'on commence (resp. où l'on termine) la remontée et $v_j \to v_i$ l'arête arrière associée (resp. $v''_j \to v'_i$ celle du cycle ayant occasionné le marquage de v'_j, avec v''_j descendant de v'_j dans l'arborescence), par hypothèse de récurrence toutes les arêtes de l'arborescence contenues dans le cycle $v''_j \rightsquigarrow v'_j \rightsquigarrow v'_i$ ont été marquées, et par hypothèse sur l'ordre de traitement des arêtes arrières le temps de pré-visite de v'_i est inférieur à celui de v_i (strictement, si l'on considère $v'_i \neq v_i$ sans perte de généralité). Il existe donc un unique chemin $v'_j \rightsquigarrow v_i \rightsquigarrow v'_i$ dans l'arborescence (il existe un unique chemin dans l'arborescence remontant d'un sommet vers la racine du parcours, v_i et v'_i sont tous deux sur le chemin débutant à v'_j (ou même v_j), et v'_i est plus proche de la racine que v_i sur ce chemin par hypothèse sur leurs temps de pré-visite) et toutes les arêtes $v'_j \rightsquigarrow v_i$ devant être marquées l'ont déjà été.
On propose ci-dessous une version formelle de cet algorithme écrite en OCaml :

```
let bd (g: graph) =
    let n = Array.length g in
    let preorder = Array.make n (-1) in
    let span_tree = Array.make n None in
    let marked = Array.make n false in
```

```
let clock = ref 0 in
let rec dfs v
    = function
    | [] -> ()
    | w::ws -> if preorder.(w) = (-1)  then begin
                   span_tree.(w) <- Some v ;
                   incr clock ;
                   preorder.(w) <- !clock ;
                   dfs w g.(w)
                end ;
                dfs v ws
in
let rec mark_til w v =
    if (w <> v) && (not marked.(w)) then begin
        marked.(w) <- true ;
        match span_tree.(w) with
        | Some w' -> mark_til w' v
        | None -> assert false
    end
in
let is_back_edge w v =
    span_tree.(w) <> Some v && preorder.(w) > preorder.(v)
in
let mark_backs w v = if is_back_edge w v then mark_til w v in
(*****)
let root = 0 in (* arbitrary *)
let () = preorder.(root) <- 0 in
let () = dfs root g.(root) in
let vip = Array.init n (fun i -> (preorder.(i), i)) in
let () = Array.sort Stdlib.compare vip in
let v_in_preorder = Array.map snd vip in
(***)
let bridges = ref [] in
Array.iter (fun w -> List.iter (mark_backs w) g.(w)) v_in_preorder ;
Array.iteri (fun i b ->
                 if not b && i <> root then
                 match span_tree.(i) with
                 | Some j -> bridges := (j, i)::!bridges
                 | None -> assert false)
            marked ;
!bridges
```