Algorithmique Avancée (CS328) – Examen Session 1 2023-24

Durée totale : 1 heure 30 / Tiers Temps = 2 heures

Toute communication (orale, téléphonique, par messagerie, etc.) avec les autres étudiant·e·s est interdite. Aucun document autorisé (la fiche de syntaxe C est inutile).

- Il est **indispensable de rédiger vos réponses, et d'expliquer vos programmes.** Un programme non expliqué n'aura pas la totalité des points.
- Les exercices sont indépendants.
- Les questions ne **SONT PAS** de difficulté croissante.
- Cet examen est **très probablement trop long**, le barème en tiendra compte. Les points donnés dans les questions (pour un total de 23 points) sont des indications de pondération respective (difficulté, longueur) de chaque question.

Solution: Attention au vocabulaire, et l'orthographe, et la grammaire On rappelle :

- Un algorithme donc "cet algorithme" (et non "cette")
- Cette fonction renvoiE.
- Une propriété donc la propriété est vraiE (ou vérifiéE)
- L'algorithme parcourT, l'ordre de parcourS

Le barème finalement utilisé est à peu près celui mentionné. La note finale est calculée avec $1,3 * (exo1 + exo2) + exo3$.

1 Analyse d'algorithme

Soit l'algorithme (décrit en C) suivant :

```
1  int approx_racine(int n){
2     c = 0;
3     s = 1;
4     while (s <= n) {
5        c = c + 1;
6        s = s + 2 * c + 1;
7     }
8     return c;
9  }
```

On cherche dans la suite à montrer que cet algorithme calcule une valeur approchée de la racine carrée de n.

Question #1 (1 point)
Donner les étapes de calcul pour $n = 16$ et $n = 24$.

Question #2 (1 point)
Expliquer pourquoi cet algorithme termine pour tout entier $n > 0$.

Question #3 (1 point)
Montrer que $(c+1)^2 = s$ est un invariant de la boucle. *On précisera en quel point.*

Question #4 (2 points)
Préciser d'avantage "valeur approchée de la racine", et conclure sur la correction de cet algorithme.

Solution:

1. On "logue" c, et s.
   — Pour n=16 $(0,1)$, $(1,4)$, $(2,9)$, $(3,16)$, $(4,25)$ et on retourne 4.
   — Pour n=25 $(0,1)$, $(1,4)$, $(2,9)$, $(3,16)$, $(4,25)$ et on retourne 4 aussi
2. L'algorithme termine pour $n > 0$ car s croit strictement à chaque tour de boucle ($c \geq 0$, donc $s' > s$), et la valeur n est donc forcément dépassée au bout d'un certain nombre d'occurrences de la boucle.
3. On note c_i, s_i les valeurs de c à la fin de la ième boucle while (avant l'accolade fermante de la ligne 7). La boucle 0 peut par commodité être la fin de la ligne 3.
   — $i = 0$, ok (à faire)
   — Supposons l'invariant vrai à la fin de la boucle numéro i. On a donc $(c_i+1)^2 = s_i$. Ensuite, on a $c_{i+1} = c_i + 1$, et $s_{i+1} = s_i + 2c_{i+1} + 1 = HR = (c_i+1)^2 + 2c_{i+1} + 1 = c_{i+1}^2 + 2c_{i+1} + 1 = (c_{i+1}+1)^2$.
4. La valeur retournée est en fait la valeur c telle que $c^2 \leq n < (c+1)^2$. C'est directement la conséquence de l'invariant précédent, couplé à la condition de boucle. On demandait de justifier.

2 Fonctions récursives sur les arbres

On donne une la définition des arbres binaires suivante (les noeuds ne possèdent pas de valeur) : `type arbre = Vide | Noeud (int, arbre, arbre)` Comme dans le cours, on suppose l'existance de constructeurs *Vide*() et *Noeud*(...,...) retournant un objet de type arbre, d'une méthode `est_vide` permettant de savoir si arbre est vide, et d'accesseurs $SAG(arbre), SAD(arbre), valeur(arbre)$ permettant de récupérer le sous-arbre gauche (resp. sous-arbre droit, la valeur) d'un noeud *qui n'est pas vide*.

Solution: Il était explicitement demandé d'utiliser les fonctions d'accès (et d'écrire des fonctions récursives). Cette partie, malgré sa simplicité, n'a pas été bien traitée par la majorité.

La hauteur d'un arbre est 1 dans le cas d'une feuille, et le nombre de noeuds d'un chemin maximal à partir de la racine sinon.

Question #1 (1 point)

Une feuille est un noeud dont les 2 enfants sont vides. Écrire une fonction pour décider si un arbre est une feuille.

Question #2 (1 point)

Écrire une fonction récursive qui calcule le nombre de noeuds d'un arbre binaire (un arbre vide a 0 noeud).

Question #3 (1 point)

Écrire une fonction récursive qui calcule la longueur du plus court chemin de la racine à une feuille (0 si l'arbre est vide, 1 si c'est une feuille ...)

Question #4 (2 points)

La profondeur d'un noeud est la distance à sa racine (la racine est à la profondeur 0, les enfants de la racine sont à la profondeur 1, les enfants de ces enfants à la profondeur 2, ...). Écrire une fonction qui renvoie la somme des profondeurs de l'ensemble des noeuds (non vides) d'un arbre. On pourra écrire une fonction annexe qui "se souvient des profondeurs".

Solution:
Les trois premieres fonctions sont très simples :

1. `si est_vide(arbre) : return False`
   `sinon`
   `return est_vide(SAG(arbre)) et est_vide(SAD(arbre));`
2.
```
int nb_noeuds(Arbre arbre):
    si est_vide(arbre) : return 0
    sinon
      return nb_noeuds(SAG(arbre))+nb_noeuds(SAD(arbre)) +1
```
3.
```
int longueur_min(Arbre arbre):
    si est_vide(arbre) : return 0
    si est_feuille(arbre) : return 1
    sinon
      return min(SAG(arbre), SAD(arbre))+1
```
4. La dernière fonction est légèrement plus compliquée. Pour se souvenir des profondeurs on peut utiliser un accumulateur dans un des paramètres de la fonction

```
int sommeprof_aux(Arbre arbre, int prof){
    if est_vide(arbre) return 0;
    sinon return (prof + sommeprof_aux(SAG(arbre), prof+1)
                       + sommeprof(SAD(arbre), prof+1))
}

sommeprof(arbre)=sommeprof_aux(arbre, 0);
```

3 Algorithmique de graphes

On rappelle que l'on peut stocker les graphes de deux façons (au moins), et ici on stockera par listes d'adjacence L : chaque noeud $u \in \mathcal{S}$ possède la liste de ces voisins directs ($L[u]$).

Le parcours en profondeur d'un graphe G fait usage des constructions suivantes :

— A chaque sommet du graphe est associée une *couleur* : au début de l'exécution de la procédure, tous les sommets sont blancs. Lorsqu'un sommet est rencontré pour la première fois, il devient gris. On noircit enfin un sommet lorsque l'on est en fin de traitement, et que sa liste d'adjacence a été complètement examinée.

— Chaque sommet est également *daté* par l'algorithme, et ceci deux fois : pour un sommet u, $d[u]$ représente le moment où le sommet a été rencontré pour la première fois, et $f[u]$ indique le moment où l'on a fini d'explorer la liste d'adjacence de u. On se servira par conséquent d'une variable entière **globale** "temps" comme compteur événementiel pour la datation.

— La manière dont le graphe est parcouru déterminera aussi une relation "parent-enfant" entre les sommets, et l'on notera $\pi[v] = u$ pour dire que u est le parent de v (selon l'exploration qui est faite, c'est à dire pendant le parcours v a été découvert directement à partir de u).

On définit alors le parcours en profondeur (**PP**) à l'aide des Algorithmes 1 et 2.

```
début
    pour tous les sommets u ∈ S/* Initialisation                    */
     faire
        couleur[u] ← BLANC;
        π[u] ← NIL;
    temps ← 0;
    pour tous les sommets u ∈ S/* Exploration                       */
     faire
        si couleur[u] = BLANC alors
            Visiter_PP(u);
        fin
fin
```

Algorithme 1 : PP(G) avec G= (S, L)

```
début
    couleur[u] ← GRIS;
    d[u] ← temps;
    temps ← temps + 1;
    pour tous les v ∈ Adj[u] faire
        si couleur[v] = BLANC alors
            π[v] ← u;
            Visiter_PP(v);
        fin
    couleur[u] ← NOIR;
    f[u] ← temps;
    temps ← temps + 1;
fin
```

Algorithme 2 : Fonction auxilliaire : Visiter_PP(u), u sommet de G

Question #1 (1 point)

Donner la réprésentation par listes d'adjacence L du graphe G de la figure 1 **attention, le graphe est orienté**

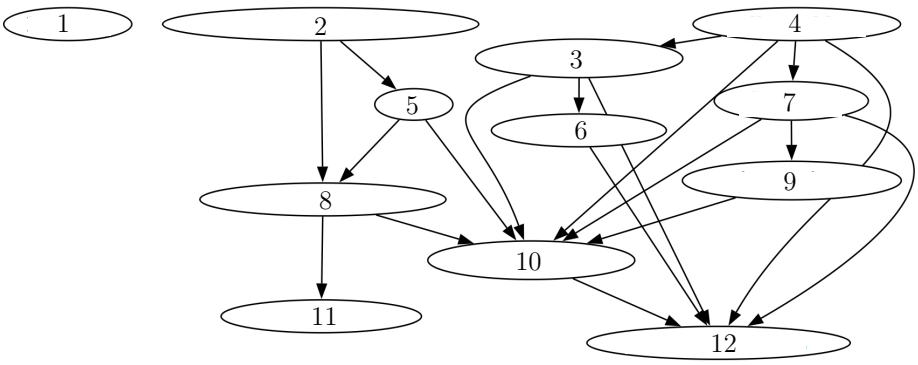


FIGURE 1 – Graphe exemple G, avec $S = \{1, \ldots, 12\}$.

Solution: Pour chaque noeud on donne la liste de ses voisins. Attention le graphe est orienté, donc par exemple $L[2] = [5, 8]$.

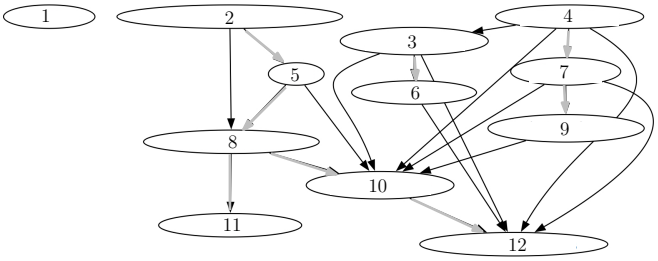
Question #2 (2 points)

Faire tourner l'algorithme sur ce graphe $PP(S, L)$. Lorsqu'il y a un choix entre deux sommets, on prendra le sommet de plus petit numéro. Il est demandé de :

— recopier le graphe et ajouter la relation $\pi[u]$ via des flèches d'une couleur différente.
— donner l'ordre de visite des sommets de ce graphe.
— donner f, d.

Vous pouvez ne pas tout détailler, mais faire proprement le début. Faites attention au calcul du "temps" - la valeur de ce temps (variable globale) est modifiée par visiter_PP.

Solution: Surtout ne pas tout détailler, faire le début, des "et ainsi de suite" et la fin. En gris la relation inverse de π ("parent-enfant") :



sinon, dans l'ordre $d = [0, 2, 14, 18, 3, 15, 19, 4, 20, 5, 9, 6]$ et
$f = [1, 13, 17, 23, 12, 16, 22, 11, 21, 8, 10, 7]$ remarquer qu'on incrémente 2 fois le "temps"

Question #3 (1 point)

Justifiez le fait que l'on n'appelle **Visiter_PP**(u) qu'une et une seule fois par sommet u du graphe, puis une et une seule fois par arc.

Question #4 (1 point)

En déduire la complexité de PP en fonction de |S| et |A| (nombre de sommets et nombre d'arêtes).

Solution: L'initialisation se fait en $\mathcal{O}(|S|)$. Grâce au système de coloriage, on appelle la procédure **Visiter_PP** sur chaque sommet une fois et une seule, soit directement depuis **PP**, soit depuis le traitement d'un autre sommet. Le traitement d'un sommet se fait en $\mathcal{O}(1)$ plus une comparaison pour chaque arc qui part de ce sommet. Donc chaque arc est également visité une fois et une seule car il est visité uniquement depuis son sommet de tête et chaque sommet n'est traité qu'une fois. La complexité du parcours est donc en $\mathcal{O}(|S| + |A|)$.

On dit que v est descendant de u s'il existe une filiation $u \to v_1 \to v_2 \to \ldots \to v$ où chaque $\to$ est une arête du graphe *empruntée par l'algorithme*.
On admet le résultat suivant. Supposons $d[u] < d[v]$ (on rappelle que d est la "date" de première visite)

1. Si v est un descendant de u. On rencontre alors v avant d'avoir terminé u, et on termine de l'explorer avant également. Donc
$$d[u] < d[v] < f[v] < f[u]$$
2. Si v n'est pas un descendant de u, ce qui signifie que l'on termine l'exploration de u avant de commencer celle de v. D'où :
$$d[u] < f[u] < d[v] < f[v]$$

Question #5 (1 point)

Donner 2 couples (u, v) de l'exemple pour illustrer ce résultat (1 pour chaque cas), et vérifier les inégalités.

Solution:

— Pour le couple (2,11) avec $v = 11$ descendant de $u = 2$:

$$d[u] = 2 < d[v] = 9 < f[v] = 10 < f[u] = 13$$

— Pour le couple (1,2) avec $v = 2$ non descendant de $u = 1$

$$d[u] = 0 < f[u] = 1 < d[v] = 2 < f[v] = 13$$

Question #6 (3 points)

Montrer que v est un descendant de u si et seulement si il existe un *chemin blanc* (c'est à dire un chemin qui ne contient que des noeuds coloriés par Blanc) de u à v à l'instant $d[u]$. *Le sens direct est relativement simple, la réciproque, un peu moins, et il est conseillé de la montrer par l'absurde.*

Solution:

Preuve : Pour le sens direct, soit v un descendant de u. Il existe donc un chemin de u à v emprunté lors de l'exécution de l'algorithme. L'algorithme n'empruntant que des sommets blancs, tous les sommets de ce chemin étaient blancs à l'instant $d[u]$. Il existe donc un chemin blanc de u à v à l'instant $d[u]$.

Montrons la réciproque par l'absurde : on suppose qu'il existe un chemin blanc de u à v, mais que v n'est pas descendant de u. Prenons alors v comme le sommet le plus proche de u tel qu'il existe un chemin blanc de u à v à l'instant $d[u]$ et tel que v ne soit pas un descendant de u. Prenons w le sommet précédent v sur ce chemin. On a donc w descendant de u (ou $w = u$) ce qui implique $f(w) \leq f(u)$. De plus v est blanc à l'instant $d[u]$ donc $d[u] < d[v]$. On explore l'arc wv avant de terminer w donc finalement $d[u] < d[v] < f(w) \leq f(u)$. Le seul cas possible pour $f(v)$ est donc $d[u] < d[v] < f(v) < f(u)$ et v est un descendant de u. D'où contradiction avec l'hypothèse de départ.

Nous allons maintenant donner une application de l'algorithme précédent : le **Tri topologique.** n tâches sont données avec des contraintes de précédence $A < B$ signifie que la tâche A doit être effectuée avant la tâche B. L'objectif est de trouver un ordre des tâches qui respecte les contraintes de précédence. Pour cela, on modélise le problème par un graphe orienté :

— les sommets sont les tâches et
— il y a un arc $A \to B$ si et seulement si $A < B$, i.e. si A doit être exécuté avant A.

Par exemple, sur la figure 2, on donne le graphe de contraintes pour permettre à Clark Kent de s'habiller le matin avec son habit de Superman sous son costume (par exemple, le slip est mis après les collants et avant le pantalon noir).

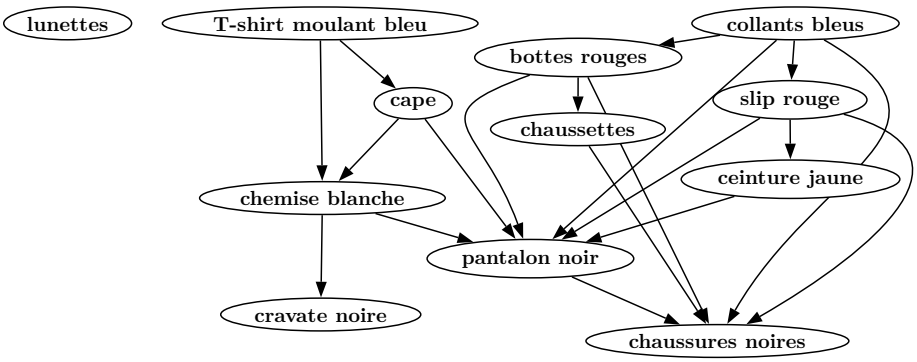


FIGURE 2 – Graphe de dépendances pour habiller superman

Il s'agit alors de trouver un ordre des sommets qui respecte les dépendances, ie qui respecte l'ordre "parent-enfant" dans le graphe. On remarque que si le graphe admet un circuit, ce n'est pas possible.

Question #7 (1 point)

Rajouter dans le graphe exemple des premières questions l'arc $12 - 8$ (afin de créer un circuit). Que se passe-t-il lors du parcours en profondeur ?

> **Solution:** Lorsque l'on regarde les voisins de 12 dans VisiterPP(12), le noeud voisin 8 est déjà marqué en Gris, il n'y a donc pas revisite de ce noeud. Par contre on a détecté un "arc arrière".

On admet le résultat suivant :
Si le parcours en profondeur visite un arc arrière (c'est-à-dire un arc uv tel que v est un ancêtre de u dans l'arborescence produite π), alors il y a un circuit dans le graphe. Réciproquement, si le graphe a un circuit, alors le parcourt *voit* forcément un arc arrière.

> **Solution:**
> On donne une preuve de ce résultat. Et pour un arc arrière, on regarde un enfant qui est GRIS. On suppose qu'il existe un arc arrière (u, v). Alors, le sommet v est un ancêtre du sommet u dans la forêt de parcours en profondeur. Il existe donc un chemin de v à u dans G, et l'arc arrière (u, v) complète un circuit. (il manque un dessin).
>
> Pour la réciproque
> Faire un dessin ! Supposons que G contienne un circuit c. On va montrer qu'un parcours en profondeur de G génère un arc arrière. Soit v le premier sommet découvert dans c, et soit (u, v) l'arc précédent dans c. A l'instant $d[v]$, les sommets de c forment un chemin entre v et u composé de sommets blancs. D'après le théorème du chemin blanc, le sommet u devient un descendant de v dans la forêt de parcours en profondeur. (u, v) est donc un arc arrière.

Passons maintenant à l'algorithme proposé pour ce tri topologique, décrit à la figure 3.

TRI-TOPOLOGIQUE(G)
1 appeler PP(G) pour calculer
les dates de fin de traitement $f[v]$ pour chaque sommet v
2 chaque fois que le traitement d'un sommet se termine,
insérer le sommet début d'une liste chaînée
3 **retourner** la liste chaînée des sommets

FIGURE 3 – Description du tri topologique.

Question #8 (2 points)

Appliquer cet algorithme pour habiller Superman grâce à l'exemple de la question 1.

> **Solution:** Ordre : 4,7,9,3,6,2,5,8,11,12,1. voir correspondance sur les dessins.

Question #9 (1 point)

Quelle est la complexité de cet algorithme ?

> **Solution:** La même que PP.

BonusBonusBonus

Question #10 (Bonus points)

Expliquer pourquoi cet algorithme donne le résultat voulu.

Solution: On raisonne sur les chemins et les couleurs des noeuds,

Supposons que PP soit exécutée sur un graphe orienté sans circuit $G = (S, A)$ donné pour déterminer les dates de fin de traitement de ses sommets. Il suffit de montrer que, pour toute paire de sommets distincts $(u, v) \in S$, s'il existe un arc dans G menant de u à v, alors $f[v] < f[u]$.

On considère un arc (u, v) quelconque exploré par PP(G). Lorsque cet arc est exploré, v ne peut pas être gris, car alors v serait un ancêtre de u, et (u, v) serait un arc arrière, ce qui contredit une des questions précédentes. v est donc soit blanc, soit noir :

— Si v est blanc, il devient un descendant de u, et donc $f[v] < f[u]$.

— Si v est noir, c'est que son traitement est achevé, et donc que $f[v]$ a déjà une valeur. Comme on est encore en train d'explorer à partir de u, il reste encore à dater $f[u]$, et une fois que ce sera fait, on aura derechef $f[v] < f[u]$.

Donc, pour un arc (u, v) quelconque du graphe orienté sans circuit, on a $f[v] < f[u]$, ce qui démontre le théorème.