Analyse d'algorithmes et programmation - 2e contôle continu

Enseignants: Christina Boura, Gaëtan Leurent

Solutions

1 QCM : vrai ou faux ?

Exercice 1
Indiquer si chaque proposition est vraie, fausse, ou si c'est un problème ouvert.

La fonction suivante calcule le n-ième nombre de Fibonacci :

```
def fibo(n):
    if n==0:
        return 1
    else:
        return fibo(n-1)+fibo(n-2)
```

Faux, `fibo(1)` essaye de calculer `fibo(-1)`

La fonction suivante calcule la n-ième puissance de a :

```
def power(a,n):
    if n==0:
        return 1
    else:
        return a*power(a,n-1)
```

Vrai

Il existe un algorithme pour multiplier deux matrices de taille n en temps $\mathcal{O}(n^3)$.

Vrai, l'algorithme naif est en $\mathcal{O}(n^3)$

Il existe un algorithme pour multiplier deux matrices de taille n en temps $\mathcal{O}(n^{2.1})$.

On ne sait pas, le meilleur algorithme connu est en $\mathcal{O}(n^{2.373})$

Il existe un algorithme polynomial pour résoudre le problème 3-SAT
(satisfaisabilité d'une formule sous forme normale conjonctive d'ordre 3).

On ne sait pas ; 3-SAT est NP-Complet

Il existe un algorithme polynomial pour résoudre le problème 2-SAT
(satisfaisabilité d'une formule sous forme normale conjonctive d'ordre 2).

Vrai

$P \subseteq NP$

Vrai

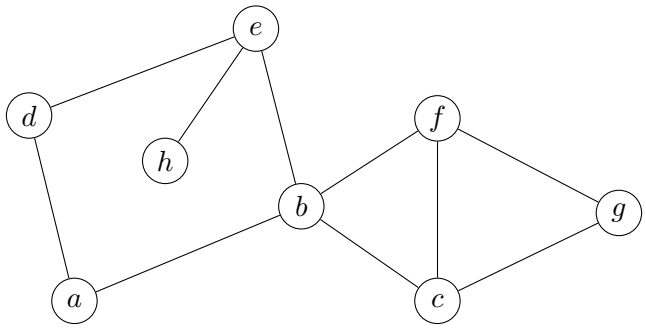
$NP \subseteq P$

On ne sait pas, c'est un problème ouvert à 1 million de dollars.

---

2 Parcours de graphe

On considère le graphe suivant :

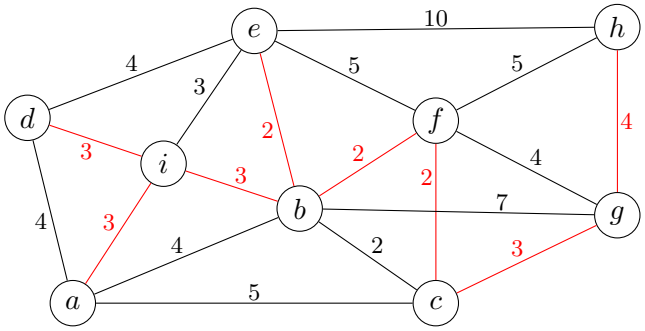


Exercice 2
Donner l'ordre de visite des sommets suivant un parcours en largeur commençant par le sommet "a".
***Réponse** : Il y a plusieurs ordres possibles, par exemple :* `a, b, d, e, f, c, h, g`

Exercice 3
Donner l'ordre de visite des sommets suivant un parcours en profondeur commençant par le sommet "a".
***Réponse** : Il y a plusieurs ordres possibles, par exemple :* `a, b, e, h, d, f, g, c`

3 Arbre couvrant minimal (Algorithme de Kruskal)

On étudie un algorithme glouton pour construire un arbre couvrant minimal (différent de l'algorithme vu en cours). Cet algorithme parcours les arrêtes par ordre croissant de poids, et ajoute une arrête à l'arbre en cours de construction si et seulement si cela ne crée pas de cycle.
Exercice 4 (sur papier)
Appliquer cet algorithme sur le graphe suivant :



Marquer sur le graphe les arrêtes choisies, et donner l'ordre dans lequel elles sont choisies.
Quel est le poids d'un arbre couvrant minimal ?
***Réponse** : Le poids d'un arbre couvrant minimal est de 22.*
Il y a plusieurs choix possibles, par exemple : $(e,b),(f,c),(b,f),(d,i),(c,g),(i,a),(i,b),(g,h)$.

Pour détecter les cycles, on va maintenir une structure d'ensembles disjoints qui correspond aux composantes connexes de l'arbre en cours de construction. Initialement, chaque sommet est dans sa propre composante connexe, et quand une arrête est sélectionnée, les composantes des deux sommets sont fusionnées. Par exemple, on peut utiliser un attribut `.set` sur les noeuds, qui indique à quelle composante il appartient.

L'algorithme peut donc être décrit ainsi :

1: **fonction** ArbreCouvrant(G, w)
2: $A \leftarrow \varnothing$
3: **pour tout** sommet x **faire**
4: MakeSet(x)
5: $L \leftarrow \{\text{arrêtes } (u, v)\}$
6: Sort(L, w) ▷ Trier les arrêtes par poids
7: **pour** $0 \leq i < |L|$ **faire**
8: $(u, v) \leftarrow L[i]$
9: **si** FindSet(u) $\neq$ FindSet(v) **alors**
10: $A \leftarrow A \cup (u, v)$
11: FusionSet(u, v)
12: **retourner** A

On peut construire une structure d'ensembles disjoints tel que les opérations MakeSet, FindSet et FusionSet aient une complexité en $\mathcal{O}(\log n)$ pour une structure contenant n éléments.

Exercice 5 (sur papier)

Quel est alors la complexité de l'algorithme ?

***Réponse** : Sur un graphe avec S sommets et A arrêtes, l'algorithme effectue :*

ligne 4 : S opérations MakeSet	$\mathcal{O}(S\log(S))$
ligne 6 : Un tri d'une liste de taille A	$\mathcal{O}(A\log(A))$
ligne 9 : $2A$ opérations FindSet	$\mathcal{O}(A\log(S))$
ligne 11 : Au plus A opérations FusionSet	$\mathcal{O}(A\log(S))$

Dans un graphe connexe, $S \leq A$, et la complexité totale est donc $\mathcal{O}(A\log(A))$.

Exercice 6 (sur machine)

Toto a implémenté l'algorithme en python, mais son programme ne marche pas.
Corriger le code de toto.
Note : une version informatique du code est disponible sur la page du cours.

```
def arbre_couvrant(G):
    for u in G.noeuds:
        u.set = u
    L = []
    for u in G.noeuds:
        for v,w in zip(u.out, u.weight):
            L.append(((u,v),w)) # Arrete (u,v) de poids w
    L.sort(key=(lambda x: x[1]))
    for (u,v),w in L:
        if u.set != v.set:
            Arbre.append((u,v))
            poids += w
            # Fusion des composantes: on remplace u.set par v.set
            for s in G.noeuds:
                if s.set == u.set:
                    s.set = v.set
    return Arbre,poids
```

Il y a plusieurs erreurs :

1. *Les variables* `Arbre` *et* `poids` *ne sont pas initialisée*
2. *La fusion des composantes ne fonctionne pas correctement : quand* `s` *dans la boucle interne est égal à* `u`*, on modifie* `u.set` *et le test (*`if s.set == u.set`*) ne fonctionne pas*

correctement pour les choix restants de `s`.

À cause de ce problème, la fonction ne renvoie pas un arbre couvrant minimal.

Voici une version corrigée qui calcule bien un arbre couvrant minimal :

```
def kruskal(G):
    Arbre = []
    poids = 0
    for u in G.noeuds:
        u.set = u
    L = []
    for u in G.noeuds:
        for v,w in zip(u.out, u.weight):
            L.append(((u,v),w)) # Arrete (u,v) de poids w
    L.sort(key=(lambda x: x[1]))
    for (u,v),w in L:
        if u.set != v.set:
            Arbre.append((u,v))
            poids += w
            u_set = u.set # On copie la valeur de u.set
            for s in G.noeuds:
                if s.set == u_set:
                    s.set = v.set
    return Arbre,poids
```

Exercice 7 (sur papier)

Prouver par induction qu'à chaque itération de l'algorithme, il existe un arbre couvrant minimal qui contient A.

Indice : on se place au moment où une nouvelle arrête (u, v) est ajoutée dans A. En supposant qu'il existe un arbre couvrant minimal qui contient A, on construit un arbre couvrant minimal qui contient $A \cup (u, v)$.

Au début de l'exécution, A est vide, et la propriété est donc vraie (il existe bien un arbre couvrant minimal, et il contient l'ensemble vide).

Quand on ajoute une arrête, on va montrer que l'arrête choisie par cet algorithme conserve la propriété. Soit A, l'ensemble construit avant d'ajouter l'arrête (u, v), et soit T un arbre couvrant minimal contenant A (qui existe par hypothèse d'induction).

Si $(u, v) \in T$, alors T contient $A \cup \{(u, v)\}$, et la propriété reste vraie.

Sinon, on considère le chemin $u \overset{p}{\rightsquigarrow} v$ dans T (qui existe car T est un arbre couvrant, donc connexe). Il existe une arrête (x, y) dans p qui n'est pas dans A ; sinon l'arrête (u, v) ajouterait un cycle à A et ne serait donc pas choisie par l'algorithme.

De plus, le graphe $A \cup p$ est un sous-graphe de T ; or T est acyclique (car c'est un arbre), donc $A \cup p$ est acyclique. En particulier, ajouter (x, y) ne créerait pas de cycle dans A. On en déduit que le poids de (x, y) est plus faible que celui de (u, v) ($w(x, y) \leq w(u, v)$). Dans le cas contraire, (x, y) aurait été considéré avant (u, v) (car les arrêtes sont considérées par ordre croissant), et serait déjà dans A.

On peut donc construire le graphe $T' = T \setminus \{(x, y)\} \cup \{(u, v)\}$. T' est connexe car $T \setminus \{(x, y)\}$ possède deux composantes connexes qui sont reconnectées par (u, v). Comme T' a autant d'arrêtes que T, c'est bien un arbre couvrant. De plus, on a $w(T') \leq w(T)$ car $w(u, v) \leq w(x, y)$. Donc T' est un arbre couvrant minimal, et la propriété est bien vérifiée.

On en conclut que cet algorithme calcule bien un arbre couvrant minimal.

4 Multiplication de polynômes

On veut calculer le produit de deux polynômes de degré n : $A = \sum_{i=0}^{n} a_i x^i$ et $B = \sum_{i=0}^{n} b_i x^i$.

$$C = A \times B = \sum_{i=0}^{2n} c_i x^i \qquad \text{avec : } c_i = \begin{cases} \sum_{j=0}^{i} a_j \cdot b_{i-j} & \text{si } i \leq n \\ \sum_{j=i-n}^{n} a_j \cdot b_{i-j} & \text{si } i \geq n \end{cases}$$

On représente un polynôme par la liste de ses coefficients $[a_0, a_1, ...a_n]$.

Exercice 8 (sur machine)
Implémenter une fonction `add(A,B)` *qui calcule la somme de deux polynômes.*

```
def add(A,B):
    n = min(len(A), len(B))
    C = [ A[i] + B[i] for i in range(n) ]
    C += A[n:] + B[n:]
    return C
```

Exercice 9 (sur machine)
Implémenter une fonction `multiply_x(A,m)` *qui multiplie un polynôme par* x^m.

```
def multiply_x(A,m):
    return [0]*m + A
```

Pour calculer le produit, on utilise une décomposition en polynômes de degré $n/2$:

$$A = A_1 \cdot x^{n/2} + A_0 \qquad B = B_1 \cdot x^{n/2} + B_0$$

On peut alors calculer C avec

$$C = (A_1 \cdot x^{n/2} + A_0) \times (B_1 \cdot x^{n/2} + B_0)$$
$$C = \underbrace{(A_1 \times B_1)}_{C_2} \cdot x^n + \underbrace{(A_1 \times B_0 + A_0 \times B_1)}_{C_1} \cdot x^{n/2} + \underbrace{(A_0 \times B_0)}_{C_0}$$

On obtient ainsi l'algorithme diviser-pour-régner suivant :

fonction MULTIPLY(A, B)
  $\ell \leftarrow |A|$
  si $\ell = 1$ **alors**
    retourner $[A[0] \times B[0]]$
  sinon
    $A_0 \leftarrow A[0, \ldots, \ell/2 - 1]$
    $A_1 \leftarrow A[\ell/2, \ldots, \ell - 1]$
    $B_0 \leftarrow B[0, \ldots, \ell/2 - 1]$
    $B_1 \leftarrow B[\ell/2, \ldots, \ell - 1]$
    $C_0 \leftarrow$ MULTIPLY(A_0, B_0)
    $C_1 \leftarrow$ ADD(MULTIPLY(A_1, B_0), MULTIPLY(A_0, B_1))
    $C_2 \leftarrow$ MULTIPLY(A_1, B_1)
    retourner ADD$(C_0,$ MUTLIPLYX$(C_1, \ell/2),$ MULTIPLYX$(C_2, \ell))$

Exercice 10 (sur papier)
Quel est la complexité de cet algorithme ?

Pour multiplier deux polynômes de degré n, on utilise 4 multiplications de polynômes de degré $n/2$, et des opération en temps linéaire (additions, multiplications par une puissance de x). La complexité suit donc la formule de récurrence suivante :

$$T(n) = 4T(n/2) + \mathcal{O}(n)$$

D'après le théorème donné en cours, on en déduit $T(n) = \Theta(n^2)$

On peut améliorer cet algorithme en utilisant la relation :

$$C_1 = (A_0 + A_1)(B_0 + B_1) - C_2 - C_0$$

Exercice 11 (sur papier)
Décrire l'algorithme amélioré. Quel est sa complexité ?

On obtient l'algorithme suivant :

1: ***fonction*** MULTIPLY(A, B)
2: $\ell \leftarrow |A|$
3: ***si*** $\ell = 1$ ***alors***
4: ***retourner*** $[A[0] \times B[0]]$
5: ***sinon***
6: $A_0 \leftarrow A[0, \ldots, \ell/2 - 1]$
7: $A_1 \leftarrow A[\ell/2, \ldots, \ell - 1]$
8: $B_0 \leftarrow B[0, \ldots, \ell/2 - 1]$
9: $B_1 \leftarrow B[\ell/2, \ldots, \ell - 1]$
10: $C_0 \leftarrow$ MULTIPLY(A_0, B_0)
11: $C_2 \leftarrow$ MULTIPLY(A_1, B_1)
12: $C_1 \leftarrow$ MULTIPLY$($ADD$(A_0, B_0),$ ADD$(A_1, B_1))$
13: $C_1 \leftarrow$ SUB$(C_1,$ ADD$(C_0, C_2))$
14: ***retourner*** ADD$(C_0,$ MUTLIPLYX$(C_1, \ell/2),$ MULTIPLYX$(C_2, \ell))$

La complexité suit la formule de récurrence suivante :

$$T(n) = 3T(n/2) + \mathcal{O}(n)$$

D'après le théorème donné en cours, on en déduit $T(n) = \Theta(n^{\log_2(3)})$

Exercice 12 (sur machine)
Implémenter le nouvel algorithme en python.

Exercice 13 (sur machine)
Ajouter des tests pour gérer des polynômes de degré différents, et tels que ℓ ne soit pas une puissance de deux.

```
def neg(A):
    return [-a for a in A]

def multiply(A,B):
    if A == [] or B == []:
        return []
    n = max(len(A),len(B))
    if n == 1:
        return [A[0]*B[0]]
    n2 = n//2
    A0,A1 = A[:n2],A[n2:]
    B0,B1 = B[:n2],B[n2:]
    C0 = multiply(A0,B0)
    C2 = multiply(A1,B1)
    C1 = add(multiply(add(A0,A1),add(B0,B1)),neg(add(C0,C2)))
    return add(C0,add(multiply_x(C1,n2),multiply_x(C2,2*n2)))
```