CentraleSupélec

1re année 2018–2019
ST2, campus de Saclay

Vendredi 25 janvier 2018, matin

Examen écrit de

Algorithme et Compléxité

Durée : 3 heures

Numéro étudiant :

Nom : Prénom :

— Tous les documents sont autorisés.
— Les outils électroniques (calculatrices, ordinateurs et téléphones) sont interdits.
— Tous vos algorithmes peuvent être écrits en pseudo-code ou en Python, selon ce que vous préférez.
— Le barème est donné à titre indicatif.
— Ce sujet comporte 8 pages.
— Vous pouvez utiliser les pages supplémentaires en fin de sujet si vous manquez de place.
— Les exercices sont indépendants.

Exercice 1 : Les amis de mes amis — 3 point(s)

On considère un réseau social qui, pour chaque utilisateur, stocke sa liste d'amis. On dispose ainsi d'une fonction `friends(u)` qui retourne la liste des amis de l'utilisateur `u`.

On souhaite écrire un algorithme qui propose de nouveaux contacts à partir des amis de nos amis.

Question 1 — 2 point(s)

Écrivez, en pseudo-code ou en Python, une fonction `propose(u)` qui prend en argument un utilisateur `u` et qui propose un **autre** utilisateur qui :

1. ne figure pas déjà dans la liste des amis de `u` ;
2. apparaît le plus fréquemment dans les listes d'amis des amis de `u`.

Correction :

Il faut parcourir les amis d'amis tout en calculant un maximum.

```
def propose(u):
    best_candidate = None
    best_score = -math.inf
    scores = {}
    l = friends(u)
    for a in l:
        for b in friends(a):
            if b!=u and not b in l:
                if not b in scores:
                    scores[b] = 1
                else:
                    scores[b] = scores[b] + 1
                if scores[b] > best_score:
                    best_candidate = b
                    best_scores = scores[b]
```

```
    return best_candidate
```

Question 2 1 point(s)

Pour chaque utilisateur nous souhaitons connaître l'ensemble des utilisateurs qui sont soit ses amis directs, soit pour lesquels il existe une suite des amis qui peuvent servir d'intermédiaires. Autrement dit, nous voulons construire la *fermeture transitive* du graphe de notre réseau social.

Écrivez un algorithme qui fait cette construction.

Correction : Il s'agit d'appliquer un algorithme de parcours de graphe :

```
def fermeture(u):
  l = []
  next = [u]
  while len(next)>0:
    x = next.pop()
    for y in friends(x)
      if not y in l:
        l.append(y)
        next.append(y)
  return l
```

Exercice 2 : Couverture par les sommets 4 point(s)

La RATP souhaite installer des caméras dans la station de métro Châtelet de manière à pouvoir suivre les déplacements des voyageurs au sein de la station. Les caméras seront installées aux intersections des couloirs : elles permettent de repérer toute personne passant par l'un des couloirs reliés à cette intersection.

L'objectif est de pouvoir surveiller tous les couloirs de la station. Pour des raisons de coût, on souhaite placer le moins de caméras possibles.

Question 1 1.5 point(s)

Modélisez ce problème. Quelle est la nature du problème ? Quelles sont les entrées et quelle est la question posée ?

Correction :

C'est un problème d'*optimisation* :

MAX-VERTEX-COVER
Entrée : Un graphe $G = (V, E)$ non orienté.
Question : Quel est le plus petit ensemble $S \subseteq V$ tel que chaque arête $e \in E$ a au moins une extrémité dans S ?

Nous considérons l'algorithme suivant qui permet de construire une solution au problème posé. Il prend en paramètre la liste des couloirs *corridors* dont les éléments sont des couples d'intersections $(i1, i2)$. Chaque couloir est ainsi identifié par les deux points de la station qu'il relie entre eux.

```
def compute_solution(corridors)

  R = corridors.copy()
  S = []

  while len(R)>0:
    (i1,i2) = R.pop()

    S.append(i1)
    S.append(i2)

    R2 = []
    for r in R:
      if r[0]!=i1 and r[1]!=i1 and r[0]!=i2 and r[1]!=i2:
        R2.append(r)
    R = R2

  return S
```

Question 2 1.5 point(s)

Que fait cet algorithme ? Expliquez son fonctionnement en français en utilisant des termes de votre propre modélisation. Indiquez à quelle famille d'algorithmes il appartient. Calculez sa complexité en temps. Justifiez chacune de vos réponses.

Correction : Cet algorithme **glouton** calcule une solution approchée au problème d'optimisation en $\mathcal{O}(|E|^2)$. Tant qu'il reste des arêtes non visitées, on en prend une quelconque. On ajoute ses deux extrémités à la solution S et on retire toutes les arêtes incidentes à l'une de ces deux extrémités.

Question 3 1 point(s)

Montrez que cet algorithme est une 2-approximation, c'est-à-dire qu'il ne donne jamais une solution S de taille supérieure à 2 fois la taille des solutions optimales.

Correction : Soit A l'ensemble des arêtes sélectionnées par l'algorithme. Pour chaque arête de A, deux sommets sont ajoutés dans la solution approchée S, donc $|S| = 2|A|$. Les arêtes de A ne partagent aucun sommet en commun, en conséquence l'ensemble optimal de sommets O contient au moins $|A|$ sommets. Par conséquent :

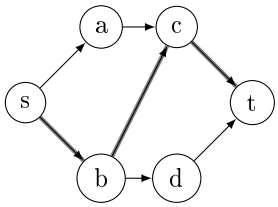
$$|A| \leq |O| \leq |S| \leq 2|A| \leq 2|O|$$

Exercice 3 : Recherche de chemins — 3 point(s)

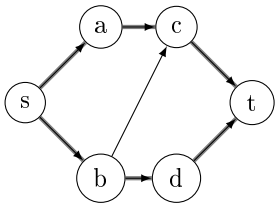
On s'intéresse au problème suivant :

> **MAX-EDGE-DISJOINT-PATHS**
> **Entrée** : Un graphe orienté $G = (V, E)$ et deux sommets s et t de G.
> **Question** : Construire le plus grand nombre possible de chemins de s à t sans utiliser deux fois le même arc.

Chaque arc ne peut donc être utilisé **que dans un seul chemin**. Le but est d'obtenir le plus de chemins possibles. Sur l'exemple ci-dessous, il est possible de construire une solution à deux chemins (c'est le maximum).



Solution à 1 chemin



Solution à 2 chemins

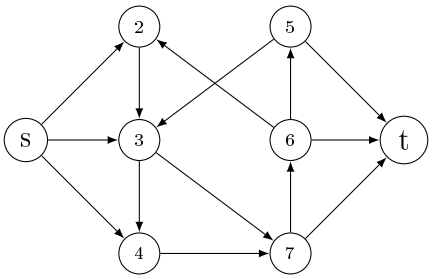
Question 1 — 2 point(s)

Donnez un algorithme polynomial qui calcule le **nombre** maximum de chemins possibles. Justifiez votre réponse.

Correction : On le transforme en graphe à flot. Nous donnons à chaque arc une capacité 1. Nous calculons le flot max sur ce graphe. La valeur du flot max est le maximum de chemins.

Question 2 — 1 point(s)

Déroulez cet algorithme sur le problème suivant en détaillant bien toutes les étapes. Quel est le nombre maximum de chemins ?



Correction : On calcule le flot-max, on trouve deux chaines augmentantes, par exemple :
- s → 4 → 7 → t
- s → 3 → 7 → 6 → t

Exercice 4 : Une soirée « fun » — 5 point(s)

Une entreprise organise une soirée pour ses employés. Les organisateurs veulent que la soirée soit la plus divertissante possible.

Note de « fun ». La capacité à plaisanter et à être de bonne humeur de chaque employé est connue et répertoriée par les organisateurs. Elle est exprimée par une note « fun » attribuée à chaque employé.

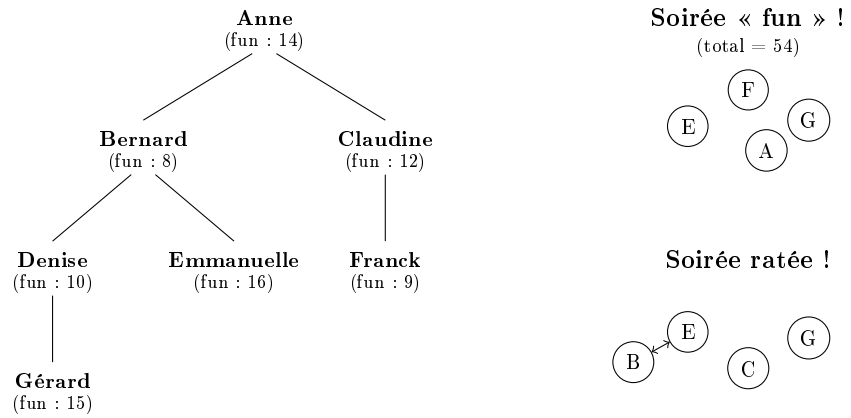


FIGURE 1 – Exemple simplifié de soirées dans une entreprise de 7 personnes

Jamais avec mon « n+1 ». Chaque employé a une position hiérarchique stricte dans l'entreprise. Il est alors impossible qu'un employé et son « n+1 » (c'est-à-dire son supérieur hiérarchique direct) soient tous les deux présents à la soirée, car cela ne serait pas « fun » du tout...

Le but est d'organiser une soirée telle que la somme des notes « fun » de tous les participants à la soirée soit maximale, tout en respectant la contrainte « jamais avec mon n+1 ». Pour ce faire, nous écrirons un algorithme en utilisant le principe de la **programmation dynamique**.

Fonctions utiles

Pour un employé i quelconque, on note :
- $\text{fun}(i)$ la valeur de sa note de « fun » ;
- $\text{sub}(i)$ la liste de ses subordonnés directs (ceux dont il est le « n+1 »).

Nous allons calculer deux valeurs :
- $\text{total}_{\text{yes}}(i)$ le score maximum de « fun » d'une soirée limitée à l'employé i et tous ses subordonnés directs et indirects, **lorsque i est présent à la soirée** ;
- $\text{total}_{\text{no}}(i)$ le score maximum de « fun » d'une soirée limitée à l'employé i et tous ses subordonnés directs et indirects, lorsque i **n'est pas présent** à la soirée.

Question 1

2 point(s)

Écrivez les formules de récurrence de $\text{total}_{\text{yes}}(i)$ et $\text{total}_{\text{no}}(i)$ en fonction des valeurs de $\text{total}_{\text{yes}}$ et total_{no} des subordonnés.

Correction :

$$\text{total}_{\text{yes}}(i) = \text{fun}(i) + \sum_{j \in \text{sub}(i)} \text{total}_{\text{no}}(j)$$

$$\text{total}_{\text{no}}(i) = \sum_{j \in \text{sub}(i)} \max(\text{total}_{\text{yes}}(j), \text{total}_{\text{no}}(j))$$

Question 2

3 point(s)

Écrivez un algorithme de programmation dynamique qui maximise la somme des notes « fun » des participants à la soirée. Vous pouvez écrire une ou plusieurs fonctions selon ce qui vous semble le plus judicieux.

Correction : Une solution possible avec trois fonctions en Python :

```
total_fun_oui = {}
total_fun_non = {}

def total_oui(i):
    if i in total_fun_oui:
        return total_fun_oui[i]
    s = fun(i)
    for j in sub(i):
        s = s+total_non(j)
```

```
    total_fun_oui[i] = s
    return s

def total_non(i):
    if i in total_fun_non:
        return total_fun_non[i]
    s = 0
    for j in sub(i):
        s = s + total(j)
    total_fun_non[i] = s
    return s

def total(i):
    if total_oui(i) > total_non(i):
        return total_oui(i)
    return total_non(i)
```

Exercice 5 : Complexité de problème

3 point(s)

On considère les deux problèmes suivants :

BI-PARTITION
Entrée : Un ensemble E de nombres.
Question : Existe-il un sous-ensemble $F \subseteq E$ tel que $\sum_{x \in F} x = \frac{1}{2} \sum_{x \in E} x$?

SUBSETSUM
Entrée : Un ensemble S de nombres et un nombre t
Question : Existe-il un sous-ensemble $T \subseteq S$ tel que $\sum_{x \in T} x = t$?

Question 1

1 point(s)

Montrez que SUBSETSUM est dans NP.

Correction : Voici un algorithme polynomial pour vérifier une solution T (complexité $\mathcal{O}(|S| * |T|)$) :

```
def verifier(S,t,T):
  S2 = S[:]         # O(|S|)
  s = 0
  for x in T:       # O(|T| * ...
    if x in S2:     #         |S| )
      S2.remove(x)
    else:
      return False
    s = s+x
  return (s==t)
```

Donc SUBSETSUM et dans NP.

On peut aussi utiliser comme structure de donnée pour T un tableau de booléens/bits indiquant la présence ou non de l'élement de S au même index. Cela dispense de vérifier que $T \subseteq S$ et réduit la complexité en $\mathcal{O}(|S|)$ (dans ce cas $|T| = |S|$).

```
def verifier(S,t,T):
    s = 0
    for i in range(len(T)):
        if T[i] :
            s = s+S[i]
    return (s==t)
```

Question 2 2 point(s)

Sachant que BI-PARTITION est NP-complet, montrez que SUBSETSUM est lui aussi NP-complet.

Correction : Prenons une instance E de BI-PARTITION. On construit alors $S = E$, $t = \frac{1}{2}\sum_{x\in E} x$ et on s'intéresse au problème SUBSETSUM sur cette instance (S, t). La transformation de (E) en (S, t) est bien polynomiale (linéaire en $|E|$).

- $\Rightarrow$ Supposons que F soit une solution au problème BI-PARTITION sur E. Alors $T = F$ est bien solution à SUBSETSUM sur (S, t) car $\sum_{x\in T} x = \sum_{x\in F} x = \frac{1}{2}\sum_{x\in E} x = t$.
- $\Leftarrow$ Supposons que T soit une solution au problème SUBSETSUM sur (S, t). Alors $F = T$ est bien solution à BI-PARTITION sur E car $\sum_{x\in F} x = \sum_{x\in T} x = t = \frac{1}{2}\sum_{x\in E} x$.

Nous avons donc montré que BI-PARTITION $<_p$ SUBSETSUM. Or BI-PARTITION est NP-complet. Donc SUBSETSUM est NP-difficile.

Or comme il est aussi NP, il est NP-complet.

Exercice 6 : Retour à l'hôpital **2 point(s)**

On s'intéresse à nouveau au problème de couverture d'un territoire par des hôpitaux que nous avons étudié pendant les deux séances de TP.

Rappels : Nous avons défini une fonction `combination_k` qui peut s'utiliser comme un itérateur dans une boucle pour écrire un algorithme de recherche exhaustive. À titre d'exemple, le code ci-dessous affiche successivement toutes les combinaisons de 10 villes parmi les villes candidates.

```
def test_combination(candidates):
    for c in combination_k(candidates,10):
        print(c) # prints a list of 10 cities
```

Nous avons aussi écrit une fonction `all_distances` qui construit un dictionnaire contenant toutes les distances entre deux villes du territoire. Les clefs du dictionnaire sont les couples de villes et les valeurs sont les distances entre les villes :

```
def test_distances(cities):
    dict_dist = all_distances(cities)
    print(dict_dist[('A','B')])
```

Le code ci-dessus affiche la distance entre les villes de noms A et B du territoire (si tant est qu'elles existent).

Définition : On appelle *diamètre* d'une liste de villes la plus grande distance entre deux villes de la liste.

Question 1 1.5 point(s)

Écrivez une fonction qui prend en argument un territoire, une liste de villes candidates et un nombre k et qui retourne la combinaison de k villes candidates qui a le plus petit diamètre (ou l'une de ces combinaisons s'il y en a plusieurs).

Correction :

```
import math
def couverture(cities,candidats,k):
  best_d = math.inf
  best_c = None
  dict_dist = distance_all(cities)
  for c in combination_k(candidats,k):
    worst_d = 0
    for v1 in c:
      for v2 in c:
        d = dict_dist([(v1,v2)])
        if d > worst_d:
          worst_d = d
    if worst_d < best_d:
     best_d = worst_d
     best_c = c
```

```
    return (best_d, best_c)
```

Question 2 0.5 point(s)

Quelle est la complexité de cet algorithme ? Justifiez votre réponse.

Correction : Le calcul des distance en n^2 (avec $n = |cities|$), puis on parcourt l'ensemble des combinaisons, donc C_n^k (autre notation : $\binom{n}{k}$) passages dans la boucle et à chaque étape, on fait un calcul polynomial en k^2 ...

Exercice 7 : Bonus — 1 point(s)

On considère le problème suivant :

SATISFIABILITY
Entrée :
- Un ensemble l de variables booléennes ;
- Une formule logique $f(l)$ reliant les variables de l.

Question : Existe-il une affectation des valeurs de l telle que $f(l)$ soit vrai ?

Question 1 1 point(s)

Écrivez un algorithme polynomial pour résoudre ce problème.

Correction : ce n'est pas possible, que si "P=NP" !

— fin —

2019-02-07