X2018 — INF411
Contrôle classant / 19 novembre 2019 / 9h–12h

Tous documents de cours autorisés (polycopié, transparents, notes personnelles). Le dictionnaire papier est autorisé pour les EV2 et EV3. Les ordinateurs, Ipad, dictionnaires électroniques, les tablettes et téléphones portables sont interdits.

L'énoncé est composé de deux problèmes indépendants, que vous pourrez traiter dans n'importe quel ordre (en revanche, merci de clairement numéroter les réponses). Vous pourrez, quand vous traiterez une question, considérer comme déjà traitées les questions précédentes du même problème.

Le correcteur prêtera attention à la qualité de la rédaction et vous remercie d'avance d'écrire lisiblement. Merci également de numéroter vos copies en indiquant leur nombre total.

Quelques éléments de la bibliothèque standard Java sont rappelés à la fin du sujet. Ils ne sont pas forcément tous nécessaires. Vous pouvez utiliser librement tout autre élément de la bibliothèque standard Java.

1 Arbres couvrants

Dans ce problème, on s'intéresse à des graphes *non orientés* dont les arcs sont étiquetés par des *poids*. On ne considère que des graphes *connexes*, c'est-à-dire où il existe toujours un chemin entre deux sommets donnés, et *sans boucles*, c'est-à-dire sans arc $a - a$.

En Java, on choisit de représenter de tels graphes à l'aide de deux classes, `Edge` et `Graph`, données figure 1. Les sommets d'un graphe sont représentés par les entiers 0,1,...,`N`-1 où `N` est le nombre de sommets, donné par le champ `N`. La structure du graphe est donnée par le tableau `adj`, où `adj[`i`]` est la liste des arcs issus du sommet i. Un arc est décrit par un objet de la classe `Edge`, où les champs `src` et `dst` sont les deux extrémités de l'arc et où le champ `weight` est le poids de l'arc. Les poids sont ici des nombres flottants, de type `double`.

Le graphe étant non orienté, un arc $a - b$ entre les sommets a et b apparaît à la fois comme un arc issu de a, c'est-à-dire un objet e de type `Edge` dans la liste `adj[`a`]`, avec e`.src` $= a$ et e`.dst` $= b$, et comme un arc issu de b, c'est-à-dire un *autre* objet e' de type `Edge` dans la liste `adj[`b`]`, avec e'`.src` $= b$ et e'`.dst` $= a$. Ces deux objets représentent le *même arc*. En particulier, on a donc e`.weight` $= e'$`.weight`.

```
class Edge {
  final int src, dst;     // un arc entre src et dst
  final double weight;    // le poids de cet arc
}

class Graph {
  final int N;            // les sommets sont les entiers 0,1,...,N-1
  LinkedList<Edge>[] adj; // un tableau de taille N
                          // adj[i] est la liste des arcs issus de i
}
```

FIGURE 1 – Représentation des graphes.

```
class UF {
  UF(int n)       // construit une structure pour les éléments 0,1,...,n-1
  int find(int i) // renvoie le représentant de la classe de i
  void union(int i, int j) // fusionne les classes des éléments i et j
}
```

FIGURE 2 – Structure union-find.

Arbre couvrant. Étant donné un graphe G, un *arbre couvrant* de G est un sous-ensemble T d'arcs de G tel que

1. T est connexe et sans cycle (c'est pourquoi on parle d'*arbre*) ;
2. chaque sommet de G est l'extrémité d'au moins un arc de T (on dit que T *couvre* tous les sommets de G).

Voici par exemple un graphe de six sommets à gauche et l'un de ses arbres couvrants à droite.

(On a ignoré pour l'instant les poids qui étiquettent les arcs.)

Question 1 Montrer que, si un graphe G possède N sommets, tout arbre couvrant de G est composé d'exactement $N - 1$ arcs.

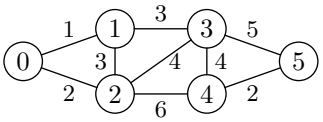
Question 2 Déduire du résultat de la question précédente une méthode

```
static boolean isSpanningTree(Graph g, LinkedList<Edge> t)
```

qui détermine si la liste d'arcs `t` constitue un arbre couvrant du graphe `g`. Indication : pour tester l'absence de cycle, on pourra utiliser une structure union-find, dont l'interface est donnée figure 2. (On ne demande pas de coder une telle structure.) Donner la complexité en temps de votre méthode, en fonction de N. On rappelle que la complexité de la méthode `size` de `LinkedList` est $O(1)$ et que les opérations `find` et `union` peuvent être considérées de complexité $O(1)$ amortie en pratique.

Arbre couvrant minimal. Étant donné un graphe G, un *arbre couvrant minimal* de G est un arbre couvrant de G dont la somme des poids des arcs est minimale.

Question 3 Donner un arbre couvrant minimal pour le graphe suivant, où chaque arc est étiqueté par son poids.



Algorithme de Kruskal. Pour construire un arbre couvrant minimal pour un graphe G, on peut utiliser l'*algorithme de Kruskal*, dont le fonctionnement est le suivant :

1. soit U une structure union-find pour les sommets $0, 1, \ldots, N - 1$ de G ;
2. soit Q une file de priorité contenant tous les arcs de G, ordonnés par leur poids ;
3. soit T une liste d'arcs, initialement vide ;
4. tant que T contient moins que $N - 1$ arcs :

(a) retirer un arc $x - y$ de poids minimal de la file de priorité Q,

(b) si x et y ne sont pas dans la même classe pour U, alors

  i. ajouter l'arc $x - y$ à T,

  ii. fusionner dans U les classes de x et y.

À la fin de l'algorithme, la liste T contient un arbre couvrant minimal pour G.

Question 4 Donner les étapes de l'algorithme de Kruskal sur le graphe de la question 3, sous la forme suivante :

étape	arc $x - y$	poids	action	U
				$\{0\},\{1\},\{2\},\{3\},\{4\},\{5\}$
1	...	...	...	...
2	...	...	...	...
⋮				

Chaque ligne indique l'arc $x - y$ retiré de la file, son poids, s'il est ignoré ou ajouté à T (action) et la structure U obtenue (ensemble des classes).

Programmation de l'algorithme de Kruskal. Pour programmer l'algorithme de Kruskal en Java, on va utiliser les classes `PriorityQueue` et `LinkedList` de la bibliothèque standard de Java (voir en annexe) pour réaliser Q et T, et la classe `UF` de la figure 2 pour réaliser U. On commence par la programmation de deux méthodes auxiliaires.

Question 5 Ajouter à la classe `Graph` une méthode `LinkedList<Edge> allEdges()` qui renvoie la liste de tous les arcs du graphe, dans un ordre arbitraire. Chaque arc doit apparaître *une seule fois* dans le résultat. Ainsi, si le graphe contient par exemple un arc $4-7$, alors la liste renvoyée doit contenir un objet `e` avec `e.src` $= 4$ et `e.dst` $= 7$, ou un objet `e` avec `e.src` $= 7$ et `e.dst` $= 4$, mais pas les deux.

Question 6 Pour construire une file de priorité de type `PriorityQueue<Edge>`, il faut que la classe `Edge` implémente l'interface `Comparable<Edge>`, avec une méthode de comparaison adéquate. Modifier la classe `Edge` en conséquence.

Question 7 Donner le code Java de l'algorithme de Kruskal sous la forme d'une méthode

```
static LinkedList<Edge> kruskal(Graph g)
```

Complexité et correction de l'algorithme de Kruskal.

Question 8 Quelle est la complexité de l'algorithme de Kruskal, en temps et en espace, dans le pire des cas, en fonction du nombre N de sommets ?

Question 9 Justifier que l'algorithme de Kruskal termine toujours et que l'étape 4a de l'algorithme n'échoue jamais sur une file vide.

Question 10 Justifier que l'algorithme de Kruskal renvoie bien un arbre couvrant minimal.

---

2 Recherche d'un mot dans un texte

Dans ce problème, on manipule des chaînes de caractères. Une chaîne S a une longueur notée $|S|$ (en Java, S.`length()`) et des caractères indexés par les entiers $0, 1, \ldots, |S|-1$. On note S_i le caractère d'indice i de la chaîne S (en Java, S.`charAt(`i`)`), pour $0 \leq i < |S|$. Pour $0 \leq i \leq j \leq |S|$, on note $S[i..j]$ la sous-chaîne de S constituée des caractères d'indices i inclus à j exclu (en Java, S.`substring(`i`,`j`)`).

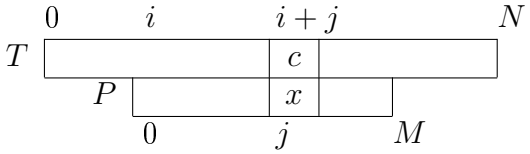
Dans tout ce problème, on s'intéresse à la recherche de toutes les occurrences d'une chaîne P (pour *pattern*), de longueur M, dans une chaîne T (pour *text*), de longueur N. Il y a une occurrence de P dans T à la position i lorsque $0 \leq i \leq N-M$ et $T[i..i+M] = P$. Nous allons étudier plusieurs algorithmes. À chaque fois, l'objectif est d'écrire une méthode qui affiche les positions de toutes les occurrences de P dans T. Par exemple, pour $P =$ `"bra"` et $T =$ `"abracadabra"`, la sortie du programme devra être la suivante :

```
occurrence à la position 1
occurrence à la position 8
```

Question 11 Écrire une méthode `static search(String p, String t)` la plus simple possible qui affiche toutes les occurrences de `p` dans `t`. Donner sa complexité en fonction de M et N.

Pré-traitement de P. Lorsque l'on cherche toujours la même chaîne P dans plusieurs textes, ou dans un texte très grand, il peut être avantageux d'effectuer un pré-traitement sur la chaîne P pour accélérer sa recherche. Un algorithme qui exploite cette idée est l'*algorithme de Boyer-Moore*. Les idées sont les suivantes :

- On va tester l'occurrence de P dans T à des positions i de plus en plus grande, en partant de $i = 0$.
- Pour une position i donnée, on va comparer les caractères de P et de T *de la droite vers la gauche*, c'est-à-dire en comparant d'abord P_{M-1} et T_{i+M-1}, puis P_{M-2} et T_{i+M-2}, etc.
- Si tous les caractères coïncident, on a trouvé une occurrence. Sinon, soit j l'indice de la première différence, c'est-à-dire le plus grand entier tel que $0 \leq j < M$ et $P_j \neq T_{i+j}$. Appelons c le caractère T_{i+j}.



L'idée de l'algorithme de Boyer-Moore consiste à augmenter alors la valeur de i de
- la grandeur $j-k$ où k est le plus grand entier tel que $0 \leq k < j$ et $P_k = c$, si un tel k existe (de manière à amener un caractère c sous le caractère T_{i+j}),
- la grandeur $j+1$ sinon.

Plutôt que de rechercher un tel k à chaque fois, on peut pré-calculer une *table de décalages* contenant, à la case indexée par l'entier j et le caractère c, le plus grand entier k tel que $0 \leq k < j$ et $P_k = c$ s'il existe, et rien sinon.

Question 12 Donner la table de décalages pour la chaîne $P =$ `"banane"`, sous la forme

	a	b	e	n
0				
1				
2				
3				
4				
5				

Certaines cases contiennent une valeur pour k, d'autres restent vides.

Mise en œuvre en Java. Pour mettre en œuvre l'algorithme de Boyer-Moore en Java, on se donne la classe suivante :

```
class BoyerMoore {
  String P;   // la chaîne que l'on cherche
  int    M;   // sa longueur
  HashMap<Character, Integer>[] table; // la table de décalages
```

La table de décalages est ici représentée par un tableau, de taille M, contenant des tables de hachage. Pour un j donné, avec $0 \leq j < M$, la table de hachage `table[`j`]` associe à un caractère c le plus grand k tel que $0 \leq k < j$ et $P_k = c$, s'il existe.

Question 13 Compléter le constructeur de la classe `BoyerMoore` pour qu'il remplisse la table de décalages.

```
  BoyerMoore(String P) {
    this.P = P;
    this.M = P.length();
    this.table = new HashMap<Character, Integer>[M];
    ...
  }
```

(En toute rigueur, il faudrait écrire `new HashMap[M]` car il s'agit d'un tableau de génériques, mais ce détail ne nous intéresse pas ici.)

Question 14 Dans la classe `BoyerMoore`, écrire une méthode `void search(String text)` qui implémente l'algorithme de Boyer-Moore et affiche toutes les occurrences de `this.P` dans `text` sous la forme demandée.

Question 15 Discuter la complexité en temps de l'algorithme de Boyer-Moore dans le cas où il n'y a aucune occurrence de P dans T. On essayera d'identifier un meilleur cas et un pire cas.

Question 16 Quelle est la place mémoire occupée par la table de décalages ? Là encore, on essayera d'identifier un meilleur cas et un pire cas.

Pré-traitement de T. Lorsque l'on cherche plusieurs chaînes dans un même texte T, il peut être avantageux d'effectuer au contraire un pré-traitement sur le texte T pour accélérer la recherche. Une solution consiste à construire un arbre de préfixes (*trie* en anglais, cf amphi 6) contenant tous les *suffixes* de la chaîne T. Pour représenter cet arbre, on se donne une classe analogue à celle vue en cours :

```
class Trie {
  int suffix; // -1 si pas un suffixe, son indice sinon
  HashMap<Character, Trie> branches;
  ...
}
```

La seule différence est qu'au lieu d'un booléen, pour indiquer la présence du mot, on utilise un entier (champ `suffix`). Si cet entier vaut -1, cela signifie que le mot correspondant à ce nœud de l'arbre n'est pas un suffixe de T. Sinon, c'est un entier i qui signifie que le mot est le suffixe $T[i..N]$, avec $0 \leq i \leq N$.

Question 17 Dessiner l'arbre des suffixes du mot $T =$ `"banane"`. (Les entiers $0, 1, \ldots, 6$ correspondant aux sept suffixes de T doivent tous apparaître dans l'arbre, chacun une seule fois.) Expliquer comment cet arbre permet de trouver facilement toutes les occurrences d'un mot donné dans T. L'illustrer avec le mot `"an"`.

Question 18 Écrire une méthode `void search(Trie st, String p)` qui affiche toutes les occurrences de `p` dans le texte dont l'arbre des suffixes `st` est passé en argument. (L'arbre `st` est supposé non `null`, car un arbre des suffixes n'est jamais vide.)

Question 19 Quelle est la complexité d'une recherche lorsque le mot P n'apparaît pas dans T ?

Question 20 Une construction de l'arbre des suffixes par insertion successive de tous les suffixes $T[i..N]$ prend clairement un temps $O(N^2)$. Qu'en est-il en revanche de l'espace mémoire occupé par l'arbre des suffixes au final ? On essayera d'identifier un meilleur cas et un pire cas.

A Bibliothèque standard Java

`class LinkedList<E>`	
`void add(E e)`	ajoute l'élément `e` à la fin de la liste
`int size()`	renvoie le nombre d'éléments de la liste

On peut parcourir tous les éléments d'une liste `l` avec la construction
`for (E x: l) ...`

`class PriorityQueue<E>`	une file de priorité dont les éléments sont de type `E`
`void add(E x)`	ajoute l'élément `x`
`void addAll(LinkedList<E> l)`	ajoute tous les éléments de `l`
`E remove()`	supprime et renvoie le plus petit élément de la file
`boolean isEmpty()`	renvoie `true` si et seulement si la file est vide

Note : la classe `E` doit implémenter l'interface `Comparable<E>`

`class String`	le type des chaînes de caractères
`int length()`	la longueur de la chaîne
`char charAt(int i)`	le caractère d'indice `i`, pour $0 \leq$ `i` $<$ `length()`
`String substring(int i, int j)`	la sous-chaîne constituée des caractères `i` inclus à `j` exclu

`class HashMap<K, V>`	un dictionnaire dont les clés sont de type `K` et les valeurs de type `V`
`void put(K k, V v)`	associe la valeur `v` à la clé `k` (en écrasant toute valeur précédemment associée à `k`, le cas échéant)
`V get(K k)`	renvoie la valeur associée à `k`, si elle existe, et `null` sinon
`Collection<V> values()`	renvoie l'ensemble de toutes les valeurs de la table

On peut parcourir toutes les valeurs d'un dictionnaire `d` avec la construction
`for (V v: d.values()) ...`

* *
*