École Normale Supérieure **2007-2008**
Département d'Informatique

Algorithmique et Programmation
Partiel

Notes de cours autorisées à l'exception de tout autre document

Exercice 1

Première Partie : structures de données pour le problème des ensembles disjointes.

Étant donné un ensemble, il est souvent utile de le partitionner en un certain nombre de sous-ensembles disjoints. Une structure de données pour le *problème des ensembles disjoints* est une structure de données qui maintient une telle partition. Un algorithme *union-find* est un algorithme qui fournit deux opérations essentielles sur une telle structure :
– Find : détermine l'ensemble contenant un élément. Notamment utile pour déterminer si deux éléments appartiennent au même ensemble.
– Union : réunit deux ensemble en un seul.
L'autre opération importante, Make, construit un ensemble contenant un unique élément (un singleton). À l'aide de ses trois opérations, beaucoup de problèmes de partitionnement peuvent être résolus.

Afin de définir ces opérations plus précisément, il faut choisir un moyen de représenter les ensembles. L'approche classique consiste à sélectionner un élément particulier, appelé le *représentant*, pour représenter l'ensemble toute entier. Dès lors, Find(x) renvoie le représentant de l'ensemble de x.

Nous ferons dépendre l'analyse du temps d'exécution des structures de données d'ensembles disjoints de deux paramètres : n, le nombre d'opérations Make et $m \geq n$ le nombre total d'opérations Make, Union et Find.

1. Proposer un algorithme simple utilisant une structure de données d'ensembles disjoints qui prenant en entrée un graphe non-orienté retourne la liste de ses composantes connexes.
2. Proposer une structure de données simple pour le problème des ensembles disjoints, utilisant des listes chaînées, pour laquelle les opérations Make et Find s'effectue en temps constant et l'opération Union en $O(m)$ opérations. Donner le temps d'exécution dans le pire des cas d'une séquence d'opérations Make et Union en fonction de m.
3. Montrer qu'en concaténant toujours la liste la plus courte à la plus longue, une séquence d'opérations prend un temps en $O(m + n \log n)$.

Pour une implantation plus efficace, nous proposons de représenter les ensembles par une famille d'arbres où chaque nœud contient un élément et chaque arbre représente un sous-ensemble. L'opération Make crée un arbre à un seul nœud, l'opérateur Find suit les pointeurs pères (notés $p[\cdot]$) jusqu'à rencontrer la racine de l'arbre et l'opération Union fait pointer la racine d'un arbre vers la racine de l'autre.

Nous définissons récursivement une fonction $F : \mathbb{N} \to \mathbb{N}$ par $F(0) = 1$ et $F(i+1) = 2^{F(i)}$ pour $i \in \mathbb{N}$ et nous utiliserons la notation $\log^*$ pour représenter le *logarithme itéré* que nous définissons comme l'inverse de F : $\log^*(n) = \min\{i \geq 0, F(i) \geq n\}$ pour tout $n \in \mathbb{N}$.

4. Dans chaque nœud, nous maintenons à jour un *rang* qui sera une approximation du logarithme de la taille du sous-arbre :

   Make(x)

   $p[x] := x$

   $rang[x] := 0$

   Find(x)

   `si` $x = p[x]$ `alors retourner` $p[x]$

   `sinon retourner` Find$(p[x])$

   Union(x,y)

   $x' :=$ Find(x) ; $y' :=$ Find(y)

   `si` $rang[x'] > rang[y']$

   `alors` $p[y'] := x'$

   `sinon` $p[x'] := y'$

   `si` $rang[x'] = rang[y']$

   `alors` $rang[y'] := rang[y'] + 1$

   Donner la complexité, en fonction de m et de n, d'une séquence d'opérations Make, Union et Find utilisant cette implantation.

5. Désormais, dans l'opération Find, nous ferons pointer chaque nœud de la route directe sur la racine de l'arbre.

   Find(x)

   `si` $x \neq p[x]$ `alors` $p[x] :=$ Find$(p[x])$

   `retourner` $p[x]$

   Le but de cette question est de montrer qu'une séquence de m opérations Make, Union et Find parmi lesquelles n sont des opérations Make s'exécute en temps $O(m \log^* n)$ dans le pire des cas.

   (a) En notant $taille[x]$ le nombre de nœuds de l'arbre enraciné en x, y compris le nœud x lui même, montrer que $taille[x] \geq 2^{rang[x]}$.

   (b) Montrer que pour tout entier $r \geq 0$, il existe au plus $n/2^r$ nœuds de rang r.

   (c) Nous partitionnons les rangs des nœuds en plusieurs *blocs* : le rang r se trouvant dans le bloc $\log^*(r)$ pour tout $r \in \{0, \ldots, \lfloor \log n \rfloor\}$ et nous notons $b(v)$ le bloc correspondant au rang d'un nœud v.

   Pour la i-ème exécution de l'opération Find, notons X_i l'ensemble des nœuds traversés et

$$\begin{aligned} W_i &= \{v \in X_i \mid v \text{ est la racine ou un fils de la racine}\} \\ Y_i &= \{v \in X_i \setminus W_i \mid b(v) < b(p[v])\} \\ Z_i &= \{v \in X_i \setminus X_i \mid b(v) = b(p[v])\} \end{aligned}$$

   i. Montrer qu'il y a au plus $2n/F(t)$ nœuds de rang t.

   ii. Montrer que

$$\sum_i \#Z_i \leq 2n(1 + \log^*(n))$$

   iii. En déduire que

$$\sum_i \#X_i = O(m \log^*(n))$$

   (d) Conclure

Deuxième partie : arbres couvrants minimaux et algorithme de Kruskal.

Étant donné un graphe non orienté et connexe, un *arbre couvrant* de ce graphe est un sous-ensemble qui est un arbre et qui connecte tous les sommets ensemble.Un *arbre couvrant de poids minimal* d'un graphe valué est un arbre couvrant dont le poids est le minimum parmi tous les arbres couvrants du graphe.

Soit $A \subseteq E$ un sous-ensemble d'arêtes incluses dans un arbre couvrant minimal de G. Une arête (u, v) est dite *sûre* pour A si l'ensemble $A \cup \{(u, v)\}$ est inclus dans un arbre couvrant minimal de G. Nous avons alors l'algorithme générique suivant de calcul d'un arbre couvrant minimal :

$A := \emptyset$
`tant que` A `ne forme pas un arbre couvrant minimal`
  `faire choisir une arête sûre` (u, v) `pour` A
    $A = A \cup \{(u, v)\}$
`retourner` A

6. Proposer un invariant pour cet algorithme et prouver sa terminaison.
7. Une *coupe* est une partition $(S, V \setminus E)$ de V. Une arête *traverse* une coupe $(S, V \setminus S)$ si elle part de S et arrive en $V \setminus S$. Une coupe respecte un sous-ensemble A d'arêtes s'il n'y a pas d'arête de A qui la traverse. Une arête est *claire* si elle traverse la coupe et est de poids minimal.
   Soit un sous-ensemble A de E qui est inclus dans un arbre couvrant minimal, soit $(S, V \setminus S)$ une coupe qui respecte A et soit (u, v) une arête claire pour la coupe. Montrer que (u, v) est sûre pour A.
8. Soit un sous-ensemble A de E qui est inclus dans un arbre couvrant minimal, et soit $C = (V_C, E_C)$ une composante connexe dans la forêt $G_A = (V, A)$. Montrer que si (u, v) est une arête claire reliant C à une autre composante connexe de G_A, alors (u, v) est sûre pour A.
9. Soit (u, v) une arête de poids minimal dans un graphe G. Montrer que (u, v) appartient à un arbre couvrant de G.
   Étant donné un graphe G et un arbre couvrant minimal T pour G, peut-il exister une arête (u, v) de G de poids minimal et qui n'appartient pas à T ?
10. En 1956, Krusal a proposé l'algorithme suivant :

    $E := \emptyset$
    `pour chaque sommet` v `de` G
      `faire` Make(v)
    `trier les arêtes de` G `par ordre croissant de poids`
    `pour chaque arête` (u, v) `de` G `prise par ordre de poids croissant`
      `faire si` Find(u) $\neq$ Find(v)
        `alors ajouter l'arête` (u, v) `à l'ensemble` E
          Union(u, v)
    `retourner` E

    Prouver la validité de l'algorithme de Kruskal et donner sa complexité.
11. Supposons que dans un graphe valué non orienté $G = (V, E)$, tous les poids d'arêtes sont des entiers compris entre 1 et $|V|$. Quel est alors le temps d'exécution de l'algorithme de Kruskal ? Que se passe-t-il si les poids sont des entiers bornés par une constante ?

Troisième partie : le problème du voyageur de commerce.

Le problème du voyageur de commerce consiste en la donnée d'un graphe complet non-orienté à n sommets, numérotés de 1 à n (et donc $n(n-1)/2$ arêtes) ; à chaque arête est associée une distance. Le voyageur de commerce réside au sommet 1 du graphe, et cherche à organiser sa tournée de façon à partir de 1, revenir en 1, passer une fois et une seule par chacun des autres sommets du graphe, et ce en parcourant la distance minimale.

12. Quelle est le nombre d'étapes de calcul à effectuer pour une énumération exhaustive des tournées possibles ?
13. Soit $S \subseteq \{2, 3, \ldots, n\}$ et $k \in S$. Soit $C(S, k)$ la distance minimale pour aller de 1 à k en passant une fois et une seule par tous les sommets de S. Exprimer $C(S, k)$ en fonction des $C(S \setminus \{k\}, \ell)$ pour $\ell \in S \setminus \{k\}$. Quelle technique utiliser pour exploiter cette relation ? Écrire le pseudo-code correspondant. Quel est le nombre d'étapes de calcul par cette méthode ?
14. Nous supposons désormais que la distance vérifie l'inégalité triangulaire : pour tous sommets u, v et w, $d(u, v) \leq d(u, w) + d(w, v)$. Considérons l'algorithme suivant :

    ```
    calculer un arbre couvrant minimal T enraciné en 1.
    retourner le trajet obtenu en regardant l'ordre dans lequel
    les sommets apparaissent lors d'un parcours en profondeur de l'arbre T.
    ```

    Soit H le trajet ainsi obtenu.
    (a) Quelle est la complexité de cet algorithme (en utilisant l'algorithme de Kruskal pour la première étape) ?
    (b) Soit H^* une tournée optimale. Montrer que le poids $c(T)$ de T est inférieur ou égal à la longueur totale de H^*.
    (c) Montrer que la longueur $c(H)$ de H est inférieure ou égale à $2c(T)$ et en déduire que l'algorithme résoud le problème du voyageur de commerce avec inégalité triangulaire à un facteur 2.
    (d) Montrer que pour tout $\varepsilon > 0$, il existe des cas où l'algorithme construit un tour de coût supérieure à $(2 - \varepsilon)c(H^*)$ et donc que l'analyse ne peut pas être améliorée.

Exercice 2 – Arbres Binaires Splay

Le but de cet exercice est d'étudier un type d'arbre binaire de recherche qui a de bonne propriété amortie en autorisant des modifications de l'arbre aussi pendant la recherche d'un élément. Les éléments qui viennent d'être recherchés sont placés à la racine de l'arbre et donc les éléments qui sont souvent recherchés se trouvent proches de la racine. On va étudier la complexité amortie de l'opération *splay* qui est utilisée dans toutes les autres opérations (*insert*, *join*, *delete*, ...).

Notations

On pourra écrire par exemple $p(x)$ pour représenter le père du nœud x et $g(x)$ pour le grand père de x. La racine de l'arbre est l'unique élément x qui n'a pas de père. Une *feuille* est un nœud x qui n'a pas de fils gauche et droit.

On suppose que les clés $u = key(x)$ des divers nœuds x sont distincts et appartiennent à un ensemble ordonné U, fixé dans la suite.

Soit x un nœud de T et $u = key(x)$. On définit une opération $splay(x, T)$ qui place x à la racine de T, en appliquant de manière récursive l'algorithme suivant tant que x n'est pas racine de l'arbre :

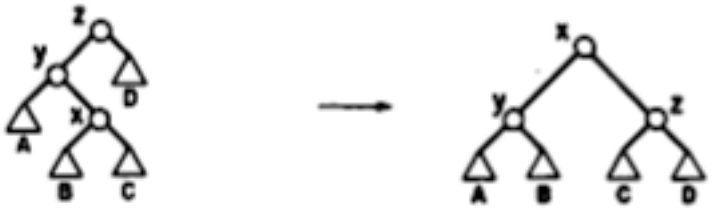
Cas 1. Si $p(x)$, le père de x, est la racine de l'arbre, on effectue une rotation de l'arête qui joint x à $p(x)$. (Ce cas est terminal.)

(a)

Cas 2. Si $p(x)$ n'est pas la racine et x et $p(x)$ sont tous deux fils gauche ou droit, on effectue une rotation des arêtes joignant le père $p(x)$ avec le grand père $g(x)$ et ensuite une rotation de l'arête joignant x avec son père $p(x)$.

(b)

Cas 3. Si $p(x)$ n'est pas la racine et x est un fils gauche et $p(x)$ un fils droit, ou vice-versa, on effectue une rotation de l'arête joignant x avec $p(x)$ et ensuite une rotation de l'arête joignant x avec son nouveau père $p(x)$.



(c)

1) Donner le résultat après une opération *splay* au nœud a sur les 3 arbres suivants :

ABRS 1 ABRS 2 ABRS 3

Ces trois cas correspondent au pire situation pour les opérations. Remarquer que l'opération l'opération *splay* non seulement amène a à la racine, mais aussi divise environ par 2 la profondeur de tous les nœuds sur le chemin de a à la racine.

2) On va étudier la complexité amortie de l'opération *splay* en utilisant la technique du potentiel. L'idée est d'attribuer à chaque configuration possible de la structure de donnée un nombre réel, appelé son *potentiel* et de définir *le temps amorti* a d'une opération par $a = t + \Phi' - \Phi$, où t est le temps de l'opération à un instant donné, Φ est le potentiel avant l'opération, et Φ' le potentiel après l'opération. Avec ces estimation, déterminer le temps total d'une suite de m opérations en fonction des a_j pour $j = 1, \ldots, m$, Φ_0 et Φ_m ainsi que le temps amorti.

3) Pour définir le potentiel d'un ABRS, on suppose que chaque clé u a un poids $w(u)$ dont la valeur est fixée. On définit la taille $s(x)$ d'un nœud x de l'arbre comme la somme des poids individuel de toutes les clés dans le sous-arbre de racine x. On définit aussi le rang $r(x)$ du nœud x comme le logarithme en base 2 de x, $\log s(x)$. Finalement, on définit le potentiel de l'arbre comme la somme des rangs de tous les nœuds. Comme une mesure du temps d'exécution d'une opération *splay*, on utilise le nombre de rotations effectuées. S'il n'y a pas de rotations, on compte 1 pour une opération *splay*.

On note s, s', r et r' les fonctions taille et le rang des nœuds juste avant et après une opération *splay*. On pourra désigner par y le père de x et par z le grand-père de x.

a) Montrer que dans le cas le cas 1) le temps amorti d'une opération splay est $1+r'(x)+r'(y)-r(x)-r(y)$ et dans les cas 2) et 3), le temps amorti est $2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$.

b) Montrer que dans le cas 1), le temps amorti est au plus $3(r'(x) - r(x)) + 1$.

c) Montrer que dans le cas 2), le temps amorti est majoré par $2 + r'(x) + r'(z) - 2r(x)$. Montrer que a fonction $\log x + \log y$ pour $x, y > 0$ et $x + y \leq 1$ est maximisée quand $x = y = 1/2$ et vaut -2. En déduire que le temps amorti dans le cas 2) est majoré par $3(r'(x) - r(x))$ en utilisant le fait que $s(x) + s'(z) \leq s'(x)$.

d) Montrer que dans le cas 3), le temps amorti est majoré par $2 + r'(y) + r'(z) - 2r(x)$. En déduire que le temps amorti dans le cas 3) est majoré par $3(r'(x) - r(x))$.

e) En conclure, alors que le temps amorti d'une opération *splay* sur l'arbre enraciné en t au nœud x est au plus $3(r(t) - r(x)) + 1$ et que ce nombre est $O(\log(s(t)/s(x)))$.

f) Montrer que si on assigne un poids $1/n$ à chacun des n nœuds d'un arbre, alors le temps total pour m opérations *splay* est $O((m + n) \log n + m)$.