ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
Section d'Informatique et de Systèmes de Communication

Corrigé de la série 5 24 Oct. 2011

1. Algorithmes récursifs : Le diamètre d'un arbre binaire

a) Notons r la racine de T. Pour un chemin dans T il y a trois possibilités :

(1) Le chemin se trouve entièrement dans T_1,

(2) Le chemin se trouve entièrement dans T_2,

(3) Le chemin passe par (ou se termine à) la racine r.

Pour pouvoir dire quelque chose sur le diamètre de T, nous considérons un chemin γ de longueur maximale. Si γ est du premier type, alors la longueur de γ est certainement égale à $\text{diam}(T_1)$, et si γ est du deuxième type, sa longueur est égale à $\text{diam}(T_2)$.
Le troisième cas nécessite une discussion un peu plus détaillée. Dénotons le dernier élément du chemin dans T_1 par x_1 et le dernier élément dans T_2 par x_2. Le chemin est alors comme suit :

$$x_1 \to \cdots \to \text{ racine de } T_1 \ \to \ \text{racine de } T \ \to \ \text{racine de } T_2 \ \to \cdots \to x_2.$$

La longueur de ce chemin est donc

$$\text{profondeur}_{T_1}(x_1) + 1 + 1 + \text{profondeur}_{T_2}(x_2),$$

où $\text{profondeur}_T(x)$ dénote la profondeur du sommet x dans l'arbre T (voir p.56 du cours pour voir la définition de profondeur).
La maximalité du chemin implique maintenant que les deux termes $\text{profondeur}_{T_1}(x_1)$ et $\text{profondeur}_{T_2}(x_2)$ doivent être maximaux, i.e. que les sommets en question se trouvent en tout en bas de l'arbre et donc que la profondeur des sommets en question est égale à la hauteur du sous-arbre correspondant. En résumé donc, la longueur du chemin est, dans ce cas, égale à

$$h_1 + 2 + h_2,$$

où h_i dénote la hauteur de l'arbre T_i.
Nous déduisons donc que la longueur du plus long chemin dans T (et alors le diamètre de T) est égale à

$$\max\{\text{diam}(T_1), \text{diam}(T_2), h_1 + h_2 + 2\}.$$

Soit h la hauteur de T. Il est alors assez clair que

$$h = \max(h_1, h_2) + 1.$$

b) Nous construisons une fonction récursive qui calcule à la fois la hauteur et le diamètre d'un arbre binaire étant donné sa racine :

Call : HauteurEtDiam
Input: r : Racine d'un arbre binaire.

Output: (h, d) : Hauteur et diamètre de l'arbre.
  if $r = 0$ **then**
    return $(-1, -1)$
  $(h_1, d_1) \leftarrow$ HauteurEtDiam(r[`left`])
  $(h_2, d_2) \leftarrow$ HauteurEtDiam(r[`right`])
  $h \leftarrow \max\{h_1, h_2\} + 1$
  $d \leftarrow \max\{d_1, d_2, h_1 + h_2 + 2\}$
  return (h, d)

Remarquons que cet algorithme a bien un temps de parcours linéaire : Si nous ignorons pour un instant les appels récursifs, nous voyons que, puisqu'il n'y a pas de boucles, l'algorithme opère en temps constant. Mais l'appel de la fonction HauteurEtDiam se fait exactement une fois pour chaque sommet ; il en résulte que l'algorithme est $O(|V|)$.

2. Diviser pour régner : Le problème du skyline

a) Nous parcourons l'axe des x de gauche à droite, en regardant les points un par un. Pour chaque point de L et de R nous devons décider s'il faut ajouter un point C, et si oui quelle va être sa hauteur.

Un algorithme est donnée ci-dessous. On remarque que le nombre d'opérations est $O(|L| + |R|)$ où $|L|$ et $|R|$ représentent les nombres de points dans les skylines de L et R.

Dans l'algorithme ci-dessous, les skylines sont données sous forme de tableau, avec les entrées étant des couples (x, h). Par convention, la dernière entrée est toujours $(\infty, 0)$ et marque la fin de la skyline.

Call : SkylineMerge
Input: Skylines L et R
Output: Skyline ret, superposition des deux skylines.
  $i_\ell, i_r \leftarrow 0$
  $h_\ell \leftarrow 0, h_r \leftarrow 0, h \leftarrow 0$
  while $\min(L[i_\ell].x, R[i_r].x) < \infty$ **do**
    if $L[i_\ell].x < R[i_r].x$ **then**
      $h_\ell \leftarrow L[i_\ell].h$
      $x \leftarrow L[i_\ell].x$
      $i_\ell \leftarrow i_\ell + 1$
    else if $L[i_\ell].x = R[i_r].x$ **then**
      $h_\ell \leftarrow L[i_\ell].h$
      $h_r \leftarrow R[i_r].h$
      $x \leftarrow L[i_\ell].x$
      $i_\ell \leftarrow i_\ell + 1$
      $i_r \leftarrow i_r + 1$
    else
      $h_r \leftarrow R[i_r].h$
      $x \leftarrow R[i_r].x$
      $i_r \leftarrow i_r + 1$
    if $\max(h_\ell, h_r) \neq h$ **then**
      $h \leftarrow \max(h_\ell, h_r)$
      ret.add((x, h))

ret.add($(\infty, 0)$)
return ret

Dans le pseudocode ci-dessus, la méthode add ajoute un élément à la fin d'un array.

b) Remarquons d'abord que l'algorithme SkylineMerge ne résout pas le problème Skyline. En effet, Skyline attend comme input un ensemble de bâtiments (i.e. de triplets (x_1, x_2, h)) et retourne comme output le skyline correspondant. Par contre, SkylineMerge a comme input deux skylines et retourne le skyline obtenu en les combinant.

MergeSort : Notre solution est très similaire à *MergeSort*, nous rappelons donc les détails de cet algorithme. Le problème à résoudre est d'ordonner une suite de nombres naturels (l'input est donc une suite de nombres naturels, et l'output une suite qui contient les mêmes éléments, mais ordonnés). Nous avons d'abord construit un algorithme Merge qui, étant donné deux suites S_1 et S_2 de nombres naturels *ordonnées*, retourne une suite ordonnée contenant les éléments de S_1 et S_2.

L'algorithme MergeSort divise la suite donnée en 2 sous-suites (plus petites), qui sont ensuite ordonnées (avec un appel récursif à MergeSort). Puis, ces 2 sous-suites ordonnées sont combinées avec l'algorithme Merge pour obtenir une version ordonnée de la suite de l'input.

Puisque nous avons fait un appel récursif à MergeSort, chacune des sous-suites sera elle même divisée en deux sous-suites, etc. Il faut donc définir un cas de base où la récurrence se terminera. Il s'agit du cas où la suite ne contient qu'un seul élément et est donc forcément déjà ordonnée.

On l'appelle un algorithme diviser-pour-régner puisqu'on divise le problème en deux instances plus petites du même problème. l'algorithme Merge est utilisé pour recoller les solutions de ces instances pour obtenir la solution du problème original.

Pour résoudre le problème Skyline nous procédons de la même façon. Notre algorithme (que nous appelons SkylineSolve) commence par diviser la liste de bâtiments en deux sous-listes (dont la taille vaut à peu près la moitié de celle de la suite originale). Nous trouvons ensuite le Skyline de chacune de ces suites avec un appel récursif à SkylineSolve.

Nous réduisons donc notre problème à deux sous-problèmes qui sont toujours des instances de Skyline, mais avec un input plus petit (il y a moins de bâtiments).

Les solutions de ces deux sous-problèmes (les deux skylines) sont ensuite combinées avec l'algorithme SkylineMerge donné dans la partie a) (ce pas correspond donc à Merge).

Comme avec MergeSort, puisqu'il y a cet appel récursif, il faut définir un cas de base. C'est le cas où il n'y a qu'un seul bâtiment, et le skyline est égal a ce bâtiment.

On obtient au final, en pseudocode :

Call : SkylineSolve
Input: Bat : Une liste de n bâtiments (de triplets (x_1, x_2, h))
Output: A : Le skyline correspondant (une suite de points (x, h))
  if $n = 1$ **then**
    return $((Bat[1].x_1, Bat[1].h), (Bat[1].x_2, 0))$
  else

$m \leftarrow \lceil n/2 \rceil$
$Bat_1 \leftarrow$ la sous-suite de Bat composée des $m-1$ premiers éléments de Bat
$Bat_2 \leftarrow$ la sous-suite de Bat composée des autres éléments (bâtiments) de Bat
$Skyline_1 \leftarrow$ SkylineSolve(Bat_1)
$Skyline_2 \leftarrow$ SkylineSolve(Bat_2)
return SkylineMerge($Skyline_1, Skyline_2$)

3. Algorithmes par induction : Le problème de la célébrité

On précise que dans nos graphes non-orientés on suppose qu'un sommet n'est jamais connecté à lui-même, c'est-à-dire que la diagonale de la matrice d'adjacence ne contient que des 0.

a) Le sommet 2 est une célébrité puisqu'il est connu de tout le monde mais ne connait personne (son in-degree vaut $n-1$ alors que son out-degree vaut 0).

b) Supposons que nous avons un graphe avec deux célébrités (que nous appelons les sommets i et j). i ne connait personne, donc en particulier i ne connait pas j. j ne peut donc pas être une célébrité et nous avons une contradiction. Un graphe ne peut donc pas avoir deux célébrités.

c) On rappelle que pour tout graphe orienté, avec comme matrice d'adjacence A, A_{ij} vaut 1 s'il y a un arc du sommet i au sommet j, et 0 sinon. Ainsi la ligne k montre les arcs qui sortent du sommet k, et la colonne k ceux qui vont vers k.

Si le sommet k est une célébrité, la ligne k ne contient donc que des 0. De même, la colonne k ne contient que des 1, sauf sur la diagonale (l'entrée A_{kk}) qui vaut 0 (puisque nous supposons qu'il n'y a jamais d'arc d'un sommet à lui-même).

d) L'algorithme naïf serait de prendre chaque sommet un par un et de regarder s'il s'agit d'une célébrité. Etant donné un sommet k, pour déterminer si c'est une célébrité nous devons :
   - (1) vérifier que tous les autres sommets connaisent k
   - (2) vérifier qu'aucun autre sommet n'est connu de k

   Pour (1) nous devons vérifier que $A_{ik} = 1$ pour tout $i \in \{1, \ldots, n\} \backslash \{k\}$ *($n-1$ questions).*
   Pour (2) nous devons véfifier que $A_{kj} = 0$ pour tout $j \in \{1, \ldots, n\} \backslash \{k\}$ *($n-1$ questions).*
   Pour vérifier si k est une célébrité il nous faut donc $2 \cdot (n-1)$ questions.

   Puisque nous devons répèter cette procedure pour tous les $k \in \{1, \ldots, n\}$ on obtient un total de $2 \cdot n \cdot (n-1)$ questions dans le pire des cas, donc un algorithme $O(n^2)$.

e) Rappelons qu'on sait qu'il existe une célébrité dans le graphe qui nous est donné.
   - *(i)* Si on a deux sommets a et b on peut poser la question "a connait-il b ?". Si la réponse est "oui" on sait que a ne peut pas être une célébrité, et puisque l'un des deux doit être une célébrité, il s'agit forcément de b. De même, si la réponse est "non", on sait que b ne peut pas être une célébrité, et donc la célébrité est a.
   - *(ii)* Nous supposons qu'étant donné un groupe de n personnes contenant une célébrité, nous pouvons la retrouver en posant $T(n)$ questions.
     Etant donné un groupe de $n+1$ personnes nous procédons comme suit :
     - nous choisissons deux personnes a et b parmi les $n+1$.

- nous posons la question "a connait-il b ?"
- Si la réponse est "oui" nous savons que a n'est pas la célébrité. Celle-ci se trouve donc parmi les n autres personnes (dont b). Nous pouvons donc la retrouver en posant $T(n)$ questions. Nous avons donc dû poser $T(n)+1$ questions au total.
- Si la réponse est "non" nous savons que b n'est pas la célébrité. Nous pouvons donc de la même façon la retrouver parmi les n personnes restantes en posant $T(n)$ questions. Nous avons donc dû poser $T(n)+1$ questions au total.

– *(iii)* L'algorithme récursif a été décrit dans la partie (ii). En pseudo-code on obtient :

Call : FindCelebrity
Input: $S = \{s_1, \ldots, s_n\}$: Un ensemble de n personnes, dont une célébrité
Output: c : la célébrité
  if $n = 2$ **then**
    if s_1 connait s_2 **then**
      return s_2
    else
      return s_1
  if s_1 connait s_2 **then**
    return FindCelebrity$(S \backslash \{s_1\})$
  else
    return FindCelebrity$(S \backslash \{s_2\})$

Si on appelle $T(n)$ le nombre de questions dont il a besoin pour trouver la célébrité parmi n personnes on a donc $T(2) = 1$ et $T(n) = T(n-1) + 1$ pour $n \geq 2$. Il est assez clair que cela nous donne

$$T(n) = n - 1$$

En effet par récurrence on voit que l'égalité est vraie pour $n = 2$ (base), et si elle est vraie pour n alors

$$T(n+1) = T(n) + 1 = (n-1) + 1 = n$$

Ainsi c'est un algorithme $O(n)$.