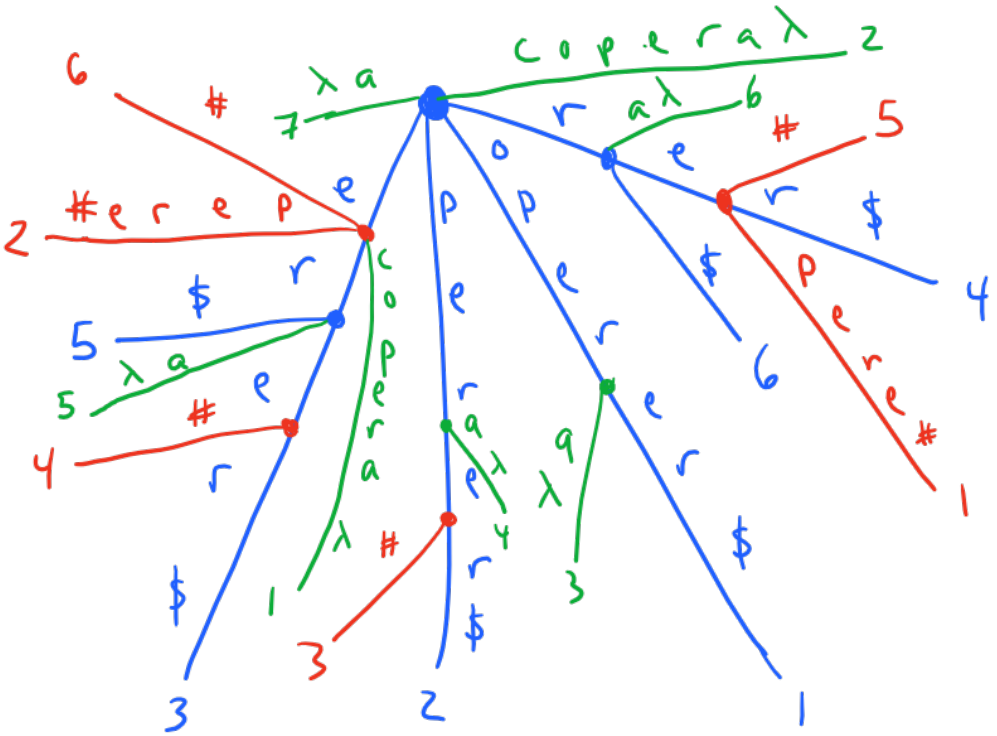
BIN702 - Exercices sur les arbres de suffixe

Manuel Lafond

Exercice 1: Construisez l'arbre de suffixe généralisé pour les trois chaînes $operer, repere, ecopera$.

Solution. On doit déterminer trois symbols de terminaison différents. Ici, j'utilise $\$, \#$ et λ.



□

Exercice 2: On a mentionné sans démonstration que puisqu'un arbre de suffixe A possède n feuilles et que chaque noeud interne a au moins 2 enfants,

alors A possède $O(n)$ noeuds au total. En fait, on peut montrer que A a au maximum $2n-1$ noeuds et au maximum $2n-2$ arêtes. Pouvez-vous le démontrer?

Solution. Vous pouvez faire une démonstration formelle par induction sur le nombre de feuilles n. Comme cas de base, si $n=1$, alors A possède $2n-1=1$ noeuds et $2n-2=0$ arêtes. Pour l'induction, soit A un arbre à n feuilles enraciné en un noeud r, et soient $r_1,\ldots,r_k$ les enfants de r. De plus, soient $n_1,\ldots,n_k$ les nombres de feuilles descendant de $r_1,\ldots,r_k$, respectivement. Par induction, ces arbres somment pour au plus $\sum_{i=1}^{k}(2n_i-1)$ noeuds et au plus $\sum_{i=1}^{k}(2n_i-2)$ arêtes. Puisque $\sum_{i=1}^{k} n_i = n$, le nombre de noeuds est donc au plus, en comptant r, $1+\left(\sum_{i=1}^{k} 2n_i\right)-\sum_{i=1}^{k} 1 = 1+2n-k \leq 2n-1$ car $k \geq 2$. De la même façon, le nombre d'arêtes est au plus $2n-2k+k \leq 2n-2$ car $k \geq 2$. □

Exercice 3: Il est possible que deux séquences S et T aient plusieurs sous-chaînes communes de taille maximum. Donnez un algorithme qui retourne le **nombre** de plus longues sous-chaînes communes entre S et T. Quelle est la meilleure complexité que vous pouvez atteindre?

Solution. Ceci peut se réaliser en suivant les étapes suivantes.

a. Construire l'arbre de suffixe généralisé A pour S et T.

b. Parcourir A de bas en haut. À chaque noeud v, retenir si v a une feuille descendante de S et/ou T.

c. Trouver la profondeur en caractère maximum p d'un noeud ayant une feuille descendante de S *et* de T.

d. Parcourir l'arbre pour compter le nombre de noeuds qui ont une profondeur p ayant une feuille descendante de S et T.

Ceci compte le nombre de chaînes distinctes de longueur p qui sont présentes dans S et T. On aurait pu intrepréter la question comme demandant le nombre de paires de positions où démarrent des sous-chaînes communes. Dans ce cas, il faudrait, pour chaque noeud v trouvé à la dernière étape, retourner le nombre de feuilles descendantes à v dans S, fois le nombre de feuilles descendantes à v dans T. J'accepterais les deux types de réponses car la question était un peu ambigue.

Chaque étape peut se faire en temps linéaire, le temps est donc $O(n)$. Ceci serait suffisant comme réponse à l'examen. Ci-bas, je vous propose un pseudo-code qui fait les 3 dernières tâches à la fois. C'est un peu laborieux, mais l'idée est que chaque noeud retourne à son parent s'il a des feuilles de S ($hasS$), des feuilles de T ($hasT$) et, s'il en a des deux, retourne la profondeur maximum d'un de ses descendants ayant les deux et le nombre de tels descendants. Il faudrait faire un appel initial avec $nbCommonSubstr(racine, 0)$.

fonction *nbCommonSubstr(u, curprof)*
  //Retourne $(maxProf, nbCommun, hasS, hasT)$;
  si *u est une feuille* **alors**
    si *u est une position de S* **alors**
      return $(0, 0, True, False)$;
    sinon
      return $(0, 0, False, True)$;
  fin
  $curMaxProf = 0$, $curNbCommun = 0$, $hasS = False$,
   $hasT = False$;
  pour chaque *enfant v de u* **faire**
    //uv.longueur est le nombre de caractères sur l'arête uv;
    $(vmaxProf, vcommun, vhasS, vhasT) =$
     $nbCommonSubstr(v, curProf + uv.longueur)$;
    si $vhasS$ *et* $vhasT$ **alors**
      si $vmaxProf > curMaxProf$ **alors**
        $curMaxProf = vmaxProf$;
        $curNbCommun = vcommun$;
      sinon si $vmaxProf = curMaxProf$ **alors**
        $curNbCommun = curNbCommun + vcommun$;
    fin
    $hasS = hasS \vee vhasS$; `// ∨ est le ‘‘ou’’ logique`
    $hasT = hasT \vee vhasT$;
  fin
  si $curMaxProf > 0$ **alors**
    return $(curMaxProf, curNbCommun, True, True)$;
  sinon si $hasS$ *et* $hasT$ **alors**
    //On entre ici quand u est le premier ayant S et T;
    return $(curProf, 1, True, True)$;
  sinon
    return $(curProf, 0, hasS, hasT)$;

□

Exercice 4: Supposez que vous avez l'arbuste de suffixes A pour une séquence S. Donnez un algorithme pour transformer A en l'arbre de suffixes de S. Le temps devrait être proportionnel au nombre de noeuds de A.

Solution. Une façon est de parcourir l'arbre de haut en bas. Pour chaque noeud u avec au moins 2 enfants et chaque enfant v de u, on regarde s'il y a un chemin à comprimer à partir de v (ceci survient quand v a un seul enfant). Si oui, on trouve le bout $curv$ de ce chemin et on le comprime. Notez que $curv$ est soit le premier noeud qui est un branchement ou une feuille.
Une difficulté est savoir quelles positions i, j font que $S[i..j]$ correspond à l'étiquette de la branche résultant de la compression. Pour ce faire, on fait un prétraitement dans lequel on étiquette chaque branche avec une position de S qui contient le caractère de sa branche parente.
Notez que la racine a toujours au moins 2 enfants à cause du caractère de terminaison.

fonction *pretraitementPositions(u)*
  //l'appel initial se fait sur chaque enfant de la racine;
  Soit p le parent de u;
  si *u est une feuille* **alors**
    $(p, u).etiquette = u.positionSuffixe$;
  sinon
    Soit v n'importe quel enfant de u;
    $(p, u).etiquette = (u, v).etiquette - 1$;

```
fonction compressArbuste(u)
    //on suppose que u n'a pas un seul enfant;
    //l'appel initial se fait sur u = racine;
    pour chaque enfant v de u faire
        si v a un seul enfant alors
            curv = v, curparent = u;
            tant que curv a 1 enfant faire
                curparent = curv;
                curv = curv.enfant;
            fin
            //À ce point-ci, curv est un branchement ou une feuille;
            //On lui demande de comprimer ses descendants;
            compressArbuste(curv);
            [pos1, pos2] = [(u, v).etiquette, (curparent, curv).etiquette];
            Supprimer tous les noeuds de v à curv (excluant curv);
            Ajouter l'arête (u, curv);
            (u, curv).etiquette = [pos1, pos2];
        sinon
            compressArbuste(v);
    fin
```

Soit n le nombre de noeuds de l'arbuste. Le prétraitement parcours chaque noeud une seule fois et est donc en temps $O(n)$. La fonction *compressArbuste* est appelée une fois par noeud qui n'a pas 1 seul enfant. Pour chaque appel, on doit parcourir les chemins démarrant à un enfant de u. On fait chaque parcours deux fois, une fois pour trouver le bout $curv$ et une autre fois pour supprimer les sommets du chemin. Au final, chaque noeud de chaque chemin est parcouru $O(1)$ fois. On peut donc associer à chaque noeud un coût en temps de $O(1)$, et la complexité résultante est $O(n)$. □

Exercice 5: Montrez comment calculer la plus longue sous-chaîne commune entre k séquences $S_1, S_2, \ldots, S_k$ en temps $O(kn)$, où n est la somme des longueurs des séquences données.
Si vous aimez les défis, faites-le en temps : $O(n)$. (mais attention ce n'est vraiment pas facile, vous pourriez y passer des heures)

Solution. Dans la version facile, on construit l'arbre de suffixe généralisé pour $S_1, \ldots, S_k$, puis de bas en haut, on étiquette chaque noeud interne avec la liste des S_i qui sont représentés parmi les descendants. Cette étiquettage peut se

faire avec un parcours post-ordre : l'étiquette est l'union des étiquettes des deux enfants. La plus longue chaîne commune est celle de profondeur maximum contenant tous les S_i comme étiquette. Puisque chaque noeud interne reçoit $O(k)$ étiquettes et qu'il y a $O(n)$ noeuds, ceci prend un temps $O(nk)$.
La version difficile est longue à expliquer. Je vous renvoie à `https://web.cs.ucdavis.edu/~gusfield/cs224f09/commonsubstrings.pdf`, car je ne pourrais pas l'expliquer d'une meilleure façon. □

Exercice 6: Un palindrome-complément est une séquence S de taille $2n$ telle que $S[1..n]$ est égale à $S[n+1..2n]$ lue de droite à gauche après avoir fait les échanges de caractères $A \leftrightarrow T$ et $C \leftrightarrow G$. Par exemple, $AACTGCAGTT$.
Montrez comment trouver tous les palindrome-compléments maximaux dans une séquence T.

Solution. Notons qu'un palindrome-complément est nécessairement pair. Il suffit de complémenter S et de la renverser, obtenant ainsi une séquence qu'on va appeler S'. C'est-à-dire, on obtient S' en changeant A en T, T en A, C en G, et G en C, puis on renverse la chaîne résultante. Une fois que c'est fait, l'approche présentée en classe fonctionne immédiatement.
En d'autres termes, pour chaque position i, on veut le plus long palindrome dont la deuxième moitié commence en i. Celui-ci est le plus long préfixe commun entre $S[i..n]$ et $S'[n-i+2..n]$, i.e. $PLPC(i, n-i+2)$.
Par exemple, soit $S = TTAGCTT$, on a le palindrome-complément $[AGCT]$. On a $S' = AAGCTAA$. Quand $i = 5$, on a $n - i + 2 = 4$, $PLPC(5, 4) = 2$ (visuellement parce que $S = TTAG\underline{CT}T$ et $S' = AAG\underline{CT}AA$). Le CT souligné dans S' correspond au AG de S, indiquant qu'il y a un palindrome-complément dont la 2ème moitié est de longueur 2 et commence à CT. □

Exercice 7: La distance Hamming entre deux séquences S et T de même longueur est le nombre de positions auxquelles S et T diffèrent. Par exemple, $S = ACCT$ et $T = AGCT$ ont une distance Hamming de 1.
Soient S un mot à chercher et T une séquence plus longue. Montrez comment on peut trouver toutes les sous-chaînes T' de T de longueur $|S|$ telles que T' et S sont à distance Hamming de 0 ou 1. (on peut le faire en temps $O(|T| + |S| + occ)$)
De façon plus générale, montrez comment trouver toutes les sous-chaînes de T à distance Hamming au plus k de S, et ce en temps $O(k|T| + |S| + occ)$.
(indice: il y a une façon qui ne fait qu'utiliser des appels à PLPC(i, j))

Solution. Soient $n = |S|$ et $m = |T|$. On définit $PLPC(i, j)$ comme la longueur du plus long préfixe commun entre $S[i..n]$ et $T[j..m]$.
Supposons $k = 1$. L'idée est d'essayer de trouver une occurrence de T à distance Hamming de 1 à chaque position de i de S. S'il y a une telle occurrence, on sait qu'on va matcher $j = PLPC(i, 1)$ caractères, que le caractère non-matché sera à la position $i + j + 1$ de S (et $j + 1$ de T), et qu'on devra avoir $PLPC(i + j + 2, j + 2) = m - j$ (c'est-à-dire, on doit matcher le reste de T). En d'autres termes, on fait des bonds de PLPC. Pour k en général, on a le droit de faire k bonds, et il faut qu'on réussisse à épuiser tous les caractères de T. Vous pouvez soit regarder le pseudo-code ci-bas, ou bien consulter `http://www.cs.tau.ac.il/~haimk/adv-alg-2014/pattern-matching-with-mismatches.pdf` qui l'explique de façon plus jolie.

fonction *rechercheInexacte(*S, T, k*)*
  Construire l'arbre de suffixe généralisé pour S et T;
  Prétraiter l'arbre pour répondre à des requêtes de LCA en temps $O(1)$;
  pour $i = 1..n$ **faire**
    $j = 0$ `// Nombre de caractères de` T `épuisés`
    $\ell = 0$ `// Nombre de mismatches`
    tant que $\ell < k$ *et* $j \leq m$ **faire**
      $j = PLPC(i + j, 1 + j) + 1$;
      $\ell = \ell + 1$;
    fin
    si $j > m$ **alors**
      Rapporter une occurrence en i;
    fin
  fin

Sachant que chaque PLPC prend un temps $O(1)$ en utilisant le LCA, ceci prend un temps $O(nk)$. □