Arbre binaire de recherche optimal

Léo Gayral

2017-2018

ref : Cormen – Introduction to algorithms, third edition – p.397

Remarque 1. Une structure de dictionnaire consiste en un ensemble fini de mots dans Σ^*, muni d'un test d'appartenance et d'un opérateur d'ajout/suppression d'élément.

On peut implémenter cette structure par un arbre binaire de recherche (*ABR*) T, muni de l'ordre lexicographique, de sorte que chaque nœud corresponde à un mot du dictionnaire. Sous ces conditions, les opérations précédentes se font toutes en $O(h(T))$ dans le pire des cas. Sans hypothèses particulières sur les données du dictionnaire, on va généralement cherche à garder un arbre équilibré, de sorte que les opérations se fassent en $O(\ln(n))$ pour un dictionnaire à n éléments, quitte à devoir rééquilibrer la structure périodiquement.

Lorsque le dictionnaire est fixé à tout jamais, sur un *CD-ROM* non réinscriptible par exemple, on peut chercher à minimiser le temps de recherche moyen.

Lemme 1. Soient $k_1 < \cdots < k_n \in \Sigma^*$. On considère que le mot k_i est recherché avec une probabilité $p(k_i) = p_i \in]0, 1]$, et qu'un mot de l'intervalle $]k_i, k_{i+1}[$ est recherché avec fréquence q_i éventuellement nulle – avec $k_0 = -\infty$ et $k_{n+1} = +\infty$ pour les *bords* de $(\Sigma^*, <)$.

Le temps de recherche au sein d'un ABR T, pondéré par la famille $(q_0, p_1, q_1, \ldots, p_n, q_n)$ est égal à $C(T) - \sum_{i=0}^{n} q_i$, avec :

$$C(T) = \sum_{x \in T} [h_T(x) + 1]\, p(x)$$

et $h_T(x)$ la hauteur du nœud au sein de l'arbre, égale à 0 en la racine. En particulier, un ABR T est dit *optimal* s'il minimise C.

Démonstration.
Quitte à *inclure* les cas d'échec au sein de l'arbre, on peut remplacer les $n+1$ feuilles *vides* par des nœuds d_i, avec $p(d_i) = q_i$.

Étant donné un mot k_i, le nombre de comparaisons avant que le dictionnaire l'accepte est $h(k_i) + 1$. Pour un mot dans $]k_i, k_{i+1}[$, le nombre de comparaisons avant rejet est $h(d_i)$.

Le nombre moyen de comparaisons est alors $\sum_{i=1}^{n} [h(k_i) + 1] p_i + \sum_{i=0}^{n} h(d_i) q_i$, d'où le résultat.

Remarquons que multiplier C par une constante positive n'influe pas sur le fait que T minimise ou non la fonction. On peut donc plus généralement considérer des ABRO pour des familles de poids positifs $(q_0, p_1, q_1, \ldots, p_n, q_n)$ quelconques. □

Théorème 1. On peut déterminer un ABRO en $O(n^3)$, par programmation dynamique.

Démonstration.
Étudions plus en détail la structure des ABR. Notre ABR contient les nœuds associés à la plage $]k_0, k_{n+1}[$. Si la racine de $T = (k_i, G, D)$, alors G est un ABR associé à la plage $]k_0, k_i[$ et D à $]k_i, k_{n+1}[$. Cette propriété se propage plus généralement à tout sous-arbre de T, jusqu'à la feuille d_i associée à la plage $]k_i, k_{i+1}[$.

On définit la masse totale de T par $p(T) = \sum_{x \in T} p(x)$, qui ne dépends que de la pondération considérée. Comme $h_T = h_G + 1$ sur G (resp. $h_D + 1$ sur D), on a la relation :

$$\begin{aligned} C(T) &= p_i + \sum_{j=1}^{i-1} [h_T(k_j) + 1] p_j + \sum_{j=0}^{i-1} [h_T(d_j) + 1] q_j \\ &\quad + \sum_{j=i+1}^{n} [h_T(k_j) + 1] p_j + \sum_{j=i}^{n} [h_T(d_j) + 1] q_j \\ &= p(T) + C(G) + C(D) \end{aligned}$$

et $p(T) = p_i + p(G) + p(D)$. En particulier, T est optimal sur la plage $]k_0, k_{n+1}[$ ssi G et D le sont sur leurs plages respectives. Ceci est plus généralement vrai sur une plage quelconque.

Lorsque $i < j$, on pose $e(i, j)$ le minimum de C parmi les ABR sur la plage $]k_i, k_j[$, et $w(i, j)$ le poids de cette plage. Pour $i \in [\![0, n]\!]$, on a toujours $T = d_i$, d'où $e(i, i+1) = w(i, i+1) = q_i$. Plus généralement, la remarque

précédente nous garantit :

$$\begin{aligned} w(i,j) &= p_i + q_i + w(i+1,j) \\ e(i,j) &= \min_{k=i+1}^{j-1} w(i,k) + w(k,j) \end{aligned}$$

donc on peut remplir les tableaux w et e diagonale par diagonale, jusqu'à obtenir $e(0, n+1)$.

En outre, si on garde une trace de quel k réalise le arg min ci-dessus, dans un troisième tableau r, on peut en déduire la racine d'un ABRO pour chaque plage, et donc explicitement construire l'arbre voulu.

On a finalement l'algorithme suivant, qui prends les listes p et q en entrée, et manipule w, e et r des tableaux indexés par $[\![0, n+1]\!]^2$:

```
1  pour i de 0 a n :
2      w(i, i+1) = q_i
3      e(i, i+1) = q_i
4  pour l de 2 a n+1 :
5      pour i de 0 a n+1-l :
6          j = i+l
7          w(i, j) = p_i + q_i + w(i+1, j)
8          e(i, j) = w(i, j) + e(i, i+1) + e(i+1, j)
9          r(i, j) = i+1
10         pour k de i+2 a j-1 :
11             si e(i, j) > w(i, j) + e(i, k) + e(k, j) :
12                 e(i, j) = w(i, j) + e(i, k) + e(k, j)
13                 r(i, j) = k
```

L'utilisation de trois boucles *for* imbriquées justifie la complexité cubique. □