TD d'algorithmique avancée

Corrigé du TD 8 : Dénombrement sur les arbres binaires

Jean-Michel Dischler et Frédéric Vivien

Dénombrement sur les arbres binaires

Dans cet exercice on notera n le nombre de nœuds d'un arbre binaire, f son nombre de feuilles et h sa hauteur. Tous les arbres considérés seront supposés non vides.

1. Quelle est la hauteur maximale d'un arbre à n nœuds ?

   Avec n nœuds on construit un arbre de hauteur maximale quand chaque nœud, excepté la feuille, a un unique fils. L'arbre n'a alors qu'une seule branche qui contient les n nœuds et qui est de longueur $n-1$. La hauteur maximale réalisable est donc $n-1$ et on a toujours : $h \leq n-1$.

2. Quel est le nombre maximal de feuilles d'un arbre de hauteur h ?

   Si $h = 0$, c'est-à-dire si l'arbre se réduit à sa racine, $f = 1$. Sinon, on raisonne par récurrence : le sous-arbre gauche (resp. droit) de la racine est un arbre de hauteur $h-1$ et il a un nombre maximal de feuilles pour un arbre de hauteur $h-1$. Si on note $f(h)$ le nombre maximal de feuilles d'un arbre de hauteur h on a donc :

   $$f(h) = \begin{cases} 1 & \text{si } h = 0, \\ 2 \times f(h-1) & \text{sinon.} \end{cases}$$

   On a donc $f(h) = 2^h$. Le nombre maximal de feuilles d'un arbre de hauteur h est donc 2^h et on a toujours : $f \leq 2^h$.

3. Quel est le nombre maximal de nœuds d'un arbre de hauteur h ?

   Si $h = 0$, c'est-à-dire si l'arbre se réduit à sa racine, $n = 1$. Sinon, on raisonne par récurrence : le sous-arbre gauche (resp. droit) de la racine est un arbre de hauteur $h-1$ et il a un nombre maximal de nœuds pour un arbre de hauteur $h-1$. Si on note $n(h)$ le nombre maximal de nœuds d'un arbre de hauteur h on a donc :

   $$n(h) = \begin{cases} 1 & \text{si } h = 0, \\ 1 + 2 \times n(h-1) & \text{sinon.} \end{cases}$$

   On a donc $n(h) = 2^{h+1} - 1$. Le nombre maximal de nœuds d'un arbre de hauteur h est donc $2^{h+1} - 1$ et on a toujours : $n \leq 2^{h+1} - 1$.

4. Quelle est la hauteur minimale d'un arbre de n nœuds ?

   La minoration de la hauteur est une conséquence directe du résultat de la question précédente :

   $$n \leq 2^{h+1} - 1 \quad \Leftrightarrow \quad n + 1 \leq 2^{h+1} \quad \Leftrightarrow \quad \log_2(n+1) \leq h + 1 \quad \Leftrightarrow \quad h \geq \log_2(n+1) - 1$$

   D'où la minoration : $h \geq \lceil \log_2(n+1) - 1 \rceil$.

5. Montrez que le nombre de branches vides — nombre de fils gauches et de fils droits vides — d'un arbre à n nœuds est égal à $n + 1$.

   Indication : on distinguera les nœuds ayant zéro, un et deux fils.

   On note :
   - *p le nombre de nœuds ayant zéro fils ;*
   - *q le nombre de nœuds ayant un fils ;*
   - *r le nombre de nœuds ayant deux fils.*

   On a les relations suivantes :

– $n = p + q + r$ *: un nœud a soit zéro, soit un, soit deux fils.*
– $0 \times p + 1 \times q + 2 \times r = n - 1$ *: tous les nœuds, la racine exceptée, ont un père ; on compte ces* $n - 1$ *nœuds à partir de leurs pères,* $0 \times p$ *étant fils d'un nœud sans fils,* $1 \times q$ *étant fils d'un nœud ayant un unique fils et* $2 \times r$ *étant fils d'un nœud ayant deux fils.*
– $2 \times p + 1 \times q + 0 \times r = b$ *: on compte les branches vides de chaque nœud.*

En additionnant les deux dernières équations on obtient :

$$2(p + q + r) = b + n - 1 \quad \Leftrightarrow \quad 2n = b + n - 1 \quad \Leftrightarrow \quad b = n + 1$$

en utilisant la première équation.

Autre solution*. On raisonne par récurrence : un arbre réduit à un unique nœud a deux branches vides, donc quand* $n = 1$, $b = n + 1$*. On suppose la propriété vérifiée pour tous les arbres contenant au plus n nœuds. Soit A un arbre à* $n + 1$ *nœuds. On supprime arbitrairement une feuille de A, ce qui nous donne un arbre B à n nœuds et* $n + 1$ *branches vides, d'après l'hypothèse de récurrence. Pour passer de A à B on a supprimé de A une feuille, et donc deux branches vides, et on a rajouté une branche vide au père de la feuille enlevée. Donc B a une branche vide de moins que A qui en a donc* $(n + 1) + 1$*, CQFD.*

6. Montrez que le nombre de feuilles est inférieur ou égal à $\frac{n+1}{2}$ ($f \leq \frac{n+1}{2}$) et qu'il y a égalité si et seulement si chaque nœud de l'arbre est soit une feuille, soit a deux fils.

   Indication : se servir du raisonnement utilisé à la question précédente.

   On a vu précédemment que l'on avait : $2 \times p + 1 \times q + 0 \times r = b$ *avec* $b = n + 1$*. Or* $p = f$ *: les nœuds sans fils sont exactement les feuilles ! q étant positif, on a* $2 \times f \leq n + 1$ *ce qui établit l'inégalité recherchée, et on a égalité quand* $q = 0$*, c'est-à-dire quand l'arbre ne contient pas de nœuds ayant un seul fils.*

7. Montrez que le nombre de feuilles d'un arbre est égal au nombre de nœuds de degré deux, plus un.

   On se sert une fois de plus des relations entre p, q et r : $p + q + r = n$ *et* $q + 2r = n - 1$*. En éliminant q entre les deux équations on obtient :* $p = r + 1$ *qui est le résultat recherché.*

Complexité du tri

Les algorithmes de tris par comparaison peuvent être considérés de façon abstraite en termes d'**arbres de décision**. Un arbre de décision représente les comparaisons (à l'exclusion de toute autre opération) effectuées par un algorithme de tri lorsqu'il traite une entrée de taille donnée. La figure 1 présente l'arbre de décision correspondant à l'algorithme de tri par insertion s'exécutant sur une liste de trois éléments.

Soient $\langle a_1, a_2, ..., a_n \rangle$ les n valeurs à trier. Dans un arbre de décision chaque nœud interne est étiqueté par une expression $a_i \leq a_j$, pour certaines valeurs de i et de j, $1 \leq i, j \leq n$. L'exécution de l'algorithme de tri suit un chemin qui part de la racine de l'arbre de décision pour aboutir à une feuille. À chaque nœud interne, on effectue une comparaison $a_i \leq a_j$ et les comparaisons suivantes auront lieu dans le sous-arbre gauche si $a_i \leq a_j$ et dans le sous-arbre droit sinon. Chaque feuille est désignée par une permutation $\langle \pi(1), \pi(2), ..., \pi(n) \rangle$: si l'algorithme de tri aboutit en la feuille $\langle \pi(1), \pi(2), ..., \pi(n) \rangle$, les valeurs à trier vérifient : $a_{\pi(1)} \leq a_{\pi(2)} \leq ... \leq a_{\pi(n)}$.

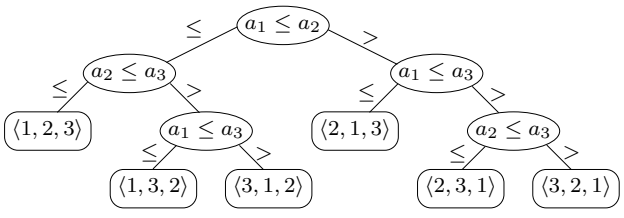

FIG. 1 – Arbre de décision correspondant au traitement de trois éléments au moyen du tri par insertion.

1. Quel est le nombre de feuilles d'un tel arbre de décisions ?
   On a $n!$ permutations possibles de n éléments, donc $n!$ feuilles possibles à notre arbre de décision.
2. En déduire une borne inférieure sur la hauteur de l'arbre de décision.
   On a vu précédemment que : $f \leq 2^h$ ce qui équivaut à $h \geq \log_2 f$ avec ici $f = n!$. Donc $h \geq \log_2(n!)$.
3. En déduire une *borne inférieure* sur la complexité du tri de n éléments.
   Indication : d'après la formule de Stirling, on a $n! > \left(\frac{n}{e}\right)^n$.
   La longueur d'un chemin de la racine à une feuille dans l'arbre de décision est égale au nombre de comparaisons nécessaires au tri pour parvenir à la réponse souhaitée. La longueur du plus long chemin de la racine à une feuille — qui est égale par définition à la hauteur de l'arbre — nous donne donc la complexité $T(n)$ de l'algorithme dans le pire cas. D'après la question précédente, la hauteur d'un tel arbre de décision, quel que soit l'algorithme de tri, est au moins de $\log_2(n!)$. Donc, en utilisant la formule de Stirling : $T(n) \geq \log_2(n!) > n\log(\frac{n}{e})$, d'où, comme $\log(ab) = \log(a) + \log(b)$:

$$T(n) = \Omega(n \log n).$$

Arbres binaires de recherche

1. Montrez que le temps de création d'un arbre binaire de recherche à partir d'une liste quelconque de n éléments est $\Omega(n \log n)$.
   Partant d'une liste quelconque de n éléments, si nous construisons un arbre binaire de recherche que nous parcourons en affichant les clés suivant un parcours (en profondeur) infixe, on obtient la liste des n éléments triés. Vu le résultat obtenu à l'exercice précédent la complexité de cette manipulation, qui est un tri, est $\Omega(n \log n)$. Le parcours avec affichage étant de complexité $\Theta(n)$, la construction de l'arbre binaire de recherche est $\Omega(n \log n)$.
2. Écrivez un algorithme qui teste si un arbre binaire est un arbre binaire de recherche.

```
EstArbreDeRecherche(x)
  (booléen, min, max) ← VérifieArbre(x)
  renvoyer booléen

VérifieArbre(x)
  (b1, min1, max1) ← VérifieArbre(gauche(x))
  si b1 = Faux alors renvoyer (Faux, 0, 0)
  (b2, min2, max2) ← VérifieArbre(droit(x))
  si b2 = Faux alors renvoyer (Faux, 0, 0)
  si max1 ≤ clé(x) ≤ min2
    alors renvoyer (Vrai, min1, max2)
    sinon renvoyer (Faux, 0, 0)
```