Probabilités, Statistiques, Combinatoire

Philippe Duchon

Université de Bordeaux – Licence Informatique
http://www.labri.fr/perso/duchon/Enseignements/Probas/

2019-2020

Arbres (binaires) aléatoires

- "arbres binaires aléatoires" : on se fixe un entier $n > 0$, et on va choisir une loi de probabilités sur l'ensemble $\mathcal{B}_n$ des arbres binaires (pas forcément complets) à n noeuds (ou arbres binaires complets à $2n+1$ noeuds).
- Il y a C_n tels arbres (nombre de Catalan), le même nombre que les mots de Dyck de longueur $2n$)

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

$1, 1, 2, 5, 14, 42 \ldots$

- Ordre de grandeur : $C_n \leq 4^n$ (les mots de Dyck de longueur $2n$, font partie de l'ensemble de tous les mots de longueur $2n$ sur l'alphabet $\{a, b\}$, donc il ne peut pas y en avoir plus)
- Plus précisément, $C_n \simeq c.4^n/n\sqrt{n}$ (une proportion de l'ordre de $1/n^{3/2}$ des mots de longueur $2n$ sont des mots de Dyck).
- À quoi ça ressemble, un "grand" arbre binaire aléatoire ?

Choix d'une loi de probabilités (n fixé)

- On a deux lois de probabilités classiques sur $\mathcal{B}_n$:
  - la loi uniforme (chaque arbre possible a probabilité $1/C_n$) : la plus simple à décrire.
  - une autre loi "naturelle" : celle des arbres binaires qu'on obtient en insérant, dans un ordre aléatoire uniforme, n nombres distincts dans un arbre binaire de recherche vide ;
- On va se poser deux questions sur chacune des deux lois :
  - comment est-ce qu'on peut les **simuler** ? (tirer un arbre selon la loi choisie)
  - peut-on prédire à quoi ressemblent des arbres aléatoires ? (les deux lois sont très différentes)

Arbres aléatoires uniformes : méthode de Rémy

- **Rappel :** bijection de Rémy entre
  - un arbre binaire complet de taille n, avec une feuille marquée et un choix parmi $\{G, D\}$, et
  - un arbre binaire complet de taille $n+1$, avec un noeud marqué
- On peut utiliser de manière répétée la bijection de Rémy pour obtenir un arbre aléatoire uniforme de taille n :
  - On part de l'arbre de taille 0
  - on répète n fois :
    - choix uniforme d'une feuille
    - choix uniforme de la direction G ou D
    - application de Rémy
    - oubli du noeud marqué
  - l'arbre aléatoire obtenu à la fin est uniforme.

Arbres aléatoires uniformes : simulation

On se fixe un entier n, par exemple $n = 1000$, ou $n = 50000$, ou $n = 1000000$.

- On sait calculer C_n, on peut facilement tirer un entier aléatoire entre 1 et C_n (pour n grand, on ne peut pas utiliser directement les fonctions classiques des bibliothèques qui ne retournent que des entiers sur 32 ou 64 bits, mais ce n'est pas un gros problème)
- En revanche, il est exclus de calculer la liste de tous les arbres de taille n, tirer un entier K, et retourner le K-ème : il y a trop d'arbres, ça ne tiendrait jamais en mémoire.
- On peut revenir à la formule de récurrence sur les nombres de Catalan : pour chaque entier k, $0 \leq k \leq n-1$, il y a $C_k \cdot C_{n-1-k}$ arbres de taille n dont l'arbre gauche est de taille k et l'arbre droit de taille $n-1-k$.

Arbres aléatoire : méthode récursive

- **Méthode récursive :** pour un n donné, on calcule les nombres de Catalan C_0 à C_n, puis, pour tirer un arbre de taille $m \leq n$:
  - on tire un entier aléatoire M entre 1 et C_m, uniforme
  - on trouve l'entier k tel que

$$\sum_{j=0}^{k-1} C_j C_{m-1-j} < M \leq \sum_{j=0}^{k} C_j C_{m-1-j};$$

  - k nous donne la taille de l'arbre gauche ; on tire récursivement un arbre aléatoire G, uniforme de taille k, et un arbre aléatoire D (indépendant de G), uniforme de taille $m-1-k$;
  - On retourne l'arbre dont l'arbre gauche est G, et l'arbre droit, D.

```
def ArbreAleatoire(C,m):
    # C: la liste des nombres de Catalan
    if m<=0:
        return [0,[],[]]
    N = GrandEntierAleatoire(1,C[m])
    k=0
    s=C[0]*C[m-1]
    while N>s:
        k += 1
        s += C[k] * C[m-1-k]
    return [m, ArbreAleatoire(C,k),
                ArbreAleatoire(C,m-1-k)]
```

Ce qu'on observe...

- Dans un arbre aléatoire uniforme de taille n, l'arbre gauche (droit) est fréquemment très petit (proche de 0) ou très grand (proche de $n-1$)
- La hauteur de l'arbre est assez élevée (généralement de l'ordre de $\sqrt{n}$)
- Ce sont des choses qu'on peut prouver (mais c'est beaucoup plus difficile)

ABR aléatoires

- Une façon d'obtenir un "ABR aléatoire" : prendre une liste comprenant $1, 2, \ldots, n$; "mélanger" (parfaitement) cette liste ; et insérer les valeurs dans un ABR vide, dans l'ordre obtenu.

ABR aléatoires

- Une façon d'obtenir un "ABR aléatoire" : prendre une liste comprenant $1, 2, \ldots, n$; "mélanger" (parfaitement) cette liste ; et insérer les valeurs dans un ABR vide, dans l'ordre obtenu.
- **Première question :** pour un arbre binaire donné, quelle est la probabilité d'obtenir cet arbre ?

ABR aléatoires

- Une façon d'obtenir un "ABR aléatoire" : prendre une liste comprenant $1, 2, \ldots, n$; "mélanger" (parfaitement) cette liste ; et insérer les valeurs dans un ABR vide, dans l'ordre obtenu.
- **Première question :** pour un arbre binaire donné, quelle est la probabilité d'obtenir cet arbre ?
- **Deuxième question :** à quoi ça ressemble, un tel arbre binaire aléatoire ? (pour n grand, disons)

ABR aléatoires

- Une façon d'obtenir un "ABR aléatoire" : prendre une liste comprenant $1, 2, \ldots, n$; "mélanger" (parfaitement) cette liste ; et insérer les valeurs dans un ABR vide, dans l'ordre obtenu.
- **Première question :** pour un arbre binaire donné, quelle est la probabilité d'obtenir cet arbre ?
- **Deuxième question :** à quoi ça ressemble, un tel arbre binaire aléatoire ? (pour n grand, disons)
- **Troisième question :** et concrètement, ça sert à quoi ?

Probabilité d'un arbre binaire

- t un squelette d'arbre binaire à n noeuds
- Dans une liste de longueur n "bien mélangée", les $n!$ ordres possibles sont équiprobables (proba. $1/n!$ chacun).
- Donc la probabilité d'obtenir t comme arbre aléatoire, c'est $p(t) = \text{ins}(t)/n!$, où $\text{ins}(t)$ est *le nombre d'ordres d'insertion qui donnent t comme arbre*.
- Peut-on calculer $\text{ins}(t)$? (donner une formule ?) Oui !
- Supposons que t a un sous-arbre gauche t_1, de taille k, et sous-arbre droit t_2, de taille $n-k-1$ $(0 \leq k \leq n-1)$
- **Proposition :** $\text{ins}(t) = \binom{n-1}{k}\text{ins}(t_1)\text{ins}(t_2)$.
- **Corollaire :** $p(t) = \frac{1}{n}p(t_1)p(t_2)$.
- **Autre formule :** $p(t) = \prod_s \frac{1}{n(s)}$, où s parcourt l'ensemble des noeuds de t, et où $n(s)$ désigne la taille (nombre de noeuds) de l'arbre dont s est la racine.

Apparence générale d'un arbre

Pour un arbre de taille n, "aléatoire" :

- La taille K du sous-arbre gauche est uniforme sur $\{0, 1, \ldots, n-1\}$
- La taille du sous-arbre gauche est $n-1-K$, elle aussi uniforme sur $\{0, 1, \ldots, n-1\}$ (mais pas indépendante de K)
- Conditionnellement à $\{K = k\}$, les sous-arbres gauche et droit sont des arbres binaires de recherche aléatoires de leurs tailles respectives, et indépendants.
- Également (admis) : **l'espérance de la hauteur d'un arbre binaire aléatoire de taille n, est d'ordre $O(\log(n))$**
- Autrement dit : les ABR aléatoires sont naturellement "à peu près équilibrés"
- C'est intéressant : ça montre que pour les ABR, "le hasard fait bien les choses" – les algorithmes dont la complexité est gouvernée par la hauteur de l'arbre, s'exécutent rapidement sur de tels arbres
- **Mais** on ne peut pas trop compter sur le fait que les insertions se fassent dans un ordre aléatoire !

Arbres binaires de recherche “randomisés”

(Martinez, Roura, 1999)

- **Scoop :** on peut réaliser les opérations (insertion/suppression) dans les arbres binaires de recherche, de manière probabiliste, de telle sorte que, **après n'importe quelle séquence d'insertions et de suppressions dans l'arbre**, l'arbre obtenu soit un “arbre aléatoire” (ne dépendant pas de la séquence exacte d'insertions/suppressions : pour toute séquence s d'opérations, la probabilité d'obtenir l'arbre t après s, en partant de l'arbre vide, est $p(t)$).
- Les algorithmes d'insertion et de suppression deviennent “randomisés” (le comportement dépend de tirages aléatoires)
- **Insertion :** dans l'algorithme classique, à chaque étape récursive on peut décider d'insérer la valeur à la racine courante, en “splittant” l'arbre
- **Suppression :** une fois trouvée la clé à supprimer, on “recolle” aléatoirement les deux sous-arbres gauche et droit
- La preuve que les algorithmes ont cette propriété est complexe ; les algorithmes eux-mêmes sont très simples.

RBST : randomized binary searth trees

- Les algorithmes utilisent la *taille* des sous-arbres ; on stocke donc dans chaque noeud, la taille du sous-arbre enraciné en ce noeud.
- **Recherche :** exactement comme dans un ABR classique.
- **Split :** étant donnés un ABR et une clé x, on découpe l'ABR en deux ABR, l'un contenant les clés plus petites que x, l'autre contenant les clés plus grandes que x

```
let rec split t x = match t with
  |Empty -> Empty,Empty
  |Bin((r,n),g,d) ->
   if r=x then g,Bin((x,1+r_size d),Empty,d)
   else if x<r then let gg,dd = split g x in
        gg,Bin((r,n-r_size(gg)),dd,d)
   else let gg,dd = split d x in
        Bin((r,n-r_size(gg)),g,gg),dd
```

(Temps : $O(h)$ où h est la hauteur de l'arbre)

RBST : suite

- **Insertion à la racine :** sépare l'arbre en deux (split) de part et d'autre de x, et met les deux arbres comme sous-arbres

```
let rec rbst_insert_at_root x t = match t with
  | Empty -> Bin((x,1),Empty,Empty)
  | Bin((_,n),_,_) ->
     let gg,dd = split t x in
     match dd with
     | Bin((y,m),Empty,d') when y=x
         -> Bin((x,n),gg,d')
     | _ -> Bin((x,n+1),gg,dd)
```

RBST : insertion

- **Insertion :** si l'arbre est de taille n : avec proba. $1/(n+1)$, on insère à la racine ; sinon, on suit l'algorithme (récursif) d'insertion classique

```
let rec insert t x = match t with
  | Empty -> Bin((x,1),Empty,Empty)
  | Bin((r,n),g,d) -> let k= Random.int (n+1) in
    if k=0 then rbst_insert_at_root x t
    else if x<r then Bin((r,n+1),(insert g x),d)
    else Bin((r,n+1),g,(insert d x))
```

RBST : join

- **join :** (probabiliste) à partir de deux arbres t_1 et t_2, dont on suppose que toute clé de t_1 est inférieure à toute clé de t_2, fusionne les deux arbres, en prenant la racine dans un des deux arbres et en descendant récursivement ; on prend la racine de chaque arbre avec proba. proportionnelle à sa taille.

```
let rec join t1 t2 = match t1,t2 with
  | Empty,_ -> t2
  | _,Empty -> t1
  | Bin((r1,n1),g1,d1),Bin((r2,n2),g2,d2) ->
    let k = Random.int (n1+n2) in
    if k<n1 then Bin((r1,n1+n2),g1,join d1 t2)
    else Bin((r2,n1+n2),join t1 g2, d2)
```

RBST : suppression

- **suppression :** on commence par chercher la clé à supprimer ; quand on la trouve, on “joint” les deux sous-arbres

```
let rec delete t x = match t with
  | Empty -> Empty
  | Bin((r,n),g,d) -> if r=x then join g d
    else if x<r then Bin((r,n-1),delete g x,d)
    else Bin((r,n-1),g,delete d x)
```

Structures de données aléatoires

- Une structure de données aléatoire (comme les RBST) s'utilise exactement comme une structure non aléatoire (ABR, arbres rouges et noirs, etc)
- On gagne sur la simplicité d'implémentation, sans trop sacrifier la performance.
- Avec les RBST : on ne garantit pas qu'un arbre ayant n noeuds est **toujours** de hauteur $O(\log(n))$, mais on le garantit avec probabilité proche de 1 : pour une constante $C \simeq 4.3$,

$$\mathbb{P}(h(T_n) \geq C \log(n)) \geq \frac{1}{n^{\alpha}}$$

- Et cette garantie, initialement valable pour les "arbres binaires aléatoires" (obtenus par n insertions dans un ordre aléatoire), reste valable pour **tout** RBST qui évolue par les algorithmes précédents.