Bornes inférieures de complexité I

Cours aux Journées ALÉA 2020

Cyril Nicaud

LIGM – Univ Gustave Eiffel & CNRS

Mars 2020

Philippe Flajolet ... 10 ans déjà

Avant propos

- C'est un cours d'algorithmique, on s'attardera plus sur les idées et les techniques de preuves que sur le formalisme mathématique
- Je ne suis pas un spécialiste du sujet (je l'enseigne au niveau M1, mais on va aller plus loin)
- Le cours est organisé ... comme un cours, en prenant son temps

- Il y a assez peu de ressources sur le sujet (hors articles de recherche). On se réfèrera à
  - La page de Jeff Erickson avec ses notes de cours sur les bornes inférieures par comptage et sur les arguments d'adversaire : `http://web.engr.illinois.edu/~jeffe/teaching/algorithms/notes/`
  - Le livre "Randomized Algorithms" de R. Motwani & P. Raghavan, pour les bornes inférieures d'algorithmes probabilistes

C'est parti !

1. Complexité des algorithmes et des problèmes

Complexité d'un algorithme (principe)

Buts: estimation de performances / comparaison d'algorithmes

```
def search(x,T):
  for y in T:
    if x == y:
      return True
  return False
```

- E_n les entrées de taille n, et $E = \cup_n E_n$
- $C(e)$ le nombre d'instructions effectuées pour l'entrée e
- C est une application de E dans $\mathbb{N}$

Comment estimer / comparer deux applications de $E \longrightarrow \mathbb{N}$?

- On agrège l'information à taille n fixée
- $C_n = \max\{C(e) : e \in E_n\}$ (pire cas) ou $C_n = \mathbb{E}_{E_n}[C]$ (cas moyen)

Comment estimer / comparer deux suites de $\mathbb{N} \longrightarrow \mathbb{N}$?

- On les estime asymptotiquement
- On utilise les notations $\mathcal{O}$, Ω et Θ

$u_n = \mathcal{O}(v_n)$ quand il existe $c > 0$ tq $u_n \leq cv_n$ à partir d'un certain rang
$u_n = \Omega(v_n)$ quand il existe $c > 0$ tq $u_n \geq cv_n$ à partir d'un certain rang
$u_n = \Theta(v_n)$ quand $u_n = \mathcal{O}(v_n)$ et $u_n = \Omega(v_n)$

Complexité d'un algorithme (exemples)

```
def search(x,T):
  for y in T:
    if x == y:
      return True
  return False
```

- Pire cas : $\Theta(n)$
- Cas moyen : $\Theta(n)$

```
def qsort(x,T):
  if len(T) <= 1: return T
  G = [x in T[1:] if x<T[0]]
  D = [x in T[1:] if x>=T[0]]
  return qsort(G)+[T[0]]+qsort(D)
```

- Pire cas : $\Theta(n^2)$
- Cas moyen : $\Theta(n \log n)$

```
def mergesort(x,T):
  if len(T) <= 1: return T
  G = mergesort(T[:n//2])
  D = mergesort(T[n//2:])
  return merge(G,D)
```

- Pire cas : $\Theta(n \log n)$
- Cas moyen : $\Theta(n \log n)$

Complexité d'un problème (définition)

"Definition" : un problème est une tâche à effectuer par ordinateur (une application de E dans un ensemble F).

Exemples : trier un tableau (**SORT**), tester s'il y a trois éléments x, y, z dans T tels que $x + y + z = 0$ (**3SUM**)

Un même problème peut être résolu par plusieurs algorithmes.

Définition (complexité d'un problème)

Un problème Π est de complexité $\mathcal{O}(u_n)$ quand il existe un algorithme de complexité $\mathcal{O}(u_n)$ qui résout Π.

Un problème Π est de complexité $\Omega(u_n)$ quand tout algorithme qui résout Π est de complexité $\Omega(u_n)$: on dit alors que $\Omega(u_n)$ est une borne inférieure de complexité pour Π.

Le problème **SORT** est de complexité $\mathcal{O}(n \log n)$, car il est résolu par l'algorithme `mergesort` qui est en $\mathcal{O}(n \log n)$ (pire cas).

Complexité d'un problème (modèle de calcul)

Définition (complexité d'un problème)

Un problème Π est de complexité $\Omega(u_n)$ quand tout algorithme qui résout Π est de complexité $\Omega(u_n)$: on dit alors que $\Omega(u_n)$ est une borne inférieure de complexité pour Π.

```
def search(x,T):
  for y in T:
    if x == y:
      return True
  return False
```

Pour estimer la complexité d'un **algorithme**, il suffit de définir ce que coûtent ses différentes instructions : ici en temps $\mathcal{O}(1)$: aller au y suivant, tester si $x == y$, ...

Pour les bornes inférieures de complexité d'un **problème** :

- Il faut dire quelque-chose sur tous les algorithmes !
- Il faut donc définir **précisément** ce qu'est un algorithme : il faut définir un **modèle d'ordinateur** (machine de Turing, RAM, ...)

Complexité d'un problème (exemple)

Rappel : SORT est le problème de trier un tableau de taille n

Exemple de théorème (tri par comparaisons)

Dans le modèle de calcul où l'on peut juste comparer les données d'un tableau, le problème **SORT** est en $\Omega(n \log n)$.

Seul accès aux données : "Est-ce que $T[i] < T[j]$?"

- ☹ Le résultat ne porte que sur les **tris par compaisons**
- ☺ Pas besoin de spécifier le modèle d'ordinateur précisément
- ☺ Pas besoin de spécifier l'encodage des données
- ☺ L'algorithme `mergesort` est optimal pour ce modèle

Dans la suite on utilisera ce type de modèles car il est très difficile d'obtenir des résultats non-triviaux dans des modèles généraux : RAM, machines de Turing (*pas réaliste pour des pb polynomiaux*), ...

Borne inférieure d'un problème (récapitulatif)

Définition (complexité d'un problème)

Un problème Π est de complexité $\Omega(u_n)$ quand tout algorithme qui résout Π est de complexité $\Omega(u_n)$: on dit alors que $\Omega(u_n)$ est une borne inférieure de complexité pour Π.

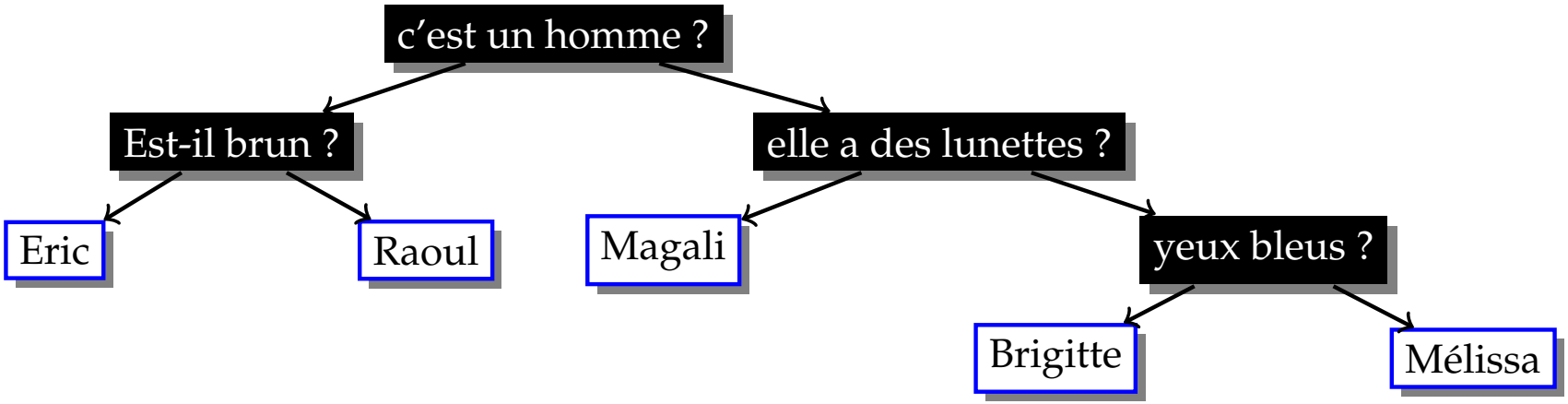
- Ce n'est pas (du tout) une complexité meilleur cas d'un algo
- On doit spécifier un modèle de calcul pour pouvoir dire quelque chose sur **tous** les algorithmes qui sont solution du problème
- On va s'intéresser à des problèmes polynomiaux : pas à des questions de type **NP**-difficulté (qui sont aussi des bornes inf.)
- On va utiliser des modèles simples de calcul, et passer du temps à interpréter les résultats obtenus

2. Comptage

Le jeu “Qui est-ce ?”

- Chaque joueur tire au sort une carte personnage
- Ils posent des **questions oui/non** tour à tour pour identifier le personnage tiré par l’autre joueur

Modèle "arbre de décision"



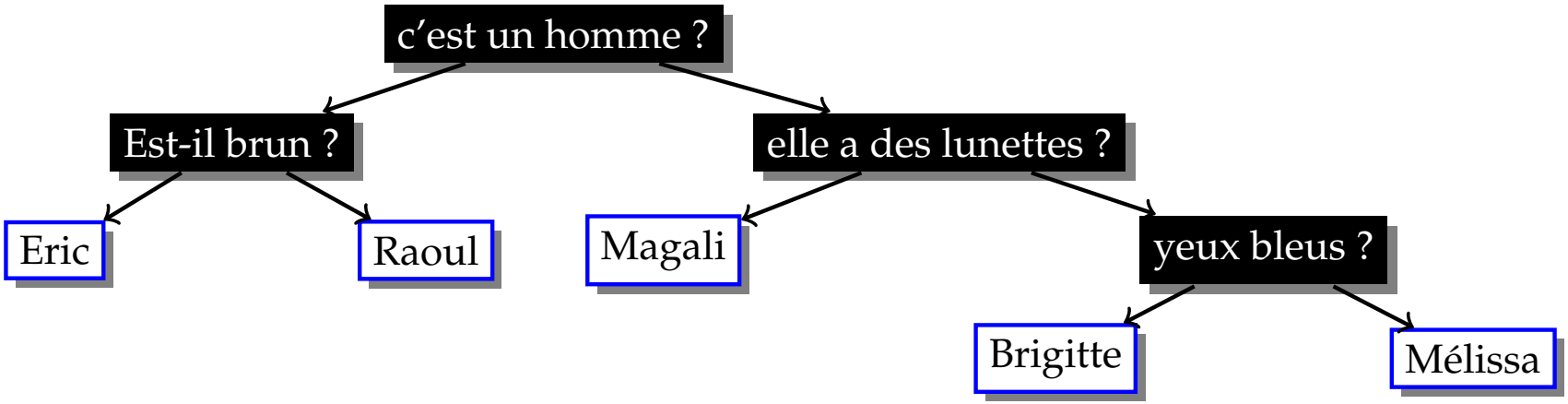
Dans le *modèle "arbre de décision" (MAD)* :

- Pour chaque n, on a un algorithme sous forme d'**un arbre de questions oui/non** pour identifier la réponse
- Le coût d'une exécution est le **nombre de questions** posées
- La complexité pire cas est la **hauteur de l'arbre**

Sur l'exemple, la complexité au pire est de **3** (*Brigitte* et *Mélissa*).

C'est un modèle simple où sont négligés tous les calculs qui ne sont pas des questions $\Rightarrow$ c'est OK car on cherche des bornes inférieures

La technique par comptage



Tous les algos pour résoudre le problème $=$ tous les arbres de décision

Théorème (argument de comptage)

Dans le **modèle "arbre de décision"**, si tout algorithme qui résout un problème doit produire au moins R_n réponses différentes pour les entrées de taille n alors le problème admet une borne inférieure de complexité en $\log_2 R_n$.

Preuve : un arbre binaire de hauteur h a au plus 2^h feuilles □

Le problème SORT (définition)

"Definition :" étant donné un tableau de n nombres, **SORT** consiste à le ranger par ordre croissant

☹ dans un MAD, on n'a pas accès directement aux valeurs stockées dans le tableau $\Rightarrow$ on ne peut pas retourner le tableau trié

Définition (SORT)

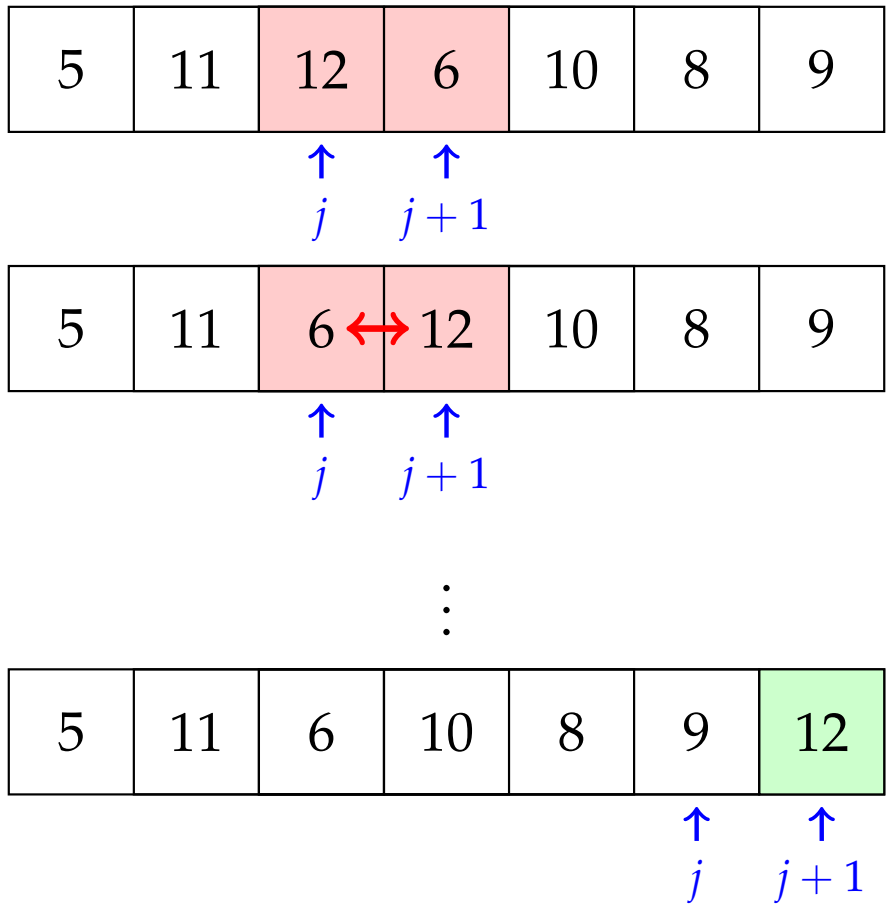
Soit T un tableau de n nombres, le problème **SORT** consiste à construire une permutation σ de $\{0 \dots n-1\}$ telle que

$$T[\sigma(0)] \leq T[\sigma(1)] \leq \dots \leq T[\sigma(n-1)]$$

c'est-à-dire de trouver comment permuter les éléments de T de sorte qu'ils soient dans l'ordre croissant.

Le problème SORT (tri bulle, présentation)

```
def triBulle(x,T):
  for i in range(len(T)):
    for j in range(len(T)-i-1):
      if T[j] > T[j+1]:
        T[j],T[j+1] = T[j+1],T[j]
  return T
```



Le problème SORT (tri bulle, reformulation)

```
def triBulle(T):
  for i in range(len(T)):
    for j in range(len(T)-i-1):
      if T[j] > T[j+1]:
        T[j],T[j+1] = T[j+1],T[j]
  return T
```

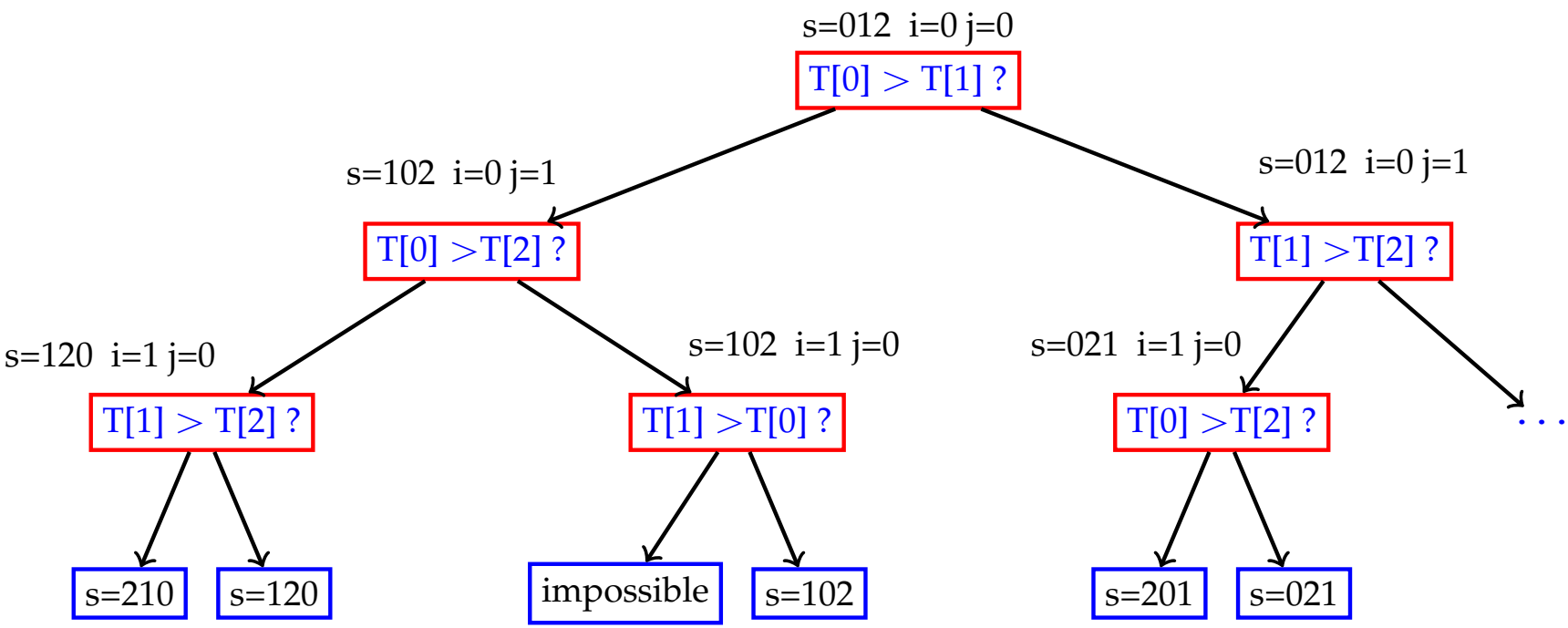
Est reformulé pour rentrer dans notre spécification (calculer une permutation qui ordonne les éléments) en :

```
def triBulle(T):
  s = list(range(len(T))) # s = perm. identité
  for i in range(len(T)):
    for j in range(len(T)-i-1):
      if T[s[j]] > T[s[j+1]]:
        s[j],s[j+1] = s[j+1],s[j]
  return s
```

On ne pose que des questions : “Est-ce que $T[i] < T[j]$?”
$\Rightarrow$ c’est un **tri par comparaisons**, donc dans le cadre MAD

Le tri bulle comme arbre de décision ($n = 3$)

```
def triBulle(T):
  for i in range(len(T)):
    for j in range(len(T)-i-1):
      if T[j] > T[j+1]:
        T[j],T[j+1] = T[j+1],T[j]
  return T
```



Borne inférieure pour SORT

Théorème (borne inf. SORT)

Dans le **modèle "arbre de décision"**, **SORT** admet une borne inférieure de complexité en $\Omega(n \log n)$.

Preuve : par comptage, car $\log n! \in \Omega(n \log n)$ □

Corollaire (borne inf. tris par comparaisons)

Tout **tri par comparaisons** nécessite $\Omega(n \log n)$ comparaisons dans le pire cas.

Attention

Théorème (argument de comptage)

Dans le **modèle "arbre de décision"**, si tout algorithme qui résout un problème doit produire au moins R_n réponses différentes pour les entrées de taille n alors le problème admet une borne inférieure de complexité en $\log_2 R_n$.

Si on considère le problème "Trouver deux indices $i \neq j$ tels que $T[i] \leq T[j]$", pour le modèle par comparaisons.

- Le problème admet tous les couples (i, j) comme solutions potentielles
- Mais un algorithme qui le résout peut ne répondre que $(0, 1)$ ou $(1, 0)$
- La borne inférieure par comptage est $\Omega(1)$!

```
def two_ordered(T):
  if T[0] <= T[1]:
    return (0,1)
  return (1,0)
```

Borne inférieure pour FIND_SORTED

Définition

Le problème **FIND_SORTED** consiste à chercher un élément x dans un tableau trié T de taille n. On doit retourner la position d'une occurrence de x si $x \in T$ et `False` sinon.

Remarque : une dichotomie résout le problème en $\mathcal{O}(\log n)$ comparaisons de x avec des éléments de T.

Théorème (borne inf. FIND_SORTED)

Dans le **modèle "arbre de décision"**, **FIND_SORTED** admet une borne inférieure de complexité en $\Omega(\log n)$.

Preuve : par comptage, car $\log_2(n+1) \in \Omega(\log n)$ □

Borne inférieure pour FIND

Définition

Le problème **FIND** consiste à chercher un élément x dans un tableau T de taille n (non nécessairement trié). On doit retourner la position d'une occurrence de x si $x \in T$ et `False` sinon.

Remarque : cette fois on ne peut pas faire une dichotomie.

Théorème (borne inf. FIND)

Dans le **modèle "arbre de décision"**, **FIND** admet une borne inférieure de complexité en $\Omega(\log n)$.

Preuve : par comptage, car $\log_2(n+1) \in \Omega(\log n)$ □

☹ La borne inférieure obtenue ne paraît pas très intéressante.

Borne supérieure pour FIND dans MAD

Théorème (borne inf. FIND)

Dans le **modèle "arbre de décision"**, **FIND** admet une borne inférieure de complexité en $\Omega(\log n)$.

En fait, dans le modèle MAD, on peut faire une dichotomie !

```
def dichoMAD(T, x):
  d, f = 0, len(T)-1
  while d <= f:
    m = (d+f) // 2
    if T[m] == x: #question MAD
      return m
    if "x est dans T entre les positions d et m-1": #question MAD
      f = m-1
    else:
      d = m+1
  return False
```

FIND : conclusion

Théorème (borne inf. FIND)

Dans le **modèle "arbre de décision"**, **FIND** admet une borne inférieure de complexité en $\Omega(\log n)$.

Théorème (borne sup. FIND)

Dans le **modèle "arbre de décision"**, **FIND** est résolu par un algorithme de complexité en $\mathcal{O}(\log n)$.

Conclusion :

- Le MAD est peu puissant car il n'autorise que les questions oui/non sur les données.
- Le MAD est très puissant car il traite ces questions en temps constant.

Technique par comptage : récapitulatif

Théorème (argument de comptage)

Dans le **modèle "arbre de décision"**, si tout algorithme qui résout un problème doit produire au moins R_n réponses pour les entrées de taille n alors le problème admet une borne inférieure de complexité en $\log_2 R_n$.

- Cela donne un outil pour obtenir des bornes inférieures
- On n'échappe pas à une discussion sur le modèle de calcul

Attention : un problème peut être résolu par des algorithmes qui ne retournent pas la même chose : il faut montrer que tout algorithme a au moins R_n réponses différentes.

Remarque : au lieu des questions binaires, on peut autoriser des k-aires, en changeant $\log_2$ en $\log_k$.

3. Adversaire

Exemple 1 : le jeu des couleurs

On considère le jeu (pas très fun) suivant :

- Alice choisit dans sa tête une couleur parmi $\{vert, blue, rouge, noir\}$
- Bob doit deviner la couleur en utilisant uniquement des questions de la forme : **"Est-ce que la couleur est x ?"**

Exemple 1 : le jeu des couleurs

On considère le jeu (pas très fun) suivant :

- Alice choisit dans sa tête une couleur parmi $\{vert, blue, rouge, noir\}$
- Bob doit deviner la couleur en utilisant uniquement des questions de la forme : **"Est-ce que la couleur est x ?"**

En trichant, mais sans que Bob puisse prouver quelle triche, Alice peut forcer Bob à poser au moins 4 questions avant de répondre **oui**.

Pour cela elle a une stratégie pour répondre aux questions, qui peut être décrite par l'algorithme :

- Initialement Alice note secrètement $\mathcal{C} = \{vert, blue, rouge, noir\}$
- À chaque fois que Bob demande si la couleur est x :
  - Si $\mathcal{C} = \{x\}$, Alice répond **oui**
  - Sinon, Alice répond **non** et enlève x de $\mathcal{C}$ (s'il y est)

On enlève au plus un élément par question et Bob ne peut pas prouver qu'Alice triche.

Exemple 2 : “Qui est-ce ?”

- Alice est une prestidigitatrice, elle peut changer sa carte à tout moment sans que Bob ne s’en aperçoive
- À chaque question de Bob, elle change éventuellement de carte pour que Bob élimine le moins de personnage possible.

Il faudra donc au moins $\log_2 n$ questions à Bob pour finir.

Preuve par adversaire : principe

- On veut montrer que tout algo nécessite au moins t étapes sur une entrée de taille n dans un certain modèle de calcul
- On donne les rôles suivants :
  - L'algorithme pose des questions et doit identifier la réponse à donner (il ne connaît l'entrée que par les questions qu'il a posées)
  - L'adversaire répond aux questions en tentant de faire durer le processus le plus longtemps possible
- On veut établir une stratégie pour l'adversaire telle que :
  - À chaque question, l'adversaire choisit une réponse telle qu'au moins une entrée satisfait ses réponses jusqu'ici (**cohérence**)
  - Après $t-1$ questions, il reste au moins deux entrées dont les résultats par l'algorithme sont différents

Remarque : en théorie il suffit de montrer l'existence d'une telle stratégie. En pratique, on la décrit souvent par un algorithme qui décrit les réponse à donner.

Retour sur FIND

Rappel : **FIND** consiste à déterminer si x est dans le tableau T

Modèle : On se place dans le modèle où l'algorithme peut poser comme question **"Quelle est la valeur de $T[i]$?"**

Stratégie de l'adversaire : répondre y à chaque fois, avec $y \neq x$

Théorème (FIND par adversaire)

Dans ce modèle, tout algorithme qui résout **FIND** utilise au moins n questions dans le pire cas.

Preuve : si l'algorithme a posé $n-1$ questions, il y a au moins une case dont il n'a pas demandé la valeur. L'adversaire peut encore choisir de placer x ou y dedans. □

Calcul du min et du max

Définition

Le problème **MIN_MAX** consiste à trouver le minimum et le maximum d'un tableau T (ou l'indice d'un minimum et l'indice d'un maximum).

Modèle : On se place dans le modèle **"par comparaisons"** où on peut poser des questions du type **"Est-ce que** $T[i] < T[j]$ **?"**

```
def naive_minmax(T):
  i_min, i_max = 0, 0
  for i in range(len(T)):
    if T[i] < T[i_min]:
      i_min = i
    if T[i] > T[i_max]:
      i_max = i
  return i_min, i_max
```

- On parcourt les indices du tableau
- On compare `T[i]` avec les min-records et max-records
- On met à jour `i_min` et `i_max`, si besoin

- $\sim 2n$ comparaisons pire cas

Min et Max : algorithme astucieux

- prendre les éléments deux par deux, les comparer entre eux
- comparer le plus petit au min courant
- comparer le plus grand au max courant

```
def minmax_group2(T):
  i_min, i_max = len(T)-1, len(T)-1
  for i in range(0,len(T)-1,2): # de 2 en 2
    if T[i] < T[i+1]:
      a, b = i, i+1
    else:
      a, b = i+1, i
    if T[a] < T[i_min]:
      i_min = a
    if T[b] > T[i_max]:
      i_max = b
  return i_min, i_max
```

Cela fait $\sim \frac{3}{2} n$ comparaisons.

Question : Peut-on faire mieux ?

Min et Max : borne inférieure (énoncé)

Rappel : **MIN_MAX** consiste à le min et le max de T

Modèle : accès aux données "par comparaisons", avec des questions du type **"Est-ce que $T[i] < T[j]$?"**

Théorème (borne inf. MINetMAX)

Dans le modèle **"par comparaisons"**, tout algorithme qui résout le problème **MIN_MAX** nécessite $\sim \frac{3}{2}n$ comparaisons dans le pire cas.

Remarque : l'algorithme astucieux `minmax_group2` est donc optimal

Min et Max : borne inférieure (preuve 1/2)

Alice remplit un tableau de taille n de symboles $\pm$. Les symboles s'interprètent de la façon suivante :

$\pm$	peut être le min ou le max	$+$	peut être le max
$-$	peut être le min	$\emptyset$	ni min ni max

Quand Bob pose une question (une comparaison) Alice donne la réponse qui enlève un minimum de signes ($\pm$ compte pour deux signes). Si Alice enlève le dernier symbole d'une case, elle y place un nombre compatible avec ses réponses antérieures.

tableau	question	réponse	signes perdus
0 1 2 3 4 $\pm$ $\pm$ $\pm$ $\pm$ $\pm$	$T[0] < T[2]$?	oui	2
0 1 2 3 4 $-$ $\pm$ $+$ $\pm$ $\pm$	$T[2] < T[4]$?	non	1
0 1 2 3 4 $-$ $\pm$ $+$ $\pm$ $-$	$T[0] < T[4]$?	oui	1
0 1 2 3 4 $-$ $\pm$ $+$ $\pm$ -1	...	...	...

Min et Max : borne inférieure (preuve 2/2)

Signes perdus :

- $\pm$ vs $\pm$: deux signes
- $\pm$ vs $+$ ou $\pm$ vs $-$: un signe
- $+$ vs $+$ ou $-$ vs $-$: un signe
- rien dans les autres cas

Remarques :

- Il y a toujours un signe $+$ et un signe $-$
- L'algo n'a pas terminé :
  - s'il y a un symbole $\pm$
  - s'il y a 3 signes

Objectif : minimiser $d + u$ avec

d = nombre de questions qui enlèvent deux signes ($\pm$ vs $\pm$)
u = nombre de questions qui enlèvent un signe

La meilleure stratégie est donc de poser un maximum de questions "$\pm$ vs $\pm$", soit $d = \lfloor n/2 \rfloor$, ce qui enlève $2\lfloor n/2 \rfloor$ signes

Comme il y a $2n$ signes initialement, il faut encore poser au moins $u = 2n - 2\lfloor n/2 \rfloor - 2$ questions qui enlèvent un signe

Au total cela fait $\lceil \frac{3n}{2} \rceil - 2 \approx \frac{3}{2}n$ questions

Remarque : si on n'avait pas l'algorithme astucieux, on pourrait le trouver à partir de l'analyse ci-dessus.

Conclusion sur les techniques

On a vu deux techniques :

- Par comptage
- Par adversaire

Qui sont des arguments **assez simples** (parfois astucieux) pour établir des bornes inférieures.

On a vu au passage qu'il est indispensable de préciser le modèle de calcul !

Plus d'exemples pendant la séance d'exercices !

Bornes inférieures de complexité II

Cours aux Journées ALÉA 2021

Cyril Nicaud

LIGM – Univ Gustave Eiffel & CNRS

19 mars 2021

4. Algorithmes probabilistes

“Définition”

Définition

Un algorithme probabiliste est un algorithme qui a accès a une source de hasard qui produit des nombres de façon i.i.d.

Soit E_n l'ensemble des entrées de taille n. On définit la complexité d'un algorithme probabiliste comme étant la suite :

$$n \mapsto \max_{e \in E_n} \mathbb{E}[C(e)],$$

où $C(e)$ est le coût de l'exécution de l'algorithme pour e. L'espérance est prise sur les tirages au sort internes à l'algorithme.

Algo probabilistes vs analyse en moyenne

Quand on **analyse un algorithme en moyenne**

- On choisit pour tout n une distribution sur E_n
- On étudie $n \mapsto \mathbb{E}[C]$, où l'espérance porte sur la distribution sur les entrées

Quand on **analyse un algorithme probabiliste**

- On cherche pour chaque n la pire entrée de taille n
- Le coût d'une entrée étant l'espérance de son coût, où les probas sont sur les tirages internes

Ce ne sont donc pas les mêmes notions, notamment

- pour l'analyse en moyenne il y a une supposition sur la distribution d'entrée (= modélisation),
- on connaît l'aléa interne pour les algorithme probabilistes

Les notions se rejoignent parfois : Quicksort

Dans certain cas, on peut construire un algorithme probabiliste pour simuler une distribution des entrées :

- Si on prend le premier élément comme pivot dans QUICKSORT, il a une complexité moyenne de $\mathcal{O}(n \log n)$ en moyenne pour la distribution uniforme
- Si on prend le pivot uniformément au hasard dans le tableau, on fait un algorithme probabiliste, dont la complexité (au sens des algo probabilistes) est $\mathcal{O}(n \log n)$ pour toute entrée de taille n

Car prendre le pivot uniformément au hasard, revient à mélanger uniformément le tableau au début.

Malheureusement (?), on ne peut pas toujours faire ça

Bornes inférieures, modèle probabiliste

On reprend les mêmes définitions qu'hier, sauf qu'on autorise les tirages au sort internes dans les algorithmes (et qu'on change la notion de complexité !)

Définition (complexité probabiliste d'un problème)

Un problème Π est de complexité probabiliste $\mathcal{O}(u_n)$ quand il existe un algorithme probabiliste de complexité $\mathcal{O}(u_n)$ qui résout Π.

Un problème Π est de complexité probabiliste $\Omega(u_n)$ quand tout algorithme probabiliste qui résout Π est de complexité $\Omega(u_n)$: on dit alors que $\Omega(u_n)$ est une borne inférieure de complexité pour Π.

Rappel : Complexité = $n \mapsto \max_{e \in E_n} \mathbb{E}[C(e)]$

Question type

On a vu que la version probabiliste de QUICKSORT a une complexité $\mathcal{O}(n \log n)$ on en déduit donc le problème **SORT** est de de complexité probabiliste $\mathcal{O}(n \log n)$.

Question : Est-ce que **SORT** a aussi une complexité probabiliste $\Omega(n \log n)$?

Modèles : dans nos modèles il faut rajouter une suite i.i.d. de nombres aléatoires qui est accessible par l'algorithme, et qui est ré-initialisée à chaque nouvelle exécution (indépendamment ...)

Comment faire ? On a enrichi nos modèles avec de l'aléa et changé la définition de la complexité

5. Le principe de Yao

Un peu de théorie des jeux

On considère un jeu à deux joueurs entre **Lamine** (L = lignes) et **Cécile** (C = colonnes). Chacun des joueurs choisit une possibilité, et on a un tableau des gains qui indique combien Cécile donne à Lamine selon les choix effectués.

	pierre	ciseaux	feuille
pierre	0	1	-1
ciseaux	-1	0	1
feuille	1	-1	0

tableau de gains classique

	pierre	ciseaux	feuille
pierre	0	1	2
ciseaux	-1	0	1
feuille	-2	-1	0

autre tableau de gains

On représente le tableau de gain par une matrice de gain M, où M_{ij} est le gain si L joue i et C joue j

Remarque : cela s'appèle un jeu à somme nulle car les gains sont transférés intégralement d'un joueur à une autre.

Objectifs et stratégies

- Le but de L est de maximiser la valeur
- Le but de C est de la minimiser

Si L choisit la stratégie i (son coup) il est assuré de gagner $\min_j M_{ij}$, et c'est ce qui se passe si C joue le meilleur coup contre i.

Donc si L joue en premier et qu'il souhaite à assurer un gain maximal, il choisit la meilleure stratégie i qui lui assure un gain de

$$V_L = \max_i \min_j M_{ij}$$

De façon symétrique, si C joue en premier, elle peut assurer un gain

$$V_C = \min_j \max_i M_{ij}$$

Lemme. Celui qui joue en premier est désavantagé :

$$V_L := \max_i \min_j M_{ij} \leq \min_j \max_i M_{ij} =: V_C$$

Objectifs et stratégies – exemples

	pierre	ciseaux	feuille
pierre	0	1	-1
ciseaux	-1	0	1
feuille	1	-1	0

tableau de gains classique

	pierre	ciseaux	feuille
pierre	0	1	2
ciseaux	-1	0	1
feuille	-2	-1	0

autre tableau de gains

- Pour le jeu classique, on a $V_L = -1$ et $V_C = 1$
- Pour la variante, on a $V_L = V_C = 0$ et on dit que le jeu a une solution. Ici, les deux joueurs ont intérêt à jouer ”pierre”

Stratégies pures & stratégies mixtes

- Quand un joueur choisit son coup, on dit qu'il choisit une stratégie pure
- Quand un joueur choisit une probabilité sur les coups, on dit qu'il choisit une stratégie mixte

On autorise maintenant les joueurs à présenter une stratégie mixte, qu'on représente par un vecteur de probabilité $\vec{p}$ pour L et $\vec{q}$ pour C.
Par exemple

$$\vec{p} = \left(\frac{1}{2}, \frac{1}{3}, \frac{1}{6}\right)$$

si L décide de jouer "pierre", "ciseaux" et "feuille" avec probas respectives $\frac{1}{2}$, $\frac{1}{3}$ et $\frac{1}{6}$
L'espérance de gain pour des stratégie mixtes $\vec{p}$ et $\vec{q}$ est

$$\mathbb{E}[\text{gain}] = \sum_i \sum_j p_i q_j M_{ij} = \vec{p}^{\,t} M \vec{q}$$

Objectifs et stratégies mixtes

On définit comme avant le gain maximisé si L joue en premier une stratégie mixte par

$$V_L := \max_{\vec{p}} \min_{\vec{q}} \vec{p}^{\,t} M \vec{q}$$

et V_C est le gain minimisé si C joue en premier une stratégie mixte par

$$V_C := \min_{\vec{q}} \max_{\vec{p}} \vec{p}^{\,t} M \vec{q}$$

Objectifs et stratégies mixtes

On définit comme avant le gain maximisé si L joue en premier une stratégie mixte par

$$V_L := \max_{\vec{p}} \min_{\vec{q}} \vec{p}^{\,t} M \vec{q}$$

et V_C est le gain minimisé si C joue en premier une stratégie mixte par

$$V_C := \min_{\vec{q}} \max_{\vec{p}} \vec{p}^{\,t} M \vec{q}$$

Théorème (Minmax de Von Neumann)

Pour tout jeu à somme zéro de matrice de gain M, on a

$$\max_{\vec{p}} \min_{\vec{q}} \vec{p}^{\,t} M \vec{q} = \min_{\vec{q}} \max_{\vec{p}} \vec{p}^{\,t} M \vec{q}$$

Avec les stratégies mixtes, il y a toujours une solution

Variante de Loomis

Si L choisit une stratégie mixte $\vec{p}$ en premier alors $\vec{p}^{\,t} M \vec{q}$ est une combinaison linéaire des q_i :

$$\vec{p}^{\,t} M \vec{q} = \sum_j \lambda_j q_j$$

Pour minimiser cette quantité, C peut simplement choisir de façon déterministe la coordonnée qui minimise λ_j : $\vec{q} = \vec{e}_j$, le vecteur de base correspondant. Ca veut dire que l'espérance de gain, une fois $\vec{p}$ révélée est minimisée par une stratégie pure (déterministe)

Corollaire (Loomis)

Pour tout jeu à somme zéro de matrice de gain M, on a

$$\max_{\vec{p}} \min_j \vec{p}^{\,t} M \vec{e}_j = \min_{\vec{q}} \max_i \vec{e}_i^{\,t} M \vec{q}$$

Retour sur les algorithmes probabilistes

Pour la suite, on va supposer qu'on a un problème dans un modèle tel que pour chaque n :

- il y a un nombre fini d'entrées de taille n
- il y a un nombre fini d'algorithmes déterministes pour résoudre le problème

Remarque : Comme on va vouloir faire des bornes inférieures, on peut assez souvent se ramener à ça (on montre que même si on restreint les entrées possibles et les algos autorisés, on a quand même une borne inférieure)

On fixe n et on note $\mathcal{E}$ l'ensemble des entrées de taille n et $\mathcal{A}$ l'ensemble des algorithmes

Reformulation

On reprend notre terminologie de théorie des jeux avec :

- les lignes sont indexées par les entrées de $\mathcal{E}$
- les colonnes sont indexées par les algorithmes déterministes de $\mathcal{A}$
- la matrice de gain contient en M_{IA} le coût de faire tourner l'algorithme déterministe $A \in \mathcal{A}$ sur l'entrée $E \in \mathcal{E}$

Avec ce formalisme, la complexité pire cas d'un algo déterministe A est $\max_I M_{IA}$. Et si on cherche le meilleur algorithme déterministe pour la complexité pire cas c'est

$$\min_A \max_I M_{IA},$$

soit la meilleure stratégie pure (déterministe) pour Cécile.

Reformulation (cadre probabiliste)

- Une stratégie mixte pour C, c'est une probabilité sur les algorithmes déterministe = un algorithme probabiliste !
- Une stratégie mixte pour L, c'est une probabilité sur les entrées

Corollaire (Principe de Yao)

Soit un problème Π ayant pour tout n un ensemble fini d'entrées $\mathcal{E}$ et un ensemble fini d'algorithmes déterministes $\mathcal{A}$. Soit $\vec{p}$ une probabilité sur $\mathcal{E}$ et $\vec{q}$ une probabilité sur $\mathcal{A}$, alors

$$\min_{A\in\mathcal{A}} \mathbb{E}_{\vec{p}}[C(A, E_{\vec{p}})] \leq \max_{E\in\mathcal{E}} \mathbb{E}_{\vec{q}}[C(A_{\vec{q}}, E)]$$

Preuve. Par Loomis on a

$$\max_{\vec{p}} \min_{A\in\mathcal{A}} \mathbb{E}_{\vec{p}}[C(A, E_{\vec{p}})] = \min_{\vec{q}} \max_{E\in\mathcal{E}} \mathbb{E}_{\vec{q}}[C(A_{\vec{q}}, E)]$$

Interprétation du principe de Yao

Corollaire (Principe de Yao)

Soit un problème Π ayant pour tout n un ensemble fini d'entrées $\mathcal{E}$ et un ensemble fini d'algorithmes déterministes $\mathcal{A}$. Soit $\vec{p}$ une probabilité sur $\mathcal{E}$ et $\vec{q}$ une probabilité sur $\mathcal{A}$, alors

$$\min_{A \in \mathcal{A}} \mathbb{E}_{\vec{p}}[C(A, E_{\vec{p}})] \leq \max_{E \in \mathcal{E}} \mathbb{E}_{\vec{q}}[C(A_{\vec{q}}, E)]$$

- en choisissant $\vec{q}$, on choisit un algo probabiliste pour Π
- la quantité de droite est la complexité de cet algo probabiliste
- en choisissant $\vec{p}$, on choisit une proba sur les entrées
- la quantité de gauche est la la complexité moyenne du meilleur algo déterministe pour cette proba sur les entrées

Interprétation du principe de Yao

Corollaire (Principe de Yao)

Soit un problème Π ayant pour tout n un ensemble fini d'entrées $\mathcal{E}$ et un ensemble fini d'algorithmes déterministes $\mathcal{A}$. Soit $\vec{p}$ une probabilité sur $\mathcal{E}$ et $\vec{q}$ une probabilité sur $\mathcal{A}$, alors

$$\min_{A \in \mathcal{A}} \mathbb{E}_{\vec{p}}[C(A, E_{\vec{p}})] \leq \max_{E \in \mathcal{E}} \mathbb{E}_{\vec{q}}[C(A_{\vec{q}}, E)]$$

Reformulation : pour tout choix de $\vec{p}$, la complexité moyenne du meilleur algorithme déterministe pour une proba $\vec{p}$ sur les entrées est une borne inférieure de complexité probabiliste pour Π

Interprétation du principe de Yao

Corollaire (Principe de Yao)

Soit un problème Π ayant pour tout n un ensemble fini d'entrées $\mathcal{E}$ et un ensemble fini d'algorithmes déterministes $\mathcal{A}$. Soit $\vec{p}$ une probabilité sur $\mathcal{E}$ et $\vec{q}$ une probabilité sur $\mathcal{A}$, alors

$$\min_{A \in \mathcal{A}} \mathbb{E}_{\vec{p}}[C(A, E_{\vec{p}})] \leq \max_{E \in \mathcal{E}} \mathbb{E}_{\vec{q}}[C(A_{\vec{q}}, E)]$$

Reformulation : pour tout choix de $\vec{p}$, la complexité moyenne du meilleur algorithme déterministe pour une proba $\vec{p}$ sur les entrées est une borne inférieure de complexité probabiliste pour Π

Pour ALÉA : c'est super, ça relie l'analyse en moyenne aux algos probabilistes (ou plutôt, aux bornes inférieures probabiliste) !

5. Application : **SORT**

Le problème SORT

Pour un n fixé on prend

- pour $\mathcal{E}$ l'ensemble des permutations de $\{1, \ldots n\}$
- pour $\mathcal{A}$ l'ensemble des arbres de comparaisons, où on ne pose pas deux fois la même question

Remarque : si on obtient une borne inférieure probabiliste dans ce cadre, elle reste valide si on autorise plus d'entrées ou des algorithmes qui répètent les questions

Le problème SORT

Pour un n fixé on prend

- pour $\mathcal{E}$ l'ensemble des permutations de $\{1, \ldots n\}$
- pour $\mathcal{A}$ l'ensemble des arbres de comparaisons, où on ne pose pas deux fois la même question

Remarque : si on obtient une borne inférieure probabiliste dans ce cadre, elle reste valide si on autorise plus d'entrées ou des algorithmes qui répètent les questions

On applique le **principe de Yao** pour la distribution uniforme sur $\mathcal{E}$: "La complexité moyenne du meilleur algorithme déterministe pour la distribution uniforme sur les entrées est une borne inférieure de complexité probabiliste pour **SORT**"

$\Rightarrow$ Il faut chercher l'algorithme déterministe le plus efficace pour la distribution uniforme sur les permutations.

Le problème SORT

$\Rightarrow$ Il faut chercher l'algorithme déterministe le plus efficace pour la distribution uniforme sur les permutations.

- Toute question dans un algorithme optimal A doit être utile : les nœuds internes ont deux fils et il n'y a pas de feuille inutile donc $n!$ feuilles
- La complexité moyenne d'un tel arbre pour la distribution uniforme c'est la moyenne des hauteurs des feuilles
- C'est donc $\frac{1}{n!}\rho(A)$, où $\rho(A)$ est la longueur de cheminement interne de A
- On montre facilement que les arbres qui minimisent la longueur de cheminement interne sont ceux qui sont les plus "tassés" : toutes les feuilles sont à hauteur h ou $h+1$ avec $h \approx \log_2 n!$
- Donc la complexité moyenne pour le meilleur arbre est supérieure à $\frac{1}{n!}n!h \approx \log_2 n!$

Donc choisir la distribution uniforme pour $\vec{p}$ donne une borne inférieure de complexité probabiliste $\Omega(n \log n)$ pour **SORT**

Le problème SORT

Théorème

Si on n'utilise que des comparaisons, le problème **SORT** a une borne inférieure de complexité probabiliste en $\Omega(n \log n)$

Cette borne est optimale, on a même des algos *déterministes* de complexité $\mathcal{O}(n \log n)$

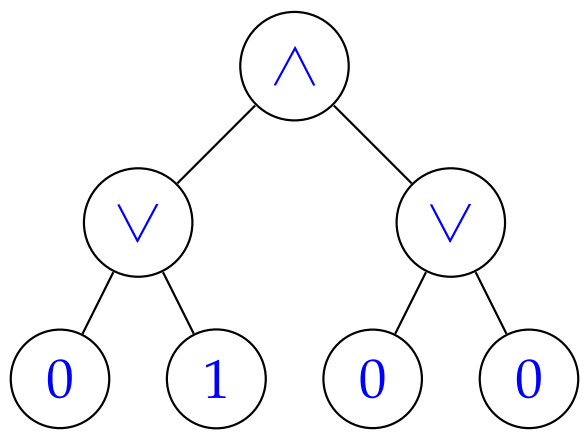
6. Application : évaluation d'arbres de jeu

Les arbres de jeu

Ce n'est pas la même notion de "jeu" que pour le principe Yao

Un arbre de jeu booléen (AJB) est un arbre binaire équilibré de hauteur $2k$: les 2^{2k} feuilles sont toutes à distance $2k$ de la racine. Les nœuds internes à distance paire de la racine sont étiquetés par $\wedge$ et ceux à distance impaire par $\vee$. Les feuilles sont étiquetées soit par vrai (1) soit par faux (0).

On s'intéresse à évaluer la formule, dans le modèle où les questions sont "quelle est la valeur de cette feuille ?" C'est le problème **EVAL**

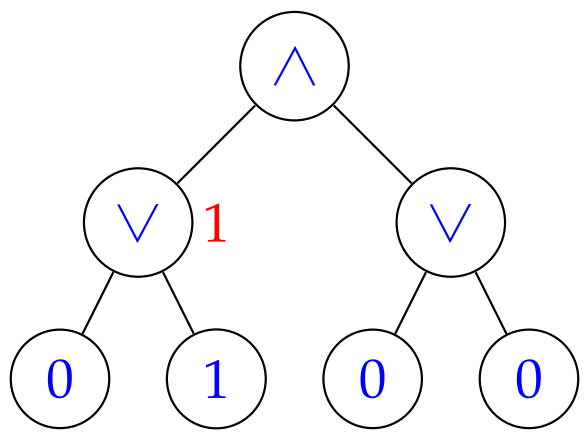


Les arbres de jeu

Ce n'est pas la même notion de "jeu" que pour le principe Yao

Un arbre de jeu booléen (AJB) est un arbre binaire équilibré de hauteur $2k$: les 2^{2k} feuilles sont toutes à distance $2k$ de la racine. Les nœuds internes à distance paire de la racine sont étiquetés par $\wedge$ et ceux à distance impaire par $\vee$. Les feuilles sont étiquetées soit par vrai (1) soit par faux (0).

On s'intéresse à évaluer la formule, dans le modèle où les questions sont "quelle est la valeur de cette feuille ?" C'est le problème **EVAL**

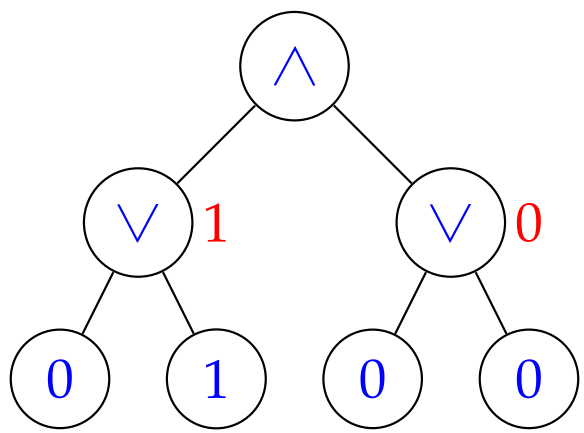


Les arbres de jeu

Ce n'est pas la même notion de "jeu" que pour le principe Yao

Un arbre de jeu booléen (AJB) est un arbre binaire équilibré de hauteur $2k$: les 2^{2k} feuilles sont toutes à distance $2k$ de la racine. Les nœuds internes à distance paire de la racine sont étiquetés par $\wedge$ et ceux à distance impaire par $\vee$. Les feuilles sont étiquetées soit par vrai (1) soit par faux (0).

On s'intéresse à évaluer la formule, dans le modèle où les questions sont "quelle est la valeur de cette feuille ?" C'est le problème **EVAL**

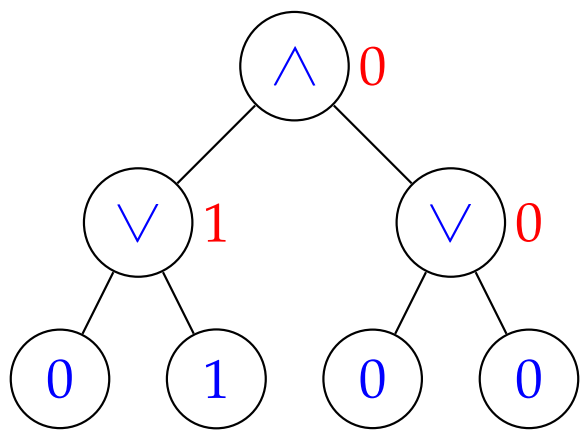


Les arbres de jeu

Ce n'est pas la même notion de "jeu" que pour le principe Yao

Un arbre de jeu booléen (AJB) est un arbre binaire équilibré de hauteur $2k$: les 2^{2k} feuilles sont toutes à distance $2k$ de la racine. Les nœuds internes à distance paire de la racine sont étiquetés par $\wedge$ et ceux à distance impaire par $\vee$. Les feuilles sont étiquetées soit par vrai (1) soit par faux (0).

On s'intéresse à évaluer la formule, dans le modèle où les questions sont "quelle est la valeur de cette feuille ?" C'est le problème **EVAL**

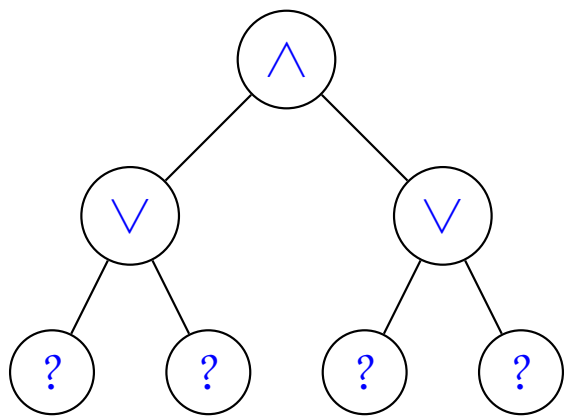


Borne inférieure (déterministe)

Théorème

Tout algorithme déterministe qui résout **EVAL** doit révéler toutes les feuilles dans le pire cas.

Preuve. Exercice ! un argument d'adversaire fonctionne.

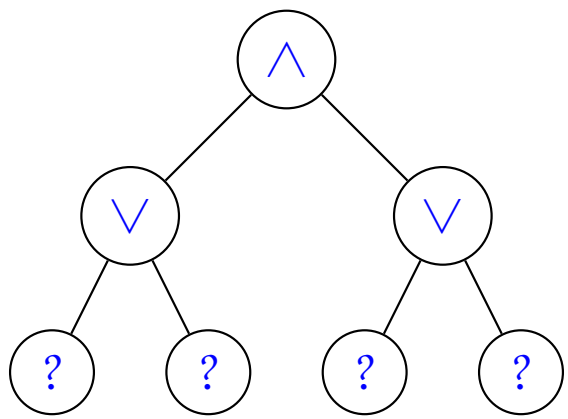


Borne inférieure (déterministe)

Théorème

Tout algorithme déterministe qui résout **EVAL** doit révéler toutes les feuilles dans le pire cas.

Preuve. Exercice ! un argument d'adversaire fonctionne.



Et si on a le droit aux algorithmes probabilistes ?

Un algorithme probabiliste

L'algorithme déterministe qu'on a utilisé sur l'exemple est récursif :

- si on est sur une feuille on retourne sa valeur
- sinon, on calcule récursivement la valeur du sous-arbre gauche et du sous-arbre droit et on combine les résultats avec $\wedge$ ou $\vee$

Remarques :

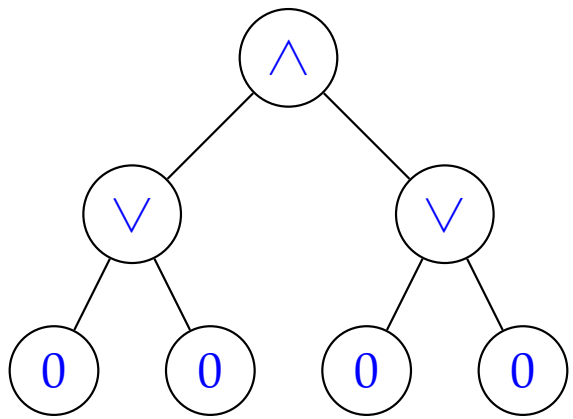
- Si le fils gauche s'évalue à 1 et que le nœud est $\vee$, inutile d'évaluer le fils droit
- Si le fils gauche s'évalue à 0 et que le nœud est $\wedge$, inutile d'évaluer le fils droit

Algorithme `random_eval` :

- On fait comme l'algo déterministe, sauf qu'on tire au sort uniformément par quel fils on commence
- On n'évalue pas l'autre fils si ce n'est pas nécessaire.

Complexité de `random_eval`

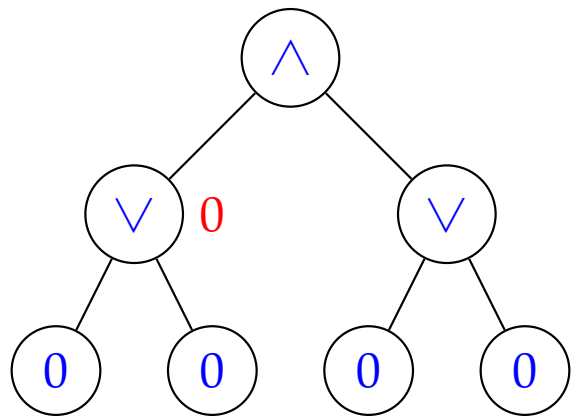
Si on regarde juste l'avant-dernier niveau on ne voit pas le gain sur les instances où toutes les feuilles sont à 0 : il faut évaluer les deux feuilles d'un nœud $\vee$ si elle sont à 0, quelque soit l'ordre dans lequel on les prend



Mais si on regarde deux niveaux, on voit apparaître un gain en moyenne : la situation précédente fait que le $\vee$ à gauche vaut 0, du coup, inutile d'évaluer le sous-arbre droit (et même chose en symétrique si on commence à droite), etc.

Complexité de `random_eval`

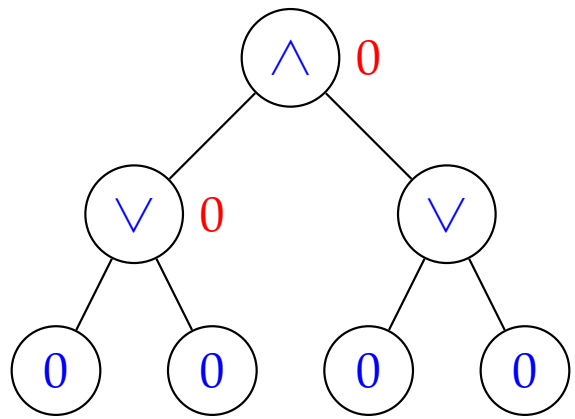
Si on regarde juste l'avant-dernier niveau on ne voit pas le gain sur les instances où toutes les feuilles sont à 0 : il faut évaluer les deux feuilles d'un nœud $\vee$ si elle sont à 0, quelque soit l'ordre dans lequel on les prend



Mais si on regarde deux niveaux, on voit apparaître un gain en moyenne : la situation précédente fait que le $\vee$ à gauche vaut 0, du coup, inutile d'évaluer le sous-arbre droit (et même chose en symétrique si on commence à droite), etc.

Complexité de `random_eval`

Si on regarde juste l'avant-dernier niveau on ne voit pas le gain sur les instances où toutes les feuilles sont à 0 : il faut évaluer les deux feuilles d'un nœud $\vee$ si elle sont à 0, quelque soit l'ordre dans lequel on les prend

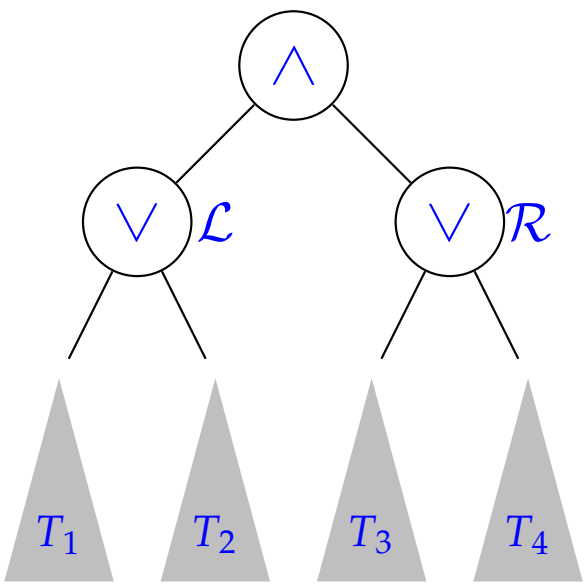


Mais si on regarde deux niveaux, on voit apparaître un gain en moyenne : la situation précédente fait que le $\vee$ à gauche vaut 0, du coup, inutile d'évaluer le sous-arbre droit (et même chose en symétrique si on commence à droite), etc.

Complexité de `random_eval`

Proposition

Pour tout AJB de hauteur de $2k$, le nombre moyen de feuilles révélées par `rand_eval` est au plus 3^k. La complexité de cet algorithme est donc en $\mathcal{O}(n^{\log_4 3})$, avec $\log_4 3 \approx 0.79$



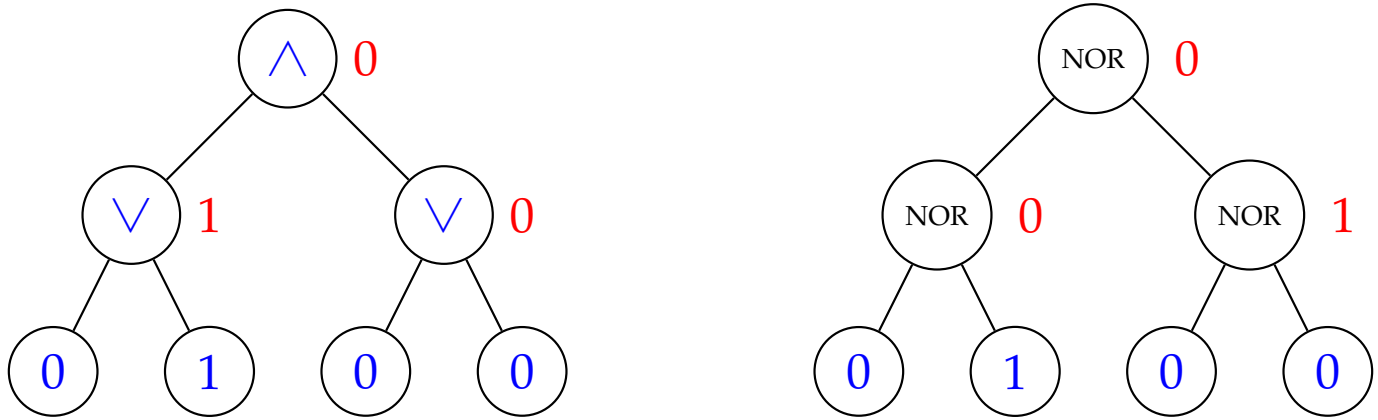
Par récurrence sur k :

- Si $eval(\mathcal{L}) = eval(\mathcal{R}) = 0$ l'algorithme va n'en évaluer qu'un des deux, pour un coût moyen au plus $2 \cdot 3^{k-1} \leq 3^k$
- Si $eval(\mathcal{L}) = 0$ et $eval(\mathcal{R}) = 1$ on a une proba $\frac{1}{2}$ de n'évaluer que $\mathcal{L}$ pour un coût moyen d'au plus $2 \cdot 3^{k-1}$ et une proba $\frac{1}{2}$ d'évaluer 3 ou 4 sous-arbres pour un coût moyen au plus $4 \cdot 3^{k-1}$. Au total ça fait une borne en $\frac{1}{2}(2 \cdot 3^{k-1} + 4 \cdot 3^{k-1}) = 3^k$
- *etc*

Passage au NOR

On définit NOR par $x \,\mathrm{NOR}\, y = 1 \Leftrightarrow x = y = 0$.

On peut placer l'opérateur NOR à la place de $\vee$ et des $\wedge$: cela calcule la même chose à la racine



$\alpha_0(h) =$ le max de la complexité de l'algo pour des arbres NOR de hauteur h qui s'évaluent à 0 ($\alpha_1(h)$ pour ceux qui s'évaluent à 1). On a

$$\begin{cases} \alpha_1(h) \leq 2\alpha_0(h-1) \\ \alpha_0(h) \leq \alpha_1(h-1) + \frac{1}{2}\alpha_0(h-1) \end{cases}$$

Amélioration du calcul

$$\begin{cases} \alpha_1(h) \leq 2\alpha_0(h-1) \\ \alpha_0(h) \leq \alpha_1(h-1) + \frac{1}{2}\alpha_0(h-1) \end{cases}$$

Cela implique la proposition suivante.

Proposition

La complexité de `rand_eval` pour un AJB à n feuilles est en $\mathcal{O}(n^\lambda)$, avec

$$\lambda = \log_2\left(\frac{1+\sqrt{33}}{4}\right) \approx 0.753$$

Amélioration du calcul

$$\begin{cases} \alpha_1(h) \leq 2\alpha_0(h-1) \\ \alpha_0(h) \leq \alpha_1(h-1) + \frac{1}{2}\alpha_0(h-1) \end{cases}$$

Cela implique la proposition suivante.

Proposition

La complexité de `rand_eval` pour un AJB à n feuilles est en $\mathcal{O}(n^\lambda)$, avec

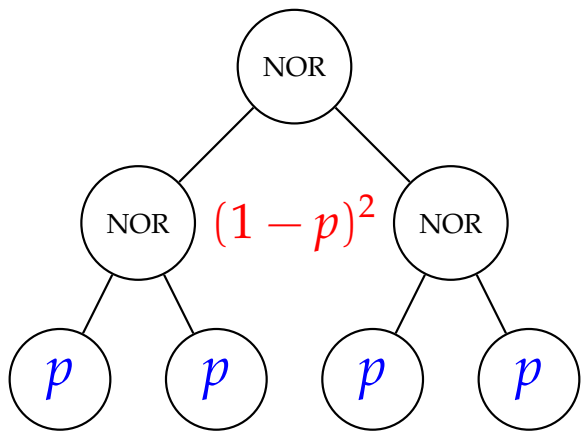
$$\lambda = \log_2\left(\frac{1+\sqrt{33}}{4}\right) \approx 0.753$$

Questions :

- Est-ce la bonne complexité ?
- Peut-on trouver un meilleur algorithme probabiliste ?

Borne inférieure probabiliste

- On veut utiliser le principe de Yao pour établir une borne inférieure probabiliste
- On choisit que chaque feuille vaut 1 avec probabilité $p = \frac{3-\sqrt{5}}{2}$, indépendamment des autres (on a $p = (1-p)^2$)



Le principe de Yao nous dit que la complexité moyenne du meilleur algorithme déterministe, pour cette distribution, est une borne inférieure probabiliste.

Meilleur algorithme déterministe?

On retrouve la difficulté d'identifier le meilleur algorithme. On admet le résultat suivant :

Théorème (Saks et Wigderson 1986)

Si chacune des feuilles vaut 1 avec probabilité $p \in (0,1)$, indépendamment, alors l'algorithme de parcours en profondeur avec élagage l'évalue de façon optimale pour la complexité en moyenne.

C'est l'algorithme qu'on avait vu au début : on évalue récursivement les deux sous-arbres puis on applique l'opération. L'élagage consiste à ne pas évaluer le deuxième sous-arbre si l'évaluation du premier permet de conclure.

Borne inférieure probabiliste

On n'a plus qu'à faire les calculs pour cet algorithme : soit $c(h)$ le coût moyen pour notre distribution sur un arbre de hauteur h, on a

$$c_h = c_{h-1} + (1-p)c_{h-1} = (2-p)c_{h-1}$$

Donc $c_h = (2-p)^h$ et comme il y a $n = 2^h$ feuilles, on trouve par Yao :

Théorème

Le problème **EVAL** admet une borne inférieure probabiliste en $\Omega(n^\gamma)$ avec $\gamma = \log_2(\frac{1+\sqrt{5}}{2}) \approx 0.694$

Récapitulatif

Pour le problème **EVAL** :

- On a un algorithme probabiliste de complexité $\mathcal{O}(n^{\lambda})$ avec $\lambda = \log_2\left(\frac{1+\sqrt{33}}{4}\right) \approx 0.753$
- Si on prend, pour le principe de Yao, les valeurs des feuilles i.i.d. Bernoulli de paramètre p, on obtient une borne inférieure probabiliste de $\Omega(n^{\gamma})$, avec $\gamma = \log_2(\frac{1+\sqrt{5}}{2}) \approx 0.694$

Récapitulatif

Pour le problème **EVAL** :

- On a un algorithme probabiliste de complexité $\mathcal{O}(n^{\lambda})$ avec $\lambda = \log_2\left(\frac{1+\sqrt{33}}{4}\right) \approx 0.753$
- Si on prend, pour le principe de Yao, les valeurs des feuilles i.i.d. Bernoulli de paramètre p, on obtient une borne inférieure probabiliste de $\Omega(n^{\gamma})$, avec $\gamma = \log_2(\frac{1+\sqrt{5}}{2}) \approx 0.694$
- Si on change la valeur de p ? on montre facilement que la borne obtenue avec le principe de Yao est moins bonne

Récapitulatif

Pour le problème **EVAL** :

- On a un algorithme probabiliste de complexité $\mathcal{O}(n^{\lambda})$ avec $\lambda = \log_2\left(\frac{1+\sqrt{33}}{4}\right) \approx 0.753$
- Si on prend, pour le principe de Yao, les valeurs des feuilles i.i.d. Bernoulli de paramètre p, on obtient une borne inférieure probabiliste de $\Omega(n^{\gamma})$, avec $\gamma = \log_2(\frac{1+\sqrt{5}}{2}) \approx 0.694$
- Si on change la valeur de p ? on montre facilement que la borne obtenue avec le principe de Yao est moins bonne
- Si on ne choisit pas les valeurs des feuilles de façon indépendantes ? ...

Récapitulatif

Pour le problème **EVAL** :

- On a un algorithme probabiliste de complexité $\mathcal{O}(n^\lambda)$ avec $\lambda = \log_2\left(\frac{1+\sqrt{33}}{4}\right) \approx 0.753$
- Si on prend, pour le principe de Yao, les valeurs des feuilles i.i.d. Bernoulli de paramètre p, on obtient une borne inférieure probabiliste de $\Omega(n^\gamma)$, avec $\gamma = \log_2(\frac{1+\sqrt{5}}{2}) \approx 0.694$
- Si on change la valeur de p ? on montre facilement que la borne obtenue avec le principe de Yao est moins bonne
- Si on ne choisit pas les valeurs des feuilles de façon indépendantes ? ...

Théorème (Saks et Wigderson 1986)

Le problème **EVAL** admet une borne inférieure probabiliste en $\Omega(n^\lambda)$ avec $\lambda = \log_2\left(\frac{1+\sqrt{33}}{4}\right) \approx 0.753$. L'algorithme probabiliste `random_eval` est optimal

C’est fini, merci !