Devoir maison facultatif : L'inégalité de Kraft

L'idée de ce projet est de faire la preuve de l'inégalité de Kraft. Une partie de la preuve se fait de manière constructive, ce qui permet de dériver un algorithme pour construire des arbres de décodage. Vous pouvez ensuite implanter cet algorithme dans votre langage de programmation préféré (cet aspect n'est pas approfondi ici, mais votre solution sera appréciée ...)

1 Aspects administratifs

Devoir maison : Vous êtes autorisés (voire encouragés) à travailler en petits groupes, jusqu'à trois personnes. Dans ce cas, vous devez indiquer sur votre copie les noms de vos collaborateurs. La solution rendue doit être manuscrite et individuelle (pas de photocopie, scan etc.), mais les contenus des solutions peuvent être identiques pour les membres d'un groupe. Chaque membre d'un groupe doit être capable d'expliquer l'intégralité des solutions sur sa copie. Vous pouvez donner votre solution à votre enseignant de cours ou de TP.

Nous sommes conscients que ce devoir maison demande un effort. N'hésitez pas à nous solliciter (directement ou par mail) pour des renseignements plus détaillés ou pour discuter de vos solutions préliminaires. Nous sommes aussi conscients que vous n'avez pas encore fait beaucoup de preuves "longues". Surtout, n'hésitez pas à nous contacter si vous pensez que votre preuve est toute fausse - on pourra vous aider !

2 Devoir Maison : Preuve de l'inégalité de Kraft

2.1 Énoncé

Le but des exercices suivants est de faire une preuve de l'inégalité de Kraft présentée en cours. Nous rappelons que l'inégalité de Kraft donne des conditions nécessaires et suffisantes pour l'existence d'un code préfixe binaire étant donnée la longueur des mots du code (voir pp. 45, 46 des transparents). Nous nous facilitons la tâche :

- nous faisons l'amalgame entre un code préfixe binaire et sa représentation comme arbre binaire de décodage. Ceci est justifié par les traductions sur les pp. 43, 44 des transparents.
- nous considérons uniquement des codes préfixes binaires *complets*, qui sont caractérisés par le fait que toute feuille de l'arbre correspond à un code. Nous verrons que dans ce cas, l'inégalité de Kraft devient une égalité.

Pour les deux exercices suivants, utilisez la méthode informelle pratiquée en cours. Nous développerons plus tard un algorithme formel que vous pouvez tester avec ces exemples.

Exercice 1 Construisez un code préfixe pour 6 mots de code avec des longueurs $n_1 = 2, n_2 = 2, n_3 = 2, n_4 = 3, n_5 = 4, n_6 = 4$.

Exercice 2 Construisez un code préfixe pour 5 mots de code avec des longueurs $n_1 = 2, n_2 = 2, n_3 = 3, n_4 = 4, n_5 = 4$. Dans ce cas, vous voyez que l'arbre est incomplet. Quel mot (chemin dans l'arbre) ne correspond pas à un code ?

Théorème 1 (Kraft) *Il existe un arbre de décodage binaire complet pour des mots de code de taille* $n_1, \ldots, n_k$ *si et seulement si*

$$\sum_{i=1}^{k} 2^{-n_i} = 1$$

2.2 Preuve

Une preuve est un argument rigoureux qui s'appuie sur des principes de raisonnement universellement reconnus comme valides. Insister sur des preuves ne nie pas l'utilité d'un exemple ou d'une illustration intuitive, mais ces derniers sont fallacieux : un exemple ne peut pas couvrir tous les cas de figure, et dans une illustration, on a tendance à oublier des cas déplaisants.

Dans vos études de mathématique et informatique, vous trouverez souvent des énoncés en langage naturel qui cachent une terminologie précise que vous devez apprendre à analyser, pour pouvoir appliquer des principes de raisonnement adéquats. Nous vous présenterons dans la suite des prototypes de cette terminologie et des méthodes de preuve typiques correspondantes. "Typiques", parce qu'il y a souvent plusieurs manières de faire une preuve. Tout au long de vos études, vous allez découvrir un arsenal de méthodes, et c'est à vous d'appliquer la plus appropriée.

Principe 1 (si et seulement si) *Le théorème 1 a la forme "P si et seulement si Q", où P et Q sont d'autres énoncés. Une écriture plus formelle est $P \leftrightarrow Q$. Une démonstration typique consiste à prouver que si P, alors Q et si Q, alors P ; formellement : $P \longrightarrow Q$ et $Q \longrightarrow P$.*

Nous appliquons ce principe, nous réduisons la démonstration du théorème 1 à la preuve des lemmes 1 et 3.

2.2.1 Condition nécessaire

Dans cette partie, nous prouvons la direction "seulement si" du théorème 1 (condition nécessaire pour l'existence d'un arbre de décodage).

Lemme 1 *S'il existe un arbre de décodage binaire complet pour des mots de code de taille $n_1, \ldots, n_k$, alors*

$$\sum_{i=1}^{k} 2^{-n_i} = 1$$

Nous avons maintenant une hypothèse de la forme "il existe un arbre ...".

Principe 2 (hypothèse "il existe") *Si on a une hypothèse de la forme "il existe une instance ... telle que ...", formellement $\exists x.P(x)$, on peut fixer cette instance et la désigner par un nom, par exemple a, et montrer qu'elle a cette propriété : $P(a)$*

Pour préciser davantage l'énoncé, nous définissons une fonction `codes` (que nous expliquerons dans la suite), et nous écrivons le lemme 1 comme suit :

Lemme 2 *Soit a un arbre de décodage binaire complet avec `codes(a) = [`$c_1 \ldots c_k$`]` et $|c_1| = n_1$, ..., $|c_k| = n_k$. Alors*

$$\sum_{i=1}^{k} 2^{-n_i} = 1$$

Nous écrivons $|\ldots|$ pour la longueur d'une chaîne de caractères, par exemple $|"110"| = 3$.

La fonction `codes`

Définition 1 (Fonction `codes`)

```
codes (L()) = [ "" ]
codes (N((), g, d)) =  prefixer(0, codes (g)) @ prefixer(1, codes (d))
```

Pour comprendre la fonction `codes`, tout d'abord quelques rappels terminologiques : Nous écrivons `[]` pour la séquence vide, et `@` pour concaténer deux séquences (c.à.d., les joindre en les mettant l'une après l'autre). La fonction `codes` prend un arbre binaire et renvoie une séquence de ses mots (chaîne de caractères) de code. Elle est définie de manière récursive sur la structure d'un arbre. La fonction auxiliaire `prefixer` attache un caractère devant chaque chaîne d'une séquence. Par exemple : `prefixer(0, ["01", "11"]) = ["001", "011"]` et `prefixer(1, ["", "10", "011"]) = ["1", "110", "1011"]`.

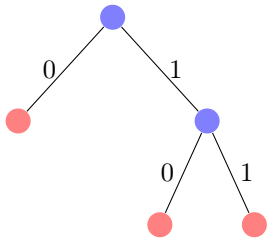

FIGURE 1: Arbre de décodage

Exercice 3

1. Représentez l'arbre de la figure 1 comme expression de ses constructeurs `L` et `N`.
2. Tracez l'exécution de la fonction `codes` sur cet arbre.

Preuve du lemme 2 Le lemme 2 parle d'une propriété E d'un arbre quelconque a. On peut prouver que $E(a)$ par induction sur la structure de l'arbre. C'est le principe d'induction sur les nombres naturels étendu aux arbres. Il exprime que si une propriété est vraie pour toutes les feuilles et préservée lorsqu'on construit des arbres de plus en plus grands, alors elle est vraie pour tout arbre.

Principe 3 (Induction sur la structure d'un arbre) *Pour prouver $E(a)$:*

- *on montre que E est vraie pour toute feuille : $E(\texttt{L}())$*
- *on montre que si E est vrai pour tout sous-arbre gauche g et droit d, alors E reste vrai pour tout arbre construit avec g et d. Donc : Pour tout g et d, si $E(g)$ et $E(d)$ sont vraies, alors aussi $E(\texttt{N}((), g, d))$*

Exercice 4 Pour avoir une intuition du raisonnement inductif et avant de faire la preuve, considérons l'exemple de l'arbre de la figure 1. Calculez :

1. la liste des codes du sous-arbre gauche, et les longueurs des codes respectifs,
2. la liste des codes du sous-arbre droit, et les longueurs des codes respectifs,
3. la liste des codes de l'arbre tout entier, et les longueurs des codes respectifs.

Vérifiez que la formule de sommation $\sum_i 2^{-n_i} = 1$ est satisfaite dans les trois cas. Comment obtenez vous le résultat pour (3) à partir des résultats pour (1) et (2) ?

Faisons maintenant la preuve en toute généralité.

Exercice 5 (Cas de base) L'arbre a du lemme 2 a la forme `L()`.
Montrez que si `codes(L()) = [`$c_1 \dots c_k$`]` et $|c_1| = n_1, \dots, |c_k| = n_k$, alors

$$\sum_{i=1}^{k} 2^{-n_i} = 1$$

Remarque : Appliquez rigoureusement la définition de `codes` pour ce cas. Vous verrez que la liste $[c_1 \dots c_k]$ peut seulement avoir une forme très spécifique. Tirez la conclusion pour les valeurs $n_1 \dots n_k$, ce qui vous permet de vérifier l'égalité.

Exercice 6 (Cas inductif) L'arbre a du lemme 2 a la forme `N(), g, d)`, où g et d sont des arbres.

Notons donc les deux hypothèses d'induction :

— pour le sous-arbre gauche : Soit `codes(`g`) = [`$c_1 \dots c_{kg}$`]` et $|c_1| = n_1$, …, $|c_{kg}| = n_{kg}$. Alors $\sum_{i=1}^{kg} 2^{-n_i} = 1$

— pour le sous-arbre droit : Soit `codes(`d`) = [`$c'_1 \dots c'_{kd}$`]` et $|c'_1| = n'_1$, …, $|c'_{kd}| = n'_{kg}$. Alors $\sum_{i=1}^{kd} 2^{-n'_i} = 1$

Sous ces hypothèses, montrez que si `codes(N(), g, d)) = [`$c''_1 \dots c''_k$`]` et $|c''_1| = n''_1$, …, $|c''_k| = n''_k$, alors

$$\sum_{i=1}^{k} 2^{-n''_i} = 1$$

Remarque : Appliquez d'abord rigoureusement la définition de `codes` pour ce cas. Si les éléments de la liste `codes(`g`)` ont respectivement la longueur n_1, …, n_{kg}, quelles sont les longueurs de `prefixer(0, codes (`g`))` ? Question similaire pour `codes(`d`)` et `prefixer(1, codes (`d`))`. Combinez ces résultats avec les hypothèses d'induction pour compléter la preuve.

Les exercices 5 et 6 complètent la preuve du lemme 2 et par conséquent du lemme 1. Nous avons donc montré la partie "seulement si" du théorème 1.

2.2.2 Condition suffisante

Dans cette partie, nous prouvons la direction "si" du théorème 1 (condition suffisante pour l'existence d'un arbre de décodage).

Lemme 3 *Si*

$$\sum_{i=1}^{k} 2^{-n_i} = 1$$

alors il existe un arbre de décodage binaire complet pour des mots de code de taille $n_1, \dots, n_k$.

2.2.3 Une preuve constructive

Il est souvent plus facile de résoudre une généralisation d'un problème que le problème original. Ainsi, nous allons démontrer :

Lemme 4 *Si*

$$\sum_{i=1}^{k} 2^{-n_i} = 2^{-p}$$

alors il existe un arbre de décodage binaire complet pour des mots de code de taille $(n_1 - p), \dots, (n_k - p)$.

Intuitivement, le p du lemme correspond à la profondeur du sous-arbre que nous sommes en train de construire.

Exercice 7 Montrez que le lemme 3 est une instance du lemme 4. Il suffit donc de prouver le lemme 4.

Pour faire la preuve, nous utilisons le principe suivant.

Principe 4 (conclusion "il existe") *Si la conclusion a la forme "il existe une instance ... telle que ...", du style $\exists y.Q(y)$, il faut trouver une instance i telle que $Q(i)$.*

Souvent, on a un énonce "pour tout x qui satisfait $P(x)$, il existe un y tel que $Q(y)$"; plus formellement : $\forall x.(P(x) \longrightarrow \exists y.Q(y))$. L'instance i à trouver dépend alors de x. Une preuve constructive *consiste à donner une fonction f qui prend x en paramètre et construit le i recherché : si $P(x)$ est vrai, alors aussi $Q(f(x))$.*

Bien sûr, il ne faut pas confondre la situation où la quantification existentielle est une hypothèse (principe 2) et celle où la quantification existentielle est une conclusion (principe 4).

Appliqué au lemme 4, nous allons développer une fonction `constr` qui prend en entrée la profondeur p et la liste des longueurs $[n_1, \ldots, n_k]$ et qui calcule l'arbre de décodage. Le lemme 4 peut donc être écrit comme suit :

Lemme 5 *Si*

$$\sum_{i=1}^{k} 2^{-n_i} = 2^{-p}$$

alors `constr(`p`,`$[n_1, \ldots, n_k]$`)` *est un arbre de décodage binaire complet pour des mots de code de taille $(n_1 - p), \ldots, (n_k - p)$.*

2.2.4 La fonction `constr`

Voici la définition de la fonction `constr`.

Définition 2 (Fonction `constr`)

```
constr (p, [n]) = L()
constr (p, ns) =  N((), constr (p + 1, nsg), constr (p + 1, nsd))
      si (nsg, nsd) = partition (p + 1, ns)
```

La lecture de la définition est la suivante : la fonction a deux cas :

- celui d'une liste à un élément `[n]` (donc, $k = 1$), et dans ce cas, nous construisons une feuille ;
- celui d'une liste `ns` avec $k > 1$ éléments. Dans ce cas, on partitionne `ns` en deux sous-listes `nsg` et `nsd` pour construire récursivement les sous-arbres gauche et droit.

Nous revenons sur les détails de la fonction auxiliaire `partition` plus tard, mais vous avez déjà une intuition de ce qu'elle doit faire.

Exercice 8 Tracez l'exécution de la fonction `constr` pour la liste des longueurs de l'exercice 1. Autrement dit, refaite la construction de l'arbre de l'exercice 1, mais cette fois ci de manière plus formelle avec la fonction `constr`.

Nous pouvons déjà faire la preuve du lemme 5 (c'est à dire, de la correction de la fonction `constr`), même sans avoir défini `partition`. Par contre, la preuve nous aidera à identifier les propriétés que `partition` doit satisfaire. L'argument de la preuve est le suivant : si `constr` est correcte pour toute liste moins longue que `ns`, alors `constr` est aussi correcte pour `ns`. Cet argument suit le principe de l'induction bien fondée, que nous énonçons ici pour les nombres naturels.

Principe 5 (Induction bien fondée sur $\mathbb{N}$) *Pour prouver que $P(k)$ pour tout $k \in N$, on montre que pour tout k, $(\forall j < k.P(j)) \longrightarrow P(k)$. Ici, $(\forall j < k.P(j))$ est* l'hypothèse d'induction.

Autrement dit, P est satisfait partout, pourvu que pour tout k, on hérite la propriété P de tous les $j < k$. L'induction "classique" sur les nombres naturels (passage de $P(k)$ à $P(k+1)$) ne suffit pas dans notre cas,

parce que les appels récursifs de `constr` font décroître arbitrairement la taille des listes (et pas seulement d'un seul élément). [1]

Appliquons ce principe pour réduire le lemme 5 au lemme suivant :

Lemme 6 *Hypothèse d'induction : Supposons que pour tout $j < k$, et pour tout p',*

$$\sum_{i=1}^{j} 2^{-n_i} = 2^{-p'}$$

implique que `constr(`p'`,`$[n_1, \ldots, n_j]$`)` *est un arbre de décodage binaire complet pour des mots de code de taille $(n_1 - p'), \ldots, (n_j - p')$.*

Soit la précondition (Pre) :

$$\sum_{i=1}^{k} 2^{-n_i} = 2^{-p}$$

Alors, la conclusion $(Concl)$ est vraie :
`constr(`p`,`$[n_1, \ldots, n_k]$`)` *est un arbre de décodage binaire complet pour des mots de code de taille $(n_1 - p), \ldots, (n_j - p)$.*

Regardons trois situations pour le k du lemme 6.

Exercice 9 Soit $k = 0$, la liste $[n_1, \ldots, n_k]$ est donc vide. Montrez que le lemme 6 est vrai dans ce cas.

Remarque : regardez la précondition (Pre) :

$$\sum_{i=1}^{k} 2^{-n_i} = 2^{-p}$$

pour $k = 0$, et appliquez le principe "ex falso quodlibet" en bas. Ce résultat justifie pourquoi nous n'avons pas de cas pour la liste vide dans la définition de `constr`.

Principe 6 (Ex falso quodlibet) *Du faux, on peut tirer n'importe quelle conclusion : Si, d'une précondition Pre, on veut tirer une conclusion $Concl$ (on veut donc montrer $Pre \longrightarrow Concl$), et la précondition Pre est fausse, alors l'implication $Pre \longrightarrow Concl$ est correcte.*

Exercice 10 Soit $k = 1$. Montrez le lemme 6 dans ce cas.

Remarque : Instanciez le lemme, et appliquez la définition de `constr` pour le cas d'une liste singleton, et montrez que vous obtenez un arbre binaire complet avec des codes de la taille souhaitée.

Avant de considérer le troisième cas, nous introduisons une simplification notationnelle : nous écrivons $\sum_{n_i \in [n_1, \ldots n_k]} 2^{-n_i}$ pour la somme $2^{-n_1} + \ldots 2^{-n_k}$ Par exemple, $\sum_{n_i \in [1,2,4]} 2^{-n_i} = 2^{-1} + 2^{-2} + 2^{-4}$

Exercice 11 Soit $k > 1$. Montrez le lemme 6 dans ce cas.

Remarque : Supposez que sous la condition (Pre) du lemme, le couple (nsg, nsd) calculé par $partition(p+1, [n_1 \ldots n_k])$ satisfait les trois conditions :

1. $ns = nsg@nsd$
2. $\sum_{n_i \in nsg} 2^{-n_i} = 2^{-(p+1)}$
3. $\sum_{n_i \in nsd} 2^{-n_i} = 2^{-(p+1)}$

Appliquez l'hypothèse d'induction du lemme 6, une fois pour la liste nsg, une fois pour la liste nsd, pour inférer des propriétés des appels récursifs `constr(p+1, nsg)` et `constr(p+1, nsd)`. Utilisez enfin la définition récursive de `constr` pour tirer la conclusion $(Concl)$ du lemme 6.

Pour résumer : nous avons prouvé le lemme 6 en considérant toutes les possibilités pour la variable k.

1. L'article de Wikipédia http://fr.wikipedia.org/wiki/Raisonnement_par_récurrence contient une discussion excellente des différentes formes d'induction.

2.2.5 La fonction `partition`

Pour finir, nous développons une fonction `partition` qui satisfait les conditions énoncées dans l'exercice 11.

Exercice 12 Regardons le cas $p = 0$ et la liste $ns = [1, 2, 2]$. Elle sera partitionné en $nsg = [1]$ et $nsd = [2, 2]$.
Vérifiez que la précondition $\sum_{n_i \in [1,2,2]} 2^{-n_i} = 2^{-p}$ du lemme 6 est satisfaite.
Vérifiez en plus les trois conditions de l'exercice 11.

On ne peut pas scinder toute liste en deux sous-listes dont les sommes des valeurs sont égales. Comme nous venons de voir, c'est possible pour $[\frac{1}{2}, \frac{1}{4}, \frac{1}{4}]$ (coupée en $[\frac{1}{2}]$ et $[\frac{1}{4}, \frac{1}{4}]$), mais pas pour $[\frac{1}{3}, \frac{1}{3}, \frac{1}{4}, \frac{1}{12}]$.

Exercice 13 Regardons le cas $p = 0$ et la liste $ns = [2, 1, 2]$. Montrez que vous ne pouvez pas la partitionner pour satisfaire toutes les trois conditions de l'exercice 11. Quelle condition n'est pas satisfaite ? (Attention : l'ordre des éléments est important - ce n'est pas le même exercice que l'exercice 12).

Ces arguments montrent que l'existence de la fonction `partition` n'est pas du tout évidente. L'exercice 13 suggère qu'il faut rajouter un critère : La liste à partitionner doit être triée en ordre croissant ; et il est essentiel de travailler avec des puissances (négatives) de 2. Résumons la spécification de `partition` dans le lemme suivant :

Lemme 7 *Soit $ns := [n_1, \ldots, n_k]$ une liste triée en ordre croissant telle que $\sum_{i=1}^{k} 2^{-n_i} = 2^{-p}$. Alors, si $(nsg, nsd) =$ `partition`$(p + 1, [n_1 \ldots n_k])$, les conditions suivantes sont satisfaites :*

1. $ns = nsg@nsd$
2. $\sum_{n_i \in nsg} 2^{-n_i} = 2^{-(p+1)}$
3. $\sum_{n_i \in nsd} 2^{-n_i} = 2^{-(p+1)}$

Nous ne prouvons pas formellement ce lemme. Dans l'exercice **??**, vous allez implanter cette fonction avec une boucle dans laquelle vous cherchez une position m telle que

$$\sum_{n_i \in [n_1, \ldots, n_{m-1}]} 2^{-n_i} = \sum_{n_i \in [n_m, \ldots, n_k]} 2^{-n_i} = 2^{-(p+1)}$$

et avec cette position, vous obtenez immédiatement nsg et nsd (pourquoi ?). Un *invariant* de la boucle est la condition suivante :

Lemme 8 *Soit $[n_1, \ldots, n_k]$ une liste triée en ordre croissant telle que $\sum_{i=1}^{k} 2^{-n_i} = 2^{-p}$. Pour tout $m < k$, si $\sum_{i=1}^{m} 2^{-n_i} < 2^{-(p+1)}$, alors $\sum_{i=1}^{m+1} 2^{-n_i} \leq 2^{-(p+1)}$*

Ce lemme dit que si vous cherchez la position m telle que la somme est égale à la borne $2^{-(p+1)}$ et vous n'avez pas encore atteint la borne, vous pouvez avancer m d'une position sans dépasser la borne.

Exercice 14 Pour la liste $[2, 1, 2]$ (non triée !) et pour $p = 0$, donnez une position m où le lemme 8 est faux. Ceci montre que la condition de tri est essentiel.

Exercice 15 Prouvez le lemme 8.