Matthieu Boyer
Planche A

Grammaires Context-Free Probabilistes (pCFG)

Une grammaire context-free probabiliste (pCFG) est un quadruplet $G = (V, \Sigma, R, S, p)$ où :

- V est un ensemble fini de symboles non-terminaux
- Σ est un ensemble fini de symboles terminaux (l'alphabet)
- $S \in V$ est le symbole de départ
- R est un ensemble fini de règles de production de la forme $A \to \alpha$ où $A \in V$ et $\alpha \in (V \cup \Sigma)^*$
- $p : R \to [0, 1]$ est une fonction qui assigne une probabilité à chaque règle

Pour tout non-terminal $A \in V$, on impose : $\sum_{A \to \alpha \in R} p(A \to \alpha) = 1$.
Un arbre de parsing (ou arbre de dérivation) pour un mot $w \in \Sigma^*$ est un arbre dont :

- la racine est étiquetée par S
- les feuilles, lues de gauche à droite, forment le mot w
- chaque nœud interne étiqueté par A avec des enfants étiquetés $X_1, \ldots, X_k$ correspond à l'application d'une règle $A \to X_1 \cdots X_k \in R$

Soit τ un arbre de parsing. La probabilité de τ est définie par :

$$P(\tau) = \prod_{r \in \text{règles}(\tau)} p(r)$$

où règles(τ) est l'ensemble des règles utilisées dans τ (avec multiplicité).

On représente une règle de production et une pCFG par les types `type regle = symbole * symbole list;; type pcfg = (regle * float) list`.

1. Donner un type qui représente un arbre de parsing en OCaml.
2. Écrire une fonction `extraire_regle : arbre -> regle option` qui extrait la règle de production utilisée à la racine d'un arbre. Renvoyer `None` si l'arbre est une feuille.
3. Écrire une fonction `toutes_regles : arbre -> regle list` qui renvoie la liste de toutes les règles utilisées dans un arbre (avec répétitions possibles).
4. Écrire une fonction `chercher_prob : pcfg -> regle -> float option` qui cherche la probabilité d'une règle dans une pCFG. Renvoyer `None` si la règle n'existe pas.
5. Écrire une fonction `probabilite_arbre : pcfg -> arbre -> float option` qui calcule la probabilité d'un arbre de parsing selon une pCFG donnée. Renvoyer `None` si une règle utilisée n'existe pas dans la grammaire.

Un mot w peut avoir plusieurs arbres de parsing différents. La probabilité du mot est :

$$P(w) = \sum_{\tau : \text{yield}(\tau) = w} P(\tau)$$

où yield(τ) désigne le mot formé par les feuilles de τ.

Le problème du parsing probabiliste consiste, étant donné un mot w, à trouver l'arbre de parsing le plus probable :

$$\tau^* = \arg \max_{\tau : \text{yield}(\tau) = w} P(\tau)$$

6. Expliquer pourquoi une approche par énumération exhaustive n'est pas viable en pratique.
7. Rappeler ce qu'est la forme normale de Chomsky d'une grammaire.
8. Adapter l'algorithme de parsing traditionnel pour résoudre le problème du parsing probabiliste. Quelle est la complexité temporelle de cet algorithme en fonction de $|w|$ et $|G|$?
9. Comment modifier l'algorithme pour calculer $P(w)$ (la somme des probabilités de tous les arbres) au lieu du maximum ?

Matthieu Boyer
Planche B

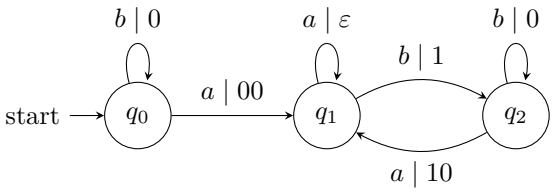
Transducteurs séquentiels

Un transducteur séquentiel est défini comme un sextuplet $T = (Q, \Sigma, \Gamma, \delta, \lambda, q_0)$ où :

- Q est un ensemble fini d'états
- Σ est l'alphabet d'entrée et Γ l'alphabet de sortie
- $\delta : Q \times \Sigma \to Q$ est la fonction de transition
- $\lambda : Q \times \Sigma \to \Gamma^*$ est la fonction de sortie
- $q_0 \in Q$ est l'état initial

On étend λ à $Q \times \Sigma^*$ par : $\lambda(q, \varepsilon) = \varepsilon$ et $\lambda(q, ua) = \lambda(q, u)\lambda(\delta(q, u), a)$ pour $u \in \Sigma^*$ et $a \in \Sigma$.
La fonction associée à T est $\varphi_T : \Sigma^* \to \Gamma^*$ définie par $\varphi_T(u) = \lambda(q_0, u)$.

1. Soit le transducteur T_0 sur $\Sigma = \{a, b\}$ et $\Gamma = \{0, 1\}$ défini par le graphe suivant :



   Calculer $\varphi_{T_0}(bbbaaa)$ et donner un antécédent de 01101 par φ_{T_0}.
2. Dessiner un transducteur séquentiel sur $\Sigma = \Gamma = \{a, b\}$ qui remplace toute séquence de a consécutifs par un seul a et laisse les séquences de b inchangées.
3. On représente les nombres en base b par des suites en little endian. Dessiner un transducteur sur $\Sigma = \{0, 1, 2, 3\}$ et $\Gamma = \{0, 1\}$ qui convertit un nombre de la base 4 vers la base 2.

On représente un transducteur en OCaml par : `type lettre = int`
`type mot = lettre list`
`type etat = int`
`type transducteur = (etat * mot) array array` où `t.(q).(a)` contient le couple `(q', v)` avec $q' = \delta(q, a)$ et $v = \lambda(q, a)$.

4. Écrire une fonction `calcul : transducteur -> mot -> mot` qui calcule l'image d'un mot par un transducteur. Justifier que la complexité est linéaire en la taille du mot d'entrée.
5. Écrire une fonction `liste_mots (p: int) (m:int): mot list` qui renvoie tous les mots de taille m sur l'alphabet $\{0, \dots, p-1\}$ dans un ordre arbitraire.
6. En déduire une fonction `antécédent(t : transducteur) (m : int) (v : mot) : mot option` `transducteur t m v` renvoie `Some u` où u est le mot le plus court possible vérifiant $|u| \leq m$ et $\varphi_T(u) = v$ s'il en existe un, et `None` sinon.
7. Soit $\varphi : \Sigma^* \to \Gamma^*$. On dit que φ est un morphisme de mots si et seulement si pour tous mots u et v de Σ^*, $\varphi(uv) = \varphi(u)\varphi(v)$.
8. Soit $\varphi : \Sigma^* \to \Gamma^*$ un morphisme de mots. Déterminer en justifiant la valeur de $\varphi(\varepsilon)$.
9. Soit φ, ψ deux morphismes de mots de Σ^* dans Γ^*. Montrer que si pour tout $a \in \Sigma, \varphi(a) = \psi(a)$, alors $\varphi = \psi$.
10. Soit $\varphi : \Sigma^* \to \Gamma^*$ un morphisme de mots. Montrer que φ est une fonction séquentielle.
11. Soit T un transducteur séquentiel dont tous les états sont accessibles (c'est-à-dire que pour tout état $q \in Q$, il existe $u \in \Sigma^*$ tel que $\delta(q_0, u) = q$). Montrer qu'il y a équivalence entre :
    (a) φ_T est un morphisme de mots
    (b) pour tout $q \in Q$ et $a \in \Sigma$, $\lambda(q, a) = \lambda(q_0, a)$

Matthieu Boyer
Planche C

Heuristiques de Boyer-Moore

L'algorithme de Boyer-Moore recherche un motif P de longueur m dans un texte T de longueur n.

1. Rappelez l'algorithme de Knuth-Morris-Pratt qui résoud ce problème.

Contrairement à l'algorithme de Knuth-Morris-Pratt, Boyer-Moore compare le motif de droite à gauche tout en se déplaçant de gauche à droite dans le texte. L'efficacité de Boyer-Moore repose sur deux heuristiques permettant de sauter des positions lors d'un échec de comparaison.

1. Soit i la position courante dans T et j la position dans P où un échec survient (on a $P[j] \neq T[i+j]$). Soit $c = T[i+j]$ le caractère qui cause l'échec. Montrer qu'on peut décaler le motif de $j+1$ positions vers la droite.
2. Montrer que de plus on peut décaler le motif pour aligner la dernière occurence de c dans $P[0 \dots j-1]$ avec $T[i+j]$. *On pourra montrer qu'aucune occurence du motif ne peut commencer dans les positions sautées.*
3. Supposons qu'on ait réussi à matcher un suffixe $t = P[j+1..m-1]$ avec $T[i+j+1..i+m-1]$, mais qu'on échoue sur $P[j] \neq T[i+j]$.
   (a) Si le suffixe t réapparaît ailleurs dans P précédé d'un caractère différent de $P[j]$, expliquer intuitivement comment on peut décaler le motif.
   (b) Si t ne réapparaît pas dans P, mais qu'un préfixe de P correspond à un suffixe de t, expliquer intuitivement le décalage possible.
   (c) Si aucune des conditions précédentes n'est satisfaite, quel décalage peut-on effectuer ?
   (d) Prouver que dans chacun de ces trois cas, le décalage proposé ne fait manquer aucune occurrence du motif dans le texte.
4. Expliquer comment combiner les deux heuristiques pour obtenir le meilleur décalage possible.
5. Donner un exemple où l'heuristique du mauvais caractère est plus efficace que celle du bon suffixe, et un exemple où c'est l'inverse.