Oral d'informatique - MPI (3h30)

Consignes

Ce sujet comporte trois parties. Les questions de programmation doivent être traitées en C ou en OCaml comme indiqué au début de chaque partie. Des rappels de programmation dans les deux langages sont donnés en annexe.

À la fin de votre préparation d'une durée de 3 heures, un échange de 30 minutes avec l'examinateur permettra d'évaluer votre travail. Vous pouvez préparer vos réponses sur un document papier qui vous servira pour cet échange.

Dans ce sujet, pour simplifier l'étude, les caractères considérés sont uniquement ceux de la table ASCII, représentés en C et OCaml par le type `char`, en ignorant tout problème relatif à l'encodage. L'annexe rappelle comment convertir un caractère en son code entier ASCII associé compris entre 0 et 127 et réciproquement.

Contexte

L'objet de ce sujet est de développer un système de complétion de texte simple. Un utilisateur qui commence à taper un texte se voit proposé, par le système, une suite à la séquence de caractères saisis. L'approche envisagée utilise les N-grammes par apprentissage sur un texte préexistant. Un *N-gramme* est une séquence de N caractères consécutifs apparaissant dans un texte. Dans ce sujet, le texte préexistant adopté pour l'apprentissage [1] est le suivant.

`Bonjour,␣comment␣allez-vous␣?␣Ca␣va,␣ca␣va␣aller␣bien␣mieux.`

La complétion procède en deux étapes en construisant d'abord les N-grammes contenus dans le texte puis en identifiant les caractères qui leur succèdent pour calculer des probabilités associées. La première étape choisit la valeur de N, taille des N-grammes. Par ordre d'apparition, l'exemple précédent contient :

- les 1-grammes : `B`, `o`, `n`, `j`, `u`, `r`, `,`, `␣`, `c`, ...
- les 2-grammes : `Bo`, `on`, `nj`, `jo`, `ou`, ...
- les 3-grammes : `Bon`, `onj`, `njo`, `jou`, ...
- etc.

La deuxième étape identifie les caractères, appelés les *successeurs*, qui suivent chaque N-gramme et compte leurs occurrences pour calculer leur probabilité d'apparition. Par exemple :

- le 1-gramme `a` est suivi des caractères `l`, `␣` , `,` avec respectivement les occurrences 2, 3, 1 ; les probabilités associées sont respectivement les $2/6$, $3/6$, $1/6$;
- le 1-gramme `u` est suivi des caractères `r`, `s`, `x` avec respectivement les occurrences 1, 1, 1 ; les probabilités associées sont respectivement les $1/3$, $1/3$, $1/3$;
- le 2-gramme `le` est suivi des caractères `z` et `r` avec respectivement les occurrences 1 et 1 ; les probabilités associées sont respectivement $1/2$, $1/2$.

En choisissant une taille de N-grammes, c'est-à-dire en fixant N, on dispose alors de probabilités conditionnelles qui permettent de prédire le caractère qui suit une séquence de caractères donnée.

Modèle de 1-grammes

Dans cette partie, le langage de programmation est le C.

Dans une version simplifiée de modèle de 1-grammes, le caractère à prédire après un autre est toujours le même, à savoir le plus fréquent parmi les successeurs. En cas d'égalité du nombres d'occurrences entre deux successeurs, on choisit arbitrairement le premier dans l'ordre alphabétique, à savoir celui qui a le plus petit code associé. De fait, ce modèle ne conserve pas en mémoire l'ensemble des successeurs de chaque caractère, mais conserve seulement celui qui est prédit. En reprenant les exemples précédents, le modèle prédit que :

- le 1-gramme `a` est suivi par le caractère `␣` qui est le plus fréquent parmi les successeurs ;
- le 1-gramme `u` est suivi par le caractère `r`, qui apparaît aussi fréquemment que `s` mais se trouve avant `s` dans l'ordre alphabétique.

En C, un modèle de 1-grammes est réprésenté par un tableau d'entiers `M`, de taille 128, tel que si `c` est un caractère de code `k`, alors `M[k]` vaut :

- le code du caractère à prédire si `c` possède des successeurs ;
- le code 0 du caractère nul `0` sinon.

1. Un apprentissage réel nécessite un texte bien plus long.

Question 1. Écrire une fonction `int plus_frequent_successeur(char c, char* chaine)` qui prend en argument un caractère `c`, une chaîne de caractères `chaine` et qui renvoie le code du caractère qui apparaît le plus fréquemment dans la chaîne comme successeur de `c`, ou le plus petit code en cas d'égalité. Si `c` n'apparaît pas dans la chaîne, la fonction renvoie le code du caractère nul.

Question 2. En déduire une fonction `int* init_modele(char* chaine)` qui construit et renvoie un modèle de 1-gramme à partir d'un texte d'apprentissage donné en argument.

Question 3. Quelle est sa complexité temporelle ? Peut-on faire mieux ? On ne demande pas d'implémenter une éventuelle solution mais de la décrire succinctement à l'examinateur.

Question 4. Écrire une fonction `int** matrice_confusion(int* M, char* test)` qui prend en argument un modèle de 1-gramme, une chaîne de caractères `test` et qui crée et renvoie une matrice de confusion de taille 128×128 comparant les prédictions données par le modèle et les successeurs de chaque caractère dans la chaîne de test.

Question 5. Évaluer le modèle de prédiction fondé sur la phrase donnée en exemple, en calculant le pourcentage d'erreurs à partir de la matrice de confusion pour la phrase de test suivante.

`Bonjour,␣ca␣va␣bien␣?␣Oui␣!␣Bien␣mieux,␣et␣vous,␣ca␣va␣?`

Modèle de N-grammes

Dans cette partie, le langage de programmation est OCaml.
Par rapport au modèle précédent, on propose deux changements :

- la prédiction se fait non plus à partir du dernier 1-gramme lu, mais à partir du dernier N-gramme, ou du dernier $(N-1)$-gramme si le N-gramme n'a pas de successeur, ou du dernier $(N-2)$-gramme si le $(N-1)$-gramme n'a pas de successeur, et ainsi de suite, en allant si besoin jusqu'au 0-gramme (chaîne vide), dont les successeurs sont l'ensemble des lettres du texte d'apprentissage ;
- la prédiction n'est plus déterministe mais choisit aléatoirement un successeur pour un N-gramme en fonction de la probabilité associée.

Par exemple :

- le modèle prédit que le 1-gramme `a` est suivi par le caractère `l` avec probabilité $2/6$, ␣avec probabilité $3/6$ et `,` avec probabilité $1/6$;
- le 3-gramme `ple` n'a pas de successeur ; le modèle utilise le 2-gramme `le` pour faire la prédiction, qui est le caractère `r` avec probabilité $1/2$ et `z` avec probabilité $1/2$;
- le 2-gramme `ay` n'a pas de successeur et le 1-gramme `y` non plus ; le modèle utilise le 0-gramme pour faire la prédiction, qui est le caractère `B` avec probabilité $1/60$, le caractère `o` avec probabilité $4/60$, le caractère `n` avec probabilité $3/60$, etc.

Dans cette partie, un modèle de N-grammes contient, pour chaque k-gramme du texte d'apprentissage, avec $k \leqslant N$, l'ensemble de ses successeurs et du nombre d'occurrences correspondant, ou des probabilités associées.

Question 6. Décrire une structure de données pour encoder un modèle de N-grammes avec un N donné. Cette structure doit permettre de retrouver les modèles dont la taille est inférieure à N. Par exemple, un modèle de 3-grammes doit contenir les successeurs des 3-grammes mais également des 2-grammes, des 1-grammes et du 0-gramme.

On donne ici quelques suggestions pour répondre à cette question.

On peut représenter un ensemble de successeurs et de leurs occurrences par :

- un tableau de taille 128 où l'élément d'indice k est le nombre d'occurrences de la lettre de code k ;
- un dictionnaire dont les clés sont les caractères et les valeurs le nombre d'occurrences.

On peut alors représenter un modèle de N-grammes comme un dictionnaire dont les clés sont les N-grammes et les valeurs les ensembles de successeurs et occurrences associés.

On peut également représenter directement un modèle de N-grammes comme un arbre préfixe (ou *trie*), c'est-à-dire un arbre d'arité quelconque vérifiant :

- la racine est étiquetée par le mot vide ;
- les autres nœuds sont étiquetés par un couple (caractère, nombre d'occurrences) ;
- à chaque nœud, on peut associer le mot constitué de la concaténation de tous les caractères lus dans le chemin de la racine à ce nœud ;
- chaque nœud interne a pour enfants tous les successeurs du mot associé, avec le nombre d'occurrences correspondant.

Question 7. Écrire une fonction `init_modele` qui construit et renvoie un modèle de N-grammes à partir d'un texte d'apprentissage et d'un entier N donnés en argument.

Question 8. Écrire une fonction `prediction` qui prend en argument un modèle, un N-gramme et qui renvoie un caractère prédit par le modèle selon le principe décrit précédemment.

Pour tester le modèle, on souhaite faire de la génération automatique de texte en *pseudo-français*, c'est-à-dire un texte dont les mots ressemblent à du français mais n'en sont pas forcément. L'idée est de partir d'une chaîne de caractères (vide ou non) et de prédire les caractères suivants, à chaque fois en considérant le dernier N-gramme, qui contient éventuellement les lettres déjà prédites. Voici par exemple un texte généré automatiquement à partir de la chaîne `Bonjour` en utilisant le modèle de la phrase donnée initialement en exemple, pour $N = 2$.

```
Bonjour, ca allez-vour, ca va, ca aller bien mien mieux.-vous ? Ca va, commen mieux.
va va aller bieux.va va, commen mient allez-vous ? Ca va va, ca va va va, ca va va va,
comment aller bieux.
```

Question 9. Écrire une fonction `generation` qui prend en argument un modèle de N-grammes, une chaîne de caractères, une taille cible et qui génère un texte aléatoire de taille cible à partir de la chaîne, en utilisant le modèle de prédiction, selon le principe décrit précédemment.

Question 10. Le fichier texte `cid.txt` contient un extrait du *Cid* de *Corneille*. Utiliser ce texte pour faire de la génération automatique, pour différentes valeurs de N.

Programmation concurrente

Dans cette partie, le langage de programmation est OCaml.

La construction d'un modèle de N-grammes prend d'autant plus de temps que le texte d'entrée est long. Pour optimiser la construction, une approche exploitant la concurrence, avec plusieurs fils d'exécution, est envisagée.

Question 11. Proposer une approche pour répartir la construction du modèle sur plusieurs fils.

Question 12. Implémenter la solution précédente.

Annexe

Rappels de C

La compilation d'un fichier `source.c` peut se faire avec la commande `gcc source.c`. On peut rajouter des options avec la commande `gcc options source.c` où `options` doit être remplacé par une ou plusieurs options de compilation. Par exemple :

- choix du nom d'exécutable : `-o nom_exec`
- avertissements : `-Wall`, `-Wextra`
- alerte mémoire : `-fsanitize=address`

Pour les conversions caractère - code ASCII :

- conversion d'un caractère `c` vers son code ASCII associé : `(int) c` ;
- conversion d'un entier `k` entre 0 et 127 vers son caractère associé : `(char) k`.

Rappels de OCaml

La compilation d'un fichier `source.ml` peut se faire avec la commande `ocamlc source.ml`. On peut rajouter des options de compilation avec `ocamlc options source.ml` où `options` doit être remplacé par une ou plusieurs options de compilation. Par exemple :

- choix du nom d'exécutable : `-o nom_exec`
- utilisation des fils d'exécution (en dernière option) : `-I +threads unix.cma threads.cma`

- L'instruction `Random.int n` renvoie un entier pseudo-aléatoire entre `0` inclus et `n` exclu. `Random.self_init ()` génère une graine aléatoire.
- `Char.code : char -> int` et `Char.chr : int -> char` convertit un code en son caractère associé.
- La fonction `String.sub : string -> int -> int` renvoie une sous-chaîne : `String.sub s deb taille` renvoie la sous-chaîne de `s` commençant à l'indice `deb` et de taille `taille`.

Le module `Hashtbl` permet l'utilisation de dictionnaires, avec notamment les instructions suivantes.

- `Hashtbl.create : int -> ('a, 'b) Hashtbl.t` prend en argument un entier correspondant à une estimation du nombre de clés qui vont être associées dans la table (en pratique on peut simplement utiliser `1` car les tables sont redimensionnables dynamiquement) et crée une table vide.
- `Hashtbl.add : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit` prend en argument une table, une clé et une valeur et rajoute une association.
- `Hashtbl.mem : ('a, 'b) Hashtbl.t -> 'a -> bool` teste si une table contient une association pour une clé donnée.
- `Hashtbl.find : ('a, 'b) Hashtbl.t -> 'a -> 'b` prend en argument une table et une clé et renvoie la valeur associée, ou lève l'exception `Not_found` si la clé n'a pas d'association.
- `Hashtbl.iter : ('a -> 'b -> unit) -> ('a, 'b) t -> unit` prend en argument une fonction f et une table et applique la fonction f à chaque couple (clé, valeur) de la table.

Pour simplifier l'utilisation des sémaphores, on peut renommer le module avec :

```
module Sem = Semaphore.Counting
```

L'utilisation des outils de synchronisation peut se faire à l'aide des commandes suivantes.

- `Thread.create : ('a -> 'b) -> 'a -> Thread.t` prend en argument une fonction `f` et un argument `x` et renvoie un fil d'exécution qui exécute `f x`.
  Attention : la fonction `f` ne doit avoir qu'un seul argument. Si on souhaite la faire travailler avec plusieurs arguments, on peut utiliser un tuple.
- `Thread.join : Thread.t -> unit` prend en argument un fil d'exécution et attend qu'il ait terminé son exécution.
- `Mutex.create : unit -> Mutex.t` renvoie un nouveau mutex.
- `Mutex.lock : Mutex.t -> unit` verrouille un mutex.
- `Mutex.unlock : Mutex.t -> unit` déverrouille un mutex.
- `Sem.make : int -> Sem.t` prend en argument un entier k et renvoie un sémaphore avec un compteur initialisé à k.
- `Sem.acquire : Sem.t -> unit` acquière un sémaphore (c'est-à-dire décrémente le compteur).
- `Sem.release : Sem.t -> unit` libère un sémaphore (c'est-à-dire incrémente le compteur).

Attention, il est nécessaire d'utiliser l'option de compilation prévue à cet effet. On peut également utiliser un interpréteur OCaml au lieu de compiler auquel cas il est nécessaire d'exécuter les commandes suivantes.

```
#directory "+threads";;
#load "unix.cma";;
#load "threads.cma";;
```