Modèles de N-grammes

Proposition de corrigé

Question 1 On utilise ici un tableau d'occurrences, initialisé à zéro. On parcourt le texte, et chaque fois qu'on rencontre le caractère `c`, le caractère suivant (éventuellement celui de fin de chaîne) voit son occurrence augmentée de 1. On garde en mémoire au fur et à mesure l'indice de la lettre ayant le nombre d'occurrences maximal.

```
int plus_frequent_successeur(char c, char* chaine){
    int occ[K] = {0};
    int n = strlen(chaine);
    int b_max = 0;
    for (int i=0; i<n; i++){
        if (chaine[i] == c){
            int b = (int) chaine[i+1];
            occ[b]++;
            if (occ[b] > occ[b_max]){
                b_max = b;
            }
        }
    }
    return b_max;
}
```

Question 2 On définit une constante littérale K, correspondant au nombre de codes distincts, par :

```
#define K 128 //nombre de codes
```

On y fera référence dans les fonctions suivantes. Pour l'initialisation du modèle, on se contente de créer un tableau de taille `K`, et pour chaque caractère (via son code), on détermine le successeur le plus fréquent.

```
int* init_modele(char* chaine){
    int* M = malloc(K * sizeof(int));
    for (int a=0; a<K; a++){
        M[a] = plus_frequent_successeur((char) a, chaine);
    }
    return M;
}
```

Question 3 La fonction `plus_frequent_successeur` a une complexité en $\mathcal{O}(K+n)$, où n est la longueur de la chaîne d'apprentissage. On y fait appel K fois. La complexité totale est donc en $\mathcal{O}(K(K+n))$. On aurait pu faire ça en $\mathcal{O}(K^2+n)$ en créant initialement tous les tableaux d'occurrences nécessaires, et en ne parcourant qu'une seule fois la chaîne (au lieu de K fois).

Question 4 On commence par créer une matrice de zéros. On modifie ensuite cette matrice en lisant la chaîne : quelle est la lettre prédite après la i-ème, sachant que ça aurait dû être la $i+1$-ème ?

```
int** matrice_confusion(int* M, char* test){
    int** mat = malloc(K * sizeof(int*));
    for (int a=0; a<K; a++){
        mat[a] = malloc(K * sizeof(int));
        for (int b=0; b<K; b++){
            mat[a][b] = 0;
        }
    }
    int n = strlen(test);
    for (int i=0; i<n; i++){
        int a = (int) test[i + 1];
        int b = M[(int) test[i]];
        mat[a][b]++;
    }
    return mat;
}
```

On pense à écrire une fonction permettant de libérer la matrice :

```
void liberer_mat(int** mat){
    for (int a=0; a<K; a++){
        free(mat[a]);
    }
    free(mat);
}
```

Question 5

On peut écrire une fonction calculant le taux d'erreurs. Pour ce faire, on calcule le nombre total d'occurrences (qui correspondant en fait à la taille de la chaîne, mais ici on ne donne en argument que la matrice) et le nombre d'erreurs (c'est-à-dire la somme des coefficients en dehors de la diagonale).

```
double taux_erreur(int** mat){
    int occ = 0, erreurs = 0;
    for (int a=0; a<K; a++){
        for (int b=0; b<K; b++){
            occ += mat[a][b];
            if (a != b){
                erreurs += mat[a][b];
            }
        }
    }
    return 1.0 * erreurs / occ;
}
```

Il faut penser à convertir en flottants avant de faire la division (sinon c'est une division entière).

Pour les tests, on obtient :

```c
int main(void){
    char chaine[] = "Bonjour, comment allez-vous ? Ca va, ca va aller bien mieux.";
    char test[] = "Bonjour, ca va bien ? Oui ! Bien mieux, et vous, ca va ?";
    int* M = init_modele(chaine);
    int** mat = matrice_confusion(M, test);
    printf("Taux erreurs : %lf\n", taux_erreur(mat));
    free(M);
    liberer_mat(mat);
    return 0;
}
```

Ce qui donne environ 54% d'erreurs. C'est décevant, mais le modèle est très simple, on pouvait s'y attendre.

1 Modèle de N-grammes

Question 6 On choisit ici la représentation la plus simple parmi celles proposées, c'est-à-dire un dictionnaire de tableaux. On choisit des tableaux de taille $K + 1 = 129$, comme ça on peut mettre le nombre total de successeurs en dernière case, pour calculer les fréquences en temps constant. On a alors juste :

```
type modele = (string, int array) Hashtbl.t
let _K = 128
```

Question 7 Pour chaque taille de k-grammes et pour chaque indice de début d'un k-gramme, on détermine ce k-gramme, le code du caractère qui suit, on crée nouveau tableau de 0 dans la table de hachage s'il n'y en a pas déjà un, et on incrémente les deux bonnes cases.

```
let init_modele s _N : modele =
    let modele = Hashtbl.create 1 in
    for k = 0 to _N do
        (* Pour chaque taille de k-grammes *)
        for i = 0 to String.length s - 1 - k do
            (* Pour chaque indice de début d'un k-gramme, on détermine ce k-gramme,
               le code du caractère qui suit, on crée nouveau tableau de 0 dans la
               table de hachage s'il n'y en a pas déjà un, et on incrémente les deux
               bonnes cases. *)
            let kgramme = String.sub s i k in
            let code = Char.code s.[i + k] in
            if not (Hashtbl.mem modele kgramme) then
               Hashtbl.add h kgramme (Array.make (_K + 1) 0);
            let occ = Hashtbl.find modele kgramme in
            occ.(code) <- occ.(code) + 1;
            occ.(_K) <- occ.(_K) + 1
        done
    done;
    modele
```

Question 8 Tant qu'on n'a pas trouvé un k-gramme présent dans le modèle, on enlève la première lettre. On suppose qu'il existe au moins la chaîne vide. On tire ensuite un entier aléatoire entre 0 et le nombre total d'occurrences moins 1, et on renvoie le caractère correspondant à ce numéro d'occurrence.

```
let prediction modele ngramme =
    let k = ref (String.length ngramme) and
        kgramme = ref ngramme in
    while not (Hashtbl.mem modele !kgramme) do
        decr k;
        kgramme := String.sub !kgramme 1 !k
    done;
    let t = Hashtbl.find modele !kgramme in
    let alea = Random.int t.(_K) in
    let nb_occ = ref 0 and i = ref (-1) in
    while !nb_occ <= alea do
        incr i;
        nb_occ := !nb_occ + t.(!i)
    done;
    Char.chr !i
```

Question 9 On se contente de rajouter, lettre à lettre, les prédictions du modèle. À chaque itération, on met à jour le dernier N-gramme lu.

```
let generation modele _N graine taille =
    let gen = ref graine in
    let len = ref (String.length graine) in
    while !len < taille do
        let m = min !len _N in
        let ngramme = String.sub !gen (!len - m) m in
        gen := !gen ^ (String.make 1 (prediction modele ngramme));
        incr len
    done;
    !gen
```

Question 10 En supposant que le texte est chargé dans la variable `cid`, on fait les tests avec :

```
let test _N =
    let modele = init_modele cid _N in
    let gen = generation modele _N "" 1000 in
    Printf.printf "%s\n" gen
```

Avec $N = 2$, on obtient un texte qui n'est clairement pas en français, mais dont les mots ont une structure similaire à celle du français. Avec $N = 3$, la plupart des mots sont en français, mais on trouve parfois quelques erreurs, comme `fletriompher`. Avec $N = 10$, on obtient des extraits du texte, différents à chaque fois puisqu'on part toujours d'une chaîne vide.

Question 11 On propose de garder la même structure de données, mais d'ajouter un mutex à chaque tableau d'occurrences, pour être sûr que deux modifications ne soient pas simultanées. Cela ne risque pas de trop ralentir le calcul, car, à part pour les 1-grammes, deux fils auront peu de chances de travailler sur le même k-gramme. Par ailleurs, on protège également la table en écriture par un mutex, pour éviter que deux fils essaient simultanément d'ajouter une entrée correspondant au même N-gramme.

Dès lors, si on a p fils d'exécutions, on donne à chacun la charge de remplir le modèle avec une fraction $\frac{1}{p}$ du texte. Il faut faire attention à prendre en compte les N-grammes qui chevauchent deux fractions différentes du texte (on laissera le fil de la fraction gauche s'en occuper).

Question 12 La fonction suit le même schéma que la première sans qu'on prend en compte la gestion des mutex.

```
let init_modele2 s _N nb_fils : modele =
    let modele = Hashtbl.create 1 and mutex = Mutex.create () in
    let len = String.length s in
    let init_fil (deb, fin) =
        (* Fonction qui prend en argument une tranche de texte et se charge
           de remplir le modèle. *)
        for k = 0 to _N do
            (* Pour chaque taille de k-grammes *)
            for i = deb to min (len - 1 - k) (fin - 1) do
                (* Pour chaque indice de début de k-grammes (c'est-à-dire tel qu'on
                   ne dépassera pas la fin de la chaîne) *)
                let kgramme = String.sub s i k in
                let code = Char.code s.[i + k] in
                if not (Hashtbl.mem modele kgramme) then begin
                    Mutex.lock mutex;
                    (* On vérifie à nouveau que le k-gramme n'a pas été rajouté
                       le temps de verrouiller le mutex. *)
                    if not (Hashtbl.mem modele kgramme) then
                      Hashtbl.add modele kgramme (Array.make (_K + 1) 0, Mutex.create ());
                    Mutex.unlock mutex
                end;
                let (occ, mut) = Hashtbl.find modele kgramme in
                (* On verrouille le mutex avant la modification. *)
                Mutex.lock mut;
                occ.(code) <- occ.(code) + 1;
                occ.(_K) <- occ.(_K) + 1;
                Mutex.unlock mut
            done
        done
    in
    let indices i = ((i * len) / nb_fils, ((i + 1) * len) / nb_fils) in
    let fils = Array.init nb_fils (fun i -> Thread.create init_fil (indices i)) in
    Array.iter Thread.join fils;
    (* On recrée la table sans les mutex. *)
    let sans_mutex = Hashtbl.create 1 in
    Hashtbl.iter (fun kgramme (occ, mut) -> Hashtbl.add sans_mutex kgramme occ) modele;
    sans_mutex
```