POLYTECH MARSEILLE Aix*Marseille Université **HUGO3**

Algorithmique

TD n°2 : preprocessing

Exercice 1 – Algorithme de Knuth-Morris-Pratt

Recherchez à l'aide de l'algorithme de Knuth-Morris-Pratt les motifs $p_1 = ababacacab$ et $p_2 = ababab$ respectivement dans les textes $t_1 = ababcababadabac$ et $t_2 = abbabaababbababaababab$.

Exercice 2 – Algorithme de Boyer-Moore simplifié

L'algorithme de Boyer-Moore est un autre algorithme pour rechercher des chaînes de caractères dans d'autres chaînes. En principe, il est un peu plus rapide que celui de Knuth-Morris-Pratt :

- $O(m+n)$ dans le pire des cas, comme Knuth-Morris-Pratt,
- $O(n/m)$ dans le meilleur cas
- En pratique, plus le motif à rechercher est long ou contient peu de lettres identiques, plus il se rapproche du meilleur cas.

Nous allons en voir ici une version simplifiée. L'algorithme s'appuie sur une distance d (en nombre de lettres) entre la dernière occurrence d'une lettre quelconque et la dernière lettre du motif P que l'on recherche. Si la lettre n'apparaît pas dans le motif ou bien si elle n'apparaît qu'en dernière position, cette distance est égale à la longueur m du motif. Par exemple, pour le motif $P = aababab$, $d(a) = 1$, $d(b) = 2$ et $d(c) = 7$. L'idée sous-jacente est que Boyer-Moore va essayer de « matcher » le motif P et le texte T dans lequel on le recherche *de la droite vers la gauche*, donc en commençant par le dernier caractère du motif et en remontant vers le premier. Dès lors que l'on rencontre un caractère qui ne « matche » pas, on doit décaler le motif d'au moins d caractères vers la droite.

La fonction suivante a pour but de déterminer la distance d. On la stocke comme un tableau dont chaque cellule correspond à un caractère :

```
1  fonction derniere_occurrence (chaîne P, tableau d) :
2    m ← longueur(P)
3
4    # par défaut, les lettres ∉ P ont une distance de m
5    pour toute lettre x de l'alphabet faire
6      d[x] ← m
7    fait
8
9    # on place dans d[P[i]] sa distance par rapport au dernier caractère
10   pour i variant de 0 à m-2 faire
11     d[P[i]] ← m-1 - i
12   fait
```

Q 2.1 Montrez qu'après exécution de `derniere_occurrence`, tous les éléments de d sont strictement positifs.

Après avoir utilisé la procédure `derniere_occurrence` pour calculer la distance d, on peut appliquer l'algorithme de `Boyer_Moore_simplifie` :

```
1  fonction Boyer_Moore_simplifie (chaîne P,chaîne T) :
2    m ← longueur(P)
3    n ← longueur(T)
4    j ← m - 1
5    tant que j < n faire
6      i ← m - 1
7      tant que i >= 0 et T[j] = P[i] faire
8        i ← i-1
9        j ← j-1
10     fait
11     si i = -1 alors
12       retourner j+1
13     sinon
14       j ← j + max(d[T[j]], m-i)
15     finsi
16   fait
```

Q 2.2 Recherchez à l'aide de l'algorithme de Boyer-Moore les motifs $p_1 = ababacacab$ et $p_2 = ababab$ respectivement dans les textes $t_1 = ababcababadabac$ et $t_2 = abbabaababbbababaababab$.

Q 2.3 À quoi sert le `m-i` sur la ligne 14 ? (idée : regardez bien les lignes 6 et 7)

Q 2.4 Montrez que `Boyer_Moore_simplifie` trouve bien la première occurrence de la chaîne P dans T (si elle existe).

Q 2.5 Comment pourrait-on améliorer cet algorithme ?

Exercice 3 – Compression : algorithme de Huffman

L'algorithme de Huffman est très utilisé en compression de données et, depuis son développement en 1952, il a suscitée beaucoup de recherches. Il sert de base à de nombreux programmes. JPEG s'en sert par exemple dans une de ses étapes de compression.

Dans la suite, on considérera que ce que l'on essaye de compresser est une séquence de « symboles » (par exemple, cela peut être des caractères, mais cela peut également être des bits d'une image). L'idée de cet algorithme consiste à affecter un code (une séquence de bits) « de petite taille » aux symboles fréquents et un code « plus gros » aux moins fréquents. Si l'on prend l'exemple d'une chaîne de caractères, on affectera donc des codes de tailles différentes à différents caractères. On peut, par exemple, encoder certains caractères sur 1 ou 2 bits et d'autres sur 10, au lieu des 8 bits standards d'un code ASCII. Ainsi, le code en sortie est souvent beaucoup plus petit que la séquence des symboles d'entrée. Pour obtenir cela, l'algorithme consiste :

1. à dresser une liste des symboles de l'alphabet triée par ordre décroissant de fréquence d'apparition des symboles dans le texte (on en profite pour calculer ces fréquences). L'exemple de la page suivante illustrera cela.
2. puis à construire un arbre, dans lequel chaque symbole se trouve sur une feuille, de la manière suivante :
   (a) on part d'un arbre vide ;

(b) on prend les deux symboles de la liste ayant les plus petites fréquences, appelons-les a_i et a_j, et on les enlève de la liste des symboles ;

(c) on crée un symbole représentant la réunion des deux choisis, appelons-le a_{ij}, auquel on affecte la somme des fréquences de a_i et a_j, et on le rajoute à la liste ;

(d) on rajoute alors à l'arbre un arbre binaire constitué d'une racine, a_{ij}, et de deux feuilles, a_i et a_j ;

(e) on itère à partir de l'étape (b) jusqu'à ce qu'il ne reste plus qu'un seul symbole.

3. à parcourir l'arbre ainsi créé afin de déterminer le codage de chaque symbole de l'alphabet, comme le montre l'exemple ci-dessous.

Les deux premières étapes constituent le préprocessing de cet algorithme.

Exemple : considérons cinq symboles, a_1, a_2, a_3, a_4 et a_5 apparaissant dans un texte avec les fréquences (probabilités) suivantes :

$$a_1 \equiv 40\% = 0,4 \qquad a_2 \equiv 0,2 \qquad a_3 \equiv 0,2 \qquad a_4 \equiv 0,1 \qquad a_5 \equiv 0,1.$$

L'algorithme de Huffman effectue alors les opérations suivantes :

1. a_4 est combiné avec a_5 car ce sont les caractères ayant les plus petites fréquences d'apparition. Leur combinaison forme alors un nouveau symbole que nous appellerons a_{45} et dont la fréquence d'apparition dans le texte est $0,1 + 0,1 = 0,2$. On créé un arbre binaire ayant pour racine a_{45} et comme feuilles a_4 et a_5 (cf. la figure 1.a).
2. On a maintenant 3 symboles avec les probabilités 0,2 : a_2, a_3 et a_{45}. On doit en combiner deux, peu importe lesquels [1]. On va sélectionner arbitrairement les symboles a_3 et a_{45}. Ceux-ci en forment donc un nouveau, que nous appellerons a_{345} et dont la fréquence d'apparition est $0,2 + 0,2 = 0,4$. Cela nous donne l'arbre de la figure 1.b.
3. On a maintenant deux symboles avec des fréquences de 0,4 : a_1 et a_{345}, plus un symbole, a_2, avec une fréquence de 0,2. On doit donc combiner a_2 avec, au choix, a_1 ou a_{345}. Choisissons de combiner a_2 et a_{345}. On obtient le symbole a_{2345} dont la fréquence est 0,6 (cf. figure 1.c).

1. En fait, l'ordre peut être important pour certaines applications. Cela dit, quels que soient les deux symboles que l'on combine, la taille du flux de sortie sera toujours la même.

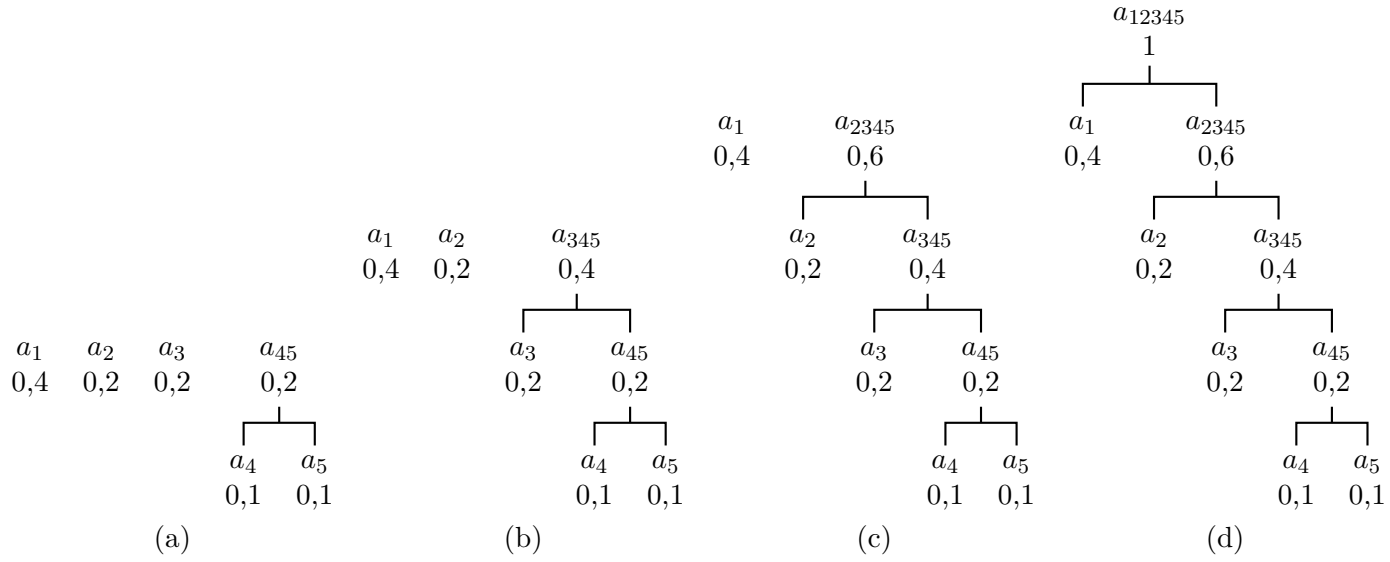


FIGURE 1 – Construction de l'arbre de Huffman.

4. Enfin, il ne reste plus que deux symboles, a_1 et a_{2345}. On les combine pour obtenir a_{12345} dont la fréquence d'apparition est 1. À cette étape, l'arbre déterminant les codes des symboles est construit : les symboles d'origine sont les feuilles et chaque « combinaison » représente un nœud de l'arbre (cf. figure 1.d).
5. À chaque nœud de l'arbre, on code l'une des deux arêtes sortantes avec un 1 et l'autre avec un 0 (cf. figure 2).

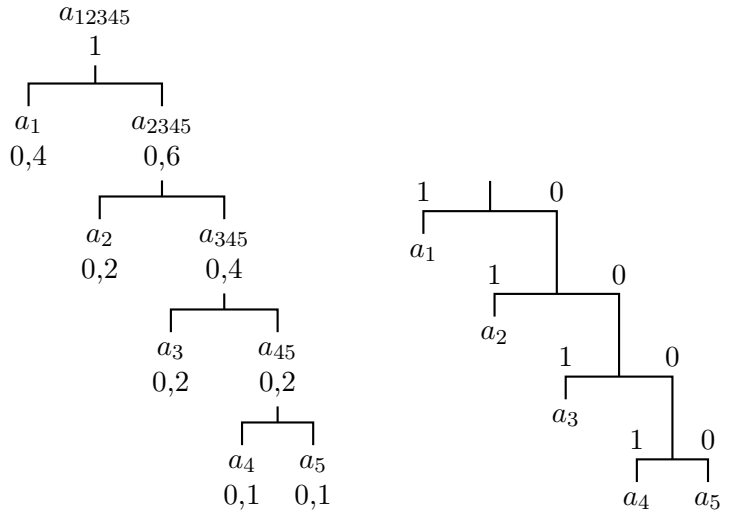


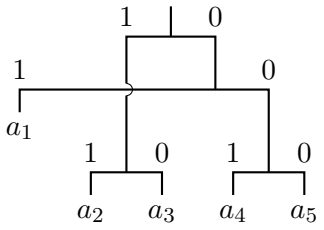
FIGURE 2 – Arbre de Huffman avec codes.

Le préprocessing est maintenant terminé.

Pour obtenir le codage d'un symbole, il suffit de partir du nœud-symbole ayant une fréquence de 1 (autrement dit, la racine tout en haut de l'arbre), et de parcourir l'arbre jusqu'à ce qu'on atteigne le symbole désiré. Au fur et à mesure du cheminement, on note les 0/1 des arcs parcourus. Ceux-ci forment le code correspondant au symbole. Ainsi, le code de a_3 sera obtenu en partant de a_{12345}, en allant sur la branche vers a_{2345} (on obtient le premier bit du code : 0), puis en allant vers a_{345} (bit 0), puis en allant vers a_3 (bit 1). Le code de a_3 sera donc 001. Les arbres ci-dessus nous donnent donc les codes suivants :

$$a_1 \equiv 1 \qquad a_2 \equiv 01 \qquad a_3 \equiv 001 \qquad a_4 \equiv 0001 \qquad a_5 \equiv 0000.$$

Notons que l'on aurait pu obtenir d'autres codes si l'on avait fait d'autres choix de combinaisons. Par exemple, si l'on avait combiné a_4 et a_5, puis a_2 et a_3, puis a_1 et a_{45}, puis a_{145} et a_{23}, on aurait obtenu l'arbre suivant :



On obtiendrait alors le codage suivant :

$$a_1 \equiv 01 \qquad a_2 \equiv 11 \qquad a_3 \equiv 10 \qquad a_4 \equiv 001 \qquad a_5 \equiv 000.$$

Quoi qu'il en soit, les codes de sortie auront la même taille. Pour montrer cela, il suffit de calculer l'espérance (moyenne) de gain (en bits/octet) des deux codages :

$$E(\text{code 1}) = 0,4 \times 1 + 0,2 \times 2 + 0,2 \times 3 + 0,1 \times 4 + 0,1 \times 4 = 2,2,$$
$$E(\text{code 2}) = 0,4 \times 2 + 0,2 \times 2 + 0,2 \times 2 + 0,1 \times 3 + 0,1 \times 3 = 2,2.$$

En moyenne, un octet du fichier de départ sera donc codé en sortie sur 2,2 bits. En fait, cette égalité des taux de compression n'est pas surprenante puisque ce qui importe dans l'algorithme, ce n'est pas vraiment le symbole qu'on code, mais sa fréquence d'apparition.

Q 3.1 Vous voulez compresser un texte à l'aide de Huffman. Vous avez effectué une première passe sur l'ensemble du texte, qui vous a permis de déterminer les nombres d'apparition des caractères suivants :

caractère	`a`	`c`	`d`	`e`	`l`	`n`	`o`	`q`	`r`	`s`	`t`	`u`	`v`	`z`	`'`	␣
nombre d'occurrences	70	8	22	80	33	23	2	3	34	55	9	3	7	7	2	20

Dessinez un arbre de Huffman correspondant puis compressez à l'aide de cet arbre le texte suivant : « `vous l'avez trouvee` ». Dans la séquence de bits que vous écrirez, vous séparerez les caractères par des barres verticales. Par exemple, si un caractère `f` a pour code 0101 et si `g` a pour code 111, alors le code correspondant à « `fgf` » sera « 0101|111|0101 ».

Q 3.2 calculez l'espérance de gain de cette compression.

Q 3.3 Pensez-vous que l'algorithme de Huffman peut réduire la taille de n'importe quelle séquence de symboles ?