Algorithmique du texte - L2 Informatique

TD : feuille 1

Ces exercices reprennent les références-sources indiquées en cours.

1 Exercices sur feuille

1.1 Complexité des algorithmes de recherche

On exprime la complexité en nombre de comparaisons effectués par l'algorithme. On note t la longueur du texte et m la longueur du motif.

1. Combien de comparaisons effectue la recherche naïve : dans le meilleur cas, le pire cas? Donner des exemples dans laquelle ces deux bornes sont atteintes.
2. Déterminer de même les bornes du nombre de comparaisons effectuées par l'algorithme de Boyer-Moore-Horspool et donner un exemple où elles sont atteintes.

1.2 Algorithme naïf avec motif à caractères uniques

Montrer qu'il est possible d'améliorer l'algorithme de recherche naïf si on suppose que les caractères du motif apparaissent une seule fois dans le motif.

- Faire par exemple la recherche de abcd dans le texte abceababccabcdabb et observer ce qu'il se passe lorsqu'on trouve un début de correspondance.

1.3 Algorithme de Boyer-Moore-Horspool (BMH) : table de décalages

Construire la table de décalages de BMH pour les motifs suivants.

1. banane
2. chercher

1.4 Algorithme BMH

1. Donner la table de décalage du motif tele
2. Donner les étapes du déroulement de l'algorithme BMH pour rechercher toutes les occurrences de ce motif dans le texte : le telephone ou la television.

1.5 Hachage de chaine de caractères

Dans ce qui suit, les fonctions suivantes à valeur entière sont définies pour s une chaine de M caractères codés en ascii. La fonction code(ch) retourne le code ascii du caractère ch (un entier entre 0 et 127). On choisit aussi un entier p (souvent premier) adapté aux entiers manipulés.

- $h_0(s)$ est la somme (modulo p) des codes ascii de s :

$$h_0(s_0 s_1 ... s_{M-1}) = (\sum_{j=0}^{M-1} c_j) \mod p,$$

où $c_j = \text{code}(s_j)$.
- $h_1(s)$ est la somme pondérée (modulo p) des codes ascii de s qui se calcule efficacement avec l'algorithme de Horner :

$$h_1(s_0 s_1 ... s_{M-1}) = (\sum_{j=0}^{M-1} B^{M-1-j} \times c_j) \mod p,$$

où $c_j = \text{code}(s_j)$. La "base" B est choisie empiriquement, par exemple $B = 31$ ou $B = 256$

1. Pour chacune des fonctions de hachage h_0 et h_1, écrire une relation entre les empreintes (les hachés) de deux sous-chaines de longueur M de s, l'une commençant au caractère s_i et l'autre au caractère s_{i+1}.
2. En déduire le coût arithmétique du calcul d'une empreinte à partir d'une empreinte déjà connue ?
3. Expliquer le rôle et la pertinence du "modulo p" dans ces fonctions de hachage ? Indication : de quelle limitation souffre ce calcul effectué dans un type uint arbitraire (sans modulo p) ?

2 TP avec codage en C

Afin de tester les algorithmes sur les textes, on dispose d'un fichier texte encodé au format ascii standard, c'est-à-dire qu'on utilise 128 caractères tous encodés sur 8 bits. De cette façon pour les algorithmes de recherche, on pourra représenter chaque caractère dans une table de 128 entiers – c'est-à-dire identifier un caractère à son code ascii.

On trouvera un exemple de fichier au format ascii sous le moodle de ce cours.

En pratique, il est assez pénible de produire de tels fichiers, par exemple à partir de fichiers textes au format utf-8 comme ceux des livres téléchargeables sur le site du projet Gutenberg. Sous linux, l'utilitaire iconv permet une conversion au format ascii, avec translittération (c'est-à-dire en remplaçant 'à' par 'a' ou encore 'é' par 'e') :

iconv -f utf-8 -t ascii//TRANSLIT <source> -o <destination>

2.1 Algorithme naïf et premiers traitements automatiques

1. Ecrire l'algorithme de recherche naïf qui renvoie l'indice de la première occurrence d'un motif m dans la chaine t, ou -1 si le motif ne se trouve pas dans la chaine.
   - On rappelle la signature de la fonction attendue :

   `int naive(char* m, char *t);`

   On écrira deux versions de cette recherche naive dont une avec la fonction strncmp de <string.h> pour comparer directement le motif à une sous-chaine de longueur donnée.
2. Ecrire une fonction naive_occ qui renvoie les indices de toutes les occurrences sous la forme d'une liste chainée d'entiers ou la chaine vide si aucune occurence n'est trouvée.
3. Modifier la fonction naive pour compter le nombre de comparaisons effectuées lors du traitement.
4. Tester la fonction de recherche naive en écrivant un programme main qui prend en arguments sur la ligne de commande un motif et un texte.
5. Expliciter des exemples pertinents du nombre de comparaisons effectuées.

2.2 Comparaison expérimentale des temps de traitement

Les expérimentations utiliseront le fichier ascii du texte disponible sur moodle (Notre dame de Paris, Victor Hugo, 1831). Ce fichier contient 1 068 362 caractères.

On pourra par exemple rechercher : a. une première occurrence de Quasimodo dans tout ou partie de ce fichier, b. les derniers mots du roman (hors notes et références) : “il tomba en poussiere.”

Pour les mesures de temps d'exécution, on pourra utiliser ce dépôt.

Méthodologie. Indiquons par exemple l'extrait suivant de la documentation de timeit de Python.

> Il est tentant de vouloir calculer la moyenne et l'écart-type des résultats et notifier ces valeurs. Ce n'est cependant pas très utile. En pratique, la valeur la plus basse donne une estimation basse de la vitesse maximale à laquelle votre machine peut exécuter le fragment de code spécifié ; les valeurs hautes de la liste sont typiquement provoquées non pas par une variabilité de la vitesse d'exécution de Python, mais par d'autres processus interférant avec la précision du chronométrage. Le min() du résultat est probablement la seule valeur à laquelle vous devriez vous intéresser. Pour aller plus loin, vous devriez regarder l'intégralité des résultats et utiliser le bon sens plutôt que les statistiques.

1. Ecrire une fonction filetostr qui prend en argument un nom de fichier et deux entiers pourcentage et start renvoie la chaine de caractères extraite des pourcentage % du fichier à partir du caractère start. Le fichier complet est obtenu par pourcentage=100 et start=0.

2. En faisant varier les paramètres de lecture du fichier ascii, effectuer des mesures du temps d'exécution de la recherche “modèle” par naive. On extraira du fichier ascii des portions successives et de pourcentages variables sur lesquelles on pourra par exemple rechercher (sans succès) les derniers mots du roman. pourcentage pourra prendre les valeurs : 0.1, 0.2, 0.5, 0.7, 0.9 et 1.0.
3. Tracer et analyser les résultats obtenus.

2.3 Algorithme naïf avec motif à caractères uniques

On suppose que tous les caractères du motif apparaissent une seule fois dans le motif.

1. Coder et valider naiveopt, nouvelle version de naive.
2. Modifier naiveopt pour compter le nombre de comparaisons effectuées lors du traitement.
3. Comparer naive et naiveopt sur deux exemples pertinents (sans ou avec réduction de nombre de comparaisons).
4. Adapter la méthodologie déjà suivie pour naive afin de compléter la comparaison expérimentale des temps de traitement de naiveopt. Produire une figure qui permette d'analyser succinctement les résultats obtenus.

2.4 Algorithme de Boyer-Moore-Horspool

1. Ecrire une fonction decalage qui produit la table des décalages d'un motif m arbitraire. Vérifier cette fonction avec les exercices de TD.
   - On rappelle qu'on utilise 128 caractères. On connait donc la taille de la table des décalages.
2. Ecrire et tester une fonction bmh qui applique l'algorithme BHM et renvoie l'indice de la première occurrence d'un motif m dans la chaine t, ou -1 si le motif ne se trouve pas dans la chaine.
3. Modifier l'implémentation de bmh pour compter le nombre de comparaisons effectuées lors du traitement.
4. Expliciter des exemples pertinents du nombre de comparaisons effectuées.

5. Compléter la comparaison expérimentale des temps de traitement avec cet algorithme. Produire une figure qui permette d'analyser succinctement les résultats obtenus.

2.5 Hachage de chaine de caractères

On reprend les fonctions de hachages h_0 et h_1 ici définies pour s une chaine de len caractères ascii (7 bits).

1. Ecrire ces fonctions selon les signatures suivantes.

```
uint64_t h0(char *s, int len);
uint64_t h1(char *s, int len);
```

2. Justifier que le choix $p = 1\ 869\ 461\ 003$ est adapté au type uint64_t.
3. Comparer expérimentalement l'efficacité de ces deux fonctions. Méthode. Dans le texte du fichier ascii du disponible sur moodle :
   - dénombrer les chaines différentes de longueur 5 et leurs collisions ;
   - faire de même avec des chaines de longueur 10 ;
   - identifier un motif d'empreinte distincte de toutes les sous chaines de même longueur extraites du texte.

2.6 Algorithme de Rabin-Karp

1. Ecrire une implémentation rk0 de l'algorithme de Rabin-Karp en utilisant la fonction de hachage h_0.
2. Modifier cette fonction afin d'obtenir en plus le nombre de collisions.
3. En recherchant ab dans le fichier disponible sur moodle, combien de collisions ne sont pas des occurrences de ce motif ?
4. Reprendre les questions précédentes pour rk1 avec la fonction de hachage h_1.
5. Compléter la comparaison expérimentale des temps de traitement avec rk1. Produire une figure qui permette d'analyser succinctement les résultats obtenus.

Sujet généré par :

pandoc -N --metadata-file=header_td.yaml --pdf-engine=xelatex -o td-t1.pdf td-t1.md