Université de Rouen

Implémentation de l'algorithme d'Aho-Corasick

Projet d'Algorithmique du texte

Auteurs :
Florian Inchingolo
Yohann Henry

Responsables :
M. Thierry Lecroq
M. Nicolas Bedon
M. Arnaud Lefebvre

13 janvier 2016

Table des matières

1 But du projet

Le but de ce projet est d'implanter l'algorithme d'*Aho-Corasick* pour compter le nombre d'occurrences exactes d'un ensemble de k mots dans un texte. Nous devions utiliser trois méthodes pour représenter l'arbre :

- une matrice de transitions ;
- un tableau de listes d'adjacence ;
- une table de transitions pour la racine et un tableau de listes d'adja- cence pour les autres noeuds de l'arbre.

Les exécutables devaient impérativement être nommés **ac-matrice**, **ac-liste**, **ac-mixte** respectivement et prendre deux paramètres, d'abord le nom du fichier qui contient les mots à rechercher (un par ligne) puis le nom du fichier qui contient le texte.

2 Contenu de l'archive

```
bin ..... Contient les exécutables
├── ac-liste ..... Aho-corasick avec des listes
├── ac-matrice ..... Aho-corasick avec des matrices
├── ac-mixte ..... Aho-corasick mixte
├── genere-mots ..... Générateur de mots
└── genere-texte ..... Générateur de texte
gen ..... Contient les mots et textes générés
obj ..... Objets issus de la compilation
scripts ..... Scripts de démonstration
├── clean.sh ..... Nettoie le projet et les fichiers générés
└── run.sh ..... Script de démonstration du projet
src ..... Sources du projet
└── include ..... En-têtes des sources
log.csv ..... Contient un rapport sur les temps d'éxécution
makefile ..... Makefile du projet
rapport.pdf ..... Ce rapport
README.txt ..... Utilisation rapide du projet
```

3 Utilisation des programmes

Les commandes ci-dessous se font depuis le dossier racine du projet décompréssé.

3.1 Script de démonstration

Le programme de démonstration construit le projet, génère pseudo-aléatoirement des textes de longueur 5 000 000 sur des alphabets de taille 2, 4, 20 et 70, puis pour chacun des alphabets génère pseudo-aléatoirement 3 ensembles de 100 mots de longueur entre 5 et 15, entre 15 et 30 et entre 30 et 60 respectivement. Ensuite, il exécute pour les 3 types de trie la rechereche des 3 ensembles dans les textes respectifs, ce qui fait en tout 36 exécutions. Ces programmes remplissent un fichier log.csv permettant de comparer les résultats.

La commande est la suivante :

```
./scripts/run.sh
```

3.2 Programme de génération de mots

Le programme affiche sur la sortie standard une série de mots séparés par un retour chariot, que l'on peut rediriger dans un fichier.

La commande est la suivante :

```
./bin/genere-mots NOMBRE MIN_SIZE MAX_SIZE ALPHA_SIZE
```

Les paramètres sont les suivants :
— **NOMBRE** : le nombre de mots à obtenir ;
— **MIN_SIZE** : la taille minimale d'un mot ;
— **MAX_SIZE** : la taille maximale d'un mot ;
— **ALPHA_SIZE** : la taille de l'alphabet utilisé.

3.3 Programme de génération de texte

Le programme affiche sur la sortie standard le texte obtenu, que l'on peut rediriger dans un fichier.

La commande est la suivante :

```
./bin/genere-texte LENGTH ALPHA_SIZE
```

Les paramètres sont les suivants :
— **LENGTH** : la taille du texte à obtenir,
— **ALPHA_SIZE** : la taille de l'alphabet utilisé.

3.4 Comptage du nombre d'occurences d'un ensemble de mots dans un texte

Pour les trois programmes suivants, **MOTS** est le fichier contenant l'ensemble de mots séparés par un espace, **TEXTE** est le fichier comprenant le texte à rechercher, et **LOGFILE** est un paramètre optionnel permettant de changer le fichier des résultats (par défaut log.csv). Si ce fichier n'existe pas, il est créé, sinon, les résultats sont ajoutés à la suite des précédents.

Avec une matrice de transisitions

La commande est la suivante :

```
./bin/ac-matrice MOTS TEXTE [LOGFILE]
```

Avec un tableau de listes d'adjacence

La commande est la suivante :

```
./bin/ac-liste MOTS TEXTE [LOGFILE]
```

En mixte

La commande est la suivante :

```
./bin/ac-mixte MOTS TEXTE [LOGFILE]
```

4 Structures de données

4.1 Liste ordonnée

Cette liste ordonnée contient un caractère (le caractère du noeud), l'identifiant du prochain noeud ainsi que l'élément suivant. Elle est triée par ordre alphabétique, car cela permet de récupérer plus rapidement une suite de lettres avec l'itérateur intégré.

```
typedef struct _listelem *ListElem;

struct _list {
    ListElem hd;
    ListElem it;
    size_t len;
};

struct _listelem {
    unsigned char c;
    size_t id;
    ListElem nx;
};

typedef struct _list *SortedList;
```

Structure d'une liste ordonnée.

4.2 Trie matriciel

```
struct _trie {
    size_t max;
    size_t next;
    size_t **trans;
    unsigned *finite;
    bool *truefinite;
    size_t *fallback;
};
```

Structure d'un trie matriciel.

4.3 Trie en liste

```
struct _trie {
    size_t max;
    size_t next;
    SortedList *trans;
    unsigned *finite;
    bool *truefinite;
    size_t *fallback;
};
```

Structure d'un trie en liste.

4.4 Trie mixte

```
struct _trie {
    size_t max;
    size_t next;
    size_t* first;
    SortedList *trans;
    unsigned *finite;
    bool *truefinite;
    size_t *fallback;
};
```

Structure d'un trie mixte.

5 Algorithmes présentés

5.1 Insertion

Algorithme présenté

```
1  Données :
2  trie : le trie utilisé,
3  mots : les mots à ajouter,
4  nb : le nombre de mots à ajouter,
5  Résultat :
6  Les mots sont ajoutés dans le trie.
7  pos <- 0;
8  while mots_non_finis() do
9  |  i <- 0;
10 |  pos <- pos + 1;
11 |  while i < nb do
12 |  |  ajouter_lettre();
13 |  |  if mot_fini() then
14 |  |  |  ajouter_etat_final();
15 |  |  end
16 |  |  substitution();
17 |  |  i <- i + 1;
18 |  end
19 end
```

Explications sur l'insertion

L'insertion dans notre trie se fait par indice plutôt que par mot. La i-ième lettre de chaque mot est lue au tour de boucle i. Les raisons sont détaillés dans le prochain point.

5.2 Fonction de suppléance

Algorithme présenté

```
1  Données :
2  trie : le trie utilisé,
3  dernier_pref : la position de la dernière suppléance du mot,
4  mot : le mot en cours d'insertion,
5  position_actuelle : la longueur des mots insérés pour le moment.
6  Résultat :
7  Le noeud de suppléance est ajouté au trie,
8  derinier_pref est mis à jour.
9  while dernier_pref <= position_actuelle do
10 |  position_trie <- position_trie(mot[dernier_pref]);
11 |  if position_trie != 0 then
12 |  |  ajouter le noeud de suppléance;
13 |  |  dernier_pref <- position_trie;
14 |  end
15 |  dernier_pref <- dernier_pref + 1;
16 end
```

Preuve sur la complexité

Le but de la fonction de suppléance est de trouver le noeud du trie contenant le suffixe le plus long de chacun des préfixes pour chaque mot inséré.

Dans une insertion "classique", cette fonction de suppléance est exécutée à la fin de l'insertion des mots, chaque mot étant découpé pour récupérer le bon suffixe pour tous les préfixes de ce mot. Le pire des cas, dans cette méthode sans optimisation, étant que le mot n'est pas compris dans le trie pour tout suffixe et pour tout préfixe, nous obtenons une complexité temporelle en $\mathcal{O}(n^2)$.

Avec notre méthode d'insertion lettre par lettre, la fonction de suppléance est exécutée après que la lettre i de chaque mot est insérée. À ce moment là, nous avons dans le trie tous les préfixes de taille i de tous les mots. Cela n'est pas un problème, car la longeur de tous les suffixes d'un mot de taille i ne peut pas être plus grand que i. En exécutant cette fonction à chaque groupement de lettres, nous l'éxécutons donc sur chacun des préfixes de tous les mots à insérer.

De plus, si un suffixe s de longueur i d'un mot m n'est pas contenu dans un trie, alors s concaténé de la lettre suivante a de longueur $i+1$ du mot ne peut pas être contenu dans un trie, vu que s n'est pas dans le trie. Il est seulement possible d'obtenir $s+a$ dans le trie si s est dans le trie. Nous pouvons donc sauvegarder à chaque exécution de la fonction de suppléance le dernier suffixe obtenu ainsi que sa position dans le trie, pour repartir de celui-ci quand le prochain groupement de lettres est inséré. Comme cela, le pire des cas est que chaque préfixe d'un mot a comme suffixe dans le trie la dernière lettre insérée, ce qui nous fait lire chaque lettre du mot deux fois. Cela nous donne une complexité temporelle en $\mathcal{O}(n)$.

L'inconvénient de cette technique est qu'il est impossible d'ajouter d'autres mots une fois le trie créé ; mais dans le cas de notre projet, cette opération n'est pas demandée.

5.3 Aho-Corasick

Algorithme présenté

```
1  Données :
2  trie : le trie utilisé,
3  texte : le texte à lire.
4  Résultat :
5  Le nombre d'occurences de tous les mots dans le texte.
6  lettre <- caractère_suivant(texte);
7  while est_valide(lettre) do
8  |  actuel <- suivant(trie, lettre, actuel) ; count <- count + etats_finaux(actuel);
9  |  lettre <- caractère_suivant(texte);
10 end
11 renvoyer count;
```

Aho-corasick.

```
1  Données :
2  trie : le trie utilisé,
3  lettre : la lettre lue,
4  actuel : la position actuelle dans le trie.
5  Résultat :
6  L'identifiant du prochain noeud à accéder.
7  prochain <- transition(actuel, lettre);
8  if prochain = 0 then
9  |  if actuel == 0 then
10 |  |  renvoyer 0;
11 |  end
12 |  renvoyer suivant(trie, lettre, prochain);
13 end
14 renvoyer prochain;
```

Fonction suivant.

6 Résultats obtenus

Avec le fichier de journal obtenu lors de la recherche des temps d'exécution, nous avons remarqué qu'il devient rapidement difficile d'obtenir ne serait-ce qu'une occurence quand l'alphabet et la longueur du mot sont grands, même sur un texte de 5 millions de caractères.

6.1 Temps d'éxécution

Pour obtenir des courbes sur les temps d'éxécution de nos trois programmes, nous avons utilisé un script utilisant 100 fois run.sh, ce qui nous a donné 3600 résultats différents en fonction du programme utilisé, des tailles d'alphabets et des longueurs des mots, tous enregistrés dans le fichier log.csv. Cette opération nous a pris environ 15 minutes, même s'il est à noter que les temps d'éxécution varient grandement selon la machine utilisée.

Graphique sur les temps d'exécution

TODO graph

Conclusion sur les temps d'éxécution

TODO mettre des nombres issus du graph

Nous voyons donc que dans tous les cas, le trie matriciel est plus rapide que le mixte et le trie en liste.

6.2 Taille dans la mémoire

Afin d'obtenir la taille utilisée dans la mémoire pour chaque programme, nous avons utilisé *Valgrind*. Cet outil permet de corriger et de monitorer les variables utilisées par un programme. Pour l'utiliser, nous avons exécuté la commande suivante :

```
valgrind --leak-check=full --show-leak-kinds=all
--track-origins=yes PROGRAMME
```

Avec **PROGRAMME** le nom du programme à vérifier. *Valgrind* donne ensuite les fuites de mémoires possible ainsi que le nombre d'octets pris dans la mémoire lors de l'éxécution du programme.

Résultats sur la taille mémoire

```
$ valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes
bin/ac-matrice  gen/mots_a4_l5 gen/texte_a4
==4816== Memcheck, a memory error detector
==4816== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==4816== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==4816== Command: bin/ac-matrice gen/mots_a4_l5 gen/texte_a4
==4816==
[bin/ac-matrice] - count : 56730 (time : 2833ms)
==4816==
==4816== HEAP SUMMARY:
==4816==     in use at exit: 0 bytes in 0 blocks
==4816==   total heap usage: 765 allocs, 765 frees, 1,370,036 bytes allocated
==4816==
==4816== All heap blocks were freed -- no leaks are possible
==4816==
==4816== For counts of detected and suppressed errors, rerun with: -v
==4816== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

$ valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes
bin/ac-liste  gen/mots_a4_l5 gen/texte_a4
==4815== Memcheck, a memory error detector
==4815== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==4815== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==4815== Command: bin/ac-liste gen/mots_a4_l5 gen/texte_a4
==4815==
[bin/ac-liste] - count : 56730 (time : 7132ms)
==4815==
==4815== HEAP SUMMARY:
==4815==     in use at exit: 0 bytes in 0 blocks
==4815==   total heap usage: 1,514 allocs, 1,514 frees, 75,596 bytes allocated
==4815==
==4815== All heap blocks were freed -- no leaks are possible
==4815==
==4815== For counts of detected and suppressed errors, rerun with: -v
==4815== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

$ valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes
bin/ac-mixte  gen/mots_a4_l5 gen/texte_a4
==4817== Memcheck, a memory error detector
==4817== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==4817== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==4817== Command: bin/ac-mixte gen/mots_a4_l5 gen/texte_a4
==4817==
[bin/ac-mixte] - count : 56730 (time : 7206ms)
==4817==
==4817== HEAP SUMMARY:
==4817==     in use at exit: 0 bytes in 0 blocks
==4817==   total heap usage: 1,510 allocs, 1,510 frees, 77,524 bytes allocated
==4817==
==4817== All heap blocks were freed -- no leaks are possible
==4817==
==4817== For counts of detected and suppressed errors, rerun with: -v
```

```
==4817== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Nous avons donc des complexités en espace, pour un fichier en entrée de *5mo* :
— **Matrice** : 1370,036 ko
— **Liste** : 75,596 ko
— **Mixte** : 77,524 ko

Conclusion sur la taille mémoire

Nous pouvons donc constater sur cet exemple, que le trie avec des matrices de transitions prend environ *17* fois plus de place que les deux autres, à cause de ses nombreuses cases vides. Même si le trie prend relativement que peu de place (*1,3mo* sur un fichier de *5mo*), s'il faut privilégier l'espace par rapport au temps, le trie mixte ou en liste serait un bon choix.