Structure de données et algorithmes

Projet 2: Scrabble

1 avril 2022

L'objectif du projet est d'implémenter, d'analyser théoriquement et de comparer empiriquement différents implémentation d'un algorithme de recherche dans un dictionnaire du plus long mot pouvant être formé à partir d'un ensemble de lettres fixées par l'utilisateur. L'objectif pédagogique est de vous faire implémenter et utiliser différentes structures de données de type dictionnaires pour résoudre un problème algorithmique. Vous apprendrez également plus sur une nouvelle structure, le Trie, non vue au cours.

Recherche d'un plus long mot

Vous aimez le scrabble et, fort de votre expérience en structures de données et algorithmes, vous aimeriez développer un outil pour vous aider à trouver des mots à partir d'un ensemble de lettres.

Le problème sera simplifié et formalisé de la manière suivante. Soit un ensemble[1] S composé de m lettres[2] de l'alphabet et une structure D contenant n mots, on souhaite écrire une fonction FindLongestWord(S, D) renvoyant un plus long mot dans D dont toutes les lettres sont dans S, avec la bonne multiplicité. La structure de données D sera construite via une fonction CreateDict(L) prenant en argument une simple liste L des n mots.

Dans nos choix d'implémentation, on veillera en priorité à minimiser les temps d'appel à la fonction FindLongestWord et à utiliser la structure de données la plus économe en espace. La fonction CreateDict devra être aussi efficace que possible mais pas au détrimant de l'efficacité de la fonction FindLongestWord.

On se propose d'étudier 4 solutions à ce problème décrite ci-dessous.

Solution par liste. L'idée de cette première solution est de garder pour la structure D la liste des n mots et ensuite, pour un ensemble S de lettres, à parcourir la liste de mots, à vérifier pour chacun d'eux s'il est couvert par l'ensemble de lettres S et à renvoyer le plus long mot trouvé lors du parcours.

Solution par table de hachage. L'idée de cette seconde solution est de stocker les mots dans un dictionnaire implémenté par une table de hachage en utilisant comme valeur de clé pour chaque mot ses lettres *triées par ordre alphabétique*. On peut ensuite générer tous les sous-ensembles de lettres de l'ensemble S, pour chacun d'eux vérifier s'il correspond à un mot stocké dans le dictionnaire par une simple recherche et finalement renvoyer le mot le plus long ainsi identifié.

[1] Strictement, dans la suite, S sera un multiensemble, c'est-à-dire un ensemble pouvant contenir plusieurs fois le même élément.

[2] On ne prendra en compte que les lettres minuscules de a à z et on ignorera les accents et autres symboles tels que le tiret.

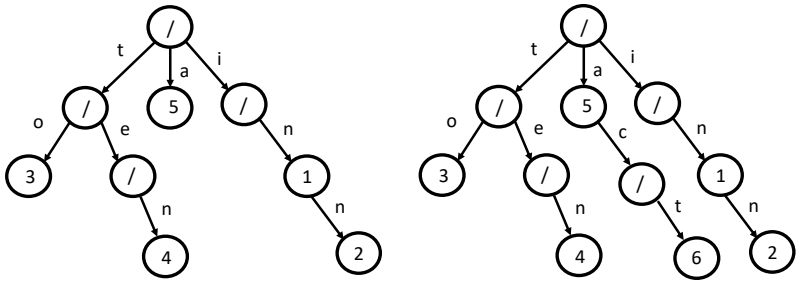

Figure 1: A gauche, un Trie qui contient les clés `to`, `ten`, `a`, `in`, et `inn` avec comme valeurs associées respectivement 3, 4, 5, 1, et 2. A droite, le même Trie après insertion de la clé `act` avec comme valeur associée 6. Une nœud marqué d'un `/` signifie que la donnée associée est un pointeur `NIL` et que donc la chaîne n'appartient pas au dictionnaire.

Solution par Trie. Cette solution est identique à la précédente si ce n'est que l'implémentation du dictionnaire pour stocker les clés (toujours obtenues du tri des lettres de chaque mot) est basée sur un Trie, une implémentation d'un dictionnaire dédiée aux clés sous la forme de chaînes de caractères uniquement[3]. Un Trie est une structure d'arbre (non nécessairement binaire), dans laquelle chaque arête correspond à une lettre de l'alphabet et chaque nœud correspond à une clé qui est donnée par la concaténation des lettres rencontrées depuis la racine jusqu'à ce nœud. Chaque nœud contient ainsi une valeur correspondant à NIL lorsque la clé n'est pas dans le dictionnaire et un pointeur vers la donnée associée lorsque la clé est dans le dictionnaire. Pour implémenter la recherche d'une clé, il suffit de parcourir la branche indiquée par les lettres de la clés. Si lors de ce parcours, on doit parcourir une branche qui n'existe pas ou si on atteint un nœud contenant comme valeur NIL, le mot n'appartient pas au dictionnaire. Sinon, on renvoie la valeur associé au nœud auquel on s'arrête. Pour implémenter l'insertion, on crée si nécessaire la branche associée aux lettres de ce mot et on stocke au nœud atteint la donnée associée. Un Trie et l'effet de l'insertion d'une clé sont illustrés à la figure 1.

Solution par Trie optimisée. Comme la solution précédente, l'idée de cette solution est de stocker l'ensemble des clés (triées) dans un Trie, mais cette fois de court-cicuiter l'étape de parcours de tous les sous-ensembles de S en exploitant la structure de Trie. Il est en effet possible de parcourir la structure d'arbre de manière intelligente à la recherche du mot le plus long. Le principe est de maintenir en chaque nœud, lors d'un parcours en profondeur, l'ensemble S' des lettres de S non rencontrées entre la racine et ce nœud et de n'investiguer à partir de ce nœud que les arêtes marquées d'une lettre contenue dans S'. Chaque fois qu'un nœud contenant une clé est atteint, il s'agit d'un mot couvert par S et il suffit de renvoyer le mot le plus long rencontré lors de ce parcours.

Implémentation

Les fichiers suivants vous sont fournis:

- `LinkedList.c/LinkedList.h`: Une implémentation simple de liste (liée).
- `Dict.h`: le fichier d'entête pour un dictionnaire. Pour minimiser le nombre de fichiers à implémenter et éviter de dupliquer du code, l'interface `Dict.h` contient la fonction `dictSearchLongest` qui sera utilisée pour la solution par Trie optimisée. Cette fonction n'est pas implémentée dans le cas de la table de hachage.

[3] `https://en.wikipedia.org/wiki/Trie`

- `HashTable.c`: une implémentation d'un dictionnaire générique par table de hachage selon l'interface `Dict.h`.
- `Scrabble.h`: le fichier d'entête pour une solution au problème du scrabble.
- `ListScrabble.c`: une implémentation de la solution par liste décrite ci-dessous selon l'interface donnée dans `Scrabble.h`.
- `main.c`: un exemple d'utilisation de l'implémentation. Ce fichier crée un exécutable qui prend comme argument un ensemble de lettres et un fichier texte contenant des mots (un par ligne) et renvoie le mot le plus long en utilisant la fonction SCRABBLEFINDLONGESTWORD fournie par l'une des implémentation. Si aucun argument n'est donné à la fonction, elle réalise 1000 recherches pour des ensembles aléatoires contenant de 5 à 15 lettres et donne les temps de calcul pour la création de la structure et les 1000 recherches. Le fichier `Makefile` permet de créer les exécutables correspondant aux 4 solutions qu'on vous demande de comparer.
- `words_small.txt`, `words_large.txt`: deux listes de mots anglais pour vos expériences, de tailles différentes. `words_small.txt` contient 58110 mots, alors que `words_large.txt` contient 370102 mots.

On vous demande d'implémenter (ou de modifier) les fichiers suivants:

- `HashTable.c`: la table de hachage utilise un tableau de taille fixe. On vous demande de modifier la fonction d'insertion pour qu'elle utilise un tableau extensible plutôt qu'un tableau de taille fixe.
- `ListScrabble.c`: l'implémentation de la fonction MATCH qui vérifie qu'une ensemble de lettres couvre un mot est naïve. On vous demande de trouver une manière de la rendre plus efficace (voir la question 1 du rapport).
- `Trie.c`: implémentant le dictionnaire dont l'interface est donnée par `Dict.h` en utilisant un Trie tel que décrit ci-dessus. La fonction `dictSearchLongest` devra implémenter la recherche du mot le plus long selon le principe de la solution par Trie optimisée. Les lettres de la chaîne `letters` donnée en argument à cette fonction sont supposées triées.
- `DictScrabble.c`: implémentant les solutions par dictionnaire (table de hachage ou Trie). Le choix de l'implémentation du dictionnaire serait fait à la compilation en sélectionnant `HashTable.c` ou `Trie.c`.
- `TrieScrabble.c`: implémentant la solution par Trie optimisée.

Vos fichiers seront évalués sur les machines ms8xx avec les flags de compilation habituels (aussi utilisé dans le fichier `Makefile` fourni): `--std=c99 --pedantic -Wall -Wextra -Wmissing-prototypes`

Rapport

Dans le rapport, on vous demande de répondre aux questions suivantes:

1. Implémentez une version plus efficace de la fonction `match` fournie dans le fichier `ListScrabble.c`:
   (a) Donnez le pseudo-code de la fonction `match` que vous avez implémentée.
   (b) Comparez ses complexités en temps et en espace dans le pire cas par rapport à celle qui vous était fournie (en fonction de la taille m de l'ensemble de lettres et de la longueur l du mot).
   (c) Mesurez empiriquement l'impact de votre amélioration en termes de temps de calcul pour la recherche de 1000 mots dans le grand dictionnaire.

2. Etudiez les complexités en temps et en espace dans le pire cas des deux fonctions `CreateDict` et `FindLongestWord` pour les 4 solutions en fonction du nombre n de mots dans le dictionnaire et du nombre m de lettres dans S, la longueur l (moyenne) des mots du dictionnaires. Justifiez ces complexités en expliquant les spécificités de vos implémentations.
   Remarque: une analyse rigoureuse détaillée de ces complexité peut être très compliquée. On ne vous demande pas de faire cette analyse de manière très formelle. L'idée est surtout de mettre en évidence les différences entre les solutions.

3. Au moyen du fichier `main.c` fourni, mesurez les temps de calcul empirique des 4 solutions, pour la construction de la structure et la recherche de 1000 mots, dans le cas du petit et du grand dictionnaire fourni.

4. Sur base de l'analyse théorique et des résultats empiriques, faites une brève analyse critique des différentes solutions.

Soumission

Le projet est à réaliser ***en binôme*** pour le ***jeudi 28 avril 2022*** au plus tard. Le projet est à remettre via la plateforme de soumission de Montefiore: `http://submit.montefiore.ulg.ac.be/`.

Il doit être rendu sous la forme d'une archive `tar.gz` contenant à sa racine:

1. votre rapport (5 pages maximum) au format PDF et nommé `rapport.pdf`. Soyez bref mais précis et respectez bien la numération des questions;

2. les cinq fichiers `.c` précisés plus haut.

Respectez bien les extensions de fichiers ainsi que les noms des fichier `*.c` (en ce compris la casse). N'incluez aucun fichier supplémentaire.

Un projet non rendu à temps recevra une cote globale nulle. En cas de plagiat[4] avéré, l'étudiant se verra affecter une cote nulle à l'ensemble des projets.

Les critères de correction sont précisés sur Ecampus.

Bon travail !

[4]Des tests anti-plagiat seront réalisés.