Structures de données et algorithmes

Projet 2: Mots mêlés

29 mars 2024

L'objectif de ce second projet est d'implémenter, d'analyser théoriquement et de comparer empiriquement différentes implémentations d'un algorithme de résolution d'un jeu de type "Mots mêlés". L'objectif pédagogique est de vous faire implémenter et utiliser différentes structures de données de type dictionnaire pour résoudre un problème algorithmique. Vous y apprendrez également une nouvelle structure de ce type, l'arbre radix, non vue au cours.

Formulation du problème

Le problème sera simplifié et formalisé de la manière suivante. Soient une grille G carrée de dimension $n \times n$ dont chaque case contient une seule lettre[1] de l'alphabet et une liste L de m mots, qu'on appellera le lexique. On souhaite écrire un algorithme renvoyant le sous-ensemble des mots de L qui se trouvent dans la grille G. Un mot se trouve dans la grille s'il est possible de le construire à partir de la grille en partant d'une position arbitraire et en se déplaçant en ligne droite vers le haut, le bas, la gauche, la droite ou une des 4 directions diagonales (voir un exemple à la figure 1).

On se propose d'implémenter et comparer deux approches :

- **La recherche dirigée par le lexique**, consistant à vérifier pour chaque mot du lexique L si celui-ci est présent dans la grille.
- **La recherche dirigée par la grille**, consistant à parcourir l'ensemble des mots de la grille pour déterminer ceux qui se trouvent dans le lexique.

La première approche est plus simple à implémenter mais elle requiert le parcours (éventuellement complet) de la grille pour chaque mot de notre lexique, et est donc potentiellement très coûteuse. La deuxième approche n'effectue qu'un seul parcours de la grille mais elle requiert de pouvoir déterminer rapidement si un mot de la grille appartient au lexique. Pour rendre cette recherche efficace, les mots du lexiques L seront préalablement stockés dans une structure de données de type ensemble[2] ("Set" en anglais).

La recherche dirigée par la grille utilisera donc deux fonctions : une fonction SetCreate(L) qui créera l'ensemble S à partir du lexique et une fonction GetAllWordsFromSet(G, S) qui renverra la liste de tous les mots de la grille G qui sont présents dans S. Dans nos choix d'implémentation, on veillera en priorité à minimiser les temps de calcul et dans une moindre mesure l'espace mémoire nécessaire. La fonction SetCreate devra être aussi efficace que possible mais pas au détriment de l'efficacité de la fonction GetAllWordsFromSet, dans l'idée

[1] 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.

[2] Un type ensemble est équivalent à un dictionnaire tel que vu au cours dans lequel cependant aucune valeur n'est associée aux clés.

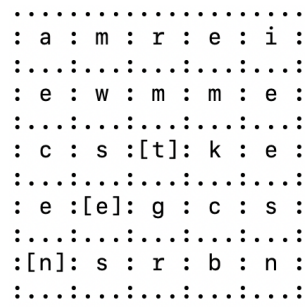


Figure 1: Un exemple de grille 5×5 dans laquelle on trouve le mot "ten".

que cette dernière pourrait être appelée sur plusieurs grilles alors que la première ne sera appelée qu'une seule fois.

Nous comparerons dans ce projet trois structures d'ensemble : la table de hachage, l'arbre binaire de recherche et l'arbre radix. Les deux premières structures ont été vues au cours. Elles seront particularisées dans ce projet à des clés sous la forme de chaîne de caractères. Pour la table de hachage, la fonction de codage utilisera un encodage en base 26 de la clé et la méthode de hachage par division vues au cours. Pour l'arbre binaire de recherche, les clés seront comparées sur base de l'ordre lexicographique[3]

L'arbre radix[4] (en anglais "radix tree" ou "radix trie") est une structure non vue cours particulièrement adaptée pour les chaînes de caractères. Nous vous donnons une description brève de cette structure ci-dessous mais vous êtes libre de consulter des sources supplémentaires pour en apprendre plus par vous-même.

Arbre radix

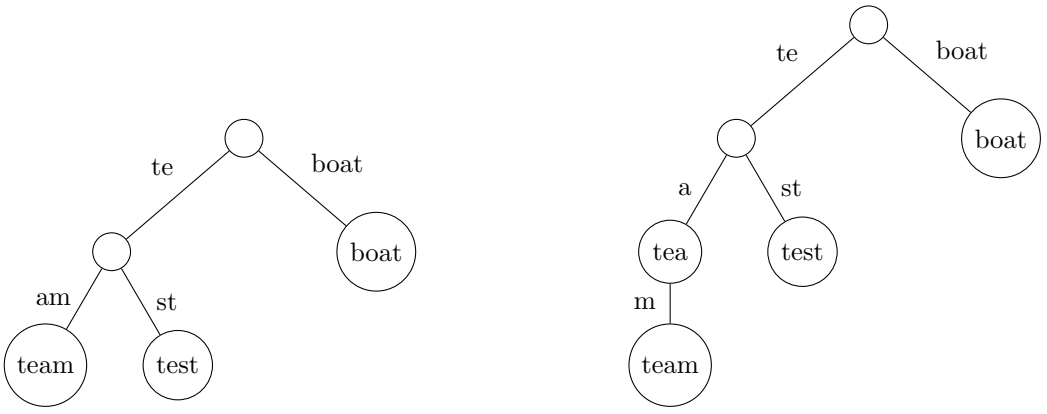
Un arbre radix représente l'ensemble sous la forme d'un arbre, non nécessairement binaire. Chaque arête de l'arbre est associée à une chaîne de caractères et certains noeuds sont associées à une clé (toutes les feuilles le sont mais pas tous les noeuds internes). Chaque nœud de l'arbre correspond à un préfixe formé en concaténant les chaînes rencontrées sur le chemin entre la racine et ce nœud. Si ce préfixe est une clé contenue dans l'ensemble, cette clé est contenue dans le nœud, sinon le nœud est vide (voir Figure 2 pour un exemple d'arbre radix).

la recherche d'une clé s'effectue en parcourant l'arbre depuis la racine tant que le préfixe correspondant à un des nœuds fils du nœud courant est un préfixe de la clé recherchée. Si la clé d'un nœud parcouru correspond à la clé recherchée, la recherche est positive. Si le parcours s'arrête parce qu'aucun fils ne correspond à un préfixe de la clé recherchée, la recherche est négative.

L'insertion d'une clé est plus complexe. On procède comme pour la recherche jusqu'à s'arrêter en un nœud. Si ce nœud contient la clé à insérer, on ne fait rien, la clé étant déjà dans l'ensemble. Si ce nœud ne contient aucune arête sortante possédant un préfixe commun au reste de clé à insérer, on crée une nouvelle arête étiquetée par le reste de la clé et un nœud fils étiqueté par la clé. Si une arête contient un préfixe du reste de la clé, on crée une nouvelle arête avec ce préfixe, un nœud intermédiaire sans clé au bout de cette arête et deux arêtes sortant de ce nœud intermédiaire avec les suffixes restant de l'ancienne arête et de la clé (voir Figure 2b pour un exemple d'insertion, où l'une des deux arêtes sortant du nœud intermédiaire correspond à une chaîne vide et peut donc ne pas être créée).

[3]Implémenté en C par la fonction `strcmp`.

[4]`https://en.wikipedia.org/wiki/Radix_tree`



(a) Arbre radix (b) Arbre radix après ajout

Figure 2: À gauche, un arbre radix contenant les clés `team`, `test` et `boat`. Un nœud vide signifie que le préfixe associé n'est pas présent dans l'ensemble. À droite, le même arbre radix auquel la clé `tea` a été ajoutée, requérant la séparation de la branche `am` afin de stocker cette nouvelle clé dans un nœud interne.

Vérification de tous les préfixes

Les trois implémentations d'ensemble permettent de vérifier plus ou moins efficacement la présence d'un mot donné (en fonction du nombre de mots dans l'ensemble et de la longueur du mot recherché). Un des intérêts de l'arbre radix pour notre application est de permettre de vérifier en une seule descente au travers d'une branche de l'arbre la présence de tous les préfixes d'une chaîne dans l'ensemble, plutôt que de faire indépendamment la recherche de chaque préfixe de cette chaîne dans l'ensemble. Par exemple, on peut déterminer efficacement que la chaîne "teamieu" contient les mots "tea" et "team" pour l'arbre de la Figure 2b. Cette possibilité permet d'accélérer l'implémentation de la recherche dirigée par la grille qui demande en effet de vérifier tous les préfixes de la plus longue chaîne partant d'une position dans chacune des huit directions possibles. Pour vous permettre d'implémenter cette solution, on ajoutera à l'interface de l'ensemble une fonction GetAllStringPrefixes(S, s) qui renverra la liste des préfixes de la chaîne de caractères s présents dans l'ensemble S. Vous pourrez ensuite utiliser cette fonction pour implémenter la recherche dirigée par la grille.

Il est intéressant de noter qu'il est possible d'implémenter également efficacement cette fonction GetAllStringPrefixes(S, s) dans le cas de la table de hachage et de l'arbre de binaire de recherche.

Implémentation

Les fichiers suivants vous sont fournis :

- `List.c/List.h` : Une implémentation simple de liste (liée).
- `Set.h` : Une interface pour une structure de type dictionnaire (ensemble). Pour simplifier l'implémentation, cette interface fait l'hypothèse que les clés sont des chaînes de caractères et on n'implémentera pas la suppression de clé, inutile pour ce projet.
- `Set_hashtable.c` : Une implémentation d'un ensemble par table de hachage selon l'interface `Set.h`.
- `Set_BST.c` : Une implémentation d'un ensemble par arbre binaire de recherche selon l'interface `Set.h`.

- `Board.h/Board.c` : Une implémentation (partielle) de la grille de jeu.
- `searchbylexicon.c` : Une implémentation, directement dans une fonction `main`, de l'approche dirigée par le lexique. La compilation de ce fichier crée un exécutable qui prend en argument un fichier texte contenant les mots du lexique (un par ligne) et une taille de grille, et qui calcule la liste des mots du lexique qui se trouvent dans une grille de cette taille générée aléatoirement (les lettres de la grille sont choisies aléatoirement sur base de leurs fréquences dans les mots du lexique). Si la grille n'est pas trop grande, elle est affichée à l'écran avec une mise en évidence du mot le plus long trouvé. Les temps de calcul sont également affichés, ainsi que le nombre total de mots trouvés.
- `searchbyboard.c` : Une implémentation similaire à la précédente pour le cas de la recherche dirigée par la grille. Cette implémentation fait cependant appel à la fonction `boardGetAllWordsFromSet` que vous devrez implémenter dans `Board.c`.
- `Makefile`, permettant de créer les exécutables correspondant aux 4 solutions qu'on vous demande de comparer.
- `english.txt` : Une liste de (128737) mots anglais à utiliser comme lexique pour vos expériences.

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

- `Board.c` : On vous demande d'implémenter la fonction `boardGetAllWordsFromSet` qui renvoie une liste de tous les mots du lexique se trouvant dans une grille. Cette fonction prend comme argument une grille et un ensemble contenant le lexique et doit renvoyer une liste (telle qu'implémentée dans `List.h/.c`) contenant tous les mots trouvés sur la grille, sans doublons et dans un ordre quelconque. Pour éviter les doublons, vous devrez vérifier qu'un mot n'a pas déjà été ajouté à la liste en cours de construction. La manière la plus efficace de gérer ça est de construire en parallèle à cette liste un `Set` contenant tous les mots déjà rencontrés.
- `Set_RadixTrie.c` : L'implémentation complète d'un arbre radix, en suivant l'interface `Set.h`.
- `Set_hashtable.c / Set_BST.c` : on vous demande d'implémenter la fonction manquante `setGetAllStringPrefixes` de sorte qu'elle permette la recherche de tous les préfixes de la chaîne en argument plus efficacement qu'en recherchant individuellement chaque préfixe de la chaîne en appelant la fonction `setContains`.

Vos fichiers seront compilés avec les flags de compilation habituels (aussi utilisé dans le fichier `Makefile` fourni) :

```
--std=c99 --pedantic -Wall -Wextra -Wmissing-prototypes
```

Toute alerte lors de la compilation entraîne une pénalité.

Rapport

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

1. Donnez les complexités en temps de la recherche et de l'insertion pour les trois implémentations d'ensemble à comparer, en fonction du nombre de mots m dans l'ensemble et de la longueur l du mot recherché.
2. Donnez la complexité en temps de **votre** implémentation de la fonction `setGetAllStringPrefixes` pour les trois ensembles, en expliquant brièvement votre implémentation. Pour simplifier l'analyse, ne prenez pas en compte dans cette complexité le coût de la construction de la liste des mots à renvoyer.

3. Analysez les complexités en temps des quatre solutions au problème général (recherche dirigée par le lexique et recherche dirigée par la grille avec les trois implémentations d'ensemble) en fonction de la taille de la grille N, et du nombre de mots dans le lexique m. Ne prenez pas en compte la construction de l'ensemble et comme au point précédent, ne prenez pas non plus en compte la construction de la liste de résultats.

4. En utilisant les fichiers de tests fournis, mesurez les temps de calcul empiriques des 4 solutions pour deux tailles de grilles croissantes[5]. Discutez de la cohérence ou non de ces résultats avec l'analyse de complexité théorique.

5. Sur base des deux points précédents, faites une brève analyse critique des différentes solutions.

Remarque: pour les analyses de complexité, vous pouvez faire l'hypothèse que le hachage est uniforme et que le facteur de charge est constant (ici la taille du lexique est connue à l'avance). Dans le cas de l'arbre binaire de recherche, vous pouvez faire l'hypothèse que vous êtes dans le cas moyen, les mots du lexique étant ordonnés de manière aléatoire dans le fichier `english.txt`.

Soumission

Le projet est à réaliser ***en binôme*** pour le ***vendredi 19 avril 2024*** à 23h59 au plus tard. Le projet est à remettre sur Gradescope (`http://gradescope.com`, code cours: KKVYNG).

Les fichiers suivants doivent être rendus :

- votre rapport (4 pages maximum) au format PDF. Soyez bref mais précis et respectez bien la numération des questions ;
- les fichiers c suivants complétés selon les instructions ci-dessus: `Board.c`, `Set_RadixTrie.c`, `Set_hashtable.c`, `Set_BST.c`.

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[6] avéré, les étudiants se verront affecter une cote nulle à l'ensemble des projets.

Les critères de correction sont précisés sur eCampus.

Bon travail !

[5] Choisissez vous-même ces tailles pour que vous puissiez tirer des conclusions à partir de l'expérience.
[6] Des tests anti-plagiat seront réalisés.