DM : String Trees

1 Introduction

En informatique, l'implémentation d'algorithmes nécessite de faire le bon choix de structure de données à utiliser : tableau, arbre, table de hachage, etc. Celle-ci dépend des paramètres du problème, notamment du type des données auxquelles on s'intéresse. Dans ce DM, nous allons voir comment implémenter une structure de données permettant d'organiser des mots (et donc des chaînes de caractères) de manière optimale. "Optimal" signifie ici que l'on veut une structure de données avec laquelle les opérations courantes, telles que l'ajout ou la recherche d'un mot, soient réalisées avec une complexité minimale.

Dans ce but, nous allons développer puis étudier une structure d'arbre nommée String Tree. Nous allons ensuite l'appliquer pour créer un système d'aide à l'écriture de type T9. Dans une dernière partie en bonus, on propose de développer une structure semblable à un String Tree inversé permettant de faire de la recherche rapide dans un texte.

2 String Tree

2.1 Principe

Pour rappel, un arbre est un ensemble de noeuds possédant chacun possiblement une valeur et des liens avec d'autres noeuds, ses enfants. Un arbre peut être binaire, auquel cas il a jusqu'à deux enfants par noeud, ou de degré r, auquel cas il a jusqu'à r enfants par noeud.

Comment alors implémenter une structure d'arbre basée sur des chaînes de caractères ? Le principe d'un String Tree va être de stocker un ensemble de mots issu d'un répertoire R. Pour cela, on décompose chaque mot en la suite de ses caractères. Ainsi, le mot *'chat'* est décomposé en $\{c, h, a, t\}$. Une première idée serait de créer un arbre où les noeuds auraient pour valeur les caractères des mots de R. Ainsi, si R contient les mots $\{$*'chat', 'chien', 'choc', 'cou', 'cours', 'course'*$\}$, l'arbre aurait pour racine un noeud de valeur c, qui est la première lettre commune à tous les mots de R. Les fils de la racine ont ensuite pour valeurs les caractères suivant c des mots de R, c'est à dire h et o. De cette manière, on obtient :

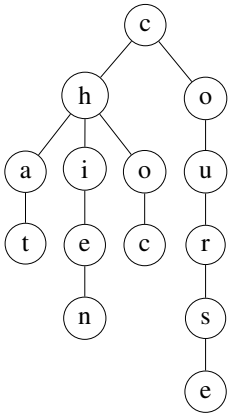

Figure 1: String Tree naïf associé à $R = \{$*'chat', 'chien', 'choc', 'cou', 'cours', 'course'*$\}$

1. On voit que c se trouve à la racine car notre répertoire R ne contient que des mots commençant par c. Si R contenait un mot commençant par une autre lettre, tel que *hamster*, comment pourrait-on faire ?

2. Expliquez comment on peut retrouver les mots de R à partir de l'arbre donné en exemple. Voyez-vous un problème ici ?

Pour remédier à ce problème, nous allons modifier un peu la structure précédente. Au lieu de placer les caractères sur les noeuds, nous allons les placer sur les liens entre les noeuds, de la manière suivante :

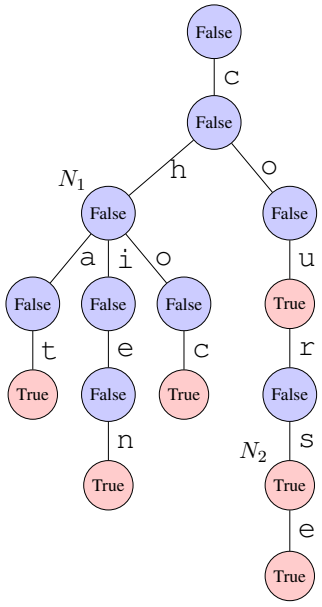

Figure 2: String Tree associé à $R = \{$*'chat', 'chien', 'choc', 'cou', 'cours', 'course'*$\}$

Ainsi, à chaque noeud on peut associer le mot construit en mettant à la suite les caractères portés par les liens allant de la racine jusqu'au noeud considéré. Par exemple ci-dessus, N_1 donne *'ch'* et N_2 donne *'cours'*. Pour reconnaître si un mot construit de cette manière appartient ou non à R, on place sur les noeuds un booléen `word` valant `True` si c'est le cas, `False` sinon.

Sur cet exemple, N_1 a 3 enfants. En fait, chaque noeud peut avoir hypothétiquement autant d'enfants qu'il y a de lettres L dans notre alphabet. Nous avons ici appliqué notre exemple à des mots du vocabulaire français. Il est important de voir que cette structure peut servir dans le cas où l'alphabet considéré est totalement arbitraire. Prenons déjà l'exemple de l'alphabet français $\{a, b, \ldots, z\}$ contenant $L = 26$ lettres. Dans un programme Python, cet alphabet ne nous permettrait pas de prendre en compte les lettres accentuées ; nous pourrions les ajouter à notre alphabet pour obtenir $\{a, b, \ldots, z, \grave{a}, \ldots, \hat{u}\}$. Nous voyons que la définition de notre alphabet est donc relativement flexible. On pourrait également y ajouter les chiffres, les ponctuations, faire la différence entre majuscules et minuscules, etc. Mais aussi il est possible de considérer des alphabets complètement différents, tels que l'alphabet binaire $\{0, 1\}$ ($L = 2$) ou encore l'alphabet hexadécimal $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$ ($L = 16$).

Si L est le nombre de lettres de notre alphabet, il faut donc pour notre structure d'arbre une implémentation permettant de stocker un nombre arbitraire d'enfants par noeud allant de 0 à L. Pour cela il existe deux solutions simples :

- Une liste : de longueur L, chaque position dans la liste correspond à une lettre de l'alphabet. La position i correspond donc à la i-ème lettre de l'alphabet (considérant l'alphabet comme ordonné). S'il existe un lien, portant cette i-ème lettre, entre le noeud considéré et un de ses enfants, alors l'enfant correspondant est en i dans la liste. Dans le cas contraire, $L[i]$ vaut `None`.

- Une table de hachage : chaque paire (clé, valeur) associe en valeur un enfant à une clé qui est le caractère porté par le lien entre le noeud considéré et cet enfant. On utilise pour cela un dictionnaire Python.

3. Expliquer les différences entre ces deux solutions et dans quelles conditions l'une ou l'autre est la plus avantageuse.

Nous choisissons pour commencer d'utiliser la solution de dictionnaire, plus flexible.

4. Implémentez la classe `StringTree` avec son constructeur qui initialise deux champs d'objet : le booléen `word` valant `False` et le dictionnaire vide `children`.

5. Implémentez les méthodes d'objet suivantes donnant la taille, la hauteur, les parcours préfixe et suffixe de l'arbre : `size()`, `height()`, `preorder()`, `postorder()`.

6. Ecrivez la méthode d'objet `contains()` qui prend en argument un mot w et renvoie `True` s'il se trouve dans l'arbre, `False` sinon.

7. Ecrivez la méthode d'objet `prefix()` qui prend en argument une chaîne de caractères s et renvoie l'ensemble des mots de l'arbre commençant par la sous-chaîne s.

8. Lorsque l'on veut ajouter un mot à un arbre, il existe différents cas de figures selon la structure déjà existante de l'arbre. Décrivez ces cas de figure possibles. Ecrivez la méthode d'objet `add()` prenant en argument un mot w et l'ajoutant à l'arbre.

9. Décrivez les cas de figure possibles pour la suppression d'un mot. Ecrivez la méthode d'objet `remove()` prenant en argument un mot w et le supprimant de l'arbre s'il s'y trouve.

10. Copiez `StringTree` dans une seconde classe `StringTreeList` qui va utiliser la solution de liste décrite plus haut. On ajoutera au constructeur un champ d'objet L précisant la taille de l'alphabet utilisé. Adaptez les méthodes de `StringTree` pour `StringTreeList`.

2.2 Complexité et application au tri

Dans la suite, on note L la longueur de l'alphabet, m la longueur moyenne d'un mot et n le nombre de mots de l'arbre.

11. Quelle est la complexité de l'opération de recherche dans un String Tree ?

12. Mêmes questions pour les opérations d'ajout et de suppression.

13. Proposez et implémentez une solution simple pour trier une liste de mots qui utilise la classe `StringTree`.

14. Quelle est la complexité de l'algorithme de tri proposé ?

15. Répétez les questions précédentes pour un String Tree sur le modèle de la liste.

3 String Tree compact

La structure que nous avons développée peut encore être optimisée, à la fois en espace et en temps. Pour cela, on propose de compresser les parties linéaires vides de l'arbre, où on appelle **partie linéaire vide** d'un arbre toute succession de noeuds ne présentant aucun embranchement ni aucune valeur `True` à l'exception du dernier noeud (qui peut avoir la valeur `True` ou `False`). Par exemple sur l'arbre de la

Figure 2, lorsque l'on descend pour former le mot *'chien'*, on observe que les trois derniers liens formant la sous-chaîne *'ien'* s'enchaînent sans embranchement et avec un dernier noeud de valeur `True`, et de même pour la sous-chaîne *'ou'* ou *'rs'* du sous-arbre droit ; ce sont des parties linéaires vides.

Pour compresser une partie linéaire vide, on opère de la manière suivante :

- On rassemble tous les caractères portés par ses liens en une chaîne de caractères qu'on attribue au premier lien.
- On rassemble tous ses noeuds en un unique noeud de valeur celle du dernier noeud.

Le résultat obtenu en compressant toutes les parties linéaires d'un String Tree est appelé un **String Tree compact**. Appliqué à l'arbre de la Figure 2, on obtient :

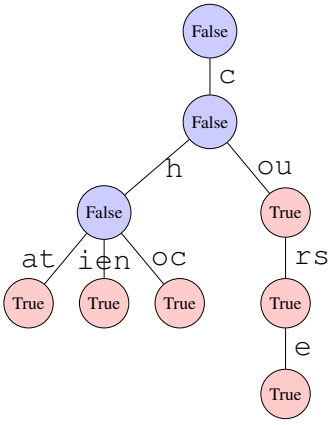

Figure 3: String Tree compact associé à $R = \{$*'chat', 'chien', 'choc', 'cou', 'cours', 'course'*$\}$

Un String Tree compact ne doit donc plus comporter que des noeuds d'embranchement et des noeuds de valeur `True`.

16. L'arbre donné en exemple ne finit qu'avec des noeuds de valeur `True`. Est-ce toujours le cas en général pour un String Tree compact ?

17. Ecrivez une nouvelle classe `StringTreeComp` avec son constructeur, en choisissant le modèle du dictionnaire ou de la liste et justifiez votre choix.

18. Ajoutez à `StringTreeComp` une méthode d'objet `compress()` prenant en argument un objet de la classe `StringTree` et qui le compresse. Le code suivant :

    ```
    stcomp = StringTreeComp()
    stcomp.compress(st)
    ```

    où `st` est un objet de la classe `StringTree`, ne doit rien renvoyer mais avoir implémenté dans `stcomp` l'arbre compressé correspondant à `st`.

19. Copiez et adaptez (si besoin) dans `StringTreeComp` les méthodes `size()`, `height()`, `preorder()`, `postorder()` de `StringTree`.

20. Décrivez les cas de figure possibles pour l'ajout d'un mot dans un String Tree compact. Ecrivez la méthode d'objet `add()` prenant en argument un mot w et l'ajoutant à l'arbre.

21. Décrivez les cas de figure possibles pour la suppresion d'un mot dans un String Tree compact. Ecrivez la méthode d'objet `remove()` prenant en argument un mot w et le supprimant de l'arbre s'il s'y trouve.

4 *Bonus* : Système d'aide à l'écriture T9

On propose maintenant de mettre en application la structure développée précédemment pour écrire un système d'aide à l'écriture de type T9. Pour rappel, T9 (pour texte en 9 touches) est un algorithme développé pour les anciens téléphones qui disposent de claviers numérotés. Généralement, ceux-ci permettent d'écrire de la manière suivante :

Touche	1	2	3	4	5	6	7	8	9	0
Caractères	ponctuations	*'abc'*	*'def'*	*'ghi'*	*'jkl'*	*'mno'*	*'pqrs'*	*'tuv'*	*'wxyz'*	espace

Avant le T9, il fallait pour écrire un mot appuyer sur chaque touche autant de fois que la position de la lettre voulue sur une touche. Par exemple, pour faire un *'r'*, il fallait appuyer 3 fois sur la touche 7. Donc pour écrire le mot *'course'*, la combinaison de touches à faire était 222 666 88 777 7777 33. Le T9 a permis de rendre l'écriture plus rapide : en appuyant une fois sur chaque touche, on définit les lettres que le mot voulu contient. En faisant la combinaison 268773, l'algorithme cherche les mots dont la première lettre se trouve dans *'abc'*, la seconde dans *'mno'*, etc. Il peut alors proposer *'course'* et tous les autres mots possibles partageant la combinaison 268773.

Pour la suite, on fournit un fichier `liste_francais.txt` contenant une grande quantité de mots issus du dictionnaire français.

22. Ecrivez une nouvelle classe `T9` dont le constructeur prend en argument un `StringTreeComp` pour initialiser un champ `R`.

23. Dans toute la suite, on considèrera que l'on écrit sans ponctuation. Ecrivez dans `T9` une méthode d'objet `combi()` qui à partir d'une combinaison de chiffres, sur le modèle donné dans la table plus haut, renvoie la liste des mots possibles associés à cette combinaison. Dans le cas où cette liste est vide, c'est qu'on a utilisé un mot qui n'est pas dans `self.R`. On utilisera alors la fonction `input()` de Python pour ajouter le mot manquant dans `self.R`.

24. Ecrivez dans `T9` une méthode d'objet `SMS()` qui ne prend aucun argument et permet d'écrire un SMS en utilisant `combi()` et la fonction `input()` de Python. Lorsque `combi()` renvoie une liste avec plusieurs mots, on prendra le premier de la liste.

25. *Question ouverte, à faire comme on veut.* Prendre le premier mot de la liste ne marche pas très bien. On pourrait pour améliorer le système permettre à l'utilisateur de choisir le bon mot à la main lorsque plusieurs sont renvoyés. Proposez à l'écrit des pistes de solutions plus automatisées. Si votre solution n'est pas trop technique, proposez une implémentation en code !

5 *Bonus* : Reversed String Tree

Nous allons nous intéresser au problème de recherche de sous-texte dans un texte. Commençons par la méthode naïve.

26. Ecrivez une fonction qui prend en argument une chaîne de caractères *S* et une chaîne *s*, et renvoie `True` si *s* est incluse dans *S*, `False` sinon. Nous voulons que les espaces ne soient pas considérés comme des caractères, c'est-à-dire que la fonction exécutée avec *S* = *'le vent se leve'* et *s* = *'sel'* renvoie `True`. La fonction doit procéder en cherchant d'abord le premier caractère de *s* dans *S*, puis une fois trouvé, comparer le caractère suivant de *S* au deuxième caractère de *s*, et ainsi de suite.

27. Si la chaîne de caractères *S* est de longueur N, et que *s* est de longueur M, quelle est la complexité de cette opération de recherche de sous-texte ?

Cette méthode brute est satisfaisante lorsque la sous-chaîne s est courte, mais devient plus coûteuse si elle est longue. Il vaut mieux alors recourir à d'autres méthodes qui vont dans un premier lieu, faire des manipulations sur le texte pour faciliter la recherche de sous-texte. La complexité finale de ces méthodes dépendra donc de la complexité du traitement initial du texte, puis de la complexité de la recherche de sous-texte. Nous allons ici procéder avec une structure d'arbre compact, comme dans la partie 2.

Jusqu'ici, nous avons construit des arbres associés aux sous-chaînes d'un ensemble de mots, prises dans le sens de lecture : la racine de l'arbre avait pour fils les premières lettres des mots considérés, et ainsi de suite jusqu'à atteindre la fin des mots. A présent nous allons nous intéresser aux sous-chaînes qui correspondent aux fins de mots, que nous appelons des *motifs finaux*. Un motif final est un substring non vide qui commence à une position quelconque d'un mot, et se termine avec la dernière lettre du mot. Par exemple, les motifs finaux du mot *'ananas'* et leurs positions associées sont : *'ananas' (0), 'nanas' (1), 'anas' (2), 'nas' (3), 'as' (4), 's' (5)*. Nous allons insérer ces motifs finaux dans une structure de Reverse String Tree, similaire au String Tree compact, où on remplace le booléen `True` qui indique une fin de mot, par un entier donnant la position du motif correspondant.

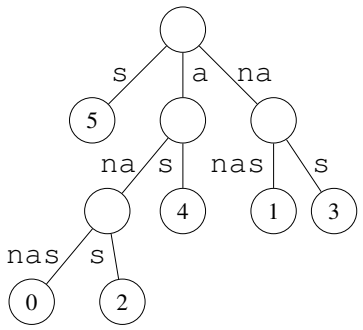

Figure 4: Reversed String Tree associé à *ananas*

28. Implémentez une classe `ReverseStringTree` avec son constructeur qui initialise deux champs d'objet : `position` initialisé à `None`, et le dictionnaire vide `children`. Le champ `position` contiendra l'entier correspondant lorsque le noeud correspond à une fin de motif final, comme dans la Figure 4.

29. Ajoutez une méthode qui prend en argument une chaîne de caractères, et renvoie le Reverse String Tree associé contenant tous les motifs finaux de la chaîne.

30. En déduire une méthode qui prend en argument deux chaînes de caractères S et s, et renvoie un booléen qui nous dit si s est inclus dans S ou non. Pour cela, on utilisera le Reverse String Tree associé à S.

31. Quelle est la complexité de l'opération de construction de Reverse String Tree associé à une chaîne de longueur N ? Quelle est la complexité de l'opération de recherche de s (de taille M) dans le Reverse String Tree de S ? Compte tenu de ces deux complexités, cette méthode de recherche de sous-texte est-elle meilleure que la méthode naïve ?

32. Il existe en fait un algorithme plus optimal de construction de Reverse String Tree, que nous ne détaillons pas ici. Il permet de construire l'arbre associé à une chaîne de longueur N avec une complexité linéaire en $\mathcal{O}(N)$. En utilisant cet algorithme, quelle serait la complexité finale de la recherche de sous-texte ?