Projet Anagrammes

[Algorithmique Avancée] [L2 INFO] [2017 2018] [Nathanael Bayard]

1. Général

Version de Python du code : *Python 3*.

Tout a été traité et testé, et tout est fonctionnel. Les réponses aux questions posées dans le PDF des instructions peuvent être visualisées en exécutant les fichiers *A.py* et *B.py*. Le code correspondant est à la fin de chaque fichier.

Stratégies de codage

- **Programmation Dynamique** : construction et utilisation d'une structure de donnée pour faciliter et accélérer les calculs effectués.
- **Backtracking** pour la grande majorité des fonctions les plus importantes, simplement parce que cela semble la seule manière possible de procéder, ou en tout cas de loin la plus simple.
- Technique de recherche **Diviser Pour Régner** : utilisée pour optimiser encore plus l'accès à la structure du dictionnaire dans le cas le plus critique (*cf* **Partie B**).

Chiffres et limitations

Le programme nécessite environ 125Mo de mémoire vive pour contenir la structure qui fait office de dictionnaire. Celle-ci se construit en environ 5 à 8 secondes (dépend de la machine). Une fois cela fait, presque tous les calculs se font de manière instantanée :

- recherche des mots ayant le plus d'anagrammes (*A.py*)
- recherche des multi-anagrammes de `CAROLINE ET FLORIAN` pour n = 2 mots maximum, de `CHAMPOLLION` pour n quelconque (*Tests_C.py*).

La recherche des mots ayant le plus de presque-anagrammes pour n = 1 lettre supplémentaire (*B.py*) demande elle beaucoup plus de temps : initialement, entre 170s et 240s. Cependant, en optimisant la structure par le biais d'un tri systématique de certaines sous-listes du dictionnaire (*Dico.py* `Dico.optimize()`), permettant par la suite une recherche en complexité logarithmique au lieu de séquentielle/linéaire, le temps pour ce calcul a pu être réduit à entre 50 et 80 secondes, selon les machines.

2. Détails

2.1. Terminologie

Cette terminologie s'applique aussi en particulier aux noms de variables et aux commentaires dans le code.

Famille

Pour une chaîne de caractères, sa **famille** désignera l'ensemble des mots du dictionnaire qui sont anagrammes de celle-ci. En terme de type, pour simplifier les cartouches, j'ai utilisé l'alias `Family` en tant que synonyme pour le type `List String`, c'est-à-dire le type d'une liste de chaînes de caractères.

Signature

Pour une chaîne de caractères `s`, sa **signature** correspond au résultat de `sorted(s)`. Pour une famille, sa **signature** est celle de n'importe quel mot contenu dans celle-ci. A toute signature donnée, correspond exactement 0 ou 1 famille du dictionnaire.

2.2. La classe `Dico`

Le coeur de l'efficacité de ce programme est basée sur l'idée d'indexer et de trier intelligemment l'intégralité du dictionnaire une fois pour toutes, afin que l'accès à la **famille** d'une quelconque chaîne de caractères se fasse aussi rapidement que possible. Dans le programme, le fichier du dictionnaire (*dico.txt*) est lu et son contenu est traité pour obtenir la variable globale `DICO`, qui est instance de la classe `Dico`. `DICO` possède un attribut `DICO.dico`, qui est une liste, et qui est le dictionnaire à proprement parler.

2.2.1. Structure interne de `DICO.dico`

`Dico.dico` contient toutes les familles d'anagrammes existantes du dictionnaire, déjà séparées dans des listes de mots individuelles, et référencées par leurs signatures. C'est un tableau à deux dimensions, pour un accès aussi rapide et direct que possible aux familles recherchées.

La première dimension correspond à la longueur des mots de la famille recherchée. La seconde dimension correspond à une formule calculée à partir des lettres des mots de la famille. A peu de chose près, elle correspond à la somme des lettres de n'importe quel mot de la famille, avec bien sûr A = 1, B = 2, etc. La fonction qui calcule cela s'appelle `wordSum()`. Donc, si l'on a une chaîne de caractères `s`, et l'on cherche ses anagrammes, la première chose à faire est de récupérer `tup := DICO.dico[len(s)][wordSum(s)]`.

`tup` est un tuple qui contient deux listes de même longueur. La seconde liste contient toutes les familles (c'est à dire les listes d'anagrammes) dont les mots ont la même longueur et la même somme que `s`. La première liste contient les **signatures** respectives des familles de la deuxième liste. Si l'on cherche la famille de `s`, il ne reste plus alors qu'à récupérer l'indice de sa signature `sorted(s)` dans la première liste. Si cet indice existe, on récupère alors la famille correspondante à cet indice dans le second élément de `tup`. Cette famille est alors directement l'ensemble des anagrammes de `s` dans le dictionnaire.

Dans le cas où la signature de `s` n'est pas dans la première liste de `tup`, ou dans le cas où l'on tombe sur la valeur `None` avant ou au moment d'atteindre `tup`, cela signifie évidemment que `s` ne possède aucun anagramme dans le dictionnaire.

2.3. Partie B

L'écriture de `presque_anagrammes()` s'est fait en *backtracking* pour balayer toutes les combinaisons possibles de `n` lettres de l'alphabet. On rajoute ces `n` letters supplémentaires au mot donné en entrée, avant d'en chercher tous les anagrammes. Les lettres sont ajoutées dans un ordre alphabétique croissant pour éviter les doublons du style `ROSE + AABC == ROSE + ABCA`.

Comme dit précédemment, le calcul de la famille ayant le plus grand nombre de presque-anagrammes pour n = 1 mettait très longtemps à l'origine. Ce temps a pu être réduit de deux tiers en créant une méthode `Dico.optimize()`, qui, lorsqu'appelée sur `DICO`, trie toutes les diverses listes de signatures (les premiers éléments des tuples mentionnées plus haut) dans un ordre alphabétique croissant, ce qui permet de remplacer une recherche séquentielle de la signature de la famille recherchée (*Tools.py* `indexOf()`), par une recherche logarithmique, avec la stratégie de **Diviser Pour Régner** (*Tools.py* `indexOf_div()`). Evidemment le tri des signatures est fait de telle sorte que l'ordre des familles correspondantes est lui aussi modifié en conséquence, pour que les indices des deux listes du tuple coincident toujours.

La méthode `Dico.optimize()` prend environ 1 seconde pour effectuer ce tri sur tout le dictionnaire, le gain est donc très net, cela dit vu qu'elle n'est pas nécessaire pour le reste des tests et des questions, elle n'a été appelée sur `DICO` que dans `B.py` — là où il y en avait réellement besoin.

2.4. Partie C

2.4.1. Algorithme pour `multiAnagrams()`

Cette fonction est le moteur principal de `A1` et `A2`. Elle récupère tous les "multi-anagrammes" de exactement `n` mots de la chaîne donnée en entrée. Elle est écrite en backtracking, et c'est de loin la plus longue et la plus compliquée. Elle donne en sortie toutes les combinaisons de `n` familles d'anagrammes du dictionnaire telles que si l'on prend un mot dans chacune des `n` familles, on obtienne un multi-anagramme de `n` mots de la chaîne d'entrée. Le type de sortie est donc une liste de listes de familles, chaque liste de familles ayant une longueur de `n`.

L'algorithme nécessite tant de paramètres (une dizaine), que j'ai préféré créer une classe `BTState` (pour *state of the backtracker*) pour contenir tous les paramètres nécessaires dans une unique variable d'état `S` (pour *state*).

On commence par créer deux listes de même taille, `S.letters` et `S.amountsLeft`, l'une contenant les différentes lettres de la chaîne passée en entrée (sans répétition), l'autre contenant la quantité restante de chaque lettre de la première liste au fur et à mesure du processus. On a aussi besoin d'une variable `S.result` qui contient toutes les solutions au fur et à mesure qu'on les trouve. `S.nWordsLeft` compte le nombre de mots restant à trouver avant d'avoir une autre solution complète ; elle vaut `n` au tout début. Les autres paramètres ont des valeurs par défaut.

L'algorithme suit le schéma qui suit :

- si `S.nWordsLeft == 0` on `return` sans rien faire.
- si `S.nWordsLeft == 1`, il ne reste plus qu'un mot, ou dit autrement, plus qu'une famille à trouver pour compléter la solution que l'on est en train de construire (la liste de `n` familles qui sera un des éléments du `S.result` final). Pour trouver cette dernière famille, on doit donc obligatoirement utiliser toutes les lettres qu'il reste de la chaîne de départ. La variable `S.selection` contient à tout instant les lettres sélectionnées pour faire partie de la signature de la prochaine famille à trouver.
  - Pour éviter les doublons, on doit tenir compte du fait que si dans une solution, deux familles qui se suivent sont de même longueur de mot, la seconde doit avoir une signature alphabétiquement supérieure ou égale à celle de la première. Si ça n'est pas le cas, on a atteint une impasse, donc on `return`.
  - Si tout va bien, on récupère les anagrammes de la `S.selection`, c'est-à-dire sa famille, à l'aide de `DICO.anagramsOf()`. Si la famille récupérée est vide, le chemin emprunté est une impasse et on ne fait rien de plus. Sinon, on ajoute à `S.result` la réunion de `S.partial`, qui contient à tout instant la solution en cours de construction, et de la famille de la `S.selection` que l'on vient de trouver.
- si `S.nWordsLeft > 1` : afin d'éviter les doublons, les familles d'une solution sont rangées par ordre croissant de longueur de mot. Il faut choisir cette longueur de mot `S.nextWordLen` pour la prochaine famille à trouver.
  - Si `S.nextWordLen == None`, c'est signe qu'on a pas encore choisi cette longueur de mot. On fait une boucle pour `L` dans `range(minL, maxL + 1)` en en définissant `newS.nextWordLen := L` puis en rappelant le backtracker avec une copie modifiée `newS` de la variable d'état `S`. `minL` est la taille minimale autorisée : comme les familles sont rangées par ordre croissant de longueur de signature, cela correspond à la longueur de la signature de la précédente famille. On a gardé cette valeur dans le paramètre

`S.minNextLen`. `maxL` est la division entière du nombre total de lettres qu'il reste (`nLettersLeft`) par le nombre de familles qu'il reste à trouver `S.nWordsLeft`. En effet, s'il reste `k` familles à trouver pour construire une solution de `n` familles, comme elles sont rangées par ordre croissant de signature, elles auront toutes une longueur de mot valant **au moins** `L`, pour `L` entre `minL` `maxL`. Donc il faut qu'il reste au moins `k*L` lettres de la chaîne de départ à distribuer, c'est-à-dire `nLettersLeft >= S.nWordsLeft*L` d'où `L` doit être inférieur ou égale à `maxL := nLettersLeft // S.nWordsLeft`.

- Si `S.nextWordLen != None`, on a déjà choisi précédemment la longueur des mots de la prochaine famille à ajouter à la solution en cours. Il nous reste à choisir les lettres à ajouter à la `S.selection`, ainsi que leur quantité.
  - Si la longueur de la `S.selection` est égale à la longueur de mot choisie `S.nextWordLen`, c'est qu'on a déjà fait ces choix. Il suffit alors de récupérer les anagrammes de `S.selection`, et, si ce n'est pas une impasse (même contraintes que précédemment), on ajoute cette famille à `S.partial`, puis on relancer le backtracking avec un nombre de mots restants `S.nWordsLeft` décrémenté.
  - Si la sélection n'a pas encore la bonne taille : pour chaque lettre dans `S.letters`, on doit choisir une quantité `quantity` de cette lettre à ajouter à `S.selection`, entre 0 et le maximum possible `maxQ`, qui est le minimum entre
    - ce qu'il reste pour atteindre la longueur de mot choisie,
    - et la quantité restante de la lettre en question dans `S.amountsLeft`.

    On rappelle le backtracker à chaque choix différent de la valeur de `quantity`, en incrémentant la variable `S.nextLetter`, qui correspond à l'indice de la prochaine lettre dont on doit choisir la quantité. Si `S.nextLetter` atteint la fin de la liste `S.letters`, mais qu'on n'a pas rempli toute la longueur requise de la `S.selection`, c'est une impasse : les choix de quantité effectués n'étaient pas les bons.

2.4.2. Traitement du résultat de `multiAnagrams()`

Les fonctions `A1()` et `A2()` sont censées renvoyer une liste de listes de mots, chaque liste de mot correspondant à un multi-anagramme de la chaîne d'entrée. Il faut donc *applatir* (*C.py* `flatten()`) le résultat de `multiAnagrams()` pour obtenir le type de résultat souhaité.

Chaque solution contenue dans le résultat retourné par `multiAnagrams()` est une liste de familles, qui est transformée par `multiplyFamilies()` en une liste de listes de mots, qui correspond à toutes les combinaisons contenant exactement un mot de chaque famille de la solution, sans tenir compte de l'ordre des mots et sans répétition. Les résultats pour toutes les solutions sont concaténés avant d'être renvoyés à `A1()` et `A2()`.

`multiplyFamilies()` nécessite un algorithme de *backtracking* pour éviter les doublons. En effet, il peut arriver que dans une solution, deux familles `F` et `G` qui se suivent soient identiques, c'est-à-dire `F == G`. Il faut donc éviter de piocher le mot `m` dans `F` et `m'` dans `G` après avoir déjà pioché (dans une autre branche du *backtracker*) `m'` dans `F` et `m` dans `G`, puisqu'en terme de multi-anagrammes, `[m, m'] == [m', m]`.

L'idée est simplement de vérifier que, lorsque l'on choisit un mot dans une famille, si la famille précédente est égale à la famille en cours (il suffit de comparer le premier mot de chaque famille pour déterminer cela), alors on doit choisir un mot d'indice `choice` supérieur ou égal à l'indice `lastChoice` du mot que l'on a choisit dans la famille précédente .