UNIVERSITÉ DE
SHERBROOKE

Guide de l'étudiant

APP5 - S2i

GIF-270 - Structures de données et complexité

Semaines 9 et 10

Département de génie électrique et de génie informatique
Faculté de Génie

Hiver 2022

Note : En vue d'alléger le texte, le masculin est utilisé pour désigner les femmes et les hommes.

Document S2i_GIF270_guide_etudiant.pdf

Version 1.0, février 2018, par Frédéric Mailhot
Version 1.1, janvier 2019, par Frédéric Mailhot
Version 1.2, janvier 2020, par Frédéric Mailhot
Version 1.3, janvier 2021, par Frédéric Mailhot
Version 1.4, janvier 2022, par Frédéric Mailhot

Réalisé à l'aide de LaTeX et TeXstudio

Table des matières

1 Éléments de compétences visés par l'unité

GIF-270 - Structures de données et complexité

1. Sélectionner et utiliser les structures de données appropriées pour solutionner un problème donné.
2. Analyser la complexité des algorithmes applicables à un problème donné.

Qualités de l'ingénieur touchées dans cette unité d'APP

Le tableau qui suit énumère les qualités de l'ingénieur qui sont touchées dans cette unité d'APP. Le tableau indique le niveau d'enseignement en lien avec les qualités identifiées (introduit, développé ou appliqué), ainsi que la présence d'évaluations spécifiques de ces qualités.

Qualité	**Description**	**Niveau d'enseignement**	**Évaluation**
Q01	Connaissances en génie	Appliqué	Oui
Q02	Analyse de problèmes	Appliqué	Oui
Q03	Investigation	-	-
Q04	Conception	-	-
Q05	Utilisation d'outils d'ingénierie	Appliqué	Oui
Q06	Travail individuel et en équipe	-	-
Q07	Communication	-	-
Q08	Professionnalisme	-	-
Q09	Impact du génie sur la société et l'environnement	-	-
Q10	Déontologie et équité	-	-
Q11	Économie et gestion de projets	-	-
Q12	Apprentissage continu	-	-

2 Énoncé de la problématique

Comment reconnaître quelqu'un par ses écrits?

Chaque jour ou presque, nous utilisons des sites et applications en ligne pour nous renseigner, nous divertir, contacter des amis, etc. Ces différentes activités, nous le savons, produisent des signatures numériques permettant à des tiers de connaître nos intérêts, nos besoins et, en partie, de nous reconnaître. Cette situation, qu'on peut apprécier ou non, peut paraître relativement nouvelle dans l'histoire de l'humanité. En fait, c'est une extension d'un problème beaucoup plus ancien: depuis l'invention de l'écriture, la question de la reconnaissance de l'auteur d'un texte s'est posée à maintes reprises. Dans cette unité d'APP, nous explorerons des structures de données et des algorithmes permettant de calculer la fréquence des mots dans un ensemble de textes, pour ensuite évaluer la possibilité qu'une certaine personne ait écrit un certain texte. Il est à noter que l'analyse des fréquences de mots dans un texte repose sur des concepts similaires à ceux utilisés par les entreprises web pour connaître nos intérêts et cibler la publicité qui nous est présentée. Ici, pour analyser les textes, nous utiliserons des unigrammes, des bigrammes et, en général des n-grammes de mots, un n-gramme représentant une séquence de "n" éléments de même type. Comme les textes analysés comprendront de grandes quantités de mots (il s'agit de textes d'auteurs francophones célèbres), l'efficacité des algorithmes sera importante et il faudra en évaluer la complexité.

Pour calculer la fréquence de mots dans un texte, il pourrait s'avérer utile d'utiliser des listes chaînées (simplement ou doublement), des arbres, des tableaux de hachage ou des graphes. Une combinaison de ces structures de données est aussi possible. Chacun des mots d'un texte étudié sera lu et comparé à l'ensemble des mots lus auparavant. Il faudra alors faire le compte des mots identiques qui apparaissent dans les textes analysés.

Chacune des structures de données étudiée implique des considérations spécifiques. Par exemple:

- l'utilisation de listes exige la comparaison et l'insertion de nouveaux éléments;
- l'utilisation d'arbres implique l'insertion et le rebalancement;
- l'utilisation des tableaux de hachage vient avec la définition de la fonction de hachage et de la taille du tableau (entre 2^8 et 2^{16} serait probablement d'intérêt ici, en utilisant la fonction modulo pour assurer une valeur d'index acceptable), ainsi que la gestion des collisions;
- l'utilisation de graphes requiert la détection et la gestion des cycles lors d'ajout de noeuds.

Pour faire l'analyse de mots dans des textes, il est important de considérer la complexité associée à la méthode utilisée pour découvrir si un élément a déjà été observé et pour faire le décompte des mots (lettres) observés. Selon la complexité de la méthode choisie, le système exigera au maximum un temps de recherche constant, logarithmique, linéaire, quadratique ou même exponentiel par rapport au nombre de mots déjà présents. On parle alors de complexité O (on prononce big-O) d'un algorithme, dont on dira, par exemple, qu'il est d'ordre $O(1)$, $O(\log n)$, $O(n)$, $O(n^2)$ ou $O(e^n)$. Il existe aussi l'analyse de complexité Ω et Θ. D'un point de vue formel, on effectue les analyses de complexité en supposant l'utilisation d'un système de calcul théorique nommé *machine de Turing*. Ceci permet d'évaluer la complexité d'un algorithme de façon indépendante du processeur spécifique utilisé pour réaliser les calculs. Le problème de l'arrêt (halting problem en anglais) a été démontré par Turing en utilisant cette machine.

À partir de la fréquence de mots calculée pour un ensemble de textes, il est possible de générer un texte "aléatoire", selon un processus qu'on appelle une chaîne de Markov, en utilisant un générateur de nombres pseudo-aléatoires. Pour ce faire, il faut trier les mots observés, du plus au moins fréquent. Plusieurs méthodes de tri existent, tels le tri fusion(*merge sort* en anglais), le tri à bulle (*bubble sort* en anglais) et le tri rapide (*quicksort* en anglais), chacune des méthodes ayant une complexité distincte. Un texte aléatoire généré à partir des fréquences observées pour un certain auteur permet de valider le fonctionnement de l'ensemble du système.

3 Connaissances à acquérir par la résolution de cette problématique

Connaissances déclaratives: QUOI

- Structures de données
  - Listes chaînées, listes doublement chaînées
  - Graphes, cycles
  - Arbres
  - Fonctions et tableaux de hachage
- Algorithmes
  - Tri
    - à bulle
    - fusion
    - rapide (quicksort)
  - Recherche binaire
  - Équilibrage
  - Largeur d'abord, profondeur d'abord
- Machine de Turing
- Complexité
  - logarithmique
  - linéaire
  - polynomiale
  - exponentielle
  - P, NP
  - Analyse O, Ω, Θ
- Language Python

Connaissances procédurales: COMMENT

- Mettre en oeuvre les structures de données appropriées, en définissant tous les éléments et paramètres nécessaires
- Concevoir les algorithmes associés à certains types de structures de données, tels la fonction de hachage, le tri, l'insertion, l'extraction

- Créer un arbre, insérer, retrancher des éléments; rebalancer
- Établir la complexité d'un algorithme
- Effectuer un tri efficace
- Utiliser le language Python pour produire un système d'analyse de texte

Connaissances conditionnelles: QUAND

- Choisir les structures de données appropriées pour un problème spécifique
- Choisir les algorithmes appropriés selon leur complexité pour un problème spécifique
- Déterminer quelles structures et bibliothèques en Python sont nécessaires pour réaliser un logiciel de complexité moyenne

4 Références

4.1 Lectures en lien avec les structures de données, la complexité et les algorithmes

- **Référence principale -** Brad Miller and David Ranum, Problem Solving with Algorithms and Data Structures Using Python:
  - Complexité: Sections 3.1 à 3.7
  - Structures de données de base: Sections 4.1, 4.2
    - Listes : Sections 4.19 à 4.23
  - Hachage : Section 6.5
  - Tri : Sections 6.6 à 6.12
  - Arbres : Sections 7.1 à 7.6, 7.11 à 7.17
  - Graphes : Sections 8.7 à 8.9, 8.19 à 8.21
- **Tableaux de hachage -**
  - Comprendre le hachage en Python
- **Complexité -** Explications au sujet de la notation de Landau sur wikipedia :
  - Comparaison des valeurs usuelles: Voir: "2.2 Échelle de comparaison"
  - Notation de Landau: Voir "3. La famille de notations de Landau O, o, Ω, ω, Θ"
- **Complexité -** Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein, Introduction to Algorithms:
  - Chapitre 3 (Introduction à la complexité - 'Growth of Functions')
- **Machine de Turing -** Explications en français :
  - La machine de Turing
  - Explication du problème de l'arrêt - Voir la preuve "classique"

4.2 Lectures en lien avec la problématique

- **Analyse et génération de texte (chaînes de Markov) -** Brian Kernighan, Rob Pike, The Practice of Programming:
  - Chapitre 3 (Calcul de fréquence de mots)

4.3 Lectures en lien avec le langage Python (au besoin)

- **Python -** Gérard Swinnen, Apprendre à Programmer en Python 3:
  - Introduction à Python: chapitres 1 à 7
  - Lecture et écriture de fichiers : chapitre 9

4.4 Lectures facultatives

- **Introduction simple aux structures de données et à la complexité -** Très simple, bonne lecture de départ
- **Tableaux de hachage -** Type et fonctionnement
- **Tableaux de hachage -** Utilisation de *dict* en Python
- **Algorithmes -** LA référence des algorithmes
- **Algorithmes -** Excellente présentation, concis mais accessible

5 Sommaire des activités liées à la problématique

Activités de la semaine 1:

- $1^{\text{ère}}$ rencontre de tutorat;
- Étude personnelle et exercices: étude des sujets issus des objectifs d'étude du tutorat;
- Formation à la pratique procédurale: *problèmes touchant les structures de données et la complexité des algorithme*;
- Formation à la pratique en laboratoire: *mise en oeuvre quelques structures de données de base et évaluation leur impact sur la performance de certains algorithmes*
- Formation à la pratique procédurale: *problèmes touchant les machines de Turing, les algorithmes de tri et leur complexité*;

Activités de la semaine 2:

- Étude personnelle et exercices: sujets issus des objectifs d'étude du tutorat;
- Validation pratique des éléments de solution;
- Rédaction du rapport d'APP;
- $2^{\text{ième}}$ rencontre de tutorat: validation des connaissances acquises;
- Remise du rapport d'APP et du code Python;
- Évaluation formative;
- Évaluation sommative.

6 Productions à remettre (équipes de deux)

Application Python et rapport à remettre avant 9h00 le matin du tutorat 2

Les consignes de rédaction du rapport de l'unité d'APP sont données à la fin de ce document (section 13).

L'application Python, nommée **markov_CIP1_CIP2.py** doit implémenter les fonctionnalités suivantes:

- Travailler avec des fichiers de texte d'un ensemble d'auteurs. Les calculs doivent être réalisés de façon indépendante pour chacun des auteurs. Les oeuvres de chacun des auteurs sont incluses dans un répertoire (au nom de l'auteur), tous ces répertoires (dont le nom correspond à l'auteur) se trouvant dans un répertoire dont le nom est passé en paramètre.
- Pour le retour du n-ième n-gramme le plus fréquent, ou pour la production d'un texte aléatoire, utiliser le nom de l'auteur passé en paramètre. Le nom de cet auteur doit correspondre à l'un des auteurs contenus dans le répertoire d'auteurs. Tous les fichier de textes se trouvant dans le répertoire correspondant à l'auteur choisi doivent être traités.
- Calculer des fréquences des mots (unigrammes de mots).
- Calculer des fréquences des séquences de deux mots (bigrammes de mots).
- Calculer le n^e n-gramme le plus fréquent dans les fichiers de texte de l'auteur choisi (1 ou 2 mots, selon le paramètres de n-gramme choisi)

- Pour calculer la fréquence des n-grammes, regrouper les n-grammes qui ont la même fréquence et retourner la liste de tous ces n-grammes. Par exemple, si les unigrammes "une", "des" et "ils" apparaissent chacun à 132 reprises, associer la liste de ces trois unigrammes au nombre 132. Si par exemple cette fréquence (132) est la troisième la plus élevée et qu'on demande de retourner le troisième unigramme le plus fréquent, on doit retourner la liste ["une", "des", "ils"].
- Déterminer la proximité d'un texte inconnu avec les statistiques de l'ensemble des auteurs traités et indiquer l'auteur le plus problable.

Votre code sera appelé par le fichier de test **testmarkov.py**. Votre code, compris dans le fichier **markov_CIP1_CIP2.py**, doit offrir les méthodes suivantes:

- analyze (pour analyser l'ensemble des textes des auteurs fournis)
- find_author (pour trouver l'auteur probable d'un texte inconnu, en utilisant les statistiques de l'ensemble des oeuvres de l'ensemble des auteurs traités)
- gen_test (pour générer un texte ayant les mêmes statistiques que celles d'un certain auteur, selon l'ensemble de son oeuvre). Valider que votre logiciel peut générer un texte d'au moins **5000 mots**
- get_nth_element (pour retourner le n-ième n-gramme le plus fréquent pour un auteur donné, pour l'ensemble de son oeuvre)

L'application de test *testmarkov* accepte les commandes suivantes:

- -d *répertoire* (pour indiquer le répertoire dans lequel se trouvent les sous-répertoires de chacun des auteurs à traiter)
- -a *auteur* (utilisé pour la génération de texte aléatoire ou pour l'extraction du n-ième n-gramme le plus fréquent de cet auteur)
- -f *fichier* (pour indiquer un fichier de texte à comparer avec les fréquences des fichiers de l'ensemble des auteurs)
- -T (pour indiquer que les résultats de l'analyse dún texte inconnu doivent être produits pour l'ensemble des auteurs)
- -m 1 (faire le calcul avec des unigrammes de mots)
- -m 2 (faire le calcul avec des bigrammes de mots)
- -F *nombre (n)* (afficher le n^e élément le plus fréquent (selon l'option -m 1, -m 2)

- -G *nombre (n)* (produire un texte aléatoire de *n* mots, selon le style de l'auteur identifié par l'option *-a auteur*
- -g *nom du fichier à générer* (pour indiquer le nom du fichier produit, si l'option -G est utilisée)
- -v (mode verbose)
- -noPonc (pour indiquer de retirer toute la ponctuation)

Pour que l'application testmarkov fonctionne adéquatement, il faut modifier le fichier **etudiants.txt** et y mettre une ligne avec vos CIPs **CIP1_CIP2**, tels qu'ils apparaissent dans le fichier **markov_CIP1_CIP2.py**.

Les commandes suivantes sont des exemples de quelques possibilités:

- *python testmarkov.py -d* TextesPourEtudiants *-f* monfichier *-m 1* (calcule la fréquence des unigrammes de mots pour toutes les oeuvres de chacun des auteurs dans le répertoire *TextesPourEtudiants*, imprime l'auteur le plus probable du fichier *monfichier*)
- *python testmarkov.py -d* TextesPourEtudiants *-a zola -m 2 -F 3* (calcule la fréquence des digrammes de mots dans l'ensemble des oeuvres des auteurs compris dans le répertoire *TextesPourEtudiants*) et imprime la 3^e séquence de 2 mots la plus fréquente dans les textes situés dans le sous-répertoire *zola*)
- *python testmarkov.py -d* TextesPourEtudiants *-f* monfichier *-m 2* (calcule la fréquence des digrammes de mots dans les fichiers de tous les auteurs présents dans des sous-répertoire du répertoire *TextesPourEtudiants* et imprime la liste des auteurs et l'évaluation de la proximité du texte *monfichier* pour chacun des auteurs) Dans le cas, l'application produira des résultats de la forme:

  balzac 0,0042

  ségur 0,0012

  zola 0,0033

  (en supposant que le répertoire *TextesPourEtudiants* comprend les répertoires balzac, ségur et zola, et que le texte inconnu est à une distance de 0,0042 des textes de balzac, etc)

Note: En Python, argparse est utilisé dans testmarkov.py pour lire les arguments donnés en ligne de commande. Notez que Python offre directement des fonctionnalités pour la génération de nombres pseudo-aléatoire.

Quelques recommandations pour assurer un fonctionnement uniforme pour votre application:

- Avant de traiter un mot, assurez-vous que toutes ses lettres sont transformées en minuscules. La méthode "lower" de Python pourrait être utile.
- Conservez (en minuscules) les lettres accentuées ainsi que tous les caractères spécifiques au français (é, è, ê, ë, î, ï, à, â, ù, û, ü, ô, ç, etc.).
- Considérez les chiffres 0-9 comme des lettres minuscules.
- Traitez les caractères qui ne sont pas des lettres ou des chiffres (par exemple: "-", ",", ";", ".", etc.) comme des espaces.
- Traitez les mots de 2 caractères ou moins comme des espaces (cela élimine les articles, qui causent des problèmes dans les unigrammes et les digrammes)
- Pour simplifier le traitement, considérez que le trait d'union est un espace. Ainsi les mots composés (par exemple, porte-voix) seront considérés comme des séquences de deux mots (pour l'exemple précédent, "porte" et "voix").
- Considérez que les espaces, les tabulations et les changements de ligne sont des séparateurs entre les mots (et ne sont donc pas des mots).
- Considérez que toutes les formes conjuguées d'un verbe ainsi que le pluriel d'un mot représentent des mots distincts (pas de racinisation, ou stemming en anglais).
- Pour comparer l'ensemble des textes d'un auteur a avec un texte inconnu, faire le calcul suivant lorsque les fréquences (unigrammes ou digrammes) sont établies pour l'auteur a et pour le texte inconnu:
  - Obtenir le vecteur $M = [|m_i|]$ pour l'ensemble des mots de l'auteur a et le vecteur $T = [|t_i|]$ pour l'ensemble des mots du texte inconnu, où $|m_i|$ et $|t_i|$ représentent le nombre de mots m_i, t_i.
  - Calculer la taille des vecteur M et T: $|M| = \sqrt{(\sum_{i=0}^{N}(|m_i|)^2)}$ et $|T| = \sqrt{(\sum_{i=0}^{N}(|t_i|)^2)}$.
  - Faire le produit scalaire de M et de T: $(M \cdot T) = (\sum_{i=0}^{N}(|m_i| \cdot |t_i|)$

  - Normaliser le résultat obtenu en le divisant par le produit des tailles des vecteurs: proximité $= (M \cdot T)/(|M| \cdot |T|)$
  - Plus cette valeur est proche de 1, plus importante est la proximité du texte inconnu avec les écrits de l'auteur sous analyse

- La méthode readline() de Python pourrait vous être utile pour obtenir les lignes d'un fichier de texte.
- La méthode split() de Python pourrait vous être utile pour obtenir les mots d'une ligne.

$\Longrightarrow$ *Pour la remise: produire et déposer le fichier compressé d'un répertoire nommé CIP1_CIP2, contenant le rapport d'APP (nommé rapport_CIP1_CIP2.pdf) et le fichier markov_CIP1_CIP2.py*

7 Semaine 1 - Formation à la pratique procédurale

But de l'activité

Le but de cette activité est de se familiariser avec certains types de structures de données ainsi qu'avec la complexité des algorithmes:

- Listes, arbres, tableaux de hachage
- Complexité des algorithmes (logarithmique, linéaire, quadratique, polynomiale, exponentielle)

NOTE : Il vous est fortement recommandé de lire les documents qui expliquent les structures de données de base et la complexité des algorithmes avant de vous présenter à cette séance.

7.1 Liste chaînée, doublement chaînée

Supposons que nous voulons représenter un ensemble d'éléments ordonnés, par exemple les différents wagons d'un certain train de métro.

a. Expliquer pourquoi une liste serait une bonne représentation d'un train de wagons

b. Supposons que le modèle que nous voulons créer sera utilisé pour représenter le nombre de passagers présents dans chacun des wagons tout au cours d'une journée. Supposons de plus que le train de métro comprend 10 wagons, numérotés 1 à 10, du wagon de tête au wagon de queue :

   1. À un certain moment, 3 passagers montent dans le wagon 5. Comment pourrons-nous accéder à l'élément représentant le wagon 5?
   2. À midi, un nouveau wagon (le wagon numéro 42) est ajouté entre les wagons 6 et 7. Quel type de liste nous permettrait de représenter cette situation? Quelles opérations devraient alors être faites?
   3. Plus tard, le wagon 7 est retiré. Comment devrons-nous modifier la liste?

7.2 Tableau de hachage

a. Quels sont les paramètres et éléments importants dans un tableau de hachage?

b. Quelle est l'utilité de la fonction de hachage?

c. Qu'est-ce qu'une collision? Est-ce que les collisions sont inévitables? Expliquer pourquoi.

d. Quelle est la différence entre la gestion des collisions par chaînage et par adressage ouvert? Quels sont les avantages et les inconvénients de chacun?

e. Supposons que N éléments sont ajoutés à un tableau de hachage. Combien d'opérations seront nécessaires pour ajouter tous ces éléments à la table (supposons que les N éléments sont différents)? Par la suite, combien d'opérations seront nécessaires pour vérifier si un élément x se trouve dans la table ou non?

7.3 Arbres

a. Qu'est-ce qu'un arbre? En quoi diffère-t-il d'une liste?

b. Qu'est-ce qu'un arbre binaire?

c. Supposons qu'on créée un arbre binaire où chaque noeud représente un nombre, et où lors de l'ajout d'un nouveau nombre, une comparaison est faite au niveau de chaque noeud: si le nouveau nombre est plus petit que celui du noeud, il va à gauche, sinon, il va à droite. Simuler le traitement de la séquence de nombres qui arrivent dans l'ordre suivant :

Cas 1 : $5, 8, 2, 1, 4, 3, 7, 6, 9, 10$

Cas 2 : $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$

d. Comment faire en sorte que le problème identifié plus haut ne se produise pas (on suppose qu'on ne peut modifier l'ordre d'arrivée des nombres)?

7.4 Complexité

Dans cette question, nous étudierons certains concepts en lien avec la complexité des algorithmes et nous ferons une brève évaluation de la complexité de quelques méthodes déjà codées.

a. Que représentent O, Ω et Θ? Donner des exemples avec des courbes.

b. Considérez la création d'une liste chaînée, où les éléments doivent être ordonnés du plus petit au plus grand. Faites l'analyse O et Ω (pour vous aider dans votre réflexion, considérez la création de deux listes (ordonnées) distinctes, l'une avec l'ordre des données suivante: $(1, 2, 3, \ldots, n)$, l'autre avec l'ordre des données $(n, n-1, n-2, \ldots, 3, 2, 1)$

c. En supposant qu'un algorithme effectue $5n^2 + 3n + 2$ opérations lorsqu'il traite n données, quelle est sa complexité O? Expliquer pourquoi. L'utilisation de logarithmes peut être intéressante pour cadrer l'explication.

d. Compléter le tableau suivant, en indiquant le nombre d'opérations et le temps d'exécution pour les différentes valeurs de n et les différents niveaux de complexité:

		Complexité				
Pour nombre d'éléments (n)		$O(1)$	$O(\log n)$	$O(n)$	$O(n^2)$	$O(10^n)$
Quantité de calculs (N) ou Temps d'exécution (T) (10^{-9} sec par op)						
N:	n = 10^3					
N:	n = 10^6					
N:	n = 10^9					
T:	n = 10^3					
T:	n = 10^6					
T:	n = 10^9					

Note: 1 heure représente environ 10^5 secondes, 1 an représente environ 10^8 secondes (en réalité, $3 * 10^7$ secondes), mille ans représentent environ 10^{11} secondes, et l'âge de l'univers est environ de 10^{18} secondes.

e. Déterminer la complexité O des extraits de code suivants:

Extrait de code 1:

```
for i in xrange(0,N):
  z += 1
for j in xrange(0,N-3):
        z += 1
```

Extrait de code 2:

```
for i in xrange(0,N):
        z += 1
for j in xrange(0,N*N):
  z += 1
```

Extrait de code 3:

```
for i in xrange(0,N):
  for j in xrange(0,i):
          z += 1
```

Extrait de code 4:

```
for i in xrange(0,1000):
        z += 1
```

Extrait de code 5:

```
def une_fonction(N):
        if (N == 0):
                return 1
        return 1 + une_fonction(N >> 1)
```

Extrait de code 6:

```
def une_autre_fonction(N,t):
        if (N == 0):
                print t
                return
        une_autre_fonction(N >> 1, 2 * t)
        une_autre_fonction(N >> 1, 1 + 2 * t)
```

8 Semaine 1 - Formation à la pratique en laboratoire

Dans cette activité, on collabore par équipe de deux.

But de l'activité

Le but de cette activité est de mettre en oeuvre quelques structures de données de base et de comprendre comment les utiliser

Description du laboratoire:

Dans ce laboratoire, nous utiliserons un tableau de hachage, un arbre AVL et un graphe pour résoudre trois petits problèmes. Ces strutures de données seront implémentées en Python et vous permettront de bien en maîtriser l'utilisation.

8.1 Le problème de l'échelle de mots - utilisation de graphes

Le problème de l'échelle de mots (*word ladder problem* en anglais) a été proposé le jour de Noël 1877 par Lewis Caroll (l'auteur de *Alice au pays des merveilles*). Il s'agit de trouver une séquence de mots qui ne diffèrent que d'une lettre, reliant deux mots distincts. Par exemple, la séquence " bateau - rateau - rameau " serait acceptable. Pour cet exercice, nous utiliserons un graphe pour modéliser les mots et leur proximité. Chaque mot sera représenté par un noeud, qui sera relié par des arcs à tous les mots auxquels il peut être associé. Cependant, nous résoudrons une généralisation du problème, où la distance entre deux mots pourra être de 1, 2 ou 3 lettres. Nous permettrons aussi de calculer une distance entre des mots avec des tailles différentes, le mot le plus petit étant considéré comme ayant une ou plusieurs lettre(s) "invisible(s)". Si vous le désirez, vous pourrez utiliser comme point de départ le code disponible dans le livre de référence, à la section 8.8 (la section 8.7 sera aussi intéressante à lire). Vous pourrez utiliser la liste de mots en français disponible sur le site de l'APP comme point de départ.

Lorsque le graphe sera créé, vous pourrez ensuite choisir un mot (par exemple, vous pourriez commencer par "bateau") et chercher s'il existe des séquences de 3, 4 ou 5 mots à partir du mot de départ.

Pour les besoins du laboratoire, nous nous en tiendrons à cette simple recherche, mais il serait possible d'utiliser ce graphe pour, par exemple, déterminer s'il existe un chemin entre un premier mot A et un deuxième mot B. Une recherche de type *recherche par parcours en largeur* (*breadth first search* en anglais) serait alors très utile. De plus, en marquant chaque noeud parcouru avec la distance minimale entre ce dernier et le noeud de départ, il serait possible de déterminer le chemin le plus court (s'il existe) entre 2 mots.

8.2 Utilisation d'arbres équilibrés pour ordonner des nombres

Les arbres binaires permettent entre autres choses de représenter des éléments ordonnés. Nous avons vu dans le premier procédural qu'il faut porter attention à la structure d'un arbre binaire lors de sa construction, sinon, dans le pire des cas, il peut devenir l'équivalent d'une liste chaînée et perdre sa capacité de permettre d'atteindre un élément dans un temps $O(\log n)$. Pour ce faire, il est possible d'utiliser des arbres qui s'équilibrent automatiquement pendant leur création. Un arbre de type AVL (d'après les noms des deux mathématiciens soviétiques qui l'ont proposé en 1962, Adelson-Velskii et Landis) est un type d'arbres binaires qui a la propriété de toujours être bien équilibré. En effet, lors de chaque insertion (ou, le cas échéant, de retrait) d'un noeud dans l'arbre, la différence de profondeur des sous-arbres droit et gauche est calculée, et une opération (rotation droite ou rotation gauche) est appliquée récursivement si cette différence est plus grande que 1. Dans cet exercice, vous utiliserez (en le complétant d'abord) un algorithme qui réalise un arbre AVL pour ordonner une suite de 100 nombres. Le code disponible à la section 7.13 du livre de référence (sur les arbres binaires) sera très utile comme point de départ, suivi du code disponible à la section 7.17, sur les arbres AVL. Vous devrez "assembler" le code présenté à la section 7.13, puis le modifier en utilisant les éléments nécessaire pour réaliser un arbre AVL. Vous devrez réaliser vous-mêmes la méthode *RotateRight*, qui s'apparente à la méthode *RotateLeft* déjà disponible. Le fichier *nombres_a_ordonner.txt*, disponible sur la page de l'APP, pourra être utilisé pour valider votre code.

8.3 Utilisation de fonctions de hachage et de comparaison personnalisées

Nous avons vu lors du premier procédural que la fonction de hachage ainsi que la fonction de comparaison sont deux éléments essentiels d'un tableau de hachage. Dans cet exercice, nous allons créer une classe qui permet de traiter un couple de deux chaines de caractères. En y redéfinissant les fonctions *__eq__()* et *__hash__()*, cela permettra ensuite d'utiliser des instances de cette classe dans un tableau de hachage standard *dict* de Python. Vous pourrez utiliser le fichier *paires_de_mots.txt* pour valider le fonctionnement du système. Vous pourriez ensuite lire un deuxième fichier *paires_de_mots2.txt* et déterminer lesquels sont présents dans le premier fichier, et lesquels sont absents du premier fichier. Le code qui suit est un exemple des différents éléments de la classe que vous devrez définir.

```
class UneClasse:
  def __init__(self):
      self.a = None
      self.b = None
```

```
  def __eq__(self, other):
      return isinstance(other, self.__class__) and self.a == other.a and self.b ==

  def __ne__(self, other):
    return not self.__eq__(other)

  def __hash__(self):
      return hash((self.a, self.b))
```

9 Semaine 1 - Formation à la pratique procédurale

But de l'activité

Le but de cette activité est d'étudier les graphes, les algorithmes de tri, les arbres AVL, ainsi que la machine de Turing.

9.1 Graphes

Dans ce problème, nous étudierons l'algorithme de Dijkstra, qui permet de trouver la distance entre deux noeuds. Nous utiliserons le graphe qui suit pour l'ensemble des sous-questions.

a. Déterminer la distance minimale entre 2 noeuds, en partant du noeud 1. Dans un premier temps, nous supposerons que les poids sur les arcs sont tous de 1. Quel est le noeud le plus éloigné du noeud 1?

b. Refaire le même exercice, mais en utilisant les poids (2, 2 et 3) indiqués sur les arcs entre les noeuds $(1,2)$, $(13,14)$ et $(18,21)$. Quel est le noeud le plus éloigné du noeud 1?

c. Refaire les calculs de la sous-question b. en démarrant du noeud 17. Quel est le noeud le plus distant du noeud 17?

d. Qu'arrive-t-il si le graphe contient une boucle dont la somme des poids est négative?

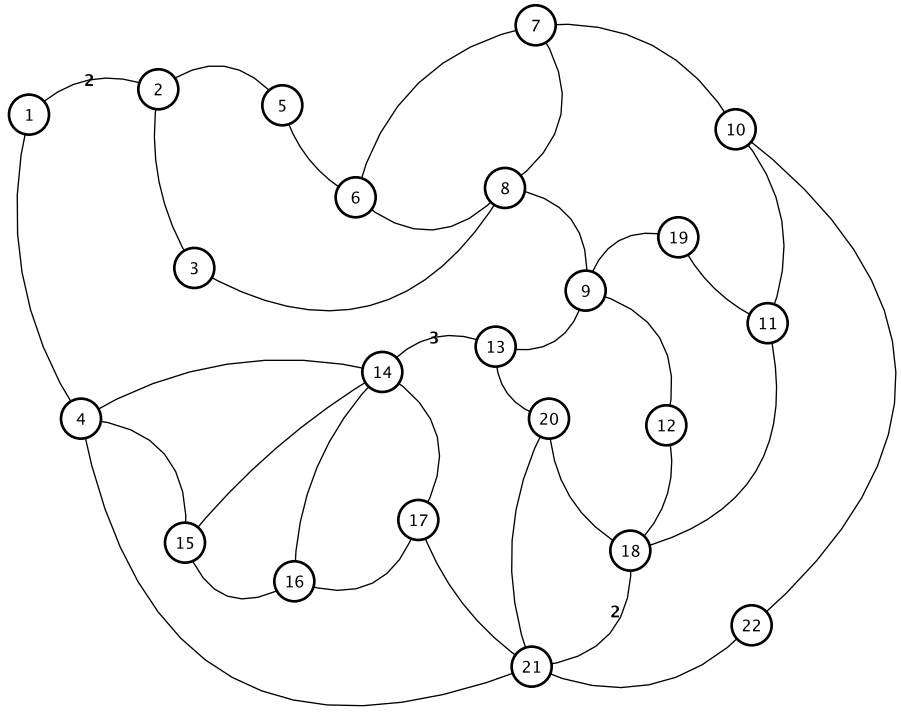
9.2 Tri

Dans cette question, nous étudierons les trois méthodes de tri suivantes: tri à bulles, tri fusion et tri rapide.

a. Utiliser le tri à bulles pour ordonner (du plus petit au plus grand) les nombres suivants: $[22,34,17,55,3,18,72,2]$

b. Utiliser le tri fusion pour ordonner (du plus petit au plus grand) les nombres suivants: $[33,12,4,28,1,31,75,5]$

c. Utiliser le tri rapide (quicksort) pour ordonner (du plus petit au plus grand) les nombres suivants: $[55,21,7,77,21,15,2,53]$



Graphe à analyser

9.3 Arbres AVL

Les arbres AVL permettent, lors de leur création, de garantir que l'arbre demeurera "balancé". Nous explorerons pourquoi, si la propriété est respectée, le temps de recherche sera toujours proche d'être logarithmique.

a. Comment calcule-t-on le facteur d'équilibre (*balance factor en anglais*)? Consulter le livre au besoin

b. À l'aide d'un arbre AVL, créez (et ordonnez) ls nombres suivants: $[55, 21, 7, 77, 15, 2, 53]$

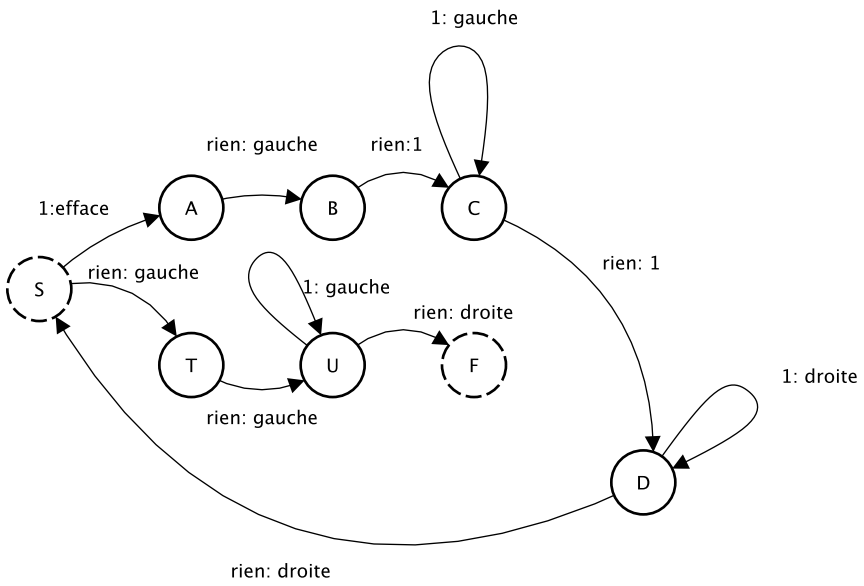
9.4 Machine de Turing

La machine de Turing est un modèle mathématique formel du calcul, et est à la base du fonctionnement (théorique) des ordinateurs classiques modernes.

a. Comment est définie la machine de Turing, et que représent les différents éléments de cette définition?

b. Supposons la machine suivante, déterminer le calcul qu'elle effectue:

   - Le ruban contient une séquence de deux signes: $1, 1$, représentant le nombre 2 en base unaire.
   - La position de départ de la tête de lecture est sur le signe le plus à gauche, l'état étant S.
   - Les opérations possibles sont: aller à droite, aller à gauche, écrire, ou terminer (dans l'état F).
   - La fonction de transition est représentée par le graphe qui suit:



Fonction de transition de la machine de Turing à analyser

10 Semaine 2 - Validation pratique de la solution en laboratoire

Activité de laboratoire

Le but de cette activité est de valider expérimentalement la solution à la problématique que vous avez développée. Vous devez démontrer que votre application a la capacité d'analyser un ensemble de textes, et expliquer son fonctionnement:

- Structures de données utilisées et explication du choix
- Complexité des éléments importants du système
- Courte démonstration

11 Semaine 2 - Travail de rédaction des productions exigées

Rédaction du rapport d'APP suivant les directives données à la section 13. Finalisation du schéma de concept.

12 Semaine 2 - 2e rencontre de tutorat: validation des connaissances acquises

Validation des connaissances - Bilan de groupe - Travail personnel de synthèse et d'études correctives.

13 Rapport d'APP

13.1 Consignes pour la préparation du rapport sur papier

Rapport d'APP à remettre avant 9h00, le jour du 2^e tutorat

Vous devez remettre un fichier zip contenant les éléments suivants:

- Le rapport d'APP, incluant les éléments suivants:
  - Noms et matricules des membres de l'équipe d'APP
  - Description des différentes structures de données utilisées, et les raisons derrière leur choix
  - Évaluation de la complexité des algorithmes utilisés pour l'ajout des mots (lettres).
  - Mot(s) le(s) plus souvent utilisé(s) par chacun des auteurs (pour l'ensemble de leurs textes)
  - Di-gramme(s) le(s) plus souvent utilisé(s) par chacun des auteurs (pour l'ensemble de leurs textes)
- Votre code python, nommé **markov_CIP1_CIP2.py**

14 Évaluation sommative

L'évaluation sommative porte sur tous les objectifs d'apprentissage de l'unité.

15 Évaluation de l'unité

La note attribuée aux activités pédagogiques de l'unité est une note individuelle. L'évaluation portera sur les compétences figurant dans la description des activités pédagogiques. Ces compétences, ainsi que la pondération de chacune d'entre elles dans l'évaluation de cette unité, sont :

Activités et éléments de compétence		*Validation d'APP*	*Rapport d'APP*	*Examen sommatif*	*Examen final*
GIF-270 - Structures de données et complexité					
1	Compétence 1	2,5 %	5 %	20 %	22,5 %
1	Compétence 2	2,5 %	5 %	20 %	22,5 %
	Total: GIF-270	5 %	10 %	40 %	45 %

Compétence 1 : Sélectionner et utiliser les structures de données appropriées pour solutionner un problème donné.

Compétence 2 : Analyser la complexité des algorithmes applicables à un problème donné.