LEPL1503 – Projet 3
Énoncé du projet – K-means
Partie 1 – Monothread

Matthieu Pigaglio Axel Legay Magali Legast Serena Lucca
Lambert Misselyn Tom Rousseaux Mathias de Schietere de Lophem

Année académique 2023 - 2024

1 Introduction

Vous avez, au cours de ces dernières semaines, appris les bases de la programmation en C. Comme vous avez pu le voir, le langage C offre un contrôle sur la gestion de la mémoire du programme. Le langage C permet aussi une gestion de plusieurs threads d'exécution dans un même programme. L'utilisation des threads est bien moins flexible en langage python et parfois même impossible.

Dans ce projet vous allez devoir implémenter, en langage C et à partir d'une implémentation Python existante, un algorithme de partitionnement de données de type k-moyennes (K-means en anglais). Dans cette première partie du projet, vous implémenterez une version séquentielle de l'algorithme. On s'intéressera au multithreading ainsi qu'aux performances énergétiques dans la seconde partie du projet.

L'algorithme des k-moyennes est utilisé dans plusieurs domaines mais principalement dans le traitement d'images et l'analyse de communautés. Dans le cas du traitement d'images, on peut par exemple, grâce à l'algorithme des K-moyennes, réduire la taille de la palette de couleurs d'une image afin d'optimiser sa taille. Pour ce projet, nous allons nous concentrer sur la deuxième application qui est l'analyse de clusters. Nous pouvons, grâce aux K-moyennes, découper un nuage de points en k clusters.

2 Le problème des k-moyennes

Il existe plusieurs algorithmes pour analyser des clusters à l'aide de k-moyennes. Dans ce projet, nous utiliserons l'algorithme de Lloyd [1]. Vous recevrez un ensemble de points en deux dimensions dans un espace et un nombre k donné. Votre code devra répartir ces points en k clusters (ou "groupes") réduisant le plus possible la distance entre les points du cluster et le centre de celui-ci, appelé un centroïde. Pour mesurer la qualité de vos clusters, l'algorithme utilise la distorsion qui est définie comme la somme du carré des distances entre chaque point et le centroïde de son cluster. L'algorithme reçoit un ensemble de points et un nombre k en entrée et répartit les points en k clusters.

Soit les points $(x_0; x_1; \dots x_i)$ et $(y_0; y_1; \dots y_i)$ dans un espace de dimension i.

Le carré de la distance euclidienne entre ces points se calcule avec la formule suivante :

$$Squared\ euclidian\ distance = \sum_{n=0}^{i-1}(x_n - y_n)^2$$

Le carré de la distance de Manhattan entre ces points se calcule avec la formule suivante :

$$Squared\ manhattan\ distance = (\sum_{n=0}^{i-1}|x_n - y_n|)^2$$

Ci-dessous se trouve la formule de distortion de k clusters avec $\mathcal{C}$ l'ensemble de tous les clusters, c_i le centroïde du cluster $\mathcal{C}_i$ et $squared\_distance(p_1; p_2)$ la fonction de distance au carré entre deux points p_1 et p_2(soit la distance de manhattan, soit la distance euclidienne).

$$Distortion(\mathcal{C}) = \sum_{0 \leq i < |\mathcal{C}|} squared\_distance(p_1; p_2)$$

Pour éviter les erreurs d'arrondis, nous travaillerons uniquement avec des nombres entiers. La division entre deux nombres entiers sera donc une division entière (le résultat ne contient pas les chiffres après la virgule). La division entière est la division de base utilisée par le langage C lorsque l'on divise deux nombre entiers.

3 Algorithme de Lloyd

Il existe plusieurs algorithmes pour résoudre le problème des k-means. Le premier et probablement plus simple d'entre-eux a été proposé par Lloyd en 1957. Son implémentation commentée est donnée dans l'Algorithme 1. C'est ce dernier que nous avons implémenté en Python pour vous et que nous vous demandons d'implémenter en code C.

Algorithm 1 Lloyd's algorithm.

Require: $\mathcal{P}$, the set of points
Require: k, the number of clusters to determine
Require: $\mathcal{I}$, the set of initial centroids
Require: $\mathcal{C}$, the set of clusters
Require: *distance*, the function computing the distance between two points
  $oldcentroids \leftarrow \varnothing$
  $centroids \leftarrow \mathcal{I}$ ▷ Initialization
  while $oldcentroids \neq centroids$ **do**
    for $i : 0 < i < |\mathcal{C}|$ **do** ▷ Reset the current clusters
      $\mathcal{C}_i \leftarrow \varnothing$
    end for
    for $p \in \mathcal{P}$ **do** ▷ Assign the points to their closest cluster
      $i \leftarrow \arg\min_j distance(p, centroids_j)$ ▷ Find the index i of the closest centroid
      $\mathcal{C}_i \leftarrow \mathcal{C}_i \cup$p ▷ Assign p to its closest cluster
    end for
    for $i : 0 < i < |\mathcal{C}|$ **do** ▷ Compute the new centroids by doing an average
      $centroids_i \leftarrow \frac{\sum_{p \in \mathcal{C}_i} p}{|\mathcal{C}_i|}$
    end for
  end while
  return $\mathcal{C}$

Comme vous le constaterez, la formule de distorsion n'est pas utilisée explicitement dans l'algorithme. L'algorithme minimise naturellement la distorsion en assignant chaque point au cluster disposant du centroïde le plus proche. Comme le résultat de l'algorithme dépend fortement de $\mathcal{I}$, l'initialisation des centroïdes, la formule de distorsion permet de mesurer la qualité d'une solution de l'algorithme et de les comparer entre elles. C'est pourquoi le programme que nous vous demandons d'implémenter calcule des solutions pour plusieurs initialisations possibles de l'algorithme.

4 Programme Python initial

Nous vous fournissons un programme Python qui résout déjà le problème de k-means. Le programme reçoit comme input un fichier binaire contenant un ensemble de points à répartir en k clusters. Comme

la solution trouvée par l'algorithme de Lloyd dépend fortement de l'initialisation des centroïdes, le programme va lancer l'algorithme de Lloyd sur plusieurs initialisations possible dans l'espoir de trouver une initialisation donnant une basse distorsion une fois la convergence atteinte. Le programme ne peut pas essayer toutes les combinaisons de points possibles car le nombre de combinaisons de k points parmi l'ensemble de points disponibles est bien trop élevé. **Le programme comprend donc l'option -p p qui précise au programme de tester toutes les initialisations possibles des k centroïdes parmis les p premiers points présents dans dans le fichier d'input.** Cet argument aura donc pour effet de calculer la solution de l'algorithme de Lloyd pour $\binom{n}{k}$ differentes initialisations.

5 Livrable

Pour la partie 1 du projet, on vous demande de faire les tâches suivantes :

— Écrire une version C du code Python. Le code C doit se comporter strictement de la même manière que le code Python : il prend les mêmes formats de fichier en entrée et produit les mêmes fichiers en sortie. Vous ne devez cependant pas obligatoirement procéder de la même manière que le code Python. Cette version devra tourner sur les machines UDS sans installation supplémentaire hormis CUnit. Seules les libraires standards ainsi que CUnit sont autorisés ;

— Maintenir un repository gitlab qui contiendra les versions du code C, et le lier à l'outil d'intégration jenkins. Chaque build de votre code devra être validé par les outils valgrind, cppcheck ainsi que par des tests unitaires (cunit) que vous devrez créer. Vous serez rajoutés dans un projet sur gitlab nommé lepl1503-2024-groupe-XX, où XX est l'identifiant de votre groupe attribué par votre tuteur. Ce dernier ainsi que l'équipe pédagogique seront aussi membres de ce gitlab. **Ce repository contiendra à sa racine un Makefile qui contient les cibles suivantes** :

  — `make` : compile le programme et produit l'exécutable nommé `kmeans` ;

  — `make tests` : compile le programme et lance les tests que vous avez écrits ;

  — `make clean` : nettoie le projet en supprimant l'exécutable et les fichiers objets produits lors de la compilation (typiquement, les exécutables et les `.o`).

  Attention ! Le Makefile doit impérativement compiler le code avec les flags `-Wall` et `-Werror` de gcc et doit fonctionner correctement sur les machines UDS sans installation supplémentaire hormis CUnit.

— Écrire un rapport de cinq pages qui contiendra au minimum les points suivants :

1. Une description de l'algorithme implémenté en C et de ses améliorations par rapport à la version Python ;
2. Une brève description des tests unitaires produits pour tester le code C (en utilisant CUnit), ainsi que des outils d'analyse de code (valgrind, cppcheck) appliqués à votre réalisation [1] ;
3. Des métriques de comparaison de performance entre les codes C et Python exécutés sur les machines UDS. Nous nous attendons **au minimum** à des analyses sur la performance en terme de rapidité, sur la consommation de mémoire de votre programme (Python vs C). Vous veillerez aussi à bien référencer tout code existant réutilisé ainsi que tout document que vous avez consulté. Ces références ne sont pas comptées dans les cinq pages.

La date de soumission de l'implémentation de la partie 1 du projet final en C est fixée à la fin de la semaine 7, le 22/03 à 18h. La partie 2 du projet vous sera donnée par une mise à jour de cet énoncé.

6 Spécifications du programme

Dans le squelette, que nous vous fournissons, le Makefile va générer un exécutable "kmeans" qui peut prendre les arguments suivants. Ce sont exactement les mêmes arguments que pour le programme

1. L'écriture de tests pertinents sera prise en compte dans votre évaluation.

Python, à l'exception de l'argument -n qui permet de préciser le nombre de threads de calcul (le programme python, lui, ne permet d'utiliser qu'un seul thread à la fois).

`./kmeans [-q][-k n_clusters][-p n_combinations_points][-n n_threads][-d distance_metric ][-f output_file][input_filename]`

-q si précisé, le programme n'affiche pas le contenu des clusters dans la sortie

-k n_cluster (par défaut : 2) : le nombre de clusters à calculer

-p n_combinations (par défaut : la même valeur que n_clusters) : on considère les n_combinations premiers points présent en entrée pour générer les centroïdes initiaux de l'algorithme de Lloyd.

-n n_threads (par défaut : 1) : le nombre de threads de calcul qui sont utilisés pour résoudre k-means (**Cette option est fixée à 1 pour la première partie du projet**)

-d distance_metric (par défaut : "manhattan") : soit "euclidean" ou "manhattan". Il s'agit du nom de la formule à utiliser pour calculer la distance entre deux points.

-f output_file (par défaut, on écrit sur la sortie standard) : le chemin vers le fichier pour écrire le résultat.

input_filename (par défaut, on lit l'entrée standard) : le chemin vers le fichier binaire contenant la liste des points à grouper

6.1 Format du fichier d'entrée

Le fichier d'input est un fichier binaire (ce n'est pas un fichier texte !) qui contient les éléments suivant dans l'ordre :

- Le nombre de dimensions qui est un entier non signé encodé sur 4 bytes (32 bits)
- Le nombre de points à grouper qui est un entier non signé encodé sur 8 bytes (64 bits)
- Une succession d'entiers signés sur 8 bytes qui donnent chaque coordonnée de chaque point à grouper. Par exemple, si on veut encoder les points $(x1; y1)$, $(x2; y2)$, $(x3; y3)$ dans ce format. On mettra le nombre de dimensions à 2, le nombre de points à 3 et la succession des coordonnées à $x1$, $y1$, $x2$, $y2$, $x3$, $y3$. Le fichier binaire d'entrée pour cet exemple contiendra donc 60 bytes au total. Comme mentionné dans le syllabus, il n'y a pas une façon unique d'encoder un entier sur 2, 4 ou 8 bytes en mémoire. Certaines machines utilisent little-endian et d'autres big-endian. Nous choisissons ici d'encoder ce fichier en big-endian. Comme vous ne pouvez pas deviner à l'avance l'endianness de la machine qui fera tourner votre programme, il faudra convertir chaque champ lu dans l'endianness de la machine faisant tourner le programme. Concrètement, après avoir lu chaque champ du fichier, vous devez utiliser les fonctions be32toh et be64toh pour convertir chacun de ces nombres entiers de big-endian vers le format de la machine sur laquelle le programme s'exécute.

6.2 Format du fichier de sortie

Le format de sortie du programme est un fichier texte encodé en CSV (pour Comma-Separated Values). La première ligne contient le nom des "colonnes" et les lignes suivantes représentent les clusters trouvés par l'algorithme de Lloyd pour chaque combinaison d'initialisation de l'algorithme. Par exemple, l'output suivant est valide :

```
initialization centroids, distortion, centroids, clusters
"[(1,1),(2,2)]",11,"[(1,1),(4,5)]","[[(1,1),(2,2)],[(3,4),(4,5)]]"
"[(1,1),(3,4)]",11,"[(1,1),(4,5)]","[[(1,1),(2,2)],[(3,4),(4,5)]]"
"[(1,1),(5,7)]",11,"[(1,1),(4,5)]","[[(1,1),(2,2)],[(3,4),(4,5)]]"
"[(1,1),(3,5)]",11,"[(1,1),(4,5)]","[[(1,1),(2,2)],[(3,4),(4,5)]]"
```

Dans cet exemple, la deuxième ligne nous apprend qu'en initialisant les 2 centroïdes à (1, 1) et (2, 2), on a obtenu une distorsion de 11. Les centroïdes finaux sont (1, 1) et (4, 5). Le premier cluster, contient (1, 1) et (2, 2) tandis que le deuxième contient (3, 4) et (4, 5). Si le programme a été exécuté avec l'argument -q, la dernière colonne n'est simplement pas présente dans l'output. N'hésitez pas à

utiliser l'implémentation Python sur de petits exemples pour voir ce que vous devez générer dans votre implémentation C.

6.3 Gestion de la taille des nombre entiers sur des machines différentes

En fonction de l'architecture de processeur pour laquelle un programme C a été compilé, la taille des types primitifs en C tels que les `int` et les `long int` pourra varier. Par exemple, sur un ordinateur portable récent, un int sera stocké sur 32 bits et un `long int` sera stocké sur 64 bits. Sur un Raspberry Pi, ces deux types sont stockés sur 32 bits. Utiliser ces types primitifs pour manipuler les nombres entiers de votre programme n'est donc pas générique et sera source de bugs dans votre programme sur certaines machines, étant donné que les coordonnées des points peuvent être plus grande qu'un entier de 32 bits. Pour éviter tout problème de la sorte, nous vous demandons de ne pas utiliser les types int et long int, mais plutôt d'utiliser des types qui assurent la taille utilisée pour les manipuler. Ces types sont présents dans <stdint.h> et sont les suivants :

- `int32_t` : un entier signé stocké sur 32 bits
- `uint32_t` : un entier non-signé stocké sur 32 bits
- `int64_t` : un entier signé stocké sur 64 bits
- `uint64_t` : un entier non-signé stocké sur 64 bits

En utilisant ces types plutôt que `int` et `long int`, vous assurerez un comportement identique de votre programme, quelle que soit l'architecture de processeur pour laquelle le programme a été compilé.

7 Portabilité et macOS

Le projet peut être implémenté sur macOS. Les fonctions d'*endianess* telles que `be32toh` ne sont pas identiques à ce que nous retrouvons sur un système Linux. Il en est de même avec les sémaphores POSIX, qui n'existent pas sur macOS. Une alternative aux sémaphores est d'utiliser des sémaphores nommées. Pour permettre à tous les utilisateurs de macOS et Linux de travailler sur le même projet, nous avons ajouté deux fichiers *headers* dans le squelette du projet : `portable_endian.h` et `portable_semaphore.h`. En incluant ces deux *headers*, il sera possible d'implémenter le projet de manière identique sur macOS et sur Linux. Il ne faut donc **pas inclure les *headers* `<semaphore.h>` et `<endian.h>`** puisque nos *headers* s'en occupent.

Même si vous n'utilisez pas macOS dans votre groupe, nous vous demandons d'utiliser les *headers* `portable_endian.h` et `portable_semaphore.h` dans votre projet. En effet, ce dernier sera évalué par d'autres étudiants, potentiellement sur macOS.

Références

[1] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2) :129–137, 1982.