Structures de données et algorithmes

Projet 2: arbres binaires de recherches

2 avril 2023

Ce projet porte sur la partie structure de données du cours, en particulier les arbres binaires de recherche. On vous demande à la fois d'utiliser l'implémentation d'arbre vue en cours mais également de l'étendre pour résoudre un problème concret. Le projet peut être réalisé seul ou par groupe de deux étudiants maximum. Tous les fichiers associés au projet sont disponibles sur Ecampus.

1 Objectif

Une société de taxi de Porto au Portugal a collecté l'ensemble des trajets effectués par ses taxis pendant une année (voir la figure 1) et souhaiterait pouvoir stocker cette information pour déterminer pour une position donnée dans la ville la liste des trajets ayant démarré près de cette position (afin par exemple de déterminer l'endroit idéal où placer une nouvelle station de taxis ou de pouvoir indiquer à un client via une application les endroits où il a le plus de chance de renconter d'autres gens afin de partager un taxi). On vous demande de mettre au point ce système. Vous voulez faire de l'excès de zèle et souhaiter minimiser au maximum le temps de réponse du système à une requête.

De manière plus abstraite, on souhaite mettre au point une structure de données de type dictionnaire permettant de stocker un ensemble potentiellement important de paires clé-valeur où la clé est une position (x, y) dans le plan et la valeur associée est quelconque. Ce système doit être capable de répondre à deux types de requêtes :

- Renvoyer la valeur associée à une paire donnée (x_q, y_q) si elle a été stockée dans la structure (*recherche exacte*)
- Renvoyer l'ensemble des valeurs associées à des clés situées à une distance (euclidienne) maximale d_{max} d'une position (x_q, y_q) données (*recherche dans une boule*).

On se propose d'étudier trois solutions au problème.

Par liste. La première approche consiste à stocker les positions et les valeurs associées dans une liste (ou plusieurs) et à effectuer une recherche simple linéaire dans cette liste pour répondre aux deux types de requêtes.

Par arbre binaire de recherche simple. La deuxième approche consiste à stocker les données dans une arbre binaire de recherche standard en utilisant les positions comme clés. La fonction de comparaison utilisée comparera les clés sur base de la coordonnée x et, en cas d'égalité de la coordonnée x, effectuera la comparaison sur la coordonnée y.

Par arbre binaire de recherche à deux dimensions. La seconde approche, consiste à stocker les villes dans un arbre binaire de recherche adapté à des clés (x, y) à deux dimensions [1]. Cette structure est très proche de la structure d'arbre binaire de recherche vue au cours si ce n'est que la fonction de comparaison change à chaque profondeur dans l'arbre. A la profondeur 0 (la racine), les clés (x, y) sont

1. Cette idée peut se généraliser facilement à $d > 2$ dimensions.

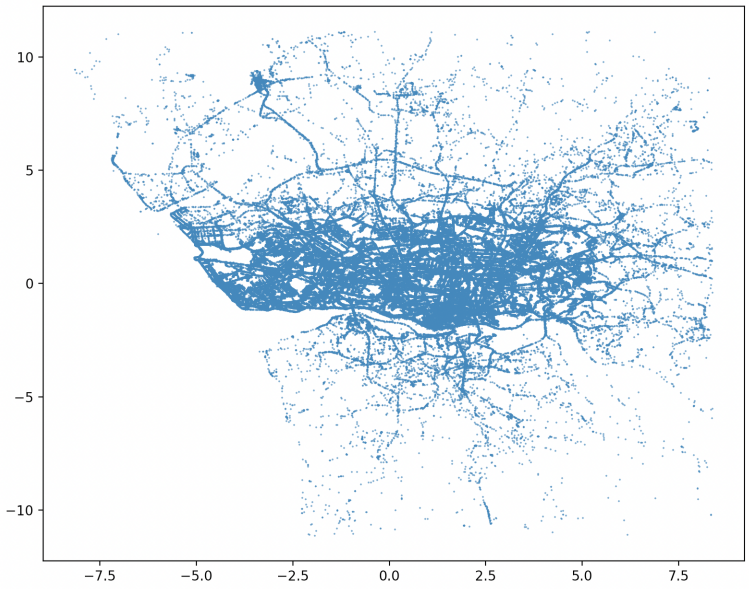

FIGURE 1 – Chaque point représente la position de départ d'une course d'un des 442 taxis de la ville de Porto effectuée entre le 1/07/2013 et le 30/06/2014. Les données ont été obtenues de Kaggle

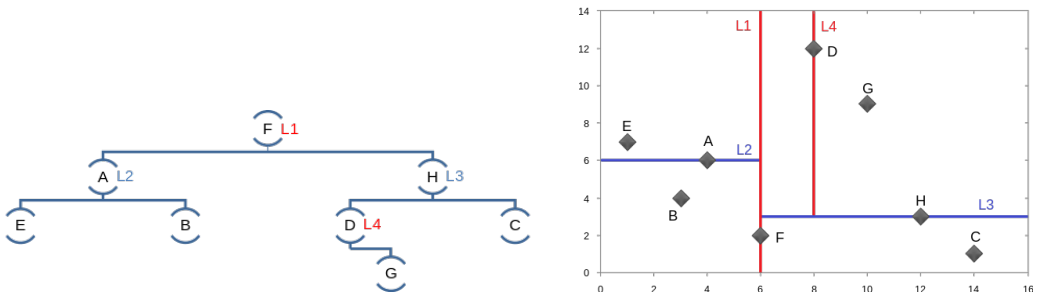

FIGURE 2 – Un arbre binaire de recherche à deux dimensions et le découpage correspondant de l'espace des coordonnées. Les positions ont été insérées, par exemple, dans cet ordre : F, A, B, E, H, D, G, C (source : Wikipedia)

comparées sur la coordonnée x, à la profondeur 1 sur la coordonnée y, à la profondeur 2 sur la coordonnée x à nouveau et ainsi de suite. En dehors de cette modification, les opérations de recherche et d'insertion sont similaires à celles implémentées pour les arbres binaires de recherche classiques (voir la figure 2 pour une illustration). L'avantage de cette structure est qu'elle rend plus efficace l'implémentation de la fonction de recherche dans une boule.

2 Implémentation

Pour implémenter ces idées, nous vous fournissons les codes suivants :

- `Point.h/.c` : un type de données correspondant à une position (x, y) permettant de calculer la distance entre deux points et de les comparer.
- `List.h/.c` : un type de données correspondant à une liste générique implémentée par une liste liée simple.
- `BST.h/.c` : une implémentation (partielle) générique d'arbre binaire de recherche
- `PointDct.h` : l'interface de la structure de dictionnaire utilisée pour stocker les positions et implémentant les deux types de requête.

— `PointDctList.c` : l'implémentation par liste de cette structure.
— `BST2d.h` : l'interface de la structure d'arbre binaire de recherche à deux dimensions.

Votre tâche est d'ajouter certains fonctions dans le fichier `BST.c`, d'implémenter l'arbre binaire de recherche à deux dimensions dans le fichier `BST2d.c` et les deux dictionnaires de points par arbre respectivement dans des fichiers `PointDctBST.c` et `PointDctBST2d.c`. Quelques précisions sont données ci-dessous sur les implémentations attendues. Consultez bien les commentaires dans les fichiers `.h` pour connaître les contraintes d'implémentation de chaque fonction.

Sauf mention contraire ci-dessous, toutes les fonctions doivent être implémentées de la manière la plus efficace possible.

Arbre binaire de recherche (`BST.c`). Vous devez ajouter dans ce fichier deux fonctions (voir `BST.h` pour une description détaillée de ces fonctions) :

— `bstRangeSearch` : une fonction renvoyant une liste liée contenant les valeurs associées aux clés k de l'arbre telles que $k_{min} \leq k \leq k_{max}$, où k_{min} et k_{max} sont deux arguments de la fonction.
— `bstAverageNodeDepth` : une fonction calculant la profondeur moyenne des clés dans l'arbre (telle que définie au cours).

La première fonction facilitera l'implémentation de la recherche dans une boule dans `PointDctBST.c` et la deuxième vous permettra de vérifier que la profondeur moyenne des clés est bien $\Theta(\log(n))$, même dans le cas d'une insertion aléatoire des clés.

Arbre binaire de recherche à deux dimensions (`BST2d.c`). Vous devez implémenter dans ce fichier l'intégralité des fonctions de l'interface donnée dans `BST2d.h`, en ce compris la fonction de recherche dans une boule :

— `bst2dBallSearch` : renvoyant une liste liée contenant les valeurs associées aux positions dans l'arbre située à une distance inférieure à un rayon donné d'un point (x_q, y_q) précisé en argument.

Pour implémenter cette fonction, l'idée est de parcourir l'arbre en profondeur d'abord mais en arrêtant l'exploration d'une branche dès qu'on peut déterminer qu'elle ne peut contenir aucune position située dans la boule. Pour déterminer ça, on exploitera le fait que les points à gauche (respectivement à droite) d'un nœud ont tous des valeurs de la coordonnée comparée au nœud (x ou y selon la profondeur du nœud) plus petite ou égale (resp. strictement plus grande) que la position stockée en ce nœud.

Dictionnaires de points (`PointDctBST.c` et `PointDctBST2d.c`). Vous devez implémenter dans ces deux fichiers les fonctions de l'interface `PointDct.h` en utilisant respectivement un arbre binaire de recherche et un arbre binaire de recherche à deux dimensions. Les fonctions à implémenter incluent :

— `pdctCreate` : stockant dans la structure les positions et valeurs données en argument. Dans le cas des deux types d'arbres, il est suffisant pour ce projet d'insérer ces clés dans un ordre aléatoire. Si vous le souhaitez, vous pouvez néanmoins essayer d'optimiser l'ordre d'insertion pour minimiser la profondeur de l'arbre.
— `pdctExactSearch` : implémentant la recherche exacte.
— `pdctBallSearch` : implémentant la recherche dans une boule.

Pour la recherche dans un boule, on utilisera directement la fonction `bst2dBallSearch` dans le cas de l'arbre binaire de recherche 2d. Dans le cas de l'implémentation par arbre binaire de recherche, il ne sera pas possible d'obtenir le résultat attendu avec un simple appel aux fonctions de l'interface de l'arbre binaire de recherche. L'idée sera de faire un premier filtrage des clés à l'aide de la fonction `bstRangeSearch` et ensuite de ne garder dans la liste filtrée que les positions effectivement dans la boule définie en argument.

Fichiers de test (`testcputime.c` et `testtaxi.c`). Le fichier `testcputime.c` fournit une fonction `main` permettant de tester chaque implémentation sur un ensemble de positions générées aléatoirement uniformément dans $[0.0; 1.0] \times [0.0; 1.0]$. Les arguments en ligne de commande de l'exécutable généré sont le nombre de points à stocker dans le dictionnaire, le nombre de recherche à effectuer et le rayon de ces recherches.

Le fichier `testtaxi.c` founit une fonction `main` permettant de tester vos implémentations sur la base de données des positions de départ des trajects de taxis de la ville de Porto (fichier `taxitripsporto.csv`).

L'exécutable prend en argument les longitudes et latitudes du centre de la boule [2] et le rayon (en km) pour la recherche. Notez que les positions dans la structure ne sont pas représentées directement par les longitudes et latitudes parce que les distances euclidiennes entre des paires de longitudes et latitudes ne représentent pas directement les distances dans le système métrique entre ces points. Une transformation est appliquées pour donner des coordonnées (x, y) qui préservent ces distances. Deux fonctions sont fournies dans le fichier (`transformToXY` et `transformToLL`) pour vous permettre de passer d'un système de coordonnées à l'autre. Ces fonctions vous seront utiles pour répondre à la dernière question dans le rapport.

Un `Makefile` vous est fourni pour automatiser la compilation. Il génère par défaut (commande `make` ou `make all`) 3 exécutables, `testlist`, `testbst` et `testbst2d` pour chaque implémentation de dictionnaire. Pour générer un exécutable à partir de `testtaxi.c`, vous devez utiliser la commande `make testtaxi`. Par défaut, l'implémentation par liste est utilisée pour ce dernier. Vous devrez modifier le makefile pour utiliser la structure la plus appropriée.

3 Analyse théorique et empirique

Dans le rapport, nous vous demandons de répondre aux questions suivantes :

1. Donnez le pseudo-code de la fonction `bstRangeSearch`.
2. Analysez sa complexité dans le pire et dans le meilleur cas en fonction de N, le nombre de nœuds dans l'arbre, et k, le nombre de valeurs dans l'intervalle. Traitez séparément le cas où l'arbre est équilibré et le cas où il ne l'est pas nécessairement.
3. Expliquez brièvement la manière dont vous avez implémenté les fonctions suivantes :
   — `pdctCreate` et `pdctBallSearch` dans le cas de l'arbre binaire de recherche simple.
   — `bst2dBallSearch` pour l'arbre binaire de rechercher 2d.
   — `pdctCreate` dans le cas de l'arbre binaire de recherche 2d
4. En partant du code fourni (`testcputime.c`, qui génère des points aléatoires), calculez et reportez dans un tableau la profondeur moyenne des noeuds dans l'arbre binaire de recherche et l'arbre binaire de recherche 2d pour des nombres de points croissants (par exemple, 10^4, 10^5, et 10^6). Discutez le résultat obtenu en fonction de votre algorithme de construction et la théorie.
5. Comparez empiriquement les temps de calcul des trois approches, à la fois pour la construction du dictionnaire et la réponse à des requêtes de type exactes et de type boules. Étudiez l'impact du nombre de points stockés dans la structure et du rayon de la recherche. Commentez brièvement vos résultats.
   Suggestion : Faites deux expériences. Fixez le nombre de recherches à 10^4 *et le rayon à 0.01 et faites varier le nombre de points dans le dictionnaire dans* $\{10^4, 10^5, 10^6\}$. *Fixez ensuite le nombre de requêtes à* 10^4 *et le nombre de points dans le dictionnaire à* 10^5 *et faites varier le rayon dans* $\{0.01, 0.05, 0.1\}$
6. Concluez quant à l'intérêt relatif des trois approches.
7. Utilisez l'implémentation la plus efficace pour trouver l'endroit à Porto qui a la densité de taxis la plus élevée. Définissez pour cela une grille de points, entre les coordonnées x et y (après transformation) minimales et maximales, espacés de 100m selon chaque coordonnée et calculez le nombre de taxis partant de chaque point de la grille dans un rayon de 100m également. Précisez le nombre de taxis maximal et la position (en terme de longitude et latitude) trouvée. Utilisez google maps pour déterminer à quel endroit à Porto correspond cette position.

4 Deadline et soumission

Le projet est à réaliser **individuellement** pour le **30 avril 2023, à 23h59** au plus tard. Il doit être soumis via Gradescope.

Les fichiers suivants doivent être remis :

2. Le centre de la ville de Porto est situé à une longitude et latitude respectives de -8.6291 et 41.1579.

1. Votre rapport au format PDF. Soyez bref mais précis et respectez bien la numération des questions.
2. Les fichiers suivants (uniquement) : `BST.c`, `BST2d.c`, `PointDctBST.c`, et `PointDctBST2d.c`.

Respectez bien les extensions de fichiers ainsi que les noms des fichier `*.c` (en ce compris la casse).

Vos fichiers seront évalués avec les flags de compilation habituels (`--std=c99 --pedantic -Wall -Wextra -Wmissing-prototypes -o boxsearch`) qui sont aussi utilisés dans le fichier `Makefile`. La présence de *warnings* impactera négativement la cote finale. Un fichier qui ne compile pas recevra une cote nulle.

Des tests anti-plagiats seront réalisés. En cas de plagiat avéré, l'étudiant se verra affecter une cote nulle à l'ensemble des projets.

Les critères de correction sont précisés sur la page web des projets.

Bon travail !