Projet Programmation 1A IR

Julie Bonnail & Thomas Bouyer

Analyse d'une base de données d'articles de recherche en informatique.

École d'ingénieurs
Télécom Physique Strasbourg

Projet Programmation 1A IR

Julie Bonnail & Thomas Bouyer

Table des matières

Quelques statistiques

Nombre de papiers scientifiques

8 749 944

Nombre d'auteurs

2 928 877

Plus grand nombre d'auteurs par publication

449

Nombre d'auteurs n'ayant écrit que deux articles

458 254

Plus grand nombre de co-auteurs par auteur

4 236

Introduction

Le but de ce projet de programmation est l'analyse d'une base de données de dblp qui regroupe un ensemble d'articles scientifiques. Cette base de données est au format portable xml et comporte (sur la version utilisée) 8 749 944 papiers scientifiques. L'analyse à proprement parler repose sur la construction d'un graphe construit sur la relation de co-écriture d'articles.

La première étape a naturellement été la conception d'un parseur (outil de transformation d'un flux de données en objets) pour le format spécifique du fichier dblp.xml. Au-delà de l'aspect temporel de l'algorithme, la construction de la classe d'équivalence de chaque auteur pour la relation choisie a posé de nombreux problèmes. Les algorithmes de parcours de graphe – quant à eux, sont pour la totalité des parcours en largeur et leur mise en place a pu être facilité à la suite des cours avec M. Theoleyre.

Le graphe utilisé est codé de manière naturelle par un ensemble non ordonné de nœuds. Chaque nœud correspond à un auteur distinct, et la structure comporte des références à un certain nombre d'éléments nécessaires aux analyses demandées, notamment à un ensemble de co-auteurs.

Répartition du travail

Ci-dessous, une liste non exhaustive des principaux aspects du projet traités.

- Création d'un parseur pour le fichier xml : Thomas
- Création d'un graphe : Thomas
- Parcours de graphe grâce à l'algorithme de Dijkstra : Julie
- Recherche d'informations dans un ensemble d'auteurs : Julie
- Énumération des composantes connexes du graphe avec leur diamètre : Julie
- Structures d'ensemble, de dictionnaire, de file de priorité et de vecteur : Thomas

Arborescence du projet

Ne sont représentés dans cette section que les dossiers publiés sur GitLab. Le fichier **`.gitignore`** peut permettre au besoin une visualisation de l'ensemble des fichiers générés. Les dossiers marqués **`(*)`** ne sont pas étendus.

```
projet_programmation_bouyer_bonnail
│
├── assets [ensemble de ressources] (*)
│
├── doc [documentations] (*)
│
├── includes [fichiers de code C .h]
│       types.h [seul .h n'ayant pas de .c, types génériques]
│       protocol.h [fichier de fonctionnalités génériques]
│       …
│
├── legacy [anciens fichiers] (*)
│
├── src [fichiers de code C .c]
│       main.c [n'a pas de .h, fichier contenant la fonction main]
│       protocol.c [implémentation de fonctionnalités génériques ou communes]
│       …
│
├── tests [répertoire de gestion des tests]
│       tests.c [fichier contenant la fonction main de lancement des tests]
│       foo.h [structure arbitraire]
│       macros.h [redéfinition des macros de test du type assert]
│       …-test.c [fonctions de tests spécifiques]
│       Makefile [le makefile avec les règles pour lancer les tests]
│
│   .gitignore
│   .gitlab-ci.yml
│   changelog.md [liste non exhaustive des modifications importantes du projet]
│   LICENSE
│   Makefile [le makefile général]
│   README.md [page de présentation pour GitLab]
│   readme.py [script python pour générer une section du fichier README]
│   setup.bash [CRLF -> LF]
```

Explications du code en français

Dans cette section, on s'attardera sur quelques aspects non triviaux du code. On verra notamment les tables de hachage dans les structures d'ensemble et de dictionnaire, les tas de Fibonacci pour les files de priorités, l'algorithme de Dijkstra appliqué à notre structure de nœud, entre autres. Les multiples algorithmes ne seront pas explicités ici, seules quelques justifications des diverses utilisations des structures seront fournies.

Structure d'ensemble et tables de hachage

L'implémentation de la structure d'ensemble (collection d'éléments uniques selon une métrique donnée) se trouve dans les fichiers **`hset.[hc]`** et implémente une table de hachage. Une table de hachage repose sur l'existence d'une injection (qu'on appelle fonction de hachage) d'un ensemble d'objets dans une sous-partie de l'ensemble des entiers naturels. Naturellement, deux difficultés majeures se présentent : la première est l'existence de cette fonction, la seconde provient du fait qu'elle n'est pas surjective.

Quelques mots rapides sur la métrique de comparaison utilisée. Dans notre projet, nous avons été amenés à construire des collections d'éléments uniques de plusieurs manières : en identifiant des structures par leur pointeur, et en identifiant des auteurs par leur nom. Les structures implémentant une table de hachage dans notre projet ont donc, dans leur méthode de création (issue d'un **`#define`** à proprement parler), un argument optionnel qui définit le type de métrique que nous souhaitons utiliser avec la structure retournée.

Retour sur les tables de hachages. Les fonctions de hachages doivent avoir un comportement pseudo-aléatoire, c'est-à-dire que pour deux entités dont la métrique est proche, leur hash doit être sensiblement différent. Les plus connues sont la famille des « secure hash algorithm » ou SHA. Ces fonctions ont l'avantage de produire des hashs très différents grâce à leur état interne qui change constamment. Ces fonctions étant pensées pour être sécurisées, elles sont plutôt lentes. Nous choisirons donc une autre fonction de hachage spécifiques aux chaînes de caractères, la fonction djb2 par Dan Bernstein. À ce stade, nous savons comment transformer des chaînes de caractères en un entier long non signé. Ainsi, lorsque nous recevons un objet, nous pouvons traiter directement son pointeur ou transformer la chaîne de caractères pointée au besoin. L'algorithme qui détermine ensuite l'indice du « bucket » fonctionne indépendamment de la métrique utilisée.

```
1.  struct hset_s {
2.      int hash_content; // 0: hash pointer, otherwise hash content
3.      size_t nbits;     // number of bits of the mask
4.      size_t mask;      // mask for the number of buckets
5.
6.      size_t capacity;        // number of buckets
7.      size_t *items;          // array of buckets
8.      size_t nitems;          // number of items
9.      size_t n_deleted_items; // number of deleted items
10. };
```

Bloc de code 1 : fichier `hset.h`, définition de la structure d'ensemble

Les éléments sont référencés par leur pointeur (**`void *`**), il est donc nécessaire de stocker des objets bien définis dans la mémoire. Dans la liste de « buckets » (définie à la ligne 7), sont stockés des pointeurs, ou les valeurs réservées **`0`** ou **`1`**, la première utilisée pour marquer des cases vides, la seconde, des cases qui ont été libérées.

Structure de dictionnaire

La structure de dictionnaire s'appuie fortement sur la structure d'ensemble. Un dictionnaire est une structure qui permet l'association de deux éléments. Un élément appelé clef, l'identifiant en quelque sorte du deuxième élément qu'on appelle valeur. Dans l'implémentation présente, deux tables sont gérées en parallèles, mais seule la table des clefs subit les tests équivalents à ceux qu'on retrouve dans la structure d'ensemble. Ainsi, lorsque l'on trouve un indice de disponible pour stocker le pointeur d'une clef, on utilise le même indice pour stocker le pointeur de la valeur.

```
1.  struct dict_s {
2.      int hash_content; // 0: hash pointer, otherwise hash content
3.      size_t nbits;     // number of bits of the mask
4.      size_t mask;      // mask for the number of buckets
5.
6.      size_t capacity;        // number of buckets
7.      size_t *keys;           // keys array
8.      size_t *values;         // values array
9.      size_t nitems;          // number of items
10.     size_t n_deleted_items; // number of deleted items
11. };
```

Bloc de code 2 : fichier `dict.h`, définition de la structure de dictionnaire

Files de priorité et tas de Fibonacci

Pour diminuer la complexité temporelle de Dijkstra, nous utilisons une file de priorité. Une file de priorité est aussi un ensemble mais qui admet une relation d'ordre. Dans le cadre de Dijkstra, nous devons trouver des plus courts chemins, les files de priorités sont donc des files minimales. La structure de tas de Fibonacci permet à la file de priorité d'atteindre les complexité temporelles moyennes suivantes :

Insérer un élément : $\Theta(1)$	Réduction de clef : $\Theta(1)$
Trouver l'élément minimal : $\Theta(1)$	Fusion de deux files : $\Theta(1)$
Extraire l'élément minimal : $O(\ln n)$	

```c
struct pqueue_s {
    dict_t *map;           // correspondence between elements and nodes
    heap_node_t *min_node; // minimum node
    size_t total_nodes;    // total number of nodes
};
```

Bloc de code 3 : fichier `pqueue.h`, définition de la structure de file de priorité

On remarque à ce stade deux choses : tout d'abord, on conserve un ensemble de correspondances entre les éléments que nous voulons stocker dans la file de priorité et leur structure de nœud utilisée pour les tas de Fibonacci, ensuite, la file de priorité ne conserve qu'un pointeur vers le nœud de clef minimale, les nœuds ayant eux-mêmes des pointeurs vers les autres nœuds. Les correspondances entre les éléments et leur nœud permet à l'utilisateur une utilisation facilitée sans se soucier de la sauvegarde manuelle des pointeurs des nœuds, en plus des éléments qu'il souhaite utiliser par la suite.

```c
struct heap_node_s {
    void *element;             // pointer to the element
    int key;                   // priority of the element
    struct heap_node_s *parent; // parent node
    struct heap_node_s *left;   // node "to the left"
    struct heap_node_s *right;  // node "to the right"
    struct heap_node_s *child;  // child node
    int degree;                // number of children
    int mark;                  // is this node mark for deletion?
};
```

Bloc de code 4 : fichier `pqueue.h`, définition de la structure de nœud pour les tas de Fibonacci

Les nœuds des tas de Fibonacci conservent en plus d'un pointeur vers l'élément qu'ils représentent des informations supplémentaires telles que leur clef, des pointeurs vers des nœuds voisins, et des informations utiles pour les nombreux parcours des tas.

Algorithme de Dijkstra et structure de nœud principale

L'algorithme de Dijkstra est sans aucun doute l'algorithme de recherche de chemin optimal le plus répandu. Il fonctionne indépendamment de la métrique utilisée pour quantifier ce que l'on appelle une distance (par exemple, dans le cadre de labyrinthes, nous pouvons utiliser la distance de Manhattan). Ici nous choisissons une métrique beaucoup plus simple encore : toutes les arrêtes de notre graphe sont étiquetées par un poids de **1**. Nous précisons ici que l'algorithme de Dijkstra ne fonctionne qu'en l'absence de circuits absorbants (i.e. ayant un poids strictement négatif). L'absence d'arrêtes étiquetées négativement nous garantit le bon fonctionnement de l'algorithme.

Notre implémentation utilise à la fois la structure d'ensemble définie plus haut ainsi que la structure de file de priorité vue précédemment. La structure d'ensemble nous permet de conserver en mémoire l'ensemble des nœuds déjà parcourus, et la file de priorité ordonne rapidement l'ensemble des nœuds à parcourir. Notre algorithme de Dijkstra atteint donc une complexité temporelle de $\Theta(|E| + |V| \log|V|)$, avec E l'ensemble des sommets et V l'ensemble des arrêtes de notre graphe.

```c
struct node_s {
    char *name;            // name of the author
    hset_itr_t *neighbors; // set of neighbor nodes
    dict_itr_t *papers;    // paper title -> year
    int weight;            // weight of the node (distance from start)
    struct node_s *parent; // parent node
};
```

Bloc de code 5 : fichier `node.h`, définition de la structure de nœud pour l'algorithme de Dijkstra

Cette structure de nœuds et la structure principale de notre programme. Chaque nœud représente un auteur. Nous stockons donc naturellement son nom, un ensemble de pointeurs vers ses co-auteurs, un ensemble de correspondances entre des titres de papiers scientifiques et leur date de parution, une distance au nœud de départ pour les parcours (initialisée à l'entier maximum du langage C), ainsi qu'un pointeur vers un nœud parent, attribué lors de l'algorithme de Dijkstra pour pouvoir facilement remonter les nœuds parcourus le long du chemin optimal trouvé.

Parcours en largeur et parcours d'ensembles

La plupart des parcours en largeur dans notre programme se font de manière similaire. Pour un ensemble de nœuds, nous stockons un ensemble de voisins que nous parcourons au fur et à mesure. Ce parcours d'ensemble est réalisé grâce à des structures d'itérateur (que l'on retrouve pour les ensembles et les dictionnaires).

```
1.  struct hset_itr_s {
2.      hset_t *set;
3.      size_t index;
4.  };
5.
6.  struct dict_itr_s {
7.      dict_t *dict;
8.      size_t index;
9.  };
```

Bloc de code 6 : fichiers `hset.h` et `dict.h`, définitions des structures d'itérateurs

Ces itérateurs gardent en mémoire l'indice du dernier élément renvoyé à l'utilisateur pour ne pas recommencer la recherche sur tout le tableau de pointeurs. Cette accélération de parcours et non négligeable, d'autant plus que nos ensembles ne sont jamais remplis à plus de 85% environ. La recherche d'éléments à l'intérieurs des tables de hachages constitue la recherche de pointeurs qui ne redirigent pas vers les adresses `0` et `1`.

Stockage du graphe

Un graphe à proprement parlé est une relation : un ensemble de sommets mis en relations. Nous avons donc choisi de stocker un ensemble de tous les nœuds. Cet ensemble de relations est construit lors du parcours de la base de données originale (le fichier `dblp.xml`). Lors de ce parcours il s'agit de stocker l'ensemble des auteurs d'un article et de rajouter les relations nécessaires entre tous les auteurs d'un même article. Cette partie en particulier a posé des problèmes notamment pour l'aspect temporel de l'algorithme. Une autre difficulté rencontrée a été la difficulté à gérer les fuites de mémoires lors du stockage du nom des auteurs. Nous devons en effet stocker les noms des auteurs de manière unique, mais chaque nom rencontré doit avoir une place réservée en mémoire pour pouvoir être utilisé, indépendamment du fait qu'il soit déjà rencontré ou non. Il faut donc être sûr de stocker tous les pointeurs pour pouvoir libérer la place associée à la fin.

Tests

Un ensemble de tests effectués peut être retrouvé dans le dossier `tests`, qui possède sa propre fonction principale de lancement de tests ainsi qu'un `makefile` à part. Les tests sont compilés avec la règle de débogage principale et sont lancés avec valgrind avec les règles « cov » et « check ». La première génère un rapport de couverture de l'ensemble des fonctions testées, la seconde lance une analyse des fuites mémoire plus approfondie et plus lente mais ne génère pas de rapport de couverture.

D'autres tests ont été effectués manuellement, par exemple pour la gestion des signaux, la recherche d'informations dans le graphe, des tests de rapidité de parcours et d'occupation mémoire sur de grandes quantité de mémoire vive. La totalité des tests a été effectuée au moins une fois avec valgrind.

Un aspect important du projet qui ne fonctionne pas comme espéré est le stockage du graphe dans le fichier binaire. En effet, le parcours de la base de données génère directement l'ensemble de toutes les relations. Leur stockage intermédiaire n'est pas assez efficace pour permettre les constructions suivantes en des temps raisonnables.

Avancement et perspectives

Les principaux aspects du projet ayant été traités (voir la section Répartition du travail) sont les suivants : parseur du fichier xml, parcours de graphe, recherche d'information dans un ensemble, prise en charge de signaux. Malheureusement, nous n'avons pas eu le temps de tout incorporer dans notre fonction principale. Ainsi, seule la construction du fichier binaire à partir de la base de données constitue une option valide dans notre programme.

Une première perspective est naturellement de terminer correctement le sujet. La construction du graphe n'est pas complète (certains champs manquent à l'appel), et la sauvegarde n'est pas très compacte. De plus le format de sauvegarde suppose un parcours du fichier binaire intermédiaire (parcours qui n'a pas été réalisé, mais qui est algorithmiquement simple puisque les données ont déjà été traitées). Une première idée est donc de générer un dump de toutes les zones mémoires où les champs de tous les nœuds du graphe sont stockés. Le chargement du graphe serait donc immédiat de cette manière.

Aussi, l'algorithme d'énumération des diamètres des composantes connexes est un algorithme glouton. Sa complexité temporelle est amortie par celle de Dijkstra qui est très efficace, mais une énumération sur un nombre important de sommets prend hélas beaucoup de temps en pratique. Il existe quelques algorithmes statistiques de déterminations du nombre de composantes de graphes à une erreur (dérisoire) prêt.

Aussi, il est possible de visualiser le graphe (ou du moins un sous-graphe) grâce à OpenGL en C++. Le choix de GLSL n'est pas anodin, étant donné qu'il s'agit de visualiser – donc de générer le rendu plusieurs fois par secondes, d'environ trois millions de nœuds. Une interface graphique pour l'utilisateur pourrait aussi être implémentée avec GTK+ ou encore avec Electron. Elle permettrait à l'utilisateur de choisir de manière plus naturelle les options du programme, et de consulter de manière interactive les résultats de l'analyse de la base de données. De manière ultime, le transfert de tout le code dans un langage plus performant et plus sûr que le langage C permettrait d'ouvrir plus de possibilités.

Conclusion

Même si ce projet n'a pas pu être complet à temps, la plupart des aspects de ce projet a été traitée. Ce projet nous a permis entre autres d'avoir un petit aperçu de la réalisation d'un projet professionnel en groupe. L'organisation a été nouvelle mais enrichissante, d'autant plus qu'il regroupe en un sujet plusieurs aspects des différents cours de première année. Quelques nouvelles idées ont même pu émerger, ce qui devrait donner matière à programmer à la plupart d'entre nous.

Documents de référence

1. Petra Berenbrink, Bruce Krayenhoff, Frederik Mallmann-Trenn, *Estimating the number of connected components in sublinear time*, Information Processing Letters, Volume 114, Issue 11, 2014, Pages 639-642, ISSN 0020-0190, https://doi.org/10.1016/j.ipl.2014.05.008.

2. Stephan Foldes, *Fundamental Structures of Algebra and Discrete Mathematics*, Wiley-Interscience, 2011, Pages 344, ISBN 0-471-57180-6

3. Cioabă S.M, *Some Applications of Eigenvalues of Graphs*, Dehmer M. (eds) Structural Analysis of Complex Networks, 2011, Pages 357-379, https://doi.org/10.1007/978-0-8176-4789-6

4. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 1996, Pages 586-589, ISBN 0-898-71366-8 urn:oclc:record:1151098525 (archive.org)

5. W. D. Maurer, G. Lewis, *ACM Computing* Surveys, Hash Table Methods, Volume 7, Issue 1, 1975, Pages 5-19 https://doi.org/10.1145/356643.356645