

# Projet Programmation 2021



## Objectifs du projet :

L'objectif de ce projet est d'analyser la base de données fournie par <https://dblp.org/> afin d'y appliquer des algorithmes de graphes. Le fichier .xml fourni contient environ 81 millions de lignes pour 8,7 millions de références liées à l'informatique et 2 947 635 auteurs distincts.

Nous avons donc répondu à cette problématique en codant un programme en C permettant d'extraire les données, afficher les informations importantes des références, générer un graphe et sur celui-ci calculer des plus courts chemins, le nombre de composantes connexes, rechercher des auteurs et leurs articles et afficher leurs voisins et leurs distances.

## Structure du projet :

Le projet est composé des dossiers :

- **database** (absent du git) contient la base de données sous forme .xml et .bin
- **src** contient les programmes .c
- **include** contient les fichiers d'en-tête .h
- **obj** stocke les fichiers .o
- **bin** contient l'exécutable du programme
- **tests** contient :
  - Le programme tests.c et son makefile avec un dossier obj stockant les .o, .gcca et .gcco
  - Plusieurs extraits du fichier .xml et leurs .bin pour les tests
  - Un dossier rapport pour le rapport de Lcov
  - Un dossier tests\_outputs contenant les résultats attendus dans le test.sh

## Structure du programme :

Le programme s'articule autour de 9 fichiers .c :

- **program.c** contient le main et s'exécute après compilation par ./bin/program et des options (voir plus loin). Il est basé sur le tableau binaire options.actions[ ] qui est rempli par la fonction parseArgs et permet d'exécuter plusieurs actions à la suite (dépendantes ou non).
- **argsParser.c** contient:
  - La fonction printUsage appelée avec l'option -h, ou aucune option, pour afficher l'aide du programme.
  - La fonction de passage des arguments basée sur getopt vers la structure options.
- **io-utils.c** contient:
  - Les fonctions pour la gestion du signal CTRL+C, des fichiers (ouverture, fermeture, vérification du type), de la fin du programme (fermetures et libération de la mémoire allouée).
  - La fonction errorToString convertissant les codes d'erreurs de la structure error\_t en message.

- **baseParser.c** contient:
  - Les fonctions d'extraction des données importantes depuis le xml vers le fichier binaire par la structure `structureBase_t` (titre, auteurs, année).
  - Le code permettant la correction des erreurs d'encodage de la base `dblp.xml`.
- **readBin.c** contient:
  - La fonction de lecture du fichier binaire en entier et celle de comptage des références pour la barre de progression.
  - Les fonctions de lecture et d'affichage d'une référence (article, thèse, livre...).
- **makeGraph.c** contient:
  - La fonction de hachage.
  - Les fonctions pour créer la liste d'adjacence.
  - Les fonctions pour libérer l'espace.
- **analyseGraph.c** contient:
  - L'algorithme de Dijkstra et les fonctions de calcul et d'affichage du plus court chemin et de tous les sommets à une certaine distance ou dans la composante connexe.
  - Les fonctions comptant le nombre de composantes connexes du graphe par exploration en largeur.
  - Les fonctions d'affichage du graphe.
- **searchingFunctions.c** contient:
  - Les fonctions de recherche des auteurs dans le graphe et des articles dans le fichier binaire pour les afficher.
  - La fonction `chooseAuthor` qui permet de réduire à un seul auteur l'appel aux fonctions lorsque celui donné en argument n'est pas unique ou suffisamment précis pour n'en renvoyer qu'un.
- **tests.c** contient 13 tests couvrant environ 90% des lignes de code (voir plus loin).

**makeGraph.c** utilise une table de hachage constituée d'un tableau de pointeurs pour avoir accès directement aux auteurs du graphe dans la liste chaînée. Cette fonction a été inspirée et adaptée depuis : <http://www.cse.yorku.ca/~oz/hash.html>. Lors de la lecture d'un auteur dans le fichier binaire pendant la génération du graphe, un appel à cette fonction le convertit en un nombre entre 0 et 100 millions (la valeur de `HT_SIZE`). Si la case à cette position dans la table est nulle, le programme ajoute un nœud à la liste chaînée et place le pointeur vers ce nœud dans la case. Si au contraire elle n'est pas nulle, le programme vérifie que l'auteur dans le nœud pointé correspond bien à celui en cours de traitement. Si oui, l'auteur est présent dans le graphe et on passe au suivant. Si non, le nom de l'auteur est haché avec un autre nombre premier (4 ont été nécessaires pour éliminer toute collision : `pr1` à `pr4`) et on réitère le procédé. Ainsi le hachage est totalement déterministe et on peut savoir grâce à une recherche en  $O(1)$  si un auteur est présent dans le graphe ou non sans tout parcourir.

## Options du programme :

Le programme comporte 14 options :

- **-i** permet de donner un fichier xml (obligatoire uniquement pour l'extraction) et **-o** un fichier binaire.
- **-x** extrait les données du xml et les écrit dans le binaire.
- **-r** lit l'ensemble du binaire (sauf si interrompu) correspondant à la base simplifiée et l'affiche à l'écran.
- **-g** créé le graphe à partir du binaire et si **-s** est présent l'affiche à l'écran sous la forme d'une liste de sommets numérotés et d'une liste d'arêtes.
- **-I MOTIF** recherche le motif donné en argument dans le graphe et affiche les auteurs correspondants.
- **-a AUTEUR** recherche l'auteur et s'il est unique (suffisamment précis pour ne donner qu'un auteur lors de la recherche) affiche tous ses articles Sinon le programme demande d'en entrer un plus précis dans la liste renvoyée. Si l'argument **-y ANNEE** est donné seuls les articles écrits pendant cette année par cet auteur seront affichés.
- **-a AUTEUR -n N** affiche tous les auteurs à distance N de l'auteur après avoir appliqué l'algorithme de Dijkstra sur toute la composante connexe de l'auteur.
- **-a AUTEUR -d** affiche les distances de tous les auteurs de la composante connexe par rapport à l'auteur donné après avoir appliqué l'algorithme de Dijkstra.
- **-p AUTEUR1 -p AUTEUR2** lance l'algorithme de Dijkstra sur le graphe et affiche le plus court chemin entre les deux auteurs s'il existe ainsi que leur distance ou un message sinon (après avoir généré le graphe et vérifié les auteurs).
- **-p AUTEUR1 -p AUTEUR2 -d** affiche les distances depuis l'auteur 1 de tous les auteurs situés à une distance inférieure à celle de l'auteur 2 à la place du plus court chemin entre eux.
- **-c** compte le nombre de composantes connexes du graphe à l'aide d'un parcours en largeur.
- **-h** affiche l'aide (de même qu'aucun argument).

**N. B.** : Certaines options sont implicitement appelées comme par exemple **-p** déclenche forcément la génération du graphe même si **-g** n'est pas donnée. Il n'est pas nécessaire d'appeler plusieurs des lignes ci-dessus ayant des options distinctes sauf si le but est d'effectuer successivement plusieurs actions.

De plus l'ordre n'a pas d'importance (utilisation de getopt).

## Structures struct utilisées dans le programme :

La structure suivante est utilisée pour lire et écrire dans le fichier binaire. Les chaînes de caractère sont coupées afin de ne pas consommer de l'espace inutilement avec des caractères vides et obtenir un fichier binaire moins volumineux que la base de données initiale en xml. Elle contient le titre, les auteurs et l'année d'un article ainsi que la longueur des chaînes de caractère pour le découpage.

```
typedef struct structureBase_t {
    int16_t year;
    int16_t titleLength;
    char title[1400];
    int16_t authornb;
    int8_t authorlengths[500];
    char author[500][85];
} structureBase_t;
```

La structure ci-dessous est la structure des sommets du graphe. Les sommets sont stockés sous la forme d'une liste chaînée avec le sommet 0 pointant vers le sommet 1 qui pointe vers le sommet 2 etc... Le sommet contient aussi un pointeur vers la liste de ses arêtes (nodeEdge). flag permet de déterminer si le sommet a été exploré ou non, il est initialisé à 0.

```
typedef struct node {
    char author[85];
    int32_t nodeNumber;
    int16_t distance;
    int8_t flag;
    struct edge *nodeEdge;
    struct node *nextNode;
} node;
```

La structure ci-dessous est la structure des arêtes du graphe. Les arêtes sont stockées sous forme de liste, chaque sommet pointe vers sa propre liste d'arêtes. linkNode est le sommet original de l'arête, otherNode est le sommet à l'autre bout de l'arête.

```
typedef struct edge {
    struct node *linkNode;
    struct node *otherNode;
    struct edge *nextEdge;
} edge;
```

Exemple de graphe : 
$$\begin{array}{c} 1 \text{ --- } 0 \text{ --- } 3 \\ | \quad / \\ 2 \end{array}$$

Sommets	Arêtes: linkNode → otherNode
0	0 → 1   0 → 2   0 → 3   NULL
1	1 → 0   NULL
2	2 → 0   2 → 3   NULL
3	3 → 0   3 → 2   NULL
NULL	



## Résultats des tests :

Toutes les commandes s'exécutent sans erreur Valgrind sur la base de données dblp.xml et sur des petits échantillons dans les tests.

Les tests sont constitués d'un script test.sh testant toutes les commandes à l'aide de valgrind et une partie des erreurs de saisie ou d'arguments puis comme test final exécute un make cov sur le tests.c.

Lcov donne une couverture proche de 90% des lignes. Les 10% restantes correspondent à des lignes de retour d'erreurs et au test de la fonction chooseAuthor qui causait une segmentation fault inexplicée puis des erreurs valgrind lors de la saisie dans le terminal d'un nom d'auteur.

### *LCOV - code coverage report*

Current view: top level		Hit		Total	Coverage	
Test: Couverture		Lines:	936	1041	89.9 %	
Date: 2022-01-21 16:53:24		Functions:	62	64	96.9 %	
Directory		Line Coverage ↕		Functions ↕		
src			87.9 %	720 / 819	96.0 %	48 / 50
tests			97.3 %	216 / 222	100.0 %	14 / 14

Generated by: LCOV version 1.14