Distance de Jaccard

Maysaloon BILAL & Tristan GROULT

14 mai 2025

Sommaire

1 Introduction

Ce projet consiste à développer un programme en langage C permettant de calculer la distance de Jaccard entre plusieurs fichiers texte, afin d'analyser leur similarité lexicale. La distance de Jaccard est une valeur comprise entre 0 et 1 : plus elle est proche de 0, plus les fichiers sont différents, et plus elle est proche de 1, plus ils sont similaires.

La distance de Jaccard entre deux ensembles A et B est définie par la formule :

$$\text{Distance de Jaccard}(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$$

Le programme propose plusieurs modes d'affichage ainsi que des calculs personnalisés. Il est possible de mesurer la similarité entre les fichiers à l'aide de l'indice de Jaccard, que nous désignerons par la suite par *Calcul de Jaccard*. Il est également possible d'afficher un graphe représentant l'appartenance des mots à leur(s) fichier(s), que nous nommerons *Graphe de Jaccard*. Nous utiliserons le terme *Calcul de Jaccard* pour faire référence à la réalisation ou la préparation de ces deux opérations.

Pour gérer efficacement les mots extraits des fichiers, nous avons utilisé une table de hachage. Sa gestion — ajout, suppression, affichage du graphe, etc. — est implémentée dans le module `jaccard`. Ce projet s'appuie également sur deux autres modules : le module `word` sous-section 2.1, dédié à la création et la manipulation des mots, et le module `opt` sous-section 2.2, responsable du traitement des options en ligne de commande.

L'ensemble de ces composants est intégré et testé dans le fichier principal `main.c`, qui constitue le point d'entrée du programme.

2 Conception et implémentation

Dans cette section, nous expliquons les différentes étapes et approches utilisées pour implémenter le programme. Nous détaillerons la conception des modules, la gestion des fichiers et l'utilisation de la table de hachage.

2.1 Module word

Le module `word` a pour objectif de construire dynamiquement un mot caractère par caractère, afin de faciliter son traitement sans avoir à connaître sa taille à l'avance. Ce module est utilisé lors de la lecture des fichiers afin d'extraire chaque mot individuellement.

2.1.1 Spécification

Pour cela, nous utilisons une structure de données définie dans la figure 1. Cette structure permet de stocker un mot, d'en gérer dynamiquement la taille, et de le manipuler facilement sans risque de dépassement mémoire.

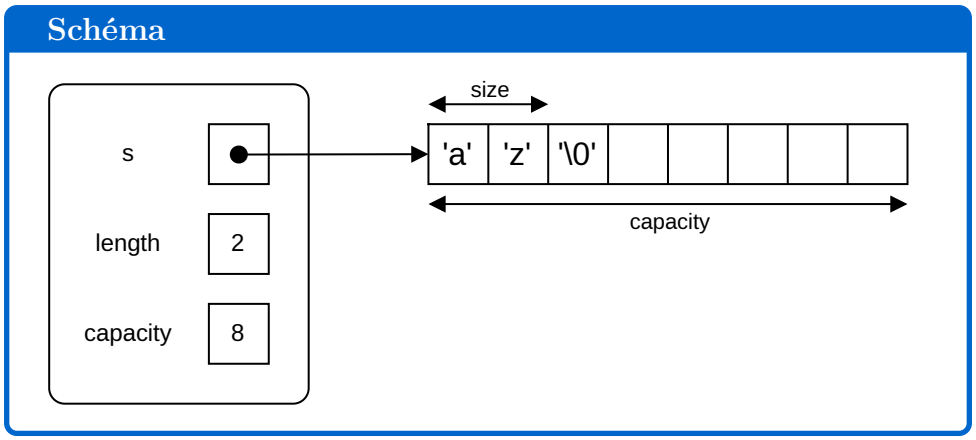

FIGURE 1 – Schéma de la structure word

2.1.2 Implantation

Lors de l'implémentation, nous avons défini une structure `struct word` (figure 2) pour représenter dynamiquement un mot. Elle est composée de trois champs :

— Un champ `s` de type `char *` qui pointe vers un tableau de caractères contenant le mot.
— Un champ `length` de type `size_t` qui représente la longueur actuelle du mot.
— Un champ `capacity` de type `size_t` qui correspond à la taille totale allouée pour le mot.

Nous avons choisi une approche avec allocation mémoire dynamique, ce qui permet de s'adapter à la longueur des mots rencontrés, en multipliant par la constante `CAPACITY_MUL` la capacité du mot lorsque celle-ci est atteinte.

```
struct word {
    char *s;
    size_t length;
    size_t capacity;
};
```

FIGURE 2 – Structure word

Nous avons développé plusieurs fonctions pour interagir avec ce module :

— `word_init()` : Initialise un mot vide en allouant une capacité minimale définie par la constante `CAPACITY_MIN`, et met la longueur à zéro.
— `word_add()` : Ajoute un caractère à la fin du mot. Si la capacité est atteinte, elle est doublée avec `realloc()` pour permettre l'ajout de nouveaux caractères.
— `word_reinit()` : Réinitialise le mot en mettant sa longueur à zéro, ce qui permet de le réutiliser sans réallouer de mémoire.
— `word_get()` : Renvoie un pointeur vers la chaîne de caractères actuelle.
— `word_get_clean()` : Copie le contenu réel du mot dans une destination passée en paramètre, sans les éventuels caractères résiduels.

2.2 Module opt

Le module `opt` a pour objectif de gérer les options de la ligne de commande, en les analysant puis en les restituant sous une forme exploitable par le reste du programme. Dans le cadre de ce projet, les options suivantes sont nécessaires :

- `-g` : Affiche le graphe de Jaccard.
- `-p` : Ignore les caractères de ponctuation lors de l'analyse.
- `-iVALUE` : Spécifie la longueur maximale des mots.
- `-` : Utilise l'entrée standard comme source.
- `--` : Indique que l'argument suivant est un nom de fichier.
- `-u` : Affiche les informations d'utilisation de l'exécutable.
- `-v` : Affiche la version de l'exécutable.
- `-?` : Affiche l'aide de l'exécutable.

2.2.1 Spécification

Pour représenter les options analysées, nous utilisons une structure de données illustrée dans la figure 3. Nous avons choisi de limiter le nombre de fichiers grâce à une constante nommée `MAX_FILES`. Ce choix permet d'assurer une consommation mémoire bornée et facilite l'intégration avec les contraintes d'implémentation du module `jaccard`.

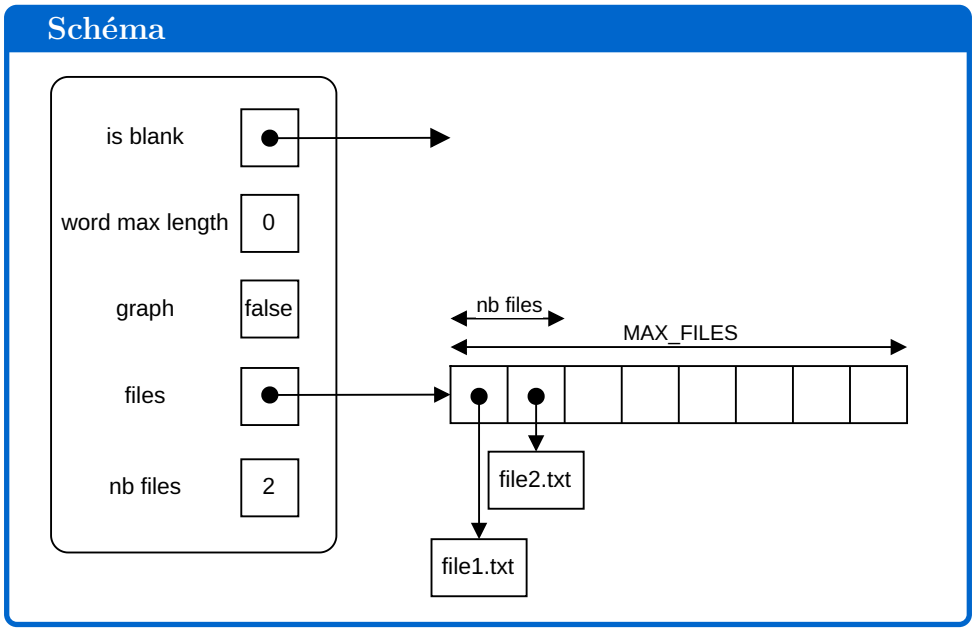

FIGURE 3 – Schéma de la structure opt

2.2.2 Implémentation

L'implémentation repose sur l'utilisation d'une structure `struct opt`, définie comme suit (figure 4) :

- `isBlank` : Pointeur vers une fonction `int(int)` déterminant les caractères séparateurs de mots.
- `word_max_length` : Entier définissant la longueur maximale d'un mot (0 signifie aucune limite ; valeur par défaut : 0).
- `graph` : Booléen indiquant si le graphe de Jaccard doit être affiché.

— `files` : Tableau de chaînes de caractères représentant les noms des fichiers à analyser.
— `nb_files` : Nombre de fichiers fournis en argument.

```
struct opt {
    int (*isBlank)(int);
    int word_max_length;
    bool graph;
    char const **files;
    int nb_files;
};
```

FIGURE 4 – Structure opt

Afin de permettre une gestion homogène de l'entrée standard, nous la représentons sous la forme d'une chaîne vide (`""`), en la définissant via une macro `STDIN`. Cette chaîne est utilisée comme nom de fichier, en tirant parti du fait qu'aucun système courant ne l'autorise comme nom valide.

Seules les options courtes ont été implémentées, comme listé plus haut. Pour l'option `-iVALUE`, la valeur doit être directement collée à l'option (par exemple `-i3`), l'espacement (`-i 3`) n'étant pas pris en charge.

Enfin, toutes les options sont définies via des macros constantes. Le préfixe des options courtes est également paramétrable, ce qui rend l'ensemble du module facilement adaptable à d'autres conventions si nécessaire.

2.3 Module Jaccard

Le module `jaccard` a pour objectif de réaliser le calcul de Jaccard en donnant la possibilité d'ajouter des mots au calcul ainsi que de gérer l'affichage du graphe ou le calcul de la distance.

2.3.1 Spécification

Pour réaliser cela, nous utilisons une structure de données définie dans la figure 5. Cette structure permet de sauvegarder les éléments utilisés pour réaliser le calcul de la distance ainsi que les autres structures que nous avons décidé d'utiliser.

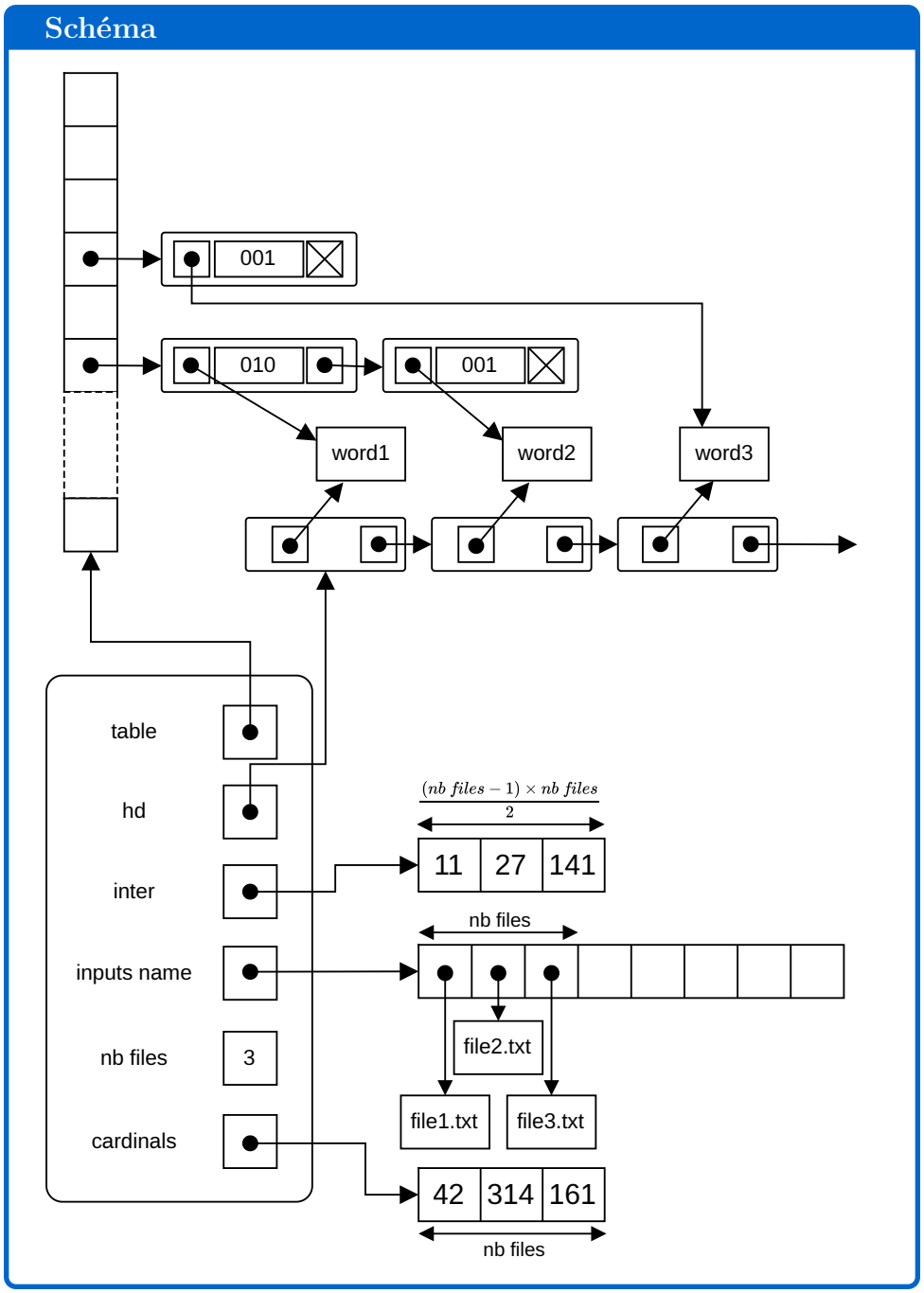

FIGURE 5 – Schéma de la structure jaccard

Pour réaliser le calcul de Jaccard, nous avons besoin :

— D'un champ $nb\ files$ permettant de connaître le nombre de fichiers utilisés pour le calcul de Jaccard courant.

— D'un tableau $inputs\ name$ contenant le nom des fichiers utilisés lors de l'affichage du graphe de Jaccard ou du calcul de Jaccard. L'ordre des fichiers est important car plusieurs autres endroits utilisent ce même ordre pour représenter ces mêmes fichiers. On considère que ce tableau de noms de fichiers est de taille $nb\ files$.

— D'un tableau constitué des cardinaux de la forme ensembliste de chaque fichier. Le cardinal à l'indice i est celui du fichier d'indice i dans inputs name. Ce tableau est donc de taille $nb\ files$.

— D'un tableau pour stocker les cardinaux des intersections entre chaque paire de fichiers. Chaque case du tableau représente une paire de fichiers différents. La taille totale du tableau est égale au nombre de paires possibles, soit $\frac{n \cdot (n-1)}{2}$ si on a n fichiers.

Pour deux fichiers avec les indices i et j (avec $i < j$), l'indice idx de la case du tableau correspondant au cardinal de leur intersection est :

$$idx(i,j) = i \cdot nb\ files - \frac{i \cdot (i+1)}{2} + (j - i - 1)$$

Cette formule transforme un couple d'indices de fichiers (i, j) en un indice unique dans le tableau, ce qui permet de toujours savoir à quelle intersection correspond chaque case.

— Un champ *table* contenant une table de hachage dans laquelle les mots vont pouvoir être recherchés et ajoutés durant le calcul de Jaccard.

— Le champ *hd* contenant une liste dynamique en mesure d'accueillir tous les mots qui sont ajoutés au calcul de Jaccard. C'est sur cette liste dynamique que doit être réalisé le tri si nécessaire car une table de hachage n'est pas triée.

La mise à jour du tableau des cardinaux des intersections est réalisée à chaque ajout d'un nouveau mot au calcul de Jaccard en suivant l'algorithme suivant :

```
Soit M le mot
Soit T le bitmap du mot M
Soit F le fichier dans lequel on a lu M
Soit j l'indice de F
Soit I le tableau des cardinaux des intersections
Soit i = 0

tant que i < j faire
    si M[i] = 1 alors
         I[idx(i, j)] <- I[idx(i, j)] + 1
    fin si
    i <- i + 1
fin tant que
```

FIGURE 6 – Algorithme de mise à jour du tableau des cardinaux des intersections

Voici un exemple de l'application de l'algorithme.

Exemple

Mot **M** lu dans le fichier **C** d'indice j = 2 et lu auparavant dans le fichier A d'indice 0

Mise à jour tableau d'intersection :

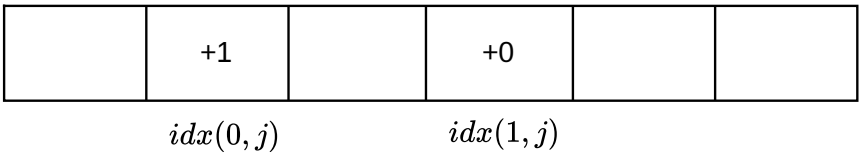

FIGURE 7 – Exemple mise à jour du tableau des cardinaux des intersections

2.3.2 Implémentation

Pour la réalisation de ce module nous avons développé de nombreuses fonctions afin de créer un calcul de Jaccard, d'ajouter des mots au calcul et finalement de procéder à l'affichage du calcul de Jaccard et du graphe de Jaccard.

```
struct jcrd {
    hashtable *table;
    holdall *hd;
    const char **inputs_name;
    size_t *inter;
    bool graph;
    int nb_files;
    size_t *cardinals;
};
```

FIGURE 8 – Exemple d'utilisation des masques sur une bitmap

Nous avons rencontré un problème sur le calcul de l'appartenance de chaque mot à son/ses fichiers. Nous avons décidé d'utiliser une bitmap afin de représenter ce tableau d'appartenance. La table de hachage a comme champ `ref` un pointeur de type `const void *`. Pour limiter notre nombre d'allocations, nous avons décidé d'écrire directement notre bitmap sur l'adresse du pointeur plutôt que d'allouer un nombre servant de bitmap. Pour réaliser cette manipulation, nous utilisons le type `uint64_t` permettant un codage assuré sur 64 bits, la taille d'une adresse mémoire. L'utilisation de ce type est aussi intéressante pour pouvoir caster ce nombre en une adresse grâce aux forceurs de type `(void *) (uintptr_t)` qui est compatible avec le type `const void *`. Nous vérifions la présence d'un bit à 0 à un indice grâce à des masques. Voici par exemple un masque permettant de détecter si le bit d'indice i est à 0 et permettant de mettre le bit d'indice i à 1 comme vous pouvez le voir dans l'exemple suivant : figure 9

```
bitmap & (1ULL << i) // Detecte si le bit d indice i est a 1
bitmap |= (1ULL << i) // Met a 1 le bit d indice i
```

FIGURE 9 – Exemple d'utilisation des masques sur une bitmap

— `jaccard_init()` Initialise un nouveau calcul de Jaccard pour le nombre de fichiers et noms passés en paramètre.
— `jaccard_add()` traite un mot lu dans un fichier, et met à jour à la fois la table de hachage, les cardinaux des fichiers, et le tableau des cardinaux des intersections. Voir figure 10 pour un exemple.
— `jcrd_print_distance()` Calcule et affiche les distances de Jaccard entre toutes les paires de fichiers analysés. Pour chaque paire de fichiers (i, j) avec (avec $i < j$), la fonction effectue les opérations suivantes :
  — calcul de l'union pour les fichiers d'indices i et j

$$CardUnion(i, j) = cardinals[i] + cardinals[j] - inter[idx(i, j)]$$

  — Calcul de la distance de Jaccard par application de la formule :

$$distance(i, j) = 1 - \frac{inter[idx(i, j)]}{CardUnion(i, j)}$$

— `jcrd_print_graph()` Affiche sur la sortie standard le graphe de Jaccard. Pour cela, les mots ont été préalablement triés grâce à la fonction de comparaison `strcol` passée en paramètre de la fonction `holdall_sort`

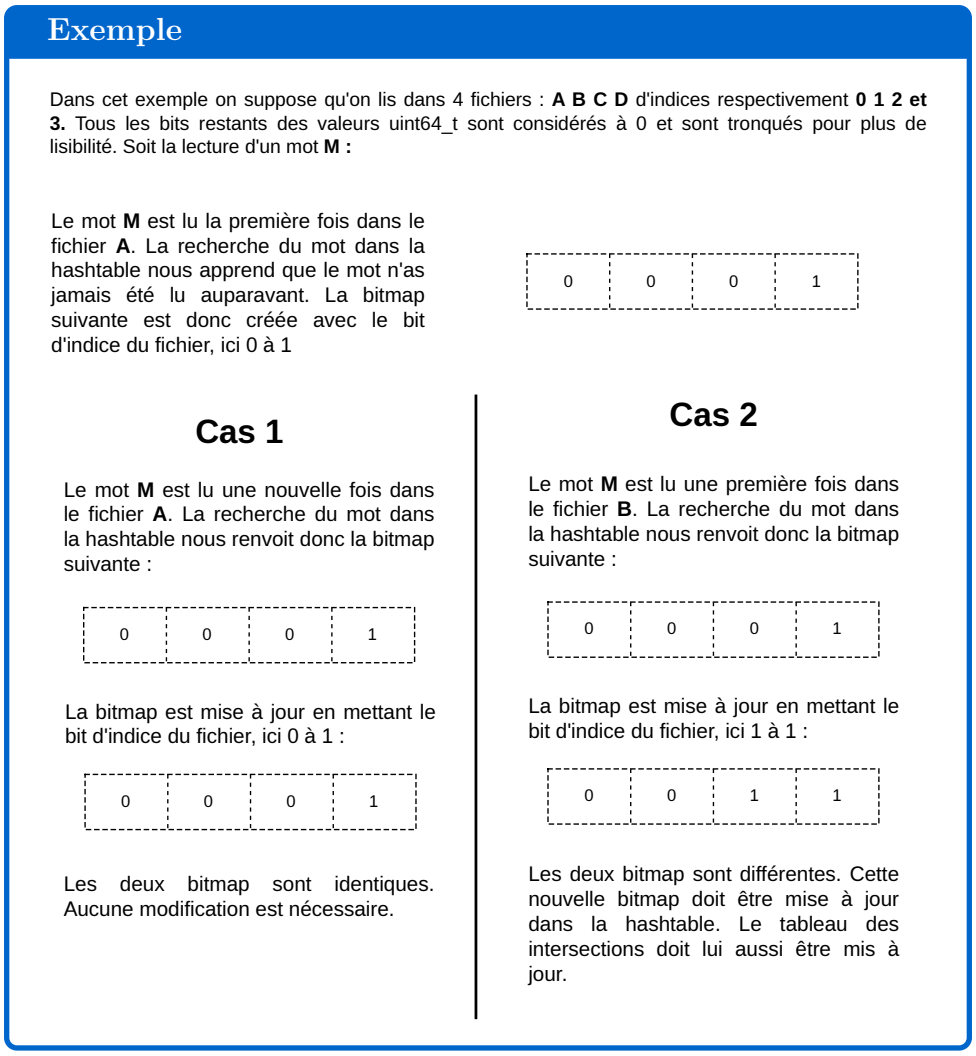

FIGURE 10 – Schéma du tableau d'intersections

2.4 Module holdall

L'implantation de ce module n'a pas été modifiée à l'exception de l'extension qui a été implémentée et qui comprend la fonction de tri `holdall_sort` permettant de trier le trousseau. Nous avons choisi d'implémenter pour ce tri, l'algorithme de tri fusion afin d'assurer une complexité en tout cas de $O(n \log n)$.

2.5 Main

2.5.1 Implémentation

— Le programme commence par initialiser les modules `opt`, `jcrd`, `word` et `holdall`.
— Chaque mot des fichiers est lu et ajouté au calcul de Jaccard
— Une fois la lecture de chaque fichier terminée, celui-ci est fermé (ou réinitialisé dans le cas de `stdin` avec `rewind(stdin)`).
— À la fin de la lecture, si l'option d'affichage de graphe n'est pas activée, le programme affiche les distances de Jaccard calculées entre toutes les paires de fichiers. Sinon, il affiche le graphe de Jaccard.

— Enfin, toutes les structures utilisées sont libérées proprement (`word`, `jcrd` et `opt` et `holdall`) pour éviter les fuites mémoire et on renvoie `EXIT_SUCCESS`. Et toute éventuelle erreur au cours de l'exécution arrête le programme en affichant le type de l'erreur et en libérant toutes les ressources allouées jusque-là. Dans ce cas la fonction renvoie `EXIT_FAILURE`.

3 Améliorations possibles

L'utilisation de `uint64_t` avec le codage de l'appartenance des mots à leur fichier directement sur l'adresse fonctionne uniquement pour les systèmes en 64 bits. Une amélioration possible serait de gérer lors de la compilation, grâce à la définition d'une macro constante, le nombre de fichiers maximum pris en compte et d'adapter en conséquence le codage de l'appartenance des mots à leur fichier afin de s'adapter aux différents codages des adresses des machines.