Université de Bordeaux — Licence 3 Informatique — Sylvain Chiron

Rapport de stage

Développement du logiciel GenGraph et d’un algorithme de reconnaissance des graphes lignes

Du 9 mai au 30 juin 2023

Sommaire

1. Contexte
    1. La structure d’accueil : le LaBRI et l’équipe GO
    2. Le logiciel et son créateur : GenGraph, par Cyril Gavoille
    3. Intérêt et motivation
        1. Portée scientifique
        2. Éthique

2. Sujet
    1. Développement des fonctionnalités du projet
        1. Mise en place d’un système de pile de requêtes
            1. Passage à une notation polonaise hybride
            2. Les instructions dans GenGraph
        2. Détermination des représentations de graphes à utiliser
            1. Le problème tel qu’il était présenté aux utilisateurs
            2. Analyse approfondie et critique du générateur
        3. Prototypage d’instructions tirant parti du nouveau système
        4. Amélioration du système d’aide
            1. Les changements à apporter
            2. La reconception du manuel
            3. Les rapports d’erreur
    2. Mise en place d’un dépôt du projet
        1. Aménagement du dépôt
        2. Programmation de la construction et de l’installation
        3. Définition des conventions de contribution
    3. Utilisation de données scientifiques
        1. Les graphes chenilles
        2. Les graphes lignes et l’opération inverse
            1. Présentation du travail fait
            2. Résumé de l’algorithme d’inversion (ILIGRA)
    4. Semaine typique

3. Déroulement
    1. Découpage en modules
        1. Vue d’ensemble du résultat
            1. Les quatre sous-projets
            2. Les dossiers
        2. Les différents modules
            1. Dans la bibliothèque GenGraph
            2. Dans les autres sous-projets
        3. Environnement et procédures de développement
            1. Logiciels utilisés
            2. Pratiques
    2. Mise en place des procédures de construction

        1. L'outil : GNU Make et les commandes POSIX
        2. Les cibles de compilation
        3. Les outils pour travailler avec un seul fichier
        4. Les procédures d'installation et de désinstallation
        5. Les plateformes utilisées pour faire les tests
    3. Travail sur l'aide
        1. Traitement du code préexistant
        2. Les fonctions du noyau
            1. Le type `Clih_ParserBuf` pour les traitements
            2. La fonction pour les substitions : `replace()`
            3. Les outils pour l'analyse de la syntaxe
            4. La génération de toutes les commandes possibles : `getOptionCase()`
        3. La nouvelle syntaxe
            1. Simplification du format d'origine
            2. Les délimiteurs de format
            3. Les styles de blocs
            4. Les formats d'instructions
        4. Les formats de sortie
            1. ANSI
            2. HTML
            3. PDF
        5. Le traitement des arguments et des erreurs
            1. La vérification de l'argument suivant
            2. Le tableau de la section en cours
            3. Le signalement des erreurs et les avertissements
        6. L'ouverture des fichiers et les accessoires
            1. L'implémentation des accessoires
            2. L'ouverture des fichiers texte et HTML
        7. La complétion
    4. Refonte de l'analyse des instructions
        1. Les fonctions, structures et macros centrales
        2. Les macros, structures et fonctions pour les instructions de graphes
        3. L'analyse des autres types d'instructions
    5. Implémentation des nouveaux algorithmes
    6. Organisation des contributions
    7. Relecture des fonctionnalités existantes
    8. Développement de nouvelles instructions

4. Bilan

5. International part
    1. Graphviz

1. Contexte

1.1. La structure d'accueil : le LaBRI et l'équipe GO

Le LaBRI, Laboratoire bordelais de recherche en informatique, est une structure en annexe de l'unité de formation Informatique de l'Université de Bordeaux, dans laquelle travaillent également des chercheurs de Bordeaux INP, du CNRS et de l'Inria. Environ 150 chercheurs et enseignants-chercheurs y sont employés, ainsi qu'une centaine de doctorants. En cette année académique 2022–2023, le laboratoire a accueilli exactement 101 stagiaires, d'élèves de troisième à étudiants d'ENS et apprentis.

Le LaBRI est structuré en 14 équipes de recherche regroupées en 5 départements :

- Combinatoire et Algorithmique (CombAlgo) ;
- Image et Son (I&S) ;

- Méthodes et Modèles Formels (M2F) ;
- Systèmes et Données (SeD) ;
- Supports et AlgoriThmes pour les Applications Numériques hAutes performanceS (SATANAS).

Le département Combinatoire et Algorithmique est découpé en les 3 équipes suivantes :

- Combinatoire et Interactions ;
- Graphes et Optimisation ;
- Algorithmique Distribuée.

L'équipe Graphes et Optimisation (GO) travaille sur le développement et les applications de la théorie des graphes. C'est dans cette équipe que Cyril Gavoille m'a invité afin que je développe GenGraph.

1.2. Le logiciel et son créateur : GenGraph, par Cyril Gavoille

GenGraph est un logiciel développé par Cyril Gavoille depuis 2007, prenant la forme d'un outil en ligne de commande permettant essentiellement de :

- générer des graphes uniformément à partir du nom d'une classe de graphes, d'éventuels paramètres et d'options diverses ;
- exécuter certains traitements dessus : algorithmes sur un graphe, filtres sur un groupe de graphes ;
- paramétrer la génération au format DOT pour le rendu avec Graphviz.

Le principal format de sortie est la liste des arêtes du graphe au format texte. Le logiciel l'accepte également comme format d'entrée. GenGraph est développé en langage C et peut interopérer avec d'autres logiciels via les formats de fichiers, mais fondamentalement ne repose sur aucune dépendance externe. Il avait été testé sur GNU/Linux et macOS avec les compilateurs GCC et Clang.

GenGraph s'utilise avec des options en anglais et fournit ses résultats en anglais. Cependant, son manuel n'existe qu'en français, et ses messages d'erreur sont également en français. Dans le code source, les identifiants (noms de fonctions et de variables) sont en anglais, mais il comporte de très nombreux commentaires qui sont tous en français.

Certaines fonctionnalités de GenGraph ne sont prévues que pour des graphes très petits, mais les plus rapides peuvent traiter en quelques secondes des graphes ayant une taille de l'ordre de la dizaine de millions, avec peu de mémoire. GenGraph implémente aussi bien des fonctionnalités simples et connues (complémentation de graphe, génération de graphe k-parti complet, graphes fixes, recherche de cycle, répartition géométrique des sommets sur un cercle…) que des fonctionnalités avancées (parmi celles qui me parlent : calcul de largeur arborescente, test d'isomorphisme, k-coloration, génération aléatoire uniforme, de graphe planaire, de graphe ligne ou de triangulations…) voire très avancées (construction de tables de routage ou de graphe des conflits, génération aléatoire uniforme de graphe de Gosset ou de treillis planaire…).

Jusqu'alors, aucune autre personne n'était intervenue sérieusement sur le projet. Cyril Gavoille le développait complètement en solitaire depuis quinze ans.

Cyril Gavoille, aujourd'hui responsable du département CombAlgo du LaBRI, est enseignant-chercheur à l'Université de Bordeaux depuis 1996. Nous sommes rentrés en contact car il a été mon enseignant d'amphithéâtre et de TD dans la matière Techniques algorithmiques et programmation au deuxième semestre de l'année 2022–2023, et il a été le tuteur et maitre de ce stage.

Au LaBRI, Cyril Gavoille est membre de l'équipe Graphes et Optimisation mais également de l'équipe Algorithmique Distribuée, qui fait elle aussi partie du département CombAlgo. Il enseigne l'algorithmique distribuée dans plusieurs cursus de master, et est coordinateur du projet ANR Descartes. Là où GenGraph est un logiciel généraliste de génération et de traitement de graphes, Descartes est une définition de modèle généraliste pour le calcul distribué. Descartes est plus récent que GenGraph mais connait davantage de partenaires.

1.3. Intérêt et motivation

1.3.1. Portée scientifique

Je souhaitais à la fois utiliser mes compétences et découvrir l'environnement et le fonctionnement d'un laboratoire de recherche en informatique. J'ai déjà démontré un intérêt pour le thème de la théorie des graphes en développant une visionneuse web d'algorithmes de graphes (instance, dépôt).

Le projet GenGraph m'a énormément captivé, d'une part parce qu'il était déjà riche en fonctionnalités et en algorithmes puissants, et d'autre part parce qu'il a une portée extrêmement générale. Il a la perspective d'être à la fois un outil professionnel et un jouet sensationnel. Il n'y a pour moi rien de plus relaxant que de pouvoir se vanter d'avoir touché des notions scientifiques en s'amusant avec un outil qui permet l'expression et le développement de l'imagination. En contribuant à GenGraph, je souhaite faire profiter

de cette richesse au plus grand nombre.

GenGraph présente ainsi selon moi une importante valeur à la fois pédagogique et ludique, car il offre une interface simple permettant de manipuler des graphes avec de nombreux paramètres, il a de nombreuses options pour configurer le rendu, et les algorithmes sont documentés. Il pourrait devenir un moyen pour n'importe quel profane de découvrir interactivement des notions de la si célèbre théorie des graphes, en acquérant dans le même temps une certaine familiarité avec les lignes de commandes et les algorithmes, grâce à la forte relation entre les mots écrits par l'utilisateur et le rendu du programme, et la facilité d'attribuer un sens aux objets générés. Voilà ce que j'en pense.

Bien que je ne puisse pas passer à côté de la théorie des graphes, je ne me connais pas le même intérêt pour elle que les personnes que j'ai rencontrées dans l'équipe GO. En revanche, GenGraph, qui est avant tout un logiciel de génération et de traitement de graphes, a besoin de solutions d'interface et de théorie des langages, qui sont actuellement au cœur de mes réflexions.

1.3.2. Éthique

Cyril Gavoille souhaite que GenGraph soit un logiciel libre mais n'avait pas encore choisi de licence, ce dont nous avons discuté. Il est pour moi essentiel que GenGraph soit muni d'un gauche d'auteur, de sorte qu'il puisse toujours être présenté comme l'objet du public et non comme celui d'organisations à but lucratif. Si nous avions choisi une licence de type permissive (c'est-à-dire sans gauche d'auteur) pour GenGraph, ma motivation au-delà du stage en aurait été considérablement amoindrie. Pour tout dire, le projet GenGraph est pour moi une sorte d'évidence, à tel point que je pensais vraiment qu'un tel logiciel était déjà développé depuis au moins deux décennies par de nombreuses organisations du domaine de l'informatique, mais que celles-ci avaient commis l'erreur de placer leur outil sous une licence permissive, de sorte à le présenter fondamentalement comme un accessoire pour les projets économiques, limitant ainsi son potentiel social et la popularisation de son utilisation. En fait, il se trouve que ledit logiciel en est au stade où il est développé par un seul chercheur dans son coin dans son laboratoire depuis des années, et qu'il est pourvu d'un gauche d'auteur.

GenGraph n'est pas déconnecté de l'écologie, car :

- il utilise le langage C, qui est très performant ;
- il n'a pratiquement aucune dépendance et peut être exécuté sur des ordinateurs et systèmes d'exploitation très anciens, il peut donc très facilement être intégré dans n'importe quel système d'information, avec très peu de ressources et une stabilité exemplaire ;
- il propose des algorithmes concurrents pour répondre à des objectifs semblables avec des contraintes de performance différentes, ce qui permet d'optimiser l'usage en fonction du besoin et donc d'éviter le gaspillage d'énergie.

2. Sujet

La dernière version de GenGraph au moment du début de mon stage était la v5.4, qui a donc été ma base de travail.

Bien que le code et les technologies informatiques ont été ce qui me mettait le plus à l'aise, afin de mieux profiter de l'environnement du LaBRI au cours du stage, je me suis initié à l'étude de documents de recherche, j'ai assisté à de nombreux exposés et j'ai profité des connaissances pointues de mon maitre de stage.

2.1. Développement des fonctionnalités du projet

Durant le stage, GenGraph a été pour moi une épreuve logicielle avant tout. Bien que, comme expliqué dans la section suivante, j'ai dû profondément retravailler l'organisation du projet pour qu'elle soit confortable, ma première finalité était d'augmenter la puissance de GenGraph en travaillant sur ses fonctionnalités fondamentales. J'ai accompli un travail considérable en ce sens.

2.1.1. Mise en place d'un système de pile de requêtes

Il s'agissait de revoir la syntaxe de GenGraph afin de lui permettre de gérer les graphes de façon beaucoup plus extensible, en les stockant dans une pile. Cela a constitué le cœur de mon travail sur GenGraph, qui a orienté toutes les autres tâches que j'ai accomplies concernant la structure et les fonctionnalités générales du logiciel.

2.1.1.1. Passage à une notation polonaise hybride

GenGraph a au départ été conçu pour analyser une liste d'arguments et générer *un* graphe.

Par la suite, il s'est vu garni d'options pour analyser ou transformer le graphe généré (via `-check`) et appliquer un filtre sur un groupe de graphes (via `-filter`). Pour gérer ces diverses options, l'analyse des paramètres donnés à GenGraph sur la ligne de commande (que nous appellerons *arguments* pour les distinguer des paramètres des instructions représentées par ces arguments) remplissait une structure de type `query` (*requête*), ainsi qu'une série de variables globales.

Pour chainer les opérations que GenGraph permettait d'exécuter, il y avait une technique : enregistrer le graphe produit dans un fichier, puis réexécuter GenGraph en chargeant ce fichier (graphe de type `load` ou `loadc`) et en mettant une autre série d'options. Cette logique avait notamment conduit Cyril Gavoille à programmer une option `-maincc`, dont la présence conduisait GenGraph à appeler la fonction C `system()` pour exécuter deux instances de lui-même connectées par un tube (`|`).

Le logiciel ayant démontré son intérêt, il a été décidé d'augmenter sa portée et sa puissance en lui permettant de générer et traiter directement plusieurs graphes en une seule exécution. La technique envisagée par Cyril Gavoille pour répondre à ce besoin a été que GenGraph gère une pile de requêtes, construite par lecture progressive des arguments, certains de ces arguments conduisant à la consommation des requêtes au sommet de la pile. Cela correspond à la lecture d'instructions en notation polonaise inversée, une opération sur deux graphes s'écrivant de la manière suivante :

```
gengraph graphe1 graphe2 opérateur
```

L'opérateur est ainsi placé en dernier, et les deux opérandes juste avant. L'opérateur peut ou non placer une requête sur la pile en résultat. Si c'est le cas, cela permet de combiner des opérations ; par exemple, si l'on écrit :

```
gengraph graphe1 graphe2 opérateur1 graphe3 opérateur2
```

alors le résultat de `opérateur1` laisse un graphe sur la pile, laissé en dessous de `graphe3` et qui peut donc être utilisé par `opérateur2` s'il requiert deux opérandes. Le traitement de cette syntaxe se conçoit simplement (ce qui rend GenGraph facile à comprendre à partir d'exemples), et le parenthésage est inutile (et donc non pris en charge).

Instruction lue	`graphe1`	`graphe2`	`opérateur1`	`graphe3`	`opérateur2`
Pile					
		`graphe2`		`graphe3`	
	`graphe1`	`graphe1`	`opération1`	`opération1`	`opération2`

État de la pile après la lecture (et le traitement) de chaque instruction

C'est pour la mise en place de ce système dans GenGraph que Cyril Gavoille a fait appel à moi.

La logique de la notation polonaise inversée est notamment employée par le langage Forth, un langage de programmation reconnu pour sa simplicité et sa performance mêlées d'une certaine puissance. Forth est apparu en 1970, a été normalisé par l'ISO en 1997 et a connu en 2014 une nouvelle version (Forth 2012). Il est ainsi utilisé par certaines niches de développeurs, notamment dans l'embarqué.

On aurait pu songer à utiliser Forth pour donner de la puissance à GenGraph. Cependant, si Forth repose complètement sur la notation polonaise inversée et le système de pile, la syntaxe choisie pour GenGraph est en fait hybride entre la notation polonaise inversée et la notation polonaise classique. Les instructions sont lues et traitées en notation polonaise inversée, mais sont adjointes d'éventuels paramètres donnés en notation polonaise classique (comme dans les appels de fonction dans la plupart des langages de programmation). Par exemple, si `graphe1` prend deux paramètres, `graphe2` zéro et `opérateur` un, on écrira :

```
gengraph graphe1 graphe1_1 graphe1_2 graphe2 opérateur opérateur_1
```

Forth reste pour moi un langage marquant surtout pour son côté ésotérique, l'utilisation absolue de la notation polonaise inversée limitant l'expressivité du langage pour les humains et donc son potentiel à faire émerger de nouvelles idées. Le concept apporté par Forth a été repris par de nombreux langages ésotériques (par exemple le langage 2D Befunge), donnant des langages à la fois simples, drôles et Turing-complets. Tout comme les langages ésotériques, Forth constitue une expérience informatique intéressante, en proposant une syntaxe simple et quand même puissante. Étant donné son efficacité prétendue, le Forth pourrait être un bon choix de langage intermédiaire (comme l'est parfois le C et comme l'est plus souvent LLVM IR) dans un processus de compilation.

2.1.1.2. Les instructions dans GenGraph

On notera que GenGraph connait deux types d'instructions (une division en sous-types sera présentée plus tard) : les *graphes* et les *options*. Ces dernières se reconnaissent par la présence d'un trait d'union `-` en préfixe de leur nom, et l'instruction spéciale `?` est également considérée comme une option. Les paramètres des instructions n'ont pas de format particulier : souvent, ce sont des nombres entiers ou des nombres à virgule (utilisant le point anglo-saxon comme dans la plupart des langages de programmation) ; il y a aussi le point terminateur de liste, les chemins d'accès vers des fichiers et parfois des mots-clés.

Le nom d'une option est parfois réparti dans plusieurs arguments. Notamment, l'option de deuxième niveau `-check routing` se décline en plusieurs options de troisième niveau, dont `-check routing bc` par exemple. On parle ainsi d'option de premier, deuxième ou troisième niveau, mais on peut aussi dire que `bc` est le premier paramètre fourni à `-check routing` et que

`routing` est le premier paramètre fourni à l'option de premier niveau `-check`.

L'utilisation du préfixe `-` pour les options est depuis longtemps la norme dans les arguments des programmes sous Unix. Aujourd'hui, la plupart des programmes utilisent `-` comme préfixe d'option courte (à un caractère, de sorte que `-abc` est équivalent à `-a -b -c`) et `--` comme préfixe d'option longue. GenGraph ne connait que les options longues avec préfixe `-`, ce qui est une convention également suivie par des programmes de traitement importants, entre autres les compilateurs GCC et Clang, le compilateur et l'interpréteur OCaml, PDFLaTeX, ou encore les programmes de la suite ImageMagick (ou le programme `gm` de GraphicsMagick). En dehors des arguments des programmes, le préfixe `-` est également utilisé en langage Tcl pour les paramètres facultatifs des fonctions. Le principal concurrent au préfixe `-` (notamment sous Ms-Dos) est le préfixe `/` qui, sous Unix, est employé pour les chemins d'accès absolus.

Dans GenGraph v5.4, l'analyse des arguments du programme était intégralement faite dans la fonction `main()`, qui comportait en tout près de 2 200 lignes, les deux tiers servant à l'analyse des arguments.

2.1.2. Détermination des représentations de graphes à utiliser

Cette problématique de nature technique recouvre quelque chose de central pour les utilisateurs de GenGraph (et qui a particulièrement trait à l'écologie) : les performances du logiciel.

2.1.2.1. Le problème tel qu'il était présenté aux utilisateurs

GenGraph v5.4 gère les graphes de deux manières : par matrice d'adjacence et par liste d'adjacence. Plus que des options possibles, les deux représentations avaient tendance à se compléter, c'est-à-dire qu'elles intervenaient pour des fonctionnalités différentes :

- la génération et l'écriture de graphes étaient faites par lecture paresseuse de la matrice d'adjacence ;
- le chargement, l'analyse et le traitement de graphe se faisaient avec la liste d'adjacence.

Cependant, l'option `-fast` permet la génération par lecture de la liste d'adjacence. La lecture n'est alors plus paresseuse, c'est-à-dire qu'elle commence par stocker le graphe en mémoire ; et la fonctionnalité n'est pas disponible pour tous les graphes et annule l'effet de certaines options.

Définitions succinctes : la liste d'adjacence d'un graphe associe à chaque sommet la liste des arêtes qui en partent (c'est une liste de listes). La matrice d'adjacence est un tableau à double entrée, carré, chaque case incarnant la relation entre deux sommets, contenant la valeur 1 si ces sommets sont reliés par une arête et la valeur 0 sinon.

Le passage d'un mode à l'autre était programmé : un graphe chargé avec `load` peut être utilisé pour générer une matrice d'adjacence ; à l'inverse, la génération par lecture de la matrice d'adjacence peut optionnellement provoquer la création de la liste d'adjacence.

Pour transformer un graphe, outre les options `-check`, il existait quelques options, entre autres `-not` (complémentation) et `-dele` (suppression d'arêtes). L'activation de ces options était simplement prise en compte au moment de la génération du graphe, complexifiant (un peu) le code du générateur. Il n'est pas indiqué dans l'historique à quelle version ces options sont apparues, donc elles existent probablement depuis la première version publique (v1.2).

En vue de développer un système d'opérations plus riche, une question se posait : pourrait-on conserver le système de génération paresseuse, ou serait-il préférable de l'abandonner complètement et donc de ne travailler que sur des graphes complètement chargés en mémoire ? J'ai été surpris que Cyril Gavoille compte sur moi pour trouver la réponse. J'ai finalement été capable de trouver une solution permettant de préserver les fonctionnalités existantes tout en satisfaisant l'objectif de modularité (toutefois, elle ne m'est venue qu'après le terme officiel du stage). Il s'agit des *graphes opérations*, dont le concept est simplement d'être des classes de graphes prenant des graphes en paramètres, tout comme les options `-check` peuvent prendre des graphes en paramètres.

2.1.2.2. Analyse approfondie et critique du générateur

Au cours de mon stage, je ne devais pas remettre en question le cœur du générateur de graphes. Je pense qu'à terme, cette fonctionnalité auparavant centrale devrait être éliminée et substituée par des fonctions bien séparées répondant aux mêmes besoins de façon plus modulaire. Notons que ce générateur constitue moins de 150 lignes dans GenGraph v5.4. En vue que ce système soit amélioré et remplacé, j'ai cherché à le structurer et établi qu'il prend en charge :

- trois modes de génération : par matrice, par liste et par initialisation seulement ;
- deux types de rendus : sortie dans un fichier et liste d'adjacence stockée en mémoire.

Le mode de génération « par initialisation seulement » est représenté dans GenGraph v5.4 par le graphe `loadc`, et j'ai créé l'option `-genc` qui repose sur le même système. C'est ainsi qu'il apparait que ces fonctionnalités n'ont pas vraiment de raison d'être regroupées. Notamment, le mode « initialisation seulement », qui est destiné à prendre beaucoup plus de place, notamment grâce

à la mise en place des graphes opérations, consiste simplement à sauter les étapes de génération autres que celle d'initialisation qui ne consiste qu'à appeler la fonction spécialisée pour chaque graphe avec les codes `INIT` et `GRAPH`.

Dans le code source de GenGraph, chaque classe de graphes est incarnée par une fonction à son nom, appelée « fonction d'adjacence » par Cyrille Gavoille, parce qu'à l'origine, leur rôle se limite à indiquer si deux sommets sont adjacents. Cependant, ces fonctions gèrent différentes fonctionnalités, selon le mode donné en paramètre :

- `INIT` : initialiser les données pour préparer les réponses, et fournir le nombre de sommets du graphe ;
- `ADJ` : indiquer si deux sommets sont adjacents ;
- `GRAPH` : fournir le graphe (codé par ses listes d'adjacence) ;
- `NAME` : donner le nom textuel d'un sommet ;
- `END` : libérer les données.

Je n'ai pas modifié ce système durant mon stage, puisqu'il m'aurait fallu vérifier l'intégralité des fonctions des classes de graphes. Aussi, mon stage a consisté à apporter un nouvel enrobage aux fonctionnalités existantes afin d'offrir de nouvelles fonctionnalités, et non à optimiser le système existant. Si le système de `ADJ` et `NAME` me semble très bon, je pense que les trois autres modes ont intérêt à être revus. Notamment, si je n'ai pas mis en place une option pour activer le mode « initialisation seulement » pour n'importe quel graphe (tout comme `-fast` active le mode de génération par liste), c'est parce que pour que ce soit propre, il m'aurait fallu revoir les cas `END` de chaque fonction de classe de graphe.

En fait, le mode `GRAPH` est utilisé par `-fast`, cependant ce dernier fait ensuite un parcours du graphe qui prend en charge diverses options (`-dele`, `-delv`, `-permute`...) et qui cause la reconstruction du graphe en mémoire (éventuellement transformé par les options). J'avais envisagé de proposer la possibilité de sauter cela (comme avec `loadc`), en se contentant du graphe généré par `GRAPH` (s'il existe). De toute manière, je pense qu'on finira par supprimer complètement le générateur, et qu'il faut remplacer `GRAPH` par une fonctionnalité générant les arêtes de manière paresseuse.

On peut remarquer que Cyril Gavoille avait lui-même prévu de modifier ou étendre le système, car parmi les modes déclarés, on en trouvait quatre complètement inutilisés et qui auraient demandé d'être implémentés dans toutes les fonctions d'adjacence, présentés comme suit :

```
// à finir ...
QUERY_DEG,   // résutlat dans ->d[Q->i] ?
QUERY_LIST,  // résultat dans ->L[Q->i] ?
QUERY_DOT,   // pour le dessin des arêtes
QUERY_LNAME, // pour la liste des noms des sommets
```

Seul `QUERY_DOT` avait une occurrence, dans la fonction `sgabriel()` marquée comme non terminée et n'apportant en effet pas de fonctionnalité.

J'ai été surpris par le choix d'utiliser la matrice d'adjacence comme représentation fondamentale pour produire les graphes, puisqu'elle fournit une complexité plus éloignée de la taille des données et ne permet pas (ou pas aussi facilement) de gérer les arêtes multiples. Ainsi, le générateur se retrouve à être terriblement lent pour générer des graphes qui n'ont pourtant pas beaucoup de données (même les graphes vides qui, dans GenGraph v5.4, n'ont pas de variante `-fast` !). Dans ma pratique, j'ai toujours utilisé les listes d'adjacence par défaut, et les matrices d'adjacence par occasions. Il semble que ce choix d'utiliser la matrice d'adjacence s'explique par le fait que les graphes sont généralement définis par la condition à remplir pour que deux sommets soient adjacents, et dès lors l'exploration de la matrice d'adjacence devient la manière naturelle d'exploiter cette définition. Mais j'ai pu remarquer que dans certains cas, le calcul de la prochaine arête serait en fait bien plus simple à coder.

Le mode de génération par liste est néanmoins disponible pour un certain nombre de classes, notamment une bonne partie des arbres. Aussi, dès que l'on sort de la simple génération et écriture des adjacences de graphes, au final, on tombe vite dans l'utilisation de la liste d'adjacence et donc le stockage du graphe en mémoire ; y échappent quelques options de coloration des sommets et la gestion des graphes géométriques ; mais même l'écriture de la matrice d'adjacence d'un graphe se base sur la liste d'adjacence, ce qui a un côté assez absurde et qui, j'espère, sera bientôt corrigé grâce à mon travail. J'ai été frustré que le mode de génération par liste ne soit pas disponible pour certains graphes très basiques (`path`, `cycle`, `stable` et `star`), alors je l'ai développé pour les trois classes de graphes dont ils sont dérivés : `ring`, `grid` et `rpartite`.

La description du générateur dans le manuel sous-entend qu'il y aurait un complexe avec les structures de données étant seulement linéaires en le nombre de sommets. J'ai tâché avec mon travail de respecter ce complexe, mais je pense qu'il faudrait s'en débarrasser. Je pense qu'il est très intéressant que GenGraph puisse générer et traiter en quelques secondes ou minutes des graphes massifs ; toutefois, les ordinateurs actuels permettent de stocker plusieurs dizaines d'octets pour une centaine de millions de sommets, et je pense que pour que GenGraph soit efficace dans davantage de cas, il faut profiter de cet espace et mettre de côté l'idée de prendre en charge des graphes d'ordre encore plus grand. Ainsi, pour le graphe `ring`, qui est le premier graphe documenté dans le manuel, pour chaque arête potentielle, le générateur fait un parcours de la liste des paramètres, qui pourrait bien être très grande ; et ces parcours pourraient être évités en stockant un tableau de bits ayant pour longueur l'ordre du graphe, ce que j'ai codé à un moment mais je suis revenu en arrière en voyant qu'il y a un complexe même avec les structures de données de cette complexité.

2.1.3. Prototypage d'instructions tirant parti du nouveau système

Pour bien saisir le potentiel d'un logiciel, il faut passer du temps, beaucoup de temps, à jouer avec. Je n'ai pas eu le temps de le faire. Cependant, en vue que cela puisse être fait, j'ai développé et documenté plusieurs instructions de GenGraph, qui offrent des fonctionnalités basiques pour manipuler la pile de requêtes et les instructions.

S'agissant du rendu de plusieurs graphes (au format principal du logiciel seulement) :

- Malgré l'existence de l'option `-filter` dans GenGraph v5.4 permettant de traiter des groupes de graphes, cette version ne permettait pas d'écrire dans des fichiers directement des groupes de graphes (sans se baser sur des groupes existants) : il était nécessaire d'écrire au moins les identifiants à la main. La nouvelle option `-output-group` permet maintenant de générer directement un groupe de graphes. Elle est implicitement invoquée à la fin du programme pour ce qui reste sur la pile.
- `-output-sum` permet d'écrire dans un fichier les adjacences d'un graphe composé par plusieurs graphes précédemment générés. Cela a rendu obsolète l'option `-shift` de GenGraph v5.4, qui permettait seulement de renuméroter les sommets d'un graphe.

Il y a également des options pour :

- dupliquer une requête (`-dup`) ou s'en débarrasser (`-discard`) ;
- générer le graphe d'une requête (`-gen[c]`, qui produit un graphe `load[c]`) afin de lui appliquer une nouvelle série d'options ou de mieux profiter de `-dup` ;
- affecter un identifiant à une requête (`-id`) ou oublier tous les identifiants (`-forget-ids`) ;
- rendre le graphe en cours (`-output`) ;
- charger un groupe de graphes depuis un fichier : il ne s'agit là pas d'une option mais plutôt d'un « graphe multiple » nommé `load+`.

Ensuite, j'ai bien entendu adapté voire développé des opérations sur plusieurs graphes ou créant un graphe :

- Il y a quelques graphes opérations : `-op not` et `-op not-wl` donnent le complémentaire d'un graphe, avec ou sans boucles ; `-op del-e` donne à chaque arête une chance d'être supprimée ; `-op half` vide le triangle inférieur de la matrice d'adjacence ; `-op transpose` donne le graphe transposé ; `-op accessible` supprime tous les sommets non accessibles depuis une liste de sommets donnée tandis que `-op maincc` ne garde que la plus grosse composante connexe ; `-op union-d`, `-op union` et `-op union-de` font l'union de graphes ; `-op line-graph` donne le graphe ligne d'un graphe. Il y a également un groupe de graphes généré par opération : `-op cc+` fournit ainsi toutes les composantes connexes d'un graphe sous la forme de graphes séparés.
- Certaines options de `-check` (`iso`, `sub`, `isub` et `minor`) analysent un graphe par rapport à un autre. Dans GenGraph v5.4, cet autre graphe était trouvé dans un fichier donné en paramètre.
- Certaines options de `-check` (`simplify`, `maincc`, `subdiv` et `prune`) rendent un graphe sur la sortie standard (au format principal seulement). La nouvelle option `-memop` prend maintenant en charge ces opérations, qui placent le graphe produit sur la pile au lieu de l'afficher directement. Il y a également les opérations `linegraph` et `invlinegraph` sur lesquelles j'ai travaillé, comme expliqué plus bas. Je n'ai imaginé l'option `-memop` qu'après le concept des graphes opérations, qui à mon avis devrait absorber complètement le système de l'option `-op` à terme.
- Le système de l'option `-filter` a été découpé, d'abord en une option `-prop` permettant de définir une propriété à étudier, et en une option `-test` permettant de définir un test à appliquer. Les tests peuvent être sur des propriétés, et ils peuvent également être combinés (ou inversés, comme auparavant avec `-filter not`), en utilisant la notation polonaise inversée comme pour les graphes. Comme indiqué dans la documentation de ces options, elles servent aux options `-print-prop`, `-print-test`, `-sort`, `-filter`, `-extract` et `-extract-all`. Ce découpage rend l'application plus puissante, et a aussi l'intérêt de structurer son manuel.

Enfin, il y a quelques accessoires :

- Les options `-n-times` et `-while` permettent de répéter des instructions en utilisant un petit système d'étiquettes. L'option `-while` utilise bien entendu le système de l'option `-test`. L'option `-quit` permet d'arrêter brutalement le programme, soit pour se débarrasser des graphes encore présents sur la pile, soit pour aider au débogage ou à l'étude d'une longue liste d'instructions.
- Les options `-chrono` et `-chrono-reset` permettent de gérer un petit chronomètre, indépendant de celui de l'option `-check info`. L'option `-print` permet d'afficher un texte quelconque, éventuellement formaté comme le texte du manuel (voir ci-dessous).

Je n'ai conçu et implémenté la plupart de ces instructions qu'au mois de juillet (donc après le terme du stage).

2.1.4. Amélioration du système d'aide

Pour que GenGraph devienne le projet accessible et pédagogique que je souhaite, il était essentiel de beaucoup travailler sur son système d'aide. Au départ, cela m'a été imposé par la structuration du dépôt (traitée section suivante) à laquelle j'ai procédé, car le manuel ne pourrait plus être lu et traité comme avant. Néanmoins, il était souhaitable que ce système d'aide soit plus puissant,

et en particulier, mieux décoré. Cyril Gavoille avait quelques requêtes à ce sujet, et j'ai pris la liberté d'apporter une série d'enrichissements supplémentaires, ce que mon maitre de stage a très bien accueilli.

2.1.4.1. Les changements à apporter

Le système d'aide est basé sur un manuel écrit en texte brut avec un certain formatage, notamment l'indentation et le marquage de certaines sections avec `....` (quatre points). D'une part, il est parcouru et affiché par GenGraph dans le terminal lorsque la commande est invoquée sans paramètre ou avec une option parmi `-help`, `?`, `-list` et `-version`, ou encore en cas de paramètre manquant. D'autre part, un script permet d'en générer une version au format HTML. On distingue ainsi l'aide intégrée au programme et au terminal (appelée simplement « aide intégrée » en général pour faire court) et l'aide HTML.

Dans l'aide intégrée, la fonctionnalité la plus intéressante est celle permettant d'obtenir de l'aide sur une section en particulier. Elle est invoquée lorsque `?` se situe après une instruction sans ses paramètres, mais également quand les paramètres sont manquants.

Les fonctionnalités désirées par M. Gavoille étaient :

- l'ajout de possibilités de formatage au sein des paragraphes, à minima pour baliser les mots du manuel correspondant à du code (c'est-à-dire les instructions, essentiellement) ;
- l'implémentation en C de l'aide intégrée, pour se débarrasser de l'invocation de programmes externes (tels que `sed`, `awk` et `grep`) ;
- la prise en charge de formules écrites en LaTeX ;
- l'ajout de couleurs dans l'aide intégrée.

Pour ce dernier point, M. Gavoille m'a fait savoir que cela se fait très facilement avec des codes d'échappement ANSI très largement supportés, offrant une gamme de 16 couleurs pour les terminaux en mode texte.

Je n'ai pas traité la question des formules LaTeX, car pour cette fonctionnalité, j'ai eu deux complexes :

- il n'existe visiblement rien pour convertir le LaTeX en une représentation convenant aux terminaux en mode texte, j'aurais dû écrire la conversion moi-même et il m'aurait fallu au moins plusieurs jours pour couvrir un nombre intéressant de fonctionnalités (et bien que je ne trouve pas cela inintéressant, j'avais bien d'autres choses à faire) ;
- pour le HTML, les principales solutions sont des scripts en JavaScript convertissant le LaTeX à chaque ouverture du document, ce qui est un peu dommage pour les performances (il vaudrait mieux faire la conversion en MathML une fois pour toutes, lors de la génération du fichier), et les autres solutions n'étaient pas faciles à utiliser et ajoutaient des dépendances.

Cependant, pour le HTML, je sais qu'il suffirait en fait de quelques lignes pour inclure un script et l'invoquer pour que les formules LaTeX fonctionnent correctement (j'ai vu la chose quand j'avais accès au code source de l'ancien site de France-IOI). Bien que je ne sois pas gaga de tout ce qui est interfaces limitées au texte brut, ce genre d'interfaces tiennent aujourd'hui une place prépondérante dans le monde de l'informatique, et la technologie GenGraph n'y échappe pas, c'est pourquoi j'ai cherché à donner la meilleure forme possible à l'aide intégrée, tout en améliorant la fonctionnalité de l'aide HTML.

Ainsi, je suis allé beaucoup plus loin que les demandes de mon professeur, en développant :

- un sommaire pliable pour l'aide HTML, et des identifiants à chaque titre de section (les sections de l'historique ne sont pas présentes dans le sommaire mais peuvent néanmoins être liées directement) ;
- un sommaire qui suit automatiquement le défilement, sur le côté de l'aide HTML ;
- un thème sombre pour l'aide HTML ;
- du formatage pour les liens, notamment les ancres au sein du manuel, qui dans la version HTML permettent de sauter rapidement à la section de l'instruction que l'on a sous les yeux ;
- la prise en charge de marques de formatage multiples (gras, italique, soulignement, barré, gros, avertissement, listes ordonnées) ;
- la création d'un style spécial pour les instructions non options (c-à-d les graphes) et leurs équivalents entre parenthèses ;
- le marquage des arguments correspondant à l'instruction en cours, pour déterminer la section à afficher en cas de paramètre manquant ou mal formaté ;
- l'ajout de l'option `-html` pour demander l'aide HTML au lieu de l'aide intégrée ;
- l'extension de l'option `-list` à l'affichage des options, et l'ajout des options `-list-options` et `-list-graphs` (cette dernière jouant le rôle que jouait `-list` dans GenGraph v5.4) ;
- l'adaptation de la longueur des lignes de l'aide intégrée à la largeur du terminal ;
- la complétion des noms d'instructions et des mots-clés (pour l'interpréteur Bash interactif) ;
- la prise en charge de motifs pour décrire les instructions dans le manuel (pour les identifiants en HTML ainsi que la complétion).

J'ai traité ce dernier point car le manuel décrivait certaines instructions avec deux syntaxes pour les motifs :

- `[abc]` pour indiquer un ou plusieurs paramètres optionnels ou un morceau de mot optionnel ;
- `mot1|mot2` pour indiquer que l'un ou l'autre mot peut être utilisé ;
- et même parfois `[abc|def]` pour indiquer que l'un ou l'autre morceau de mot peut être utilisé.

2.1.4.2. La reconception du manuel

Nous nous sommes bien entendu demandé si nous ne pouvions pas utiliser simplement un format existant, tel que celui des pages de manuel de la commande `man` ou le Markdown. Il s'est avéré que ces outils existants ne nous apporteraient pas grand-chose par rapport à nos besoins : aucun n'était fait à la fois pour être très épuré, proposer des couleurs dans le terminal, des ancrages, un sommaire auto-généré et la compilation des formules en LaTeX. Aussi, la petite syntaxe personnalisée et très simple utilisée par M. Gavoille, basée sur l'indentation, m'est apparue comme sympathique, et j'ai trouvé bien pratique de travailler avec un outil sur lequel j'avais toute l'autorité. Notons que les formats existants n'étaient pas aussi bien installés au moment où le manuel de GenGraph a vu le jour (Markdown est apparu en 2004, reStructuredText en 2002), et ceux visant les terminaux textuels (roff et Texinfo) sont anciens mais leur syntaxe est lourde.

GenGraph v5.4 utilisait deux codes bien séparés pour produire les deux formats d'aide :

- l'aide intégrée était affichée à l'aide de la fonction C `system()` exécutant une ligne de commande construite par le programme, invoquant les programmes `sed`, `awk`, `grep`, `sort` et `more`, avec quelques arguments ou petits scripts ;
- l'aide HTML était produite à l'aide d'un script Awk (/ɔːk/).

L'aide intégrée ne procédait à aucune transformation du texte, hormis le remplacement des quatre points `....` et des marques `!!!` par des espaces. Elle utilisait ces `....` pour trouver les sections à afficher. L'aide HTML reconnaissait les titres de section (balises `<h1>` à `<h4>` en HTML), des blocs préformatés (soit avec `!!!` soit avec `Ex:`, utilisant la balise `<pre>`) et des listes non ordonnées (avec `•` : balises `<ul>` et `<li>` en HTML). Le contenu de la balise `<head>` (essentiellement des règles de style) était tiré d'un fichier à part.

Pour ma part, je ne pense pas que les invocations de programmes causaient des problèmes de dépendances : ce sont des programmes Unix très standards, faisant pour l'essentiel partie desdits « coreutils » et qu'on retrouve par exemple dans la liste des utilitaires dont l'invocation est cautionnée par GNU dans les Makefile. `more` ne fait pas partie de cette liste ; cependant, j'ai fait le choix d'utiliser à la place (en option seulement) `less`, qui est nettement moins standard mais qui est le programme couramment utilisé par la commande `man` entre autres (et qui a eu tendance à devenir au moins aussi classique que Vim et `vi`, dont il a repris les conventions). Sous Android, on retrouve les cinq commandes (mais pas `less`) dans `/system/bin`, trois d'entre elles étant fournies par l'utilitaire `toybox`. Aussi, GenGraph avait quelques autres dépendances : à POSIX (notamment par l'utilisation de la fonction `random()`), et il utilisait quelques extensions GNU pour le C, qui sont, il me semble, moins standards que ces cinq programmes. Néanmoins, j'ai tout de suite été motivé à recoder le système, parce que ces invocations de programmes externes ne rendaient pas le code particulièrement propre, et m'apparaissaient comme inutilement lourdes (l'exécution d'un programme n'est pas une opération légère pour le système d'exploitation) et aux effets moins maitrisables qu'un module en C pur.

Le stage a été l'occasion pour moi de découvrir Awk, dont j'avais entendu parler mais sur lequel je ne m'étais jamais penché. Je n'ai pas été séduit par l'outil. D'une part, il est bien expliqué dans la documentation de GNU Awk qu'Awk est plutôt fait pour traiter des données structurées, et interpréter des petits scripts jetables et facilement reproductibles. Cela ne correspondait guère au traitement voulu pour l'aide qui est faite de texte formaté pour l'essentiel peu organisé, et pour laquelle nous voulions un système quand même assez élaboré et qui tienne sur la durée. D'autre part, j'ai trouvé la sémantique du langage un peu pauvre : pas de sémantique pour les booléens, et l'acte automatique des expressions rationnelles sur la ligne en cours (sauf en argument de certaines fonctions spéciales), ce qui empêche d'enregistrer ces expressions dans des variables (sauf avec une extension GNU, non disponible avec la version d'Awk de l'ordinateur de M. Gavoille par exemple) à moins de les écrire comme des chaines de caractères échappées (ce qui conduit par exemple à devoir quadrupler les contre-obliques `\\\\` pour en avoir une seule `\`). De plus, tout comme `sed`, Awk ne reconnait que les expressions rationnelles POSIX, que je n'ai pas particulièrement envie d'apprendre parce qu'elles sont moins agréables que les PCRE qui sont devenues plus populaires.

Finalement, j'ai tout codé en C, en faisant reposer les deux formats de l'aide sur un noyau commun de 400 lignes, chaque format ayant son module de 350 lignes, et il y en a encore 400 autres pour les diverses fonctions de l'aide intégrée (y compris l'ouverture de l'aide HTML, le lancement de `less` et quelques routines pour la lecture des arguments du programme) et 150 pour la complétion. Je suis globalement satisfait du résultat, et je me suis retrouvé un peu limité dans mes compétences pour faire ce travail ; ainsi, le système devrait à mon avis être revu au bout d'un moment afin que son code soit un peu plus condensé, mais en attendant il garantit de fournir un service bien plus complet qu'avant et qui promet d'être satisfaisant longtemps. Bien que l'écriture d'une telle bibliothèque en C puisse paraitre fastidieuse, le C offre un contrôle important, c'est une technologie largement connue et le code demeure lisible dès que l'on sait quelles sont ses idées fondamentales. À présent, la génération de l'aide HTML est faite en un clin d'œil à chaque exécution de la commande `make` pour la construction du projet.

Pressé de voir le manuel correctement formaté, je l'ai relu et formaté entièrement. Je passe déjà beaucoup de temps à l'explorer — par curiosité, mais aussi pour vérifier que mes améliorations fonctionnent avec suffisamment de cas —, et sa lecture m'est ainsi nettement plus agréable. L'historique des versions n'était pas du tout formaté, et je l'ai formaté avec soin, car je trouve l'histoire des logiciels très intéressante, et je trouve que c'est le meilleur endroit pour parler de ce qu'on fait. J'ai aussi décrit toutes les nou-

veautés que j'ai apportées en nommant la version v6.0a.

2.1.4.3. Les rapports d'erreur

Il convient aussi de prendre en compte la façon dont les erreurs (dans les arguments donnés au programme) étaient signalées par le logiciel. En effet, les commandes passées à GenGraph étant destinées à être complexifiées, il devient nécessaire de faciliter leur diagnostic. GenGraph traite toute erreur comme une erreur fatale : une fois qu'il en a détecté une, il affiche un message d'erreur, puis s'interrompt sur-le-champ sans plus faire aucun traitement. Sans revoir le système en profondeur, j'y ai apporté quelques améliorations en vue d'augmenter sa convivialité :

- afin d'être plus génériques et d'apporter plus d'informations contextuelles, les messages d'erreur prennent des paramètres, transmis tels quels à la fonction `vsprintf()` ;
- les messages d'erreur peuvent être garnis de couleurs et de formatage, en utilisant la même syntaxe que dans le manuel ;
- le format des arguments de la ligne de commande qui doivent être numériques est vérifié : GenGraph s'assure que l'argument contient bien un nombre et rien d'autre (GenGraph v5.4 ne signalait jamais d'erreur et utilisait 0 comme valeur à défaut de nombre en début d'argument).

Jusqu'alors, la trivialité de ce système ne m'est pas apparue comme un obstacle significatif à l'utilisation de GenGraph. Tout du moins, j'ai pu améliorer considérablement la puissance de GenGraph sans repenser son système de rapport d'erreur. Pour l'instant, il semble assez naturel que toute erreur soit perçue comme fatale et interrompe complètement l'exécution, tout comme une erreur détectée lors de la compilation d'un programme fait obstacle à la construction de celui-ci — on peut toutefois critiquer le fait que GenGraph ne détecte l'erreur que lorsqu'il arrive à l'instruction qui la contient. Néanmoins, si GenGraph devait être utilisé couramment pour des scripts plus complexes, à mon avis, il sera bon qu'un système de gestion d'erreurs plus puissant et plus robuste soit conçu et développé.

GenGraph n'est actuellement pas capable de déterminer si une liste d'arguments est correcte sans (commencer à) exécuter les instructions qu'elle décrit. Une telle fonctionnalité se montrerait utile au moins pour la complétion, pour gérer proprement des sauts en avant dans les instructions (les options `-n-times` et `-while` que j'ai développées ne permettent que les sauts en arrière), ou encore pour corriger une bizarrerie due au système d'aide intégrée ; elle se fait désirer mais demeure loin d'être indispensable. La fonctionnalité pourrait éventuellement être offerte par l'analyse du manuel ; cependant, certains paramètres ont des contraintes complexes pour lesquelles il serait compliqué de créer une syntaxe, et donc l'analyse du manuel ne permettra sans doute jamais de prendre en compte ces contraintes et détecter leur violation. Il faudra donc développer une capacité à lire les instructions tout en inhibant leurs effets. Notons aussi que certaines erreurs résident dans les propriétés du graphe à traiter, qui peut très bien être complètement différent à chaque fois qu'une même instruction est réexécutée.

2.2. Mise en place d'un dépôt du projet

Au début du stage, je me suis retrouvé seul (M. Gavoille étant en déplacement) avec un unique fichier source de 26 000 lignes, constituant l'intégralité du projet. Même le manuel en faisait partie, dans un très long commentaire en fin de fichier. M. Gavoille m'a par la suite transmis (à ma demande) trois scripts annexes, notamment celui permettant de générer la version HTML du manuel.

Le projet étant déjà assez mature et n'ayant pas connu d'autre contributeur que son créateur, je me suis senti très libre de réaliser un certain nombre de tâches visant à favoriser la participation future d'autres contributeurs, en m'appuyant sur les conventions qui m'ont été enseignées et que j'ai pu observer dans de nombreux projets de logiciels libres existants. Ce travail était en effet nécessaire à la réalisation de mon désir de faire de GenGraph un logiciel libre que les personnes motivées peuvent facilement explorer et étendre.

La première de ces tâches a consisté (bien entendu) en le découpage de l'unique fichier source en de nombreux fichiers sources. Ce faisant, il fallait que le projet demeure tout aussi facile à construire, et donc qu'un script automatise ce processus (voire le rende encore plus aisé). Cyril Gavoille ne souhaitant pas être brusqué dans les méthodes de développement qu'il avait depuis des années, il fallait aussi que la nouvelle structure du projet ne l'empêche pas de poursuivre le développement avec ses méthodes, quand bien même il serait nécessaire de développer des outils pour obtenir cette rétrocompatibilité. Par ailleurs, dans son manuel, GenGraph v5.4 était déjà décrit comme une commande (plutôt qu'un simple programme), mais il n'existait aucune procédure officielle pour l'installer, c'est pourquoi j'ai mis en place une telle procédure dans le script de construction.

Les idées que j'ai mises en application pour la réalisation de ces tâches peuvent être distinguées selon leur origine :

- la grosse partie sont des pratiques que l'on m'a enseignées, que j'ai observées dans des projets existants et que j'avais déjà mises en application ;
- une autre partie sont plus subjectives : ce sont des pratiques canoniques que j'ai adoptées après avoir expérimenté des techniques concurrentes, dans des projets et exercices personnels ;
- une petite partie ont consisté à compléter les compétences des deux points précédents en faisant des recherches que je voulais faire mais que je n'avais pas encore faites à défaut de projet dans lequel les mettre en pratique.

2.2.1. Aménagement du dépôt

Sous-tâches :

- création d'un dépôt du code source sur une forge logicielle ;
- découpage de l'unique fichier source en de nombreux fichiers (dits modules) ;
- présentation du projet via un fichier lisez-moi ;
- choix d'une licence de logiciel libre.

Comme forge logicielle, j'ai simplement utilisé l'instance de GitLab hébergée par l'université (https://gitub.u-bordeaux.fr/). Le dépôt est ainsi géré à l'aide de Git, le logiciel de gestion de versions le plus populaire en développement logiciel, créé en 2005 par Linus Torvalds. Le dépôt du code source est ainsi accessible en lecture à tout le monde. Cependant, cette instance ne permet pas à n'importe qui de s'inscrire pour rapporter des bogues ou proposer des améliorations. Mais elle permet de voir le dépôt de façon structurée et d'explorer l'historique des versions (depuis le code de GenGraph v5.4), que j'ai moi-même utilisé plusieurs fois au cours du stage.

Bien que Cyril Gavoille n'y voie pas vraiment d'intérêt, j'ai tenu à découper le fichier source d'origine, de sorte à avoir un dépôt respectant les standards logiciels courants. Il reste vrai qu'en un sens, cela complexifie le projet. Mais le développement en C est bien conçu pour pouvoir être fait plus facilement avec plusieurs fichiers. Ainsi, le découpage apporte au moins quatre avantages :

- la structure du projet est visible en un coup d'œil, et les commandes Unix standards (telles que `wc` et surtout `grep` que j'utilise en permanence) permettent de l'étudier plus en profondeur ;
- les interdépendances entre les différents composants du projet sont plus simples à gérer (notamment via les `#include`), le graphe du projet est ainsi plus facile à saisir, et on peut plus facilement envisager de découper le projet en plusieurs sous-projets ;
- la recompilation est généralement plus rapide, en particulier lorsqu'on fait de nombreux essais-erreurs sur des fichiers `.c` ;
- les noms globaux d'un fichier (notamment : fonctions `static`, énumérations, structures, macros) peuvent être masqués aux autres, ce qui permet parfois d'éviter d'utiliser des noms compliqués sans risquer de rentrer en conflit avec d'autres éléments du projet.

Le dépôt est ainsi structuré comme suit :

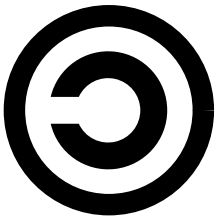
- à la racine se trouvent des fichiers présentant le projet et permettant de le construire et de l'installer ;
- le dossier `src/` contient les fichiers sources du programme principal : un *module* est ainsi généralement composé d'un fichier `.h` (entête) et d'un fichier `.c`, mais certains fichiers `.h` sont communs à plusieurs modules ;
- le dossier `tools/` contient quelques accessoires venant au soutien des scripts de construction principaux ou du logiciel GenGraph.

Les fichiers présentant le projet sont : les lisez-moi et la licence en français et en anglais, et le manuel de GenGraph. Pour le lisez-moi, je me suis au départ basé sur le commentaire tout en haut du fichier `gengraph.c` dont je suis parti, puis j'ai ajouté un certain nombre d'informations. Le manuel contient à sa fin un historique des versions du logiciel, que j'ai isolé dans le standard fichier `CHANGELOG.txt`. Dans la page principale du dépôt, on peut remarquer que GitLab offre des boutons-liens visant spécialement les fichiers `README.md`, `LICENSE.md` et `CHANGELOG.txt` du dépôt (sans que j'aie eu à faire quoi que ce soit) :

Les boutons-liens générés automatiquement par GitLab sur la page principale du projet

Pour la licence, j'avais l'impression de faire face à trois choix (qui correspondent aux trois variantes de la licence CeCILL) :

- licence permissive, sans gauche d'auteur, avec crédit (type BSD, MIT ou CeCILL-B) ;
- licence avec gauche d'auteur faible (type LGPL, MPL ou CeCILL-C) ;
- licence avec gauche d'auteur fort (type GPL ou CeCILL).



Le symbole *copyleft*

À vrai dire, dans ma pratique, tant qu'à ne pas mettre de gauche d'auteur, je ne mets pas non plus de crédit (licence type WTFPL, Unlicense ou Creative Commons Zero), mais je ne m'en sers que pour de très petits projets. Je ne pensais au départ qu'aux licences américaines, mais nous avons finalement adopté une licence CeCILL : la licence CeCILL-C. CeCILL est un choix confortable, car c'est une licence émise entre autres par le CNRS et l'INRIA qui sont des partenaires réguliers du LaBRI. Le terme « gauche d'auteur » est une traduction de l'anglais *copyleft*. Bien qu'il repose légalement sur le droit d'auteur (*copyright*), il en fait un usage opposé à l'usage classique, qui garantit des droits aux utilisateurs des produits plus qu'à leurs auteurs, notamment la recopie à souhait du produit. Le terme *copyleft* est ainsi adapté aussi bien par son opposition au *copyright* que par sa signification « copie laissée ».

2.2.2. Programmation de la construction et de l'installation

Sous-tâches :

- mise en place d'un système pour automatiser la construction du produit, aussi bien pour les simples utilisateurs ne faisant que télécharger et exécuter le logiciel, que pour le développement continu et le débogage ;
- création d'une procédure d'installation officielle et automatisée ;
- création d'un outil permettant de continuer à travailler avec un unique fichier source `gengraph.c`.

Il s'agissait essentiellement de développer un fichier `Makefile` pour GNU Make. Nous avons pu constater sur l'ordinateur de Cyril Gavoille que sous macOS, la version de GNU Make installée par défaut est une vieille version datant de 2006 (GNU Make 3.81, la version actuelle étant 4.4.1). Pour éviter les complications éventuelles pour les personnes voulant installer le produit, j'ai donc tâché d'assurer une compatibilité satisfaisante avec cette vieille version de GNU Make. Un petit programme, reposant sur le fichier source `dependencies.c`, teste l'environnement de compilation pour définir quelques drapeaux à utiliser (notamment pour l'éventuelle dépendance libbsd, requise sur les distributions Linux un peu anciennes).

Les produits de la compilation sont tous placés dans un dossier `_build/`. En particulier, les fichiers intéressants (l'exécutable et le manuel) sont placés à la racine, et une fois construit, le programme peut donc être lancé en tapant `_build/gengraph`. Ce n'est pas aussi confortable qu'avoir l'exécutable directement dans le répertoire racine, c'est pourquoi j'ai fini par développer un petit script `gengraph.sh` qui permet d'exécuter le programme ou éventuellement la version de débogage.

Une autre tâche que j'ai voulu réaliser a été la création d'une procédure officielle d'installation. Elle est exécutée par la cible `install` du Makefile et donc par la commande `make install`, avec des paramètres éventuels. Je souhaitais en particulier savoir comment le programme (situé dans un dossier `bin/`) faisait pour savoir où trouver ses fichiers de données (placés dans `share/gengraph/`). Il se trouve qu'il n'y a pas de meilleure solution que mettre le chemin directement dans le programme. J'ai codé un minimum d'interactivité dans la procédure, permettant au moins de voir comment configurer l'installation. Il y a aussi une cible `uninstall`, ainsi qu'un fichier d'installation pour le gestionnaire de paquets Pacman de la distribution Arch Linux.

Enfin, puisque Cyril Gavoille ne semblait pas à l'aise à l'idée de devoir travailler avec plusieurs fichiers, j'ai réalisé quelques développements pour que le développement de GenGraph puisse se faire en modifiant un seul fichier `gengraph.c`. Quelques lignes de code shell concatènent les fichiers sources, un programme en C redistribue les modifications dans les fichiers sources (il ne met à jour que les fichiers modifiés pour qu'il n'y ait pas besoin de les recompiler), et un autre programme en C fait la traduction des numéros de ligne dans les messages du compilateur.

2.2.3. Définition des conventions de contribution

Sous-tâches :

- délimitation des objectifs de GenGraph ;
- détermination de conventions pour l'uniformité des contributions ;
- détermination des standards technologiques à respecter ;
- définition de la procédure de livraison.

Sur le plan juridique, les logiciels libres sont garantis étudiables, adaptables et redistribuables à souhait. Cependant, le bon fonctionnement d'un projet de logiciel libre ne se limite pas à la liberté de faire des produits dérivés, mais repose aussi sur la capacité de chacun à contribuer au projet principal afin de l'aider à satisfaire ses objectifs au mieux. La version de GenGraph sur laquelle j'ai travaillé a vu ses objectifs être élargis au traitement généraliste et automatisé de graphes. Pour garantir un bel avenir à GenGraph, il est nécessaire d'une part de définir plus précisément ces objectifs, et d'autre part de déterminer les moyens validés par le projet pour les mettre en œuvre.

Afin de traiter ces questions, j'ai analysé le projet tel qu'il était, j'ai écouté attentivement les préoccupations et les projections de mon maitre de stage et je lui ai aussi posé diverses questions qui n'avaient rien à voir. Au final, le projet que je rends ne contient pas de texte résultant de ces sous-tâches que j'ai effectuées, mais j'ai produit une certaine réflexion et des règles que j'ai utilisées dans mon travail sur GenGraph et que je vais pouvoir présenter dans ce rapport.

De nombreux projets de logiciels libres contiennent au moins un fichier `CONTRIBUTING` à la racine de leur dépôt de code source, présentant les règles à suivre pour contribuer au projet principal. Ces règles, aussi peu spécifiques que possibles, facilitent la relecture du projet par ses développeurs et leur évitent aussi de se poser constamment des questions sur les paradigmes et conventions à suivre dans le développement du projet.

De plus, l'utilisabilité d'un logiciel est dépendante de sa compatibilité avec le système d'exploitation de l'utilisateur. Pour GenGraph, le choix est d'avoir un minimum de dépendances, quitte à recoder un certain nombre de fonctions courantes qui ne viendraient pas seules si on les tirait d'une bibliothèque. Aussi, le C est un bon vieux langage standardisé, cependant il connait certaines variations : après C99 est apparu C11, et C23 apparaitra bientôt ; les compilateurs C proposent bien souvent leurs propres extensions à ces normes ; enfin, la bibliothèque standard du C permet la lecture et l'écriture de fichiers ainsi que de texte dans le terminal, mais une jolie application ne saurait s'en contenter, et il y a pour cela des bibliothèques pour le C qui sont standardisées et qui peuvent ainsi être utilisées de la même manière dans de nombreux systèmes d'exploitation. Il est possible d'écrire du code dépendant du système d'exploitation, mais c'est bien plus simple si on peut s'en passer (et cela déborderait vrai-

ment de ce qu'on souhaitait faire avec GenGraph). Il était pour moi fondamental de déterminer très rapidement quels standards le code de GenGraph devait exploiter, et ainsi mes réponses sur le sujet sont complètes, et différentes en fonction des sous-projets (présentés plus loin dans ce rapport) qui composent GenGraph.

Je ne peux pas en dire autant pour les conventions de style à suivre dans le code. Cyril Gavoille a suivi dans son développement un certain nombre de conventions, mais d'après moi il vaudrait mieux en modifier certaines, et en définir davantage pour une meilleure cohérence du code et des descriptions plus synthétiques des algorithmes et des données. Remettre en question les conventions existantes n'était pas une priorité dans le stage, cependant j'en ai installé de nouvelles, telles que la diminution du nombre de variables globales (en les concentrant dans quelques structures globales), l'augmentation de la sémantique des données par l'exploitation du système de types du langage C (davantage de types `struct` et `union`, utilisation du type `bool` et des mots `true` et `false`, utilisation d'opérateurs plus naturels avec les types manipulés), et la définition de macros puissantes pour l'organisation des données écrites en dur dans le code. Notons qu'une grosse différence entre le langage C et son prédécesseur le B est la mise en place du système de types, et que le système de types du C est aujourd'hui connu pour être faible.

La procédure de livraison encadre la sortie des nouvelles versions du logiciel. GenGraph devenant important et ambitieux, il devient à mon avis adéquat de faire une petite liste de choses à prendre en compte lors de la publication d'une nouvelle version, pour le succès de celle-ci et sa cohérence avec le travail précédemment réalisé.

2.3. Utilisation de données scientifiques

Comme je le souhaitais, j'ai découvert pendant le stage la recherche telle qu'elle est faite au LaBRI, en me lançant dans l'étude de quelques travaux de recherche que j'ai mis en application dans GenGraph.

2.3.1. Les graphes chenilles

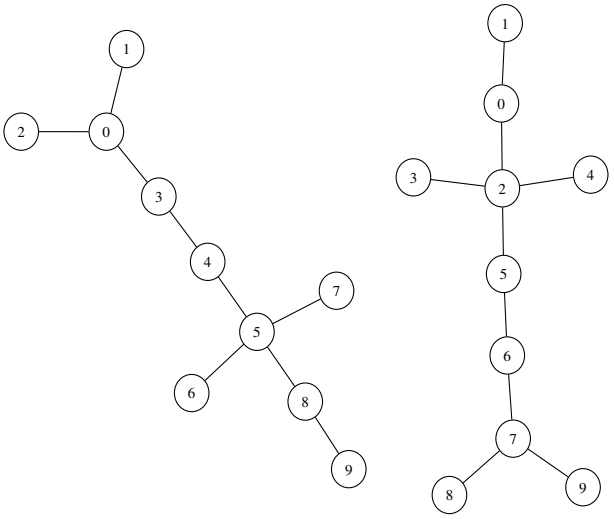
Cyril Gavoille a dans ses cahiers numériques un long fichier décrivant des idées d'algorithmes pour la prise en charge de nouvelles classes de graphes (remarquables ou connues) dans GenGraph. C'est une sorte de brouillon en texte brut dans lequel il donne un nom de graphe avec ses paramètres et décrit l'algorithme (ou donne des pistes pour le trouver), de façon semblable à ce qu'on trouve dans le manuel dans la section des GRAPHES. Bien que les classes de graphes soient correctement définies ailleurs, il est parfois difficile de trouver une définition qui peut être directement traduite en programme ; et surtout, la recherche de l'algorithme permettant la génération aléatoire uniforme (dans le respect de la définition et des paramètres) représente potentiellement beaucoup de travail.

Il semble que Cyril Gavoille n'avait rien sous la main de facile à comprendre et à programmer, à part une classe de graphes : les chenilles. Il en avait fait une première implémentation dans GenGraph mais elle n'était pas uniforme, et plus récemment il avait conçu un algorithme simple pour produire une génération aléatoire uniforme. Il semble que je me suis débarrassé du court texte présentant cet algorithme, c'est pourquoi je ne peux pas l'intégrer à ce rapport. M. Gavoille m'a expliqué l'algorithme en détail et j'ai été capable d'en faire une première implémentation et de le documenter en moins de deux heures.

Les graphes chenilles sont des arbres très simples : un chemin et des sommets pendants. La génération aléatoire uniforme consiste à donner à chaque graphe possible autant de chance d'être généré, en considérant les graphes isomorphes comme étant identiques (c'est-à-dire sans considérer d'ordonnancement des sommets ou des arêtes, alors que l'informatique d'aujourd'hui ne sait appréhender les choses que dans un certain ordre). En anglais et dans GenGraph, cette classe de graphes est appelée `caterpillar`. Elle prend un paramètre : le nombre de sommets. (Notons que si on veut choisir le nombre de sommets internes n_1 et le nombre de sommets pendants n_2, on peut utiliser `path` n_1 `-star` n_2 plutôt que `caterpillar`.)

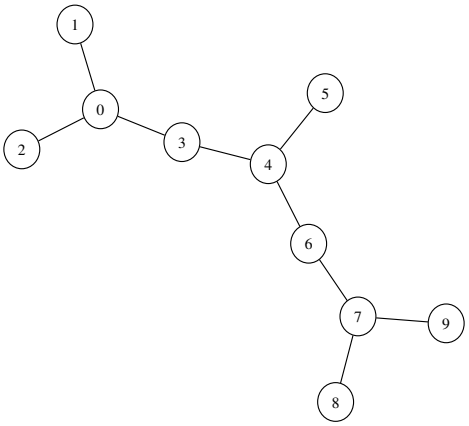
Pour obtenir des valeurs aléatoires, on utilise bien entendu une fonction existante (`random()` de la norme POSIX). Notre technique pour générer des chenilles uniformément est d'associer à chaque sommet un bit, indiquant si c'est un sommet interne ou bien un sommet pendant ; et dans tous les cas, le sommet est connecté au sommet interne précédent. Il n'y a plusieurs possibilités de chenille qu'à partir de 4 sommets, et les bits sont ainsi toujours fixés pour trois sommets : le premier sommet est toujours interne, et le deuxième et le dernier sont toujours pendants. Pour les autres sommets, les bits sont aléatoires.

Pour la génération uniforme, cette idée connait une limite, que nous avons dû gérer : certaines sous-listes de bits aléatoires correspondent au même graphe que d'autres listes de bits (par isomorphisme). Ce cas se retrouve précisément lorsque l'on retourne une sous-liste. Par exemple, la liste `1001110010` (la sous-liste est soulignée, les autres bits sont fixes) génère le même graphe que `1010011100` :



Deux graphes chenilles isomorphes, obtenus par les listes de bits `1001110010` et `1010011100`

Ainsi, la majorité des graphes chenilles ont deux manières d'être générés ; mais ceux n'ayant pas d'équivalent par retournement de sous-liste, parce que cette sous-liste est symétrique par rapport à son milieu et donc reste la même si on la retourne, n'ont qu'une seule manière d'être générés. C'est par exemple le cas du graphe suivant, obtenu avec la liste `1001101100` :



Le graphe chenille obtenu par la liste de bits `1001101100`, qui n'a pas d'équivalent isomorphe avec l'algorithme décrit

Pour éviter que ces graphes soient générés moins que les autres, la solution utilisée consiste à rejeter les sous-listes lexicographiquement supérieures à leur équivalent retourné. Ce rejet conduit à essayer une nouvelle sous-liste.

Après une première implémentation fonctionnelle, M. Gavoille m'a demandé de développer une optimisation : ne pas générer plus de bits aléatoires qu'il n'en faut pour savoir que la sous-liste doit être rejetée. Cette petite optimisation a significativement complexifié le code et m'a conduit à concevoir différents cas à gérer (j'ai dû me relire plusieurs fois pour tous bien les gérer) ; ainsi, je doute un peu de sa pertinence, mais la fonctionnalité me parait mure.

2.3.2. Les graphes lignes et l'opération inverse

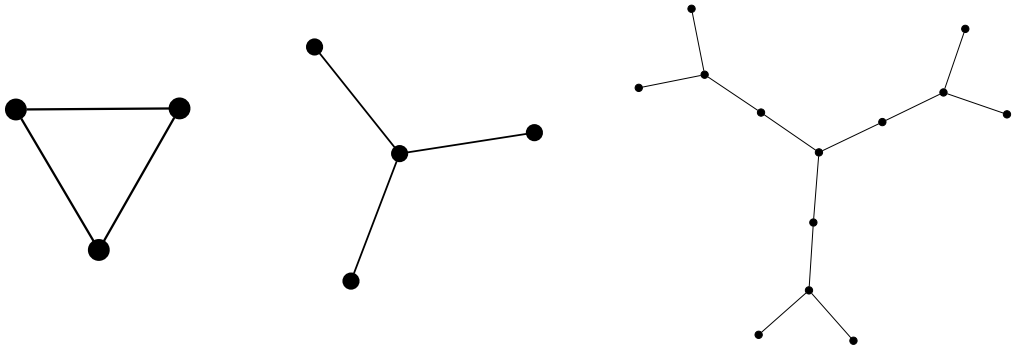
2.3.2.1. Présentation du travail fait

J'ai consacré (ou essayé de consacrer) les trois dernières semaines du stage au sujet indiqué dans son intitulé : les graphes lignes. Ils sont aussi appelés graphes adjoints, mais plus souvent par leur nom anglophone « *line graphs* ». Un graphe ligne est le graphe des arêtes d'un autre graphe, c'est-à-dire que les sommets du graphe ligne sont les arêtes du graphe racine (*root graph*). Deux arêtes sont considérées comme adjacentes si elles ont une extrémité en commun. « Ligne » est un nom parfois utilisé pour « arête », tout comme « nœud » peut être utilisé à la place de « sommet ».

GenGraph v5.4 disposait déjà d'un générateur de graphe ligne : le graphe `line-graph`. Cependant, l'algorithme pour générer le graphe ligne d'un graphe n'était pas encore codé ; il l'est maintenant, avec l'option `-memop linegraph` et le graphe opération `-op line-graph`. Notons que la fonctionnalité `line-graph` existante ne génère pas toujours des graphes lignes, parce qu'elle ne prend en charge que le mode de génération par matrice et fusionne donc les arêtes parallèles en une seule arête.

Au LaBRI, j'avais l'impression que tout le monde sauf moi savait ce qu'est un graphe ligne ; par contre, l'opération inverse (consistant à retrouver le graphe racine d'un graphe ligne) résonnait souvent moins chez les personnes autour de moi (cela a eu tendance à me surprendre, parce que pour ma part, je ne conçois jamais une opération sans penser un tant soit peu à son ou ses in-

verses). Cette opération inverse suppose l'existence d'un graphe racine ; elle n'est pas applicable à tous les graphes : notamment, les graphes lignes sont sans griffe induite (c'est-à-dire qu'ils ne comportent jamais quatre sommets dont l'un est connecté aux trois autres sans que ces autres aient la moindre connexion entre eux), sinon il n'y a pas de graphe racine. L'opération inverse est ainsi associée à celle de caractérisation des graphes lignes. Il y a un seul graphe ligne qui a deux graphes racines : le triangle. Voir la documentation d'`-op invlinegraph` pour les détails. Notons que les sommets isolés dans le graphe racine ne sont pas du tout considérés puisque ces sommets n'ont pas d'arête et n'ont donc aucune incarnation dans le graphe ligne (en revanche, une composante connexe simple à deux sommets se traduit par un sommet isolé dans le graphe ligne).



Le graphe triangle (à gauche) est le graphe ligne de deux graphes : lui-même et le graphe griffe (au milieu). La banane sur la droite, obtenue dans GenGraph avec `banana 3 3`, contient quatre griffes induites et n'est donc pas un graphe ligne. Les graphes chenilles présentés plus haut contiennent eux aussi des griffes et n'ont donc pas non plus de graphe racine.

Cyril Gavoille m'a donc chargé d'étudier au moins un document décrivant un algorithme pour l'inversion de graphe ligne. Plusieurs algorithmes existent. Le plus connu est apparemment celui de Rossopoulos (c'est aussi le moins efficace d'après les auteurs d'ILIGRA) ; il y a également celui de Lehot, celui de Degiorgi & Simon, et celui que nous avons choisi pour mon stage est ILIGRA qui signifie « *Inverse LIne GRaph Algorithm* » et qui se prétend plus simple et plus efficace que les autres. Une autre option était MARINLINGA (« *MAtrix Relabeling INverse LINe Graph Algorithm* »), qui a une complexité quadratique en le nombre de sommets du graphe à inverser ; je ne l'ai pas du tout étudié et je me suis contenté d'ILIGRA.

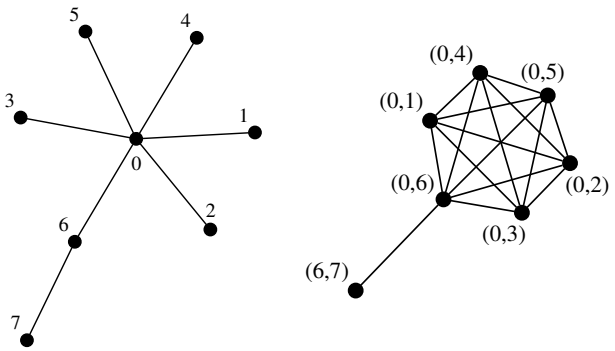
ILIGRA ne prend en charge que les graphes simples non orientés. Il sait plus rapidement trouver le graphe racine que déterminer s'il y en a vraiment un : la création du graphe racine a une complexité proportionnelle au nombre de sommets (qui deviennent les arêtes dans le graphe racine), mais pour déterminer si le graphe est un graphe ligne, il faut ajouter un parcours des arêtes. Ainsi, l'algorithme peut être bien plus rapide si on demande un graphe racine sans vérifier que le graphe en entrée est un graphe ligne, c'est pourquoi cette vérification est en option (activée par défaut).

Il existe une définition des graphes lignes pour les graphes orientés, que mon maitre de stage m'a donnée dans un second temps : chaque arc devient un nœud, et les nouveaux arcs vont des arcs entrants vers les arcs sortants. (Cela implique visiblement une perte d'informations beaucoup plus grande : impossible de savoir si deux sommets du graphe ligne n'ayant tous les deux que des arcs entrants, que des arcs sortants ou aucun arc, avaient une extrémité en commun dans le graphe racine.) J'ai donc généralisé mon implémentation de `-memop linegraph` aux graphes orientés. Alors que l'implémentation pour les graphes non orientés était relativement triviale, la généralisation aux graphes orientés, sans que je m'y attende, m'a donné un peu plus de fil à retordre, mais elle semble bien fonctionner. Plus tard, j'ai développé le graphe opération `-op line-graph` qui permet de générer un graphe ligne par matrice d'adjacence (le mode de génération par défaut).

En revanche, comme indiqué dans le manuel, je ne suis pas allé au bout de mon implémentation d'ILIGRA. J'ai cependant bien compris et implémenté la base de l'algorithme, qui permet de traiter un grand nombre de cas correctement. Pendant longtemps, j'ai cru que mon implémentation avait un gros problème ; je me suis finalement lancé dans son débogage au mois d'aout et en fait j'avais simplement oublié d'initialiser un tableau avec des zéros. Au bout du compte, il reste à traiter les cas particuliers de l'algorithme, auxquels je ne me suis jusqu'alors pas du tout attaqué.

2.3.2.2. Résumé de l'algorithme d'inversion (ILIGRA)

Le principal constat à la base d'ILIGRA est que les groupes d'arêtes incidentes à un même sommet dans le graphe racine forment une clique dans le graphe ligne. Ainsi, il suffit plus ou moins de parcourir les cliques pour retrouver les sommets du graphe racine auxquels connecter les arêtes qui, dans le graphe ligne, sont les sommets. La boucle principale d'ILIGRA est ainsi presque triviale, et la difficulté (sans rapport avec la complexité algorithmique) réside en fait dans la partie qui consiste à trouver une entrée dans ce « réseau de cliques ».

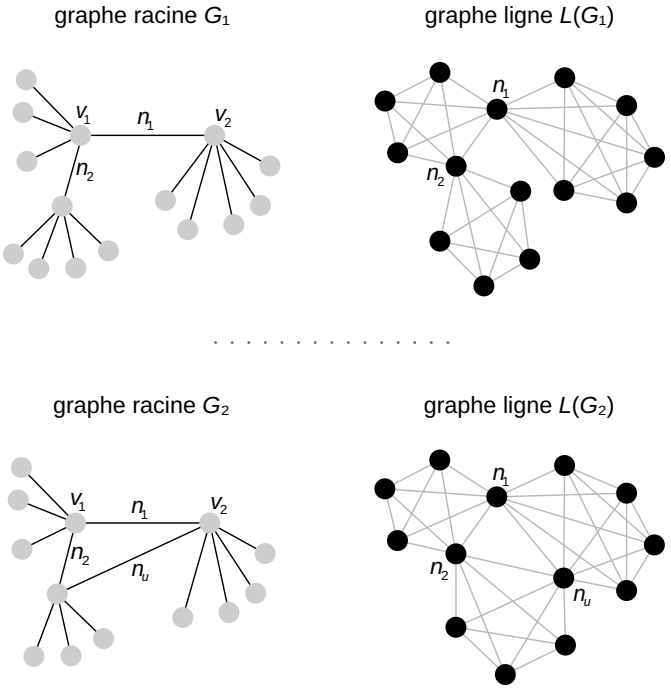


Une étoile à six branches liée par une de ses branches à un sommet pendant (à gauche), et le graphe ligne correspondant (à droite). Les six arêtes de l'étoile forment une clique dans le graphe ligne, car elles sont toutes adjacentes par le sommet 0. L'arête 6–7 n'a qu'un seul voisin : l'arête 0–6, par le sommet 6.
Notons que l'étoile comporte des griffes induites (si l'on prend le sommet central et trois de ses voisins, n'importe lesquels) ; le graphe de gauche n'est donc pas un graphe ligne.

L'illustration ci-dessus peut être obtenue avec la commande suivante (`-fast` peut éventuellement être omis) :

```
gengraph star 6 load-str '0 6-7' -op union -dup -op line-graph -op union-d -fast -label -1 -output line-graphs-2
```

J'utilise le terme « nœud-ligne » pour désigner les nœuds qui sont à la fois les arêtes du graphe ligne et les sommets du graphe racine. Tout en créant des sommets dans le graphe racine, ILIGRA va chercher à déterminer quels nœuds-lignes y sont incidents. ILIGRA commence en choisissant arbitrairement deux nœuds-lignes adjacents n_1 et n_2, et pose leur extrémité commune v_1 ainsi que la seconde extrémité v_2 du premier nœud-ligne. À cette dernière, v_2, extrémité de n_1 mais pas de n_2, l'algorithme connecte tous les nœuds-lignes qui sont adjacents à n_1 mais pas à n_2. Les voisins restants de n_1 sont aussi des voisins de n_2 : c'est là qu'il y a une multiplicité de cas pour déterminer si ces voisins communs sont incidents à v_1 ou bien à v_2, en particulier si cet ensemble des voisins communs compte seulement un ou deux nœuds-lignes. En principe, les voisins communs sont connectés à v_1, mais il peut y en avoir un, un n_u, qui relie v_2 à la seconde extrémité de n_2. Les trois théorèmes du document que j'ai étudié viennent au soutien de cette partie de l'algorithme.



Un schéma (adapté) du document sur ILIGRA, qui illustre les deux premiers théorèmes du document. Ce schéma présente deux graphes de trois étoiles connectées par leur centre, et les graphes lignes correspondants. Les nœuds-lignes sont en noir. On se concentre sur ceux qui relient les centres des étoiles : n_1, n_2 et éventuellement n_u. On voit une clique pour chaque étoile : les voisins communs à n_1 et n_2, les voisins propres à n_1 et les voisins propres à n_2. Cependant, il peut y avoir un n_u qui est également un voisin commun de n_1 et n_2, formant avec eux un triangle dans le graphe racine, et qui dans le graphe ligne se trouve à la fois dans les deux cliques des voisins propres au lieu de se trouver dans celle des voisins communs. Ce sont les deux seuls cas possibles si n_1 et n_2 ont au moins trois voisins en commun.

À partir de là, les deux extrémités de n_1 sont déterminées, et tous ses voisins ont une (et une seule) extrémité déterminée qui est leur extrémité commune avec n_1 ; n_1 est ainsi complètement traité. La boucle principale n'a plus qu'à sélectionner un nœud-ligne n

dont une seule extrémité est déterminée, lui créer son autre extrémité et y connecter tous les voisins de n dont au moins une extrémité n'était pas encore déterminée. Répéter cela jusqu'à ce qu'il n'y ait plus un nœud-ligne dont une seule extrémité est déterminée suffit à traiter toute une composante connexe. Voilà les grandes lignes d'ILIGRA. Pour traiter les autres composantes connexes, il suffit de recommencer l'algorithme en commençant avec des nœuds non encore traités.

Tous les ensembles de voisins successivement appréhendés (sauf l'éventuel n_u dans les voisins communs à n_1 et n_2) forment forcément des cliques dans le graphe ligne. Le gain si l'on est déjà sûr que l'on a bien un graphe ligne vient du fait que, ces cliques étant imposées, on peut se contenter de supposer qu'elles existent pour explorer le graphe ligne et construire le graphe racine. À contrario, pour s'assurer que le graphe est un graphe ligne, il faut (et il suffit de) relever toutes ces cliques, ce qui implique de parcourir toutes les arêtes.

On peut remarquer que l'article sur MARINLINGA présente dans son introduction quatre définitions des graphes lignes, dont la première est : « Un graphe est un graphe ligne si et seulement s'il est possible de trouver une collection de cliques dans le graphe, partitionnant toutes les arêtes, telle que tout sommet appartienne à au plus deux cliques et qu'il n'y ait pas deux cliques qui partagent plus d'un sommet. » Il ne peut y avoir qu'une seule telle partition, sauf pour le graphe triangle qui en a deux.

2.4. Semaine typique

J'ai passé l'essentiel de mon temps dans une salle informatique au CREMI, dans une salle dédiée aux stagiaires du LaBRI cet été. Les semaines ont été ponctuées par trois principaux évènements :

- les rendez-vous avec mon maitre de stage (environ un par semaine), tous d'au moins deux heures, durant lesquels nous avons surtout parlé d'algorithmique et de théorie des graphes, mais également de la reconception de GenGraph et des outils à utiliser avec le projet ;
- les réunions du groupe de travail de l'équipe GO le vendredi midi, consistant en un exposé d'une heure d'un membre de l'équipe GO (parfois un stagiaire ou un doctorant) ou d'un invité venu de l'étranger ;
- des exposés d'une demi-heure le mardi et le jeudi après le déjeuner, d'un chercheur ou d'un doctorant (travaillant au LaBRI le plus souvent).

3. Déroulement

3.1. Découpage en modules

Le découpage en modules est la première tâche à laquelle je me suis attaqué, et c'est aussi presque la dernière. En effet, au départ, j'ai passé deux semaines à parcourir le code et à essayer de regrouper les fonctions et les déclarations dans des fichiers de façon cohérente. Ensuite, à force de mieux connaitre le projet, j'ai eu plusieurs fois des impulsions de revoir le nom de certains modules, et à découper encore plus.

À présent, il n'y a plus que trois fichiers de plus de 1 000 lignes : `src/gengraph/graphs/base.c` qui contient des fonctions diverses de classes de graphes (dites « graphes de base »), `src/gengraph/algos/conflict.c` et `src/gengraph/algos/routing_schemes.c`. Les deux premiers font à peine plus de 1 000 lignes ; le troisième en fait un peu plus de 3 000, c'est une fonctionnalité complexe que je n'ai pas du tout étudiée. Il y a en tout plus de cent fichiers, et les deux tiers (principalement des entêtes `.h`) contiennent moins de 200 lignes. Il y a en tout plus de 26 500 lignes de code ; 32 000 si on ajoute le manuel ; 33 500 si on rajoute les outils et les scripts de construction.

La dernière sous-section présente l'environnement de développement (rudimentaire) que j'ai utilisé. La section ultérieure Relecture des fonctionnalités existantes explique mes choix concernant la modification des interfaces de programmation (fonctions, macros, types) préexistantes.

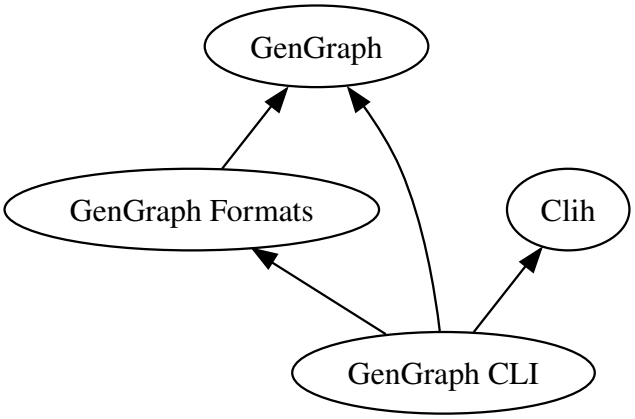
3.1.1. Vue d'ensemble du résultat

3.1.1.1. Les quatre sous-projets

La dernière tâche que j'ai réalisée, en une journée à la fin du mois d'aout, a été le découpage des sources en quatre grands dossiers correspondant à quatre sous-projets de GenGraph. En effet, à force de développer GenGraph, j'ai ressenti que l'on pouvait en dégager quatre projets et envisager un graphe de dépendances sans cycle entre eux :

- GenGraph : une bibliothèque C pour générer et traiter des graphes ;
- GenGraph Formats : une bibliothèque C dépendant de GenGraph, gérant des formats de fichiers de graphes ;
- GenGraph CLI (le programme GenGraph) : un programme en ligne de commande pour générer et traiter des graphes, qui dépend des trois autres sous-projets ;

- Clih : une bibliothèque C pour adjoindre à un programme en ligne de commande des outils d'aide puissants.



Graphe des dépendances voulu entre les quatre sous-projets, généré à l'aide de Graphviz (voir la source)

« CLI » est un acronyme anglophone signifiant *« command-line interface »*, « interface en ligne de commande ». On peut l'opposer à « GUI » qui signifie « interface graphique utilisateur ». Le H en plus dans le nom « Clih » est l'initiale de *« help »* (« aide »). J'ai plutôt tendance à prononcer « Clih » à la française mais en soufflant (légèrement) le H comme en anglais, donc au final plutôt comme on prononcerait « kleeh » en anglais.

La bibliothèque GenGraph est la grosse partie : plus de 20 000 lignes de code. Le programme GenGraph contient plus de 3 000 lignes, et les deux autres projets en contiennent plus de 1 000. J'ai très peu travaillé sur GenGraph Formats, et j'ai mis plus de temps à la concevoir comme un sous-projet indépendant. J'ai finalement décidé de le faire, car je pense qu'il y aurait beaucoup de perspectives pour développer cette bibliothèque, et notamment en lui ajoutant des dépendances (pas forcément obligatoires) qui iraient bien avec l'outil GenGraph CLI mais dont il vaudrait mieux isoler la bibliothèque GenGraph centrale. Clih, au bout du compte, peut se voir complètement indépendamment de GenGraph, et il est probable qu'au bout d'un moment, je souhaiterai que ce soit un projet séparé.

Ce découpage en sous-projets donne plus de valeur au travail : je me retrouve à avoir travaillé sur quatre projets plutôt que sur un seul. De plus, c'était à mon avis nécessaire pour assurer sa cohérence au projet global, qui fait qu'il est mieux structuré et qu'il est plus facile et plus relaxant de travailler dessus. Cependant, si Clih est déjà indépendant des autres sous-projets, il y a encore un peu de travail à faire pour que les autres relations de dépendances deviennent vraiment asymétriques.

3.1.1.2. Les dossiers

Avant de faire ce découpage, j'ai longtemps travaillé avec 60 fichiers tous dans le même dossier, et juste un sous-dossier avec 40 fichiers (de taille généralement plus modeste) pour les fonctions des classes de graphes.

La structure du projet (image animée) jusqu'à fin aout, semblable à ce qu'elle était après les deux premières semaines

Dernièrement, j'ai réparti tout cela dans quatre dossiers pour les quatre sous-projets. Afin d'équilibrer, j'ai même rajouté des sous-dossiers dans le dossier de la bibliothèque GenGraph.

La structure actuelle du projet (image animée), consécutive au découpage en quatre sous-projets

Le dépôt actuel peut être parcouru sur la page du dépôt ou éventuellement à ce lien.

En principe, je ferais beaucoup de dossiers ; cependant, avec les logiciels actuels, cela devient vite lourd, car on ne voit toujours qu'un seul dossier à la fois. Il serait souhaitable que les gestionnaires de fichiers cherchent à afficher une certaine quantité minimale de fichiers en affichant automatiquement le contenu des sous-dossiers ; mais le mieux qu'ils savent faire pour l'instant, c'est sauter les dossiers ne contenant qu'un seul sous-dossier ! Par conséquent, je pense qu'il vaut mieux se limiter à des dossiers contenant au minimum une dizaine de fichiers, et utiliser des préfixes (suivis d'un tiret bas `_`) pour signifier que certains fichiers vont ensemble. Toutefois, un sous-projet peut éventuellement contenir moins de dix fichiers ; c'est le cas de Clih.

Le projet a donc trois niveaux de dossiers ; par exemple l'un des plus profonds est : `src/gengraph/graphs`. Le dossier `_build` qui contient les produits de la compilation est plus profond, mais il n'est jamais nécessaire de l'explorer, car tous les éléments intéressants qui s'y trouvent sont à la racine (à part l'exécutable avec les symboles pour le débogage).

On peut voir facilement la longueur de tous les fichiers en composant une commande avec `wc` et `find` (ou les `globstar` de Bash). Le nombre de lignes se trouve dans la première colonne (il est suivi par le nombre de mots et d'octets), et il y a un total à la fin. Résultat :

```
$ wc `find src/gengraph -type f | sort`

    99    345   2507 src/gengraph/algos/algos.h
  1003   6360  42656 src/gengraph/algos/conflict.c
   639   3076  22751 src/gengraph/algos/exploration.c
   112    745   5268 src/gengraph/algos/exploration.h
   464   2105  13579 src/gengraph/algos/ops.c
   291   1486  11913 src/gengraph/algos/ops_linegraph.c
   221    962   6616 src/gengraph/algos/path.c
    30    113    730 src/gengraph/algos/path.h
   424   2012  14260 src/gengraph/algos/predicates.c
   562   2347  17309 src/gengraph/algos/props.c
   146    665   4096 src/gengraph/algos/props_clique.c
  3220  15532 118361 src/gengraph/algos/routing_schemes.c
   140    548   3837 src/gengraph/algos/routing_schemes.h
   226    796   6740 src/gengraph/algos/tree.c
    56    307   2126 src/gengraph/algos/tree.h
   173    947   6173 src/gengraph/common.h
   290   1323   7703 src/gengraph/conv.c
    86    413   2550 src/gengraph/conv.h
    75    277   2346 src/gengraph/debug.h
   147    711   6746 src/gengraph/errors.c
    94    148   2460 src/gengraph/errors.h
   148    494   3617 src/gengraph/graph.c
    66    390   2443 src/gengraph/graph.h
   138    358   3558 src/gengraph/graphs/base_alkane.c
    85    311   2379 src/gengraph/graphs/base_behrend.c
  1254   2976  27358 src/gengraph/graphs/base.c
   304   1168   9360 src/gengraph/graphs/base_calls_tree.c
    76    344   2541 src/gengraph/graphs/base_caterpillar.c
    87    251   1953 src/gengraph/graphs/base_debruijn.c
   556   3387  24885 src/gengraph/graphs/base_drg.c
   206    769   6073 src/gengraph/graphs/base_dyck.c
   143    531   3920 src/gengraph/graphs/base_flip.c
   439   2884  19465 src/gengraph/graphs/base_gabriel.c
   202    976   6973 src/gengraph/graphs/base_ggosset.c
    92    245   1852 src/gengraph/graphs/base_grid.c
   106    340   2480 src/gengraph/graphs/base_hexagon.c
   111    498   3677 src/gengraph/graphs/base_hyperbolic.c
```

En général, je tape juste `wc src/**`, ce qui requiert d'activer l'option `globstar` de Bash (avec `shopt`).

3.1.2. Les différents modules

Fort heureusement, le fichier `gengraph.c` originel contenait une ébauche de structure : le découpage en fonctions était déjà relativement bon (il y avait certaines exceptions, en particulier la fonction `main()` qui faisait plus de 2 000 lignes), les fonctions étaient parfois déjà regroupées par thème, et il y avait quelques grandes sections marquées par un commentaire de ce genre :

gengraph-v5.4/gengraph.c (extrait)

```
19571 /**********************************
19572
19573            FONCTIONS DE TESTS
19574               POUR -FILTER
19575
19576 **********************************/
```

Cependant, un certain nombre étaient aussi abstraits que :

gengraph-v5.4/gengraph.c (extrait)

gengraph-v5.4/gengraph.c (extrait)

```
828 /**********************************
829
830         ROUTINES EN VRAC
831
832 **********************************/
```

Le fichier contenait ainsi deux blocs de « routines en vrac », un bloc de « routines pour les fonctions adj() » et un bloc de « routines pour les fonctions d'adjacence ». Les fonctions des classes de graphes sont au cœur de GenGraph depuis sa création et vont l'être encore davantage grâce au système des graphes opérations, on peut donc espérer qu'il y ait en effet beaucoup de routines à leur support. Et puis au bout du compte, certaines de ces routines étaient finalement utilisées pour certains algorithmes de l'option `-check`.

J'ai donc dû accomplir un travail conséquent, en m'intéressant à une grosse partie des fonctions du projet, pour parvenir à une structure de modules équilibrée. Une conséquence positive de cet exercice que je me suis imposé est qu'à son terme, je connaissais bien les différents éléments du projet, et j'étais donc armé pour envisager sa reconception en profondeur. Je pourrais ainsi éviter à la fois de casser beaucoup de choses et de ne faire que complexifier le projet au lieu de simplement le repenser.

J'utilise la casse_serpent (minuscules et tirets bas) pour les noms de fichiers sources C, mais j'ai tendance à utiliser la casse ChatMotMajuscule pour désigner les modules correspondants, comme les modules OCaml et les classes Java. Le module RoutingSchemes est ainsi à associer aux fichiers `routing_schemes.c` et `routing_schemes.h`, par exemple.

3.1.2.1. Dans la bibliothèque GenGraph

Il y a trois grands dossiers :

- `graphs/` : les fonctions des classes de graphes, dites « fonctions d'adjacence » ;
- `algos/` : des fonctions de traitement de graphes, dits algorithmes ;
- `util/` : des fonctions de support ne portant pas sur la théorie des graphes.

Pour organiser les fonctions des classes de graphes, j'ai voulu utiliser la même division que dans le manuel. Arrivé au bout, j'ai cependant fini par voir que lesdits « graphes composés » n'ont aucune fonction à leur nom et sont complètement traités au même endroit que la lecture des arguments du programme. En effet, ces graphes composés sont construits à l'aide des « graphes de base », en utilisant des valeurs de paramètres particulières. Au bout du compte, ils ne font pas partie de la bibliothèque GenGraph, mais sont une fonctionnalité apportée par GenGraph CLI.

Je me suis ainsi retrouvé avec 31 fichiers dont le nom commence par `base_` et 2 commençant par `directed_`. J'ai laissé tel quel. Il y a en plus le fichier `base.c` contenant des graphes de base divers, dont le code est trop court pour justifier la création d'un fichier ; le fichier `fixed.c` qui contient des définitions de graphes fixes (qui utilisent une syntaxe spécifique) ; et j'ai depuis rajouté les graphes opérations : 2 fichiers préfixés par `op_` et le fichier `op.c`. Note : les graphes chargés depuis un fichier, notamment `load`, font partie de GenGraph Formats.

Le fichier `base.c` demeure l'un des plus gros. Pour essayer de réduire sa taille, je me suis fixé la convention suivante : un fichier peut être créé à partir de 70 lignes de code (sans compter les directives d'inclusion), ce qui fait un peu plus que la hauteur des écrans HD que j'ai l'habitude d'utiliser. Certaines fonctions un peu longues peuvent être regroupées, parce qu'elles utilisent une même fonction ou sont associées à un même concept de la théorie des graphes. Mais beaucoup de fonctions sont vraiment petites, ce qui fait que `base.c` contient encore 1 200 lignes.

Dans le dossier `graphs/`, il n'y a que deux fichiers entêtes : `common.h` qui est inclus par tous les fichiers du dossiers, et `graphs.h` qui déclare toutes les fonctions de classes de graphe (et qui n'est inclus qu'à quelques rares endroits). `graphs.h` inclut `common.h` et non le contraire, pour permettre de déclarer une nouvelle classe de graphes sans avoir à recompiler toutes les autres.

Pour les algorithmes, il y a quelques gros modules que je n'ai pas du tout creusés. Chacun a son entête `.h` correspondant. En voici la liste :

- Exploration : l'exploration de graphe ;
- Path : la construction de chemin ;
- Conflict : la construction de graphe des conflits ;
- RoutingSchemes : les schémas de routage ;
- Tree : les forêts enracinées.

Le module Tree n'est (comme je l'ai indiqué à l'intérieur) « pas encore utilisé ». Il regroupe quelques structures et fonctions de taille importante, qui étaient présentes dans le code d'origine, sans être utilisées. Au début de la principale fonction de ce module, il est d'ailleurs indiqué : « non utilisée, non testée ».

Le reste des algorithmes est constitué de fonctions diverses. Pour toutes celles-là, je n'ai utilisé qu'un entête commun `algos.h`

(comme le nom du dossier). Je les ai cependant divisées en modules (c'est-à-dire en plusieurs fichiers `.c`) comme suit :

- Ops : des opérations modifiant un graphe, voire créant un tout nouveau graphe à partir d'un ou plusieurs autres ;
- Props : le calcul de propriétés de graphes (généralement numériques), c'est-à-dire la construction de valeurs à partir de la donnée d'un graphe (ou éventuellement de plusieurs) ;
- Predicates : des tests sur les graphes, c'est-à-dire des propriétés booléennes, les fonctions renvoyant parfois la justification de la réponse booléenne.

Il y a deux sous-modules, contenant des fonctions que j'ai développées : PropsClique, pour résoudre différents problèmes sur les cliques ; et OpsLinegraph, pour exécuter des opérations relatives aux graphes lignes.

Enfin, les utilitaires sont divisés en cinq modules :

- Misc : divers, avec entre autres manipulation de bits, mots de Dyck et génération aléatoire ;
- Math : fonctions plus purement mathématiques ;
- Sort : fonctions pour les tris et la recherche d'élément ;
- Sets : fonctions sur les ensembles ;
- Query : fonctions sur le type `query` utilisé par les classes de graphes.

Pendant longtemps, j'avais laissé les fameuses « routines pour les fonctions adj() » dans un module GraphAdjacency dont le fichier source était long (1 300 lignes). Je me suis finalement mis à le découper (juste avant la division en sous-projets) en le mêlant au module Util existant, ce qui a donné ces cinq modules du dossier `util/`. L'un des gros problèmes avec ce module GraphAdjacency fourre-tout et au nom inadapté était que son entête servait d'entête général pour les fonctions des classes de graphes. Il incluait de nombreux fichiers, et était inclus dans tous les fichiers de fonctions de classes de graphes, ainsi que dans quelques autres qui utilisaient une fonction de ce module. La conséquence de cela était qu'il y avait de très nombreux fichiers d'entête dont la modification conduisait à devoir recompiler tous ces fichiers.

À présent, il y a davantage de directives `#include`, mais certains entêtes (dont notamment `util.h`, `algos.h`) dont la modification conduisait à devoir recompiler quelque chose comme 60 fichiers (soit presque tout le projet) ont laissé place à des entêtes qui peuvent être modifiés sans avoir à recompiler plus d'une quinzaine de fichiers. L'entête du module Query est cependant inclus par les fichiers des fonctions de classes de graphes ; sa modification conduit donc à devoir recompiler un peu plus de 40 fichiers.

Il y a enfin les fichiers en racine de la bibliothèque. Je les ai tous profondément revus, sauf ceux du module XY. Il y en a trois qui sont inclus par tous les fichiers du projet (même dans GenGraph Formats et GenGraph CLI, du moins pour l'instant) :

- `common.h` : entêtes principaux (c'est-à-dire une dizaine de fichiers de la bibliothèque standard communément utilisés dans le projet), et macros et petites fonctions diverses utilisées partout dans le projet, principalement pour l'allocation mémoire ;
- `errors.h` : codes des erreurs et fonction (et macros) pour afficher un message d'erreur ;
- `debug.h` : macros pour aider au débogage.

C'est le fichier `common.h` qui inclut les deux autres. Errors est en fait un module : il y a un fichier `errors.c`. `errors.h` et `debug.h` sont écrits pour ne pas dépendre de `common.h` (sinon, la bonne pratique serait de vérifier que l'autre fichier a été inclus avant et c'est moins joli).

Il y a ensuite deux modules centraux dans la bibliothèque, dont dépendent la plupart des modules de GenGraph, GenGraph Formats et GenGraph CLI :

- Graph : contient le type structure représentant un graphe par ses listes chainées stockées en mémoire, quelques types structures accessoires, et quelques fonctions permettant de créer/cloner/supprimer un graphe ;
- Query : contient la structure d'une requête de génération de graphe, et des fonctions pour en créer/cloner/supprimer.

Query dépend de Graph. Il y a un sous-module : QueryGen, qui lance la génération de graphe. Ces deux modules sont pour moi complètement à revoir pour faire de GenGraph une bonne bibliothèque pour traiter des graphes ; c'est la prochaine grande tâche à réaliser selon moi. Tous les modules de GenGraph dépendent de Graph, sauf Util/Misc, Util/Math et Util/Sets.

Enfin, il y a deux autres modules :

- XY : fonctions relatives au positionnement du contenu des graphes ;
- Conv : fonctions de conversion entre représentations de graphes, notamment une représentation calquée sur le format simple de GenGraph, et gestion de groupes de graphes.

3.1.2.2. Dans les autres sous-projets

GenGraph Formats contient les modules suivants :

- In : lecture et obtention de graphe ;
- Out, OutDot, OutHtml et OutText : écriture de graphe ;

- Color : adjonction de couleurs aux graphes écrits.

Il y a également les fichiers suivants :

- `load.c` qui exploite le module In afin de mettre à disposition les classes de graphes `load` et `load-str`, et les groupes de graphes `load+` et `load-str+` ;
- `format.h` : le fichier entête principal de la bibliothèque ;
- `internal.h` : un fichier entête interne à la bibliothèque.

Dans le dossier `tools/`, on trouve par ailleurs le fichier `dot2gg.awk` qui traduit le langage DOT vers le format simple de GenGraph (Cyril Gavoille m'avait fourni ce fichier sous le nom `dot2gen.awk`).

GenGraph CLI contient les modules suivants :

- Main : le point d'entrée du programme ;
- QueryProp : calcul de propriété sur une requête ;
- QueryTest : calcul de prédicat sur une requête ;
- Selectors : application de filtre sur des nombres entiers (utilisé seulement par QueryTest et pour la lecture de graphe) ;
- Util : fonctions de support pour l'interface ;
- Help : préparation et affichage de l'aide ;
- Args : analyse des instructions données au programme.

À l'heure actuelle, le parcours de la liste des arguments du programme est fait dans Args qui appelle les fonctions de ses sous-modules pour analyser les instructions. Args a des sous-modules pour les grosses catégories d'instructions : ArgsOutput, ArgsMisc, ArgsCheck, ArgsTest, ArgsXY et surtout ArgsGraphs. Ce dernier est un peu long : presque 700 lignes ; toutefois, en l'état, je ne pense pas le découper davantage.

Le module Config est un cas particulier : son mini fichier source `config.c` est remplacé par un autre lors de l'installation du programme, afin de contenir les chemins des dossiers utilisés par l'exécutable. Ces chemins ne sont utilisés que dans Help. Voir Les procédures d'installation et de désinstallation.

Clih a son fichier entête principal `clih.h` (c'est le seul qu'il faut inclure pour utiliser la bibliothèque), et les modules suivants :

- Parsing : un noyau centralisant des fonctions de traitement et d'analyse utiles aux autres modules ;
- Manual : écriture du manuel pour le terminal textuel ;
- ManualHtml : écriture du manuel en HTML ;
- Accessories : écritures de parties spécifiques du manuel ;
- Completion : génération de la complétion pour un argument d'un programme utilisant Clih.

Tous les modules dépendent de Parsing, et Accessories dépend de Manual. Il y a aussi un dossier pour Clih dans le dossier `tools/`, contenant :

- `bash-completion.c` : un petit programme permettant d'invoquer la complétion de Clih ;
- `bash-completion.sh` : un script mettant en place la complétion dans l'interpréteur de commandes (généralement Bash) ;
- `generate-html-manual.c` : un petit programme pour générer le manuel en HTML ;
- `manual-template.html` : le gabarit HTML dans lequel le manuel doit être écrit.

Les trois premiers fichiers font chacun moins de 30 lignes et peuvent être conçus indépendamment de GenGraph. Le gabarit HTML quant à lui fait plutôt partie du projet GenGraph que de Clih car les projets utilisant Clih doivent pouvoir personnaliser leur gabarit, néanmoins il est très générique et n'est vraiment connecté à GenGraph que par son titre et la gestion de `-html -list-graphs`. Le générateur de l'aide HTML originel avait un fichier `gengraph.head` équivalent ; ce fichier ne m'a pas été fourni mais on peut voir qu'il est appelé par le script et j'ai trouvé son contenu en prenant la bonne partie du manuel HTML de GenGraph sur le site de Cyril Gavoille.

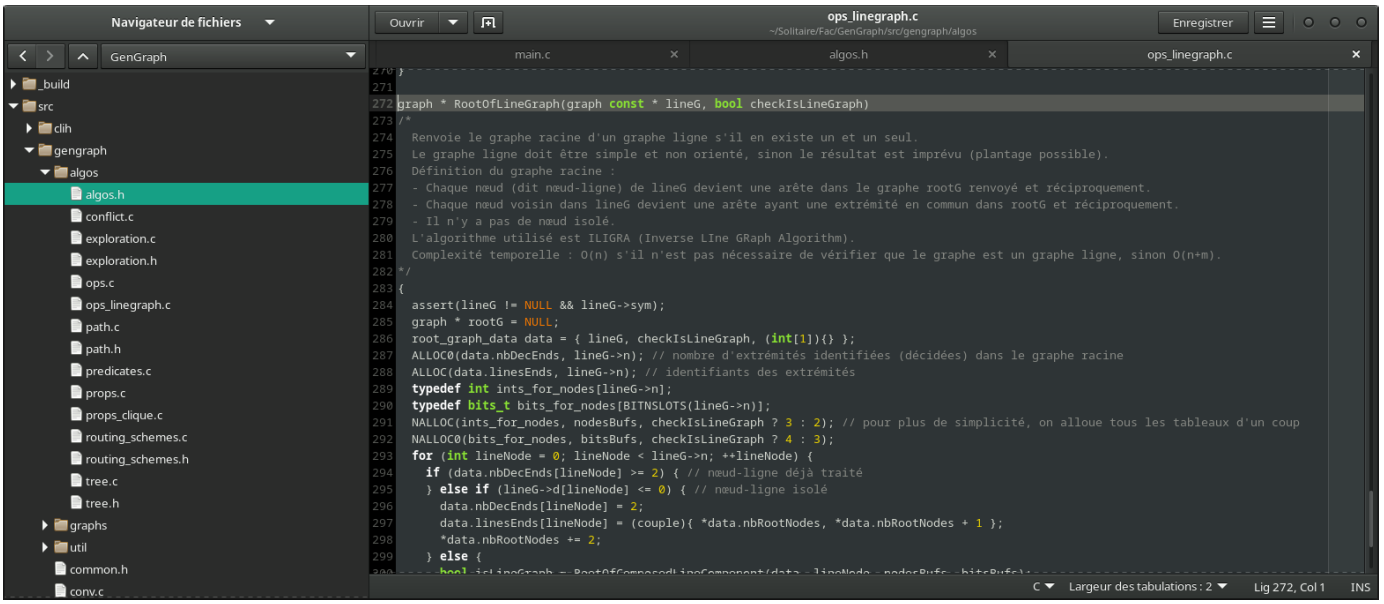
L'équivalent de `generate-html-manual.c` était à l'origine un script `do_help.awk` écrit par Cyril Gavoille. Il m'a aussi fourni un script `do_check1.awk` qui vérifiait la correspondance entre les sections du manuel et les instructions prises en charge par GenGraph. Pendant longtemps, j'ai gardé ce script (rendu complètement obsolète par mes modifications) dans `tools/` sous le nom `manual-vs-source.awk`, puis j'ai fini par me décider à le supprimer car un tel script ne correspond pas à mes méthodes de travail. En effet, j'ai tendance à constamment faire le lien entre une fonctionnalité et sa documentation, et donc je ne ressens pas le besoin d'avoir un outil automatisé pour faire de telles vérifications. Pour tout ce qui existait, j'ai vérifié la correspondance entre le code et le manuel : d'un côté pour les options, parce que je les ai toutes revues, et de l'autre pour les graphes, à l'aide de commandes telles que `grep`, `sort` et `diff`.

3.1.3. Environnement et procédures de développement

J'ai réorganisé le projet pour qu'il réponde mieux à mes logiques de développement, que je présente ici.

3.1.3.1. Logiciels utilisés

Je développe toujours avec un éditeur de code relativement basique : gedit, le vieil éditeur de code de l'environnement de bureau GNOME, utilisant la bibliothèque d'interfaces graphiques GTK et la bibliothèque GtkSourceView pour la zone d'édition. gedit est plutôt délaissé depuis plusieurs années, mais il continue de répondre à mes besoins principaux.



Capture d'écran de gedit, avec le dossier de GenGraph accessible sur la gauche, et trois onglets ouverts

Il me fournit les fonctionnalités basiques essentielles : arborescence de fichiers, onglets, saisie et sélection de texte, coloration syntaxique, surlignement des parenthèses correspondantes, préservation de l'indentation, indentation et désindentation de plusieurs lignes, édition verticale, accès direct à un numéro de ligne, recherche et remplacement de texte avec expressions rationnelles PCRE. Tous les éditeurs de code un peu avancés proposent ces fonctionnalités. L'éditeur Mousepad, beaucoup plus basique (et également basé sur GtkSourceView) les propose toutes, sauf l'édition verticale et l'arborescence de fichiers (que gedit n'offre d'ailleurs que via des greffons). J'aurais bien plus de peine à travailler sans toutes ces fonctionnalités, et je ne souhaite pas non plus m'embarrasser avec une fonctionnalité complexe qui n'est pas constamment disponible dans les éditeurs de code.

La recherche et le remplacement via expressions rationnelles se révèle extrêmement utile pour la refactorisation du code. L'édition et la sélection verticales sont souvent bien pratiques, bien plus simples que le remplacement par expressions rationnelles ou une série de copier-coller, pour modifier des lignes consécutives au format semblable. Cette fonctionnalité n'est pas encore assez plébiscitée en développement à mon sens, mais elle est bien présente en général dans les éditeurs de code. Il y a aussi des fonctionnalités de sélection et édition multiples, mais leur utilité principale se trouve quand on part d'une édition en colonne, et qu'on utilise Ctrl+flèche pour se déplacer de mot en mot (ce qui conduit les curseurs à ne plus être alignés verticalement, mais à être au même endroit dans le format des lignes modifiées).

J'utilise à côté un terminal textuel pour exécuter des commandes : compiler, tester, étudier le code et gérer les versions, principalement. gedit propose (via un greffon) d'avoir un terminal de ce type en bas de la fenêtre, mais je préfère utiliser une application spécialisée. J'aimerais que ce soit aussi le cas pour l'arborescence de fichiers, mais je n'ai pas de logiciel qui fournit d'interface aussi adaptée. J'utilise l'interpréteur de commandes GNU Bash (qui est l'interpréteur par défaut dans de nombreux environnements de type Unix, et conforme à POSIX). Pour étudier le code, j'utilise simplement `grep` constamment, avec ces options :

```
grep -Fsn 'texte à trouver' src/**
grep -Fsni 'recherche non sensible à la casse' src/gengraph/**
grep -Psn 'recherche d'expression rationnelle' src/**
```

Donc soit `grep -Fsn` soit `grep -Psn`, éventuellement avec `-i` parfois. `-P` est une extension de la version GNU de `grep`, qui permet d'utiliser les expressions rationnelles PCRE. Les autres options sont documentées sur la page liée ci-dessus.

Le gestionnaire de versions Git et la plateforme GitLab sont présentés dans la section Organisation des contributions. Comme compilateur, j'utilise principalement le compilateur C de GNU dit GCC, et je vérifie parfois que la compilation fonctionne tout aussi bien avec Clang. Les outils pour l'automatisation de la construction sont présentés au début de la section Mise en place des procédures de construction, et les différents environnements sur lesquels j'ai testé GenGraph sont présentés à la fin de cette même section. Il y est aussi question du débogage.

3.1.3.2. Pratiques

J'ai une méthode de développement très itérative et libertaire. C'est-à-dire que lorsque je suis focalisé sur une tâche, j'appré-

hende toujours le projet dans son entier. Cela a pour conséquence que lorsque je développe une fonctionnalité, je suis souvent conduit à repenser certaines structures, à passer à la correction d'un bogue qui n'a rien à voir mais que j'ai constaté en faisant des tests (surtout lorsque c'est moi qui en suis à l'origine), à nettoyer du code avec laquelle ma fonctionnalité interagit afin de mieux le comprendre, à améliorer le manuel… La plupart du temps, je ne fais pas mention de ces retouches diverses dans les messages de mes révisions. Aussi, entre un moment de travail et un autre, j'ai tendance à me rendre compte que je n'ai pas bien conscience de ce que j'ai codé, et alors quand je remets au travail, je commence par relire et améliorer (et souvent, corriger). Mes révisions, dont l'historique est entièrement consultable, contiennent ainsi le plus souvent de grosses modifications faites dans tous les sens et qui sont parfois très vite contredites (cela est davantage commenté dans la section sur l'organisation des contributions, plus bas). C'est comme cela que je conçois la qualité. Cela a aussi l'avantage de me permettre de me contenter de développer ce qui est intéressant pour le projet tel qu'il est, et donc de passer très vite aux implémentations, au lieu de m'ennuyer à faire de la conception dans l'abstrait, qui serait de toute façon remise en question par la suite. Cela s'apparente aux méthodes agiles, bien qu'en ce qui me concerne, c'était simplement ma nature de fonctionner comme ça.

En travaillant sur GenGraph, j'ai souvent commencé par reformater le code existant pour pouvoir mieux le lire et le comprendre, puis mes réflexes de programmeur m'ont poussé à améliorer le code, tout cela pour n'en garder que certaines parties parfois (voire même complètement le supprimer dans le cas de CLIFilters). De même, il m'est arrivé plusieurs fois de remettre profondément en question la structure de groupes de fonctions que j'ai développées : les interfaces pour l'utilisateur final déterminent l'objectif à atteindre, qui doit être stable, et le code doit s'y soumettre en demeurant le plus simple possible. Le code n'a pas à être stable. D'où la logique itérative, qui me conduit à restructurer le code existant pour le rendre plus pertinent avec les nouvelles fonctionnalités. Pour faire tout cela, il était important que j'aie pleinement la main sur le code, ce qui a été le cas pendant le stage. Le projet étant maintenant bien découpé, le travail coordonné avec d'autres développeurs devrait être bien plus facile.

Pour développer une nouvelle fonctionnalité ou réviser du code existant, j'ai eu plusieurs méthodes. Mais en général, je commence par bidouiller des instructions C dans des fonctions : je définis une vraie procédure qui me permet de mieux saisir si ce que je fais est intéressant et viable. Je la connecte au reste, en déclarant la fonction dans l'entête et en donnant un moyen de l'appeler, au moment où me vient l'envie de tester. Moins souvent, je commence par retravailler les structures. Je documente dans le manuel quand je suis satisfait par mes tests. Il m'est arrivé une fois de faire le contraire : pour remplacer `-filter` par les nouvelles fonctionnalités `-test` et `-prop`, j'ai commencé par écrire dans le manuel ce que je voulais obtenir, principalement pour bien savoir comment transposer les fonctionnalités préexistantes dans le nouveau système.

Je touche à tout, mais je tâche quand même de ne pas toucher au code que je ne saurais pas comment tester. Comme il est difficile de bien penser à tout quand on code, j'ai l'habitude de me concentrer pour les petites choses fines, et pour les plus grosses, j'ai tendance à faire constamment des oublis qui vont laisser de grosses erreurs que je verrai très vite, de préférence dès la compilation. J'écris parfois du code qui permet à la fois de tester et de comprendre ; le moins possible, j'écris pour ne faire soit que l'un soit que l'autre.

Je rejette complètement les gros logiciels sophistiqués ou qui ne cherchent pas à devenir des standards. Je veux des dépendances minimales et des performances ; je suis le principe KISS. Cyril Gavoille semble voir aussi les choses comme ça. J'ai ainsi eu le plaisir de travailler sur un projet codé en C, puissant et léger. La philosophie et le manifeste des logiciels suckless me correspondent bien : je trouve aussi qu'il faut avoir l'audace de repenser souvent la façon dont on répond à ses objectifs généraux, et que cela permet de s'ouvrir à de nouvelles clientèles (au lieu de passer son temps à essayer de vendre ses produits à des gens qui n'y comprendraient rien). Toutefois, je trouve que la pratique corolairement soutenue est trop extrémiste : un bon projet peut quand même avoir une certaine taille. Ainsi, pour moi, GenGraph pourrait encore grossir pas mal sans que cela devienne un problème.

Le découpage en modules demande d'ajouter les déclarations des fonctions dans les entêtes, une redondance dont le fichier unique pouvait se passer. C'est un travail facile, et il est même agréable car il conduit à construire et travailler l'interface fournie par les modules, et ainsi réfléchir à ses pratiques et envisager d'utiliser le tout pour d'autres projets. L'interface est visible, sans qu'il soit nécessaire de la poser sur un document de conception externe. Le C est ainsi à mon sens très bien fait. Il suffit de sélectionner le prototype de la fonction (avec un triple-clic), de le copier-coller (avec un clic milieu), et de rajouter un point-virgule. On ne retrouve pas le problème des méthodes du C++ dont le prototype de définition diffère de celui de la déclaration (parce qu'il faut répéter le nom de la classe et omettre certains mots-clés comme `virtual`). C'était quelque chose qui m'avait vraiment dégouté en C++ : même si l'éditeur créait les définitions de méthodes automatiquement, en relisant on ne retrouvait pas exactement la même ligne et c'était gênant. En C, l'association entre déclaration et définition est beaucoup plus simple. Pour certaines longues listes de fonctions ayant toutes le même type (cas des fonctions des classes de graphes notamment), ce type est nommé par un `typedef`, ce qui permet de se contenter d'ajouter le nom de la fonction à la liste au lieu de mettre le prototype complet.

3.2. Mise en place des procédures de construction

3.2.1. L'outil : GNU Make et les commandes POSIX

Cyril Gavoille m'a très vite expliqué qu'il veut grossièrement un projet sans dépendance et à la structure très simple. Comparativement aux projets « suckless » évoqués ci-dessus, on peut constater d'une part que GenGraph v5.4 avait déjà bien dépassé la taille d'un projet suckless (qui est de l'ordre de 3 000 lignes, avec plus des deux tiers dans le fichier source C princi-

pal), et d'autre part que ces projets ne se privent pas de fichiers présentant le produit et automatisant sa construction.

GenGraph étant un projet assez gros, il était fondamental pour moi de le découper en dossiers et en fichiers multiples, et de le garnir des fichiers que l'on retrouve classiquement dans les dépôts de logiciels libres, de sorte qu'il respecte les canons. Il est donc devenu nécessaire d'avoir un outil pour automatiser la construction. Je ne souhaitais pas du tout utiliser un outil complexe et élaboré pour cela, mais simplement le logiciel GNU Make bien connu. Je n'ai jamais bien maitrisé les moteurs plus haut niveau (tels que CMake ou Meson), ni vraiment été convaincu de leur intérêt. Je voulais juste un logiciel me permettant d'exécuter les commandes que je voulais pour construire des fichiers en fonction de dépendances.

En fait, je n'ai pas été complètement satisfait par GNU Make. Au bout du compte, j'aurais bien envie de développer mon propre outil, plus léger et répondant mieux à mes besoins. J'ai un peu galéré avec GNU Make, dont j'ai trouvé les logiques souvent peu intuitives et pas assez génériques — comme avec les interpréteurs de commandes. Pour certaines choses que je voulais et que je trouve simples, j'ai dû faire appel à des extensions GNU un peu compliquées. Il existe une norme POSIX pour Make, prônée par certains puristes ; elle est peut-être utile pour les moteurs de production plus haut niveau qui se basent sur Make, mais pour ma part, puisque j'avais déjà du mal à me satisfaire de GNU Make, je n'ai pas du tout cherché à me limiter à la norme POSIX. Je suis ainsi l'idée que pour être un bon projet proposant des fonctionnalités intéressantes, il faut quand même accepter de s'appuyer sur un autre outil spécifique — même si, idéalement, j'aimerais avoir un outil plus léger et spécialisé. Il existe en effet d'autres logiciels Make, notamment la variante BSD, mais la variante GNU est très certainement la plus utilisée (même sous macOS, on trouve par défaut une version de GNU Make).

Si on veut vraiment se passer de GNU Make, on peut en fait facilement compiler le projet avec cette ligne (donnée dans le lisez-moi) :

```
cc -o gengraph -I src `find src -name \*.c` -lm
```

Cette commande semble d'ailleurs requérir beaucoup moins d'énergie que la compilation des fichiers un par un. Avec de nombreux cœurs de processeurs, cette dernière peut néanmoins être plus rapide. Il y a ainsi un moyen de compiler le projet sans GNU Make, qui est donc surtout essentiel pour le développement.

Quand nous avons essayé de compiler le projet avec la commande `make` sur l'ordinateur de Cyril Gavoille (sous macOS Ventura, la dernière version en date de macOS), nous avons fait face à des problèmes. Le Make utilisé était GNU Make version 3.81 datant de 2006. Nous avons installé la version actuelle avec MacPorts, et la commande correspondante est `gmake`. Cependant, Cyril Gavoille m'a fait remarquer que le `make` habituel lui suffisait en général. (Il ne méconnait pas les Makefile, puisqu'il nous en a fait utiliser en cours.)

À mon avis, il est souhaitable que macOS intègre une version récente de GNU Make dans ses prochaines versions. Notons que ce GNU Make 3.81 est installé dans le dossier `/usr` comme les programmes Unix classiques, alors que la version installée par MacPorts est installée dans `/opt`. En attendant que macOS se mette à jour, j'ai quand même tâché de faire attention à ce que le projet puisse être facilement compilé avec GNU Make 3.81.

Le test était facile à réaliser, puisqu'il se trouve que quelqu'un a réalisé le travail (aberrant à mes yeux) de permettre facilement l'installation de GNU Make 3.81 avec le gestionnaire de paquets Pacman, en écrivant un petit script d'installation, sans doute pour correspondre à la version de GNU Make historiquement disponible sur macOS. J'utilise la distribution Manjaro qui dérive d'Arch Linux et qui utilise donc Pacman, j'ai donc pu avoir `make-3.81` très facilement, et je peux ainsi confirmer que ma version de GenGraph peut être aisément compilée avec cette version de GNU Make.

Comme montré dans le lisez-moi, un projet utilisant Make est compilé avec cette simple commande :

```
make
```

On peut choisir une procédure particulière (voire plusieurs), appelée « cible », en l'indiquant en paramètre. La cible `install` permet classiquement d'installer le programme sur l'ordinateur du client :

```
make install
```

La cible `uninstall` automatise la désinstallation. Si un argument contient le symbole `=`, il définit la valeur d'une variable dans le Makefile :

```
make install prefix=~/.local
```

La compilation fonctionne avec `make-3.81`. Cependant, la première fois, il est nécessaire de l'exécuter deux fois et avec `-j` (qui conduit Make à lancer simultanément plusieurs tâches de construction) :

```
make-3.81 -j
make-3.81 -j
```

La première fois, Make va créer quelques dossiers et s'arrêter tout de suite, parce que ces dossiers n'étaient pas présents au départ et sont requis pour créer les fichiers. Si on n'utilise pas `-j`, Make va s'arrêter pour chaque nouveau dossier créé (il y en a plus d'une dizaine). Une fois que les dossiers sont créés, il n'y a plus de souci. Cela fonctionne aussi si on adjoint un chiffre suffisamment élevé à `-j`, en écrivant par exemple `-j9` (comme dans le lisez-moi et dans `gengraph.sh`). C'est à mon avis bien satisfaisant.

La procédure d'installation ne fonctionne pas avec GNU Make 3.81, parce que j'utilise la cible spéciale `.ONESHELL` pour pouvoir écrire facilement un script dans le Makefile. Il faudrait travailler un peu pour déplacer ce script dans `gengraph.sh`, ce qui n'est pas évident car il utilise des variables du Makefile.

Le Makefile (le script adressé à Make) que j'ai écrit fait plus de 200 lignes, ce qui est un peu gros pour un Makefile écrit à la main. Cependant, près de la moitié est occupée par les procédures d'installation et de désinstallation, qui sont des procédures linéaires très simples et interactives. (Les moteurs de production comme CMake produisent des quantités de Makefile très longs.)

Pour les commandes les plus centrales (y échappe la génération du manuel au format PDF), je tâche de me limiter aux programmes admis par la convention GNU et dans leur forme admise par le standard POSIX. Pour la commande `find` par exemple, il faut donc regarder le manuel via `man 1p find` et non simplement `man find` qui, dans mon cas, affiche le manuel de GNU Find qui prend en charge beaucoup plus d'options que ce que demande POSIX.

Je tâche aussi de me limiter à la syntaxe des commandes POSIX. Pour cela, j'ai réalisé mes tests avec Dash, un tout petit interpréteur de commandes (150 ko) qui implémente la norme POSIX sans faire grand-chose de plus. Je l'ai découvert sur le wiki d'Arch Linux, et aussi parce qu'il est utilisé par défaut quand je lance Make sous Termux (l'environnement de compilation que j'ai sous Android). On peut choisir l'interpréteur que l'on veut avec Make en définissant la variable `SHELL` :

```
make SHELL=dash
```

Notons qu'il y a de nombreux interpréteurs de commandes qui ne respectent pas la norme POSIX. Ainsi, chaque utilisateur qui utilise un interpréteur de commandes ne respectant pas ce standard doit disposer d'un autre interpréteur de commandes qui implémente la norme.

3.2.2. Les cibles de compilation

La compilation du programme est essentiellement un assemblage, un peu à ma sauce, d'outils très standards, en particulier lorsqu'on utilise GNU Make et le compilateur GCC ou Clang. Au tout début, on trouve l'affectation des chemins des fichiers et dossiers dans des variables :

gengraph-repo/Makefile (extrait)

```
1 BINDIR := _build
2 SRC := $(shell find src -name '*.c' | sort)
3 SRCSUBDIRS := $(shell cd src/ && find * -type d)
4 CLIH_SRC := $(wildcard src/clih/*.c)
5 MANUAL := MANUAL.txt CHANGELOG.txt
6 DEPFILES := $(SRC:src/%.c=_build/obj/deps/%.dep)
```

`build` est un nom souvent usité pour le dossier dans lequel placer tous les fichiers produits par la compilation. J'ai ajouté un tiret bas, convention que j'ai piquée au projet Coq et qui m'a beaucoup plu, car cela permet de bien représenter le fait que ce dossier ne sera pas inclus dans le dépôt, est destiné à rester local, et que ce contenu est comme une sorte de cache qui peut être supprimé et reconstruit en quelques commandes. C'est semblable au `.` utilisé comme préfixe des fichiers cachés. La compilation de GenGraph crée donc à la racine du projet un dossier `_build`, et y met tous les fichiers qu'elle produit. Elle ne touche à rien d'autre, et il ne faut normalement jamais modifier directement ce qui se trouve dans `_build`.

Les fichiers et dossiers du projet sont trouvés à l'aide de la commande `find`. Elle ne fait aucun tri, donc `sort` est utilisé pour avoir un ordre plus cohérent (au moins pour que les fichiers d'un même dossier soit compilés les uns à la suite des autres).

Les fichiers `.c` sont mis dans la variable `SRC`, et sont utilisés pour créer les chemins vers les fichiers de dépendances, inclus plus tard dans le Makefile :

gengraph-repo/Makefile (extrait)

```
104 include $(wildcard $(DEPFILES))
```

L'utilisation de `wildcard` à cet endroit permet d'éviter un avertissement tant que ces fichiers ne sont pas encore créés.

Les sous-dossiers de `src/` sont mis dans la variable `SRCSUBDIRS`. Les fichiers sources de Clih sont également récupérés pour la compilation des outils. Comme il n'y a pas de sous-dossiers dans Clih et qu'il n'y en aura probablement jamais, j'ai utilisé tout simplement `wildcard` à cet endroit. Le Makefile traite bien d'autres fichiers, mais la plupart du temps, je n'ai pas trouvé pertinent de définir des variables pour chacun (à l'origine, j'avais été éduqué à le faire, mais j'ai remis en question cette pratique).

Ensuite viennent les drapeaux de compilation :

gengraph-repo/Makefile (extrait)

```
8  DEPPROG := _build/tools/dependencies
9  CWARNINGS := -Wall -Wwrite-strings -Wcast-qual -Wno-format-truncation -Wno-format-overflow -Wno-maybe-uniniti
10 CFLAGS := -std=gnu11 $(CWARNINGS) `$(DEPPROG) --cflags` -I src
11 DEPFLAGS = -MT '$$(OBJDIR)/$(1).o' -MMD -MP -MF '_build/obj/deps/$(1).dep'
12 LDLIBS := -lm `$(DEPPROG) --libs`
```

`-Wall` est essentiel pour avoir une quantité d'avertissements très intéressants, qui m'évitent souvent d'avoir à me concentrer quand je produits du code. En effet, ces avertissements permettent d'écrire beaucoup de code efficacement, de sorte à poser ses idées en oubliant éventuellement certaines choses et en laissant le compilateur voir l'incohérence. Par exemple, il m'arrive très souvent de déclarer une variable et d'oublier de l'utiliser, parce qu'elle ne sert qu'à gérer un problème technique. J'ai aussi inclus trois autres avertissements que je trouve très utiles :

- `-Wwrite-strings` : signale la conversion des littéraux de type chaine de caractères (c'est-à-dire les textes écrits en dur dans le code sous la forme `"texte"`, entre guillemets) en chaines de caractères modifiables. Le langage C permet encore de placer ces constantes dans des variables de type `char *`, alors que la modification de ces constantes conduit souvent tout bonnement à un plantage du programme. C++, quant à lui, donne à ces littéraux le type `char const *` (pointeur vers des caractères constants), et le standard interdit leur conversion implicite en `char *`.
- `-Wcast-qual` : on ne peut pas (encore) se permettre de l'utiliser dans tous les projets, mais pour GenGraph c'est approprié. Cela signale les conversions éliminant les qualifications « cvr », en particulier `const` (les deux autres sont `volatile` et `restrict`, non utilisés par GenGraph). Par exemple, les conversions de `char const *` à `char *` sont rapportées. Le cas où c'est un problème est pour certaines fonctions pouvant aussi bien traiter des données constantes que variables, par exemple `strstr()` de la bibliothèque standard, qui prend un `char const *` au cas où mais renvoie un `char *` au cas où. Dans C23, les fonctions de la bibliothèque standard comme `strstr()` préserveront le `const` éventuel.
- `-Wcast-function-type` : signale les conversions de pointeurs de fonctions vers des types incompatibles. Notamment, je fais des conversions de `void (* )(int const *)` en `void (* )(void const *)` par exemple, mais je ne souhaite pas pouvoir convertir `void (* )(int, int)` en `void (* )(double)`.

Le but d'avoir de tels avertissements est en fait de les traiter comme si c'étaient des erreurs. Il y a une option `-Werror` qui en ferait en effet des erreurs, mais elle risquerait d'empêcher des personnes utilisant d'autres compilateurs de compiler le programme, et est parfois une contrainte inutile lorsqu'on fait quelques expérimentations, c'est pourquoi je n'utilise jamais cette option. Quand je compile d'autres projets, il y a souvent des quantités inquiétantes d'avertissements, mais parfois ces projets sont bien compilés et fonctionnent à la fin.

J'ai aussi masqué certains avertissements impertinents :

- `-Wno-format-truncation` et `-Wno-format-overflow` : masquent des avertissements de GCC indiquant que les appels à respectivement `snprintf()` et `sprintf()` pourraient déborder (j'utilise souvent des tableaux de taille statique pour simplifier, et lorsqu'ils sont combinés avec `s[n]printf()`, GCC compare leurs tailles, ce qui est ennuyeux). Clang ne reconnait pas ces deux options.
- `-Wno-maybe-uninitialized` : il semble que Cyril Gavoille a eu dans GenGraph beaucoup de logiques consistant à initialiser ses variables dans une condition à l'intérieur d'une boucle. Cela trompe GCC (mais pas Clang, dont l'option équivalente est : `-Wno-unitialized`). Ce n'est vraiment pas mon habitude de désactiver cet avertissement-là : pour ma part, j'initialise mes variables classiquement, généralement dès la déclaration. Mais les techniques de programmation de Cyril Gavoille le rendent apparemment lourd.
- `-Wno-cast-function-type-strict` : c'est spécifiquement pour la version de Clang que j'ai avec Termux sur Android, qui signale toutes les conversions de pointeurs de fonction (signalements dont je n'ai même pas compris l'intérêt).
- `-Wno-unknown-warning-option` : évite à Clang de se plaindre d'options d'avertissements reconnues seulement par GCC.

Ces deux derniers ne sont inclus que pour Clang, via le petit programme `dependencies`. `-Wcast-function-type` n'est pas inclus avec GCC < 8, car cette option n'est pas reconnue par GCC 5.4 que j'ai utilisé pour compiler GenGraph sur Windows XP, et sa présence empêche carrément la compilation. La version actuelle de GCC est GCC 13, et les options `-W` non reconnues le conduisent à afficher des avertissements. Ces options risqueraient de poser des problèmes avec d'autres compilateurs, auquel cas la solution est de modifier la variable `CFLAGS` (ou `CWARNINGS`) lors de l'exécution de Make. Notons que si on change de compilateur sans supprimer/recompiler `dependencies`, on aura alors les options prévues pour l'autre compilateur.

`dependencies` est un tout petit programme C. Les outils comme CMake semble produire automatiquement de tels programmes pour tester l'environnement de compilation (et je trouve cela plutôt lourd). Une méthode toute simple pour obtenir les options de compilation que l'on veut est simplement d'écrire un petit programme C avec des directives `#ifdef`. À l'origine, je l'avais créé juste pour ajouter la dépendance libbsd (option `-lbsd`), nécessaire pour compiler GenGraph sous Linux mais pas sous les autres environnements. En haut du fichier source de GenGraph v5.4, on avait ainsi ce commentaire :

gengraph-v5.4/gengraph.c (extrait)

```
27    Comment l'installer / le compiler ?
```

gengraph-v5.4/gengraph.c (extrait)

```
28
29      (MacOs)  gcc gengraph.c -o gengraph
30      (Linux)  gcc gengraph.c -lm -lbsd -o gengraph
```

`-lm` n'est pas requis sous macOS, mais ne pose pas de problème. `-lbsd` en revanche demande à être installé, or cela n'est nécessaire que sous GNU/Linux, et même pas avec les versions les plus récentes car `arc4random()` est désormais fourni par glibc (la bibliothèque C fournie par GNU, utilisée sous GNU/Linux) à partir de la version 2.36. `-lbsd` est ainsi encore nécessaire au CREMI, mais ni sur mon ordinateur sous GNU/Linux, ni les autres plateformes sur lesquelles j'ai expérimenté la compilation.

`-std=gnu11` active C11 — c'est normalement l'option par défaut avec les compilateurs récents. J'ai tenu à demander à Cyril Gavoille si adopter C11 lui convenait, car j'ai vu que de certaines bibliothèques mettent des années à adopter les nouveaux standards, et demeurent incompatibles avec eux avant qu'un certain travail soit fait. On utilise GNU C11 et pas simplement C11 pour avoir les fonctions de POSIX. `-I src` permet d'inclure les fichiers des différents sous-projets comme si c'étaient déjà des bibliothèques installées sur le système (le compilateur cherchera dans le dossier `src/` les fichiers inclus).

La série d'options alambiquée `-MT file.o -MMD -MP -MF file.dep` demande à GCC de créer les fichiers de dépendances de `file.c` en même temps qu'il le compile. Habituellement, l'extension de ces fichiers est `.d`, mais c'est pour moi l'extension des fichiers sources en langage D, c'est pourquoi j'ai préféré utiliser une extension `.dep` à trois lettres. J'ai trouvé cette série d'options sur le net à plusieurs endroits, et en regardant dans le manuel de GCC, j'ai vu que c'est effectivement ce qu'il faut écrire. La documentation a de quoi faire mal au crâne, parce qu'on peut lire, pour `-MMD` :

> Like `-MD` except mention only user header files, not system header files.

Et pour `-MD` :

> `-MD` is equivalent to `-M -MF file`, except that `-E` is not implied. […]

Les fichiers de dépendances sont des bouts de Makefile qui ne contiennent que des déclarations de dépendances. Par exemple, pour le fichier `graph.c`, on aura le fichier `graph.dep` contenant :

```
$(OBJDIR)/gengraph/graph.o: src/gengraph/graph.c src/gengraph/graph.h \
 src/gengraph/common.h src/gengraph/debug.h src/gengraph/errors.h
```

Ensuite, certaines choses sont mises en place, distinguant la compilation des livrables de celle des programmes contenant les symboles de débogage :

gengraph-repo/Makefile (extrait)

```
14  ifneq (debug,$(filter debug,$(MAKECMDGOALS)))
15      OBJDIR := _build/obj/release
16      CPPFLAGS += -D NDEBUG
17      CFLAGS += -O2
18  else
19      BINDIR := _build/debug
20      OBJDIR := _build/obj/debug
21      CFLAGS += -Og -g
22  endif
23  OBJ := $(SRC:src/%.c=$(OBJDIR)/%.o)
24  CLIH_OBJ := $(CLIH_SRC:src/%.c=$(OBJDIR)/%.o)
```

Si `debug` est inclus dans les cibles (y compris dans une partie d'un mot seulement, comme dans `split-debug`), le dossier cible de l'exécutable devient `_build/debug` (au lieu de simplement `_build`). Les fichiers objets (`.o`) sont placés dans `_build/obj`, dans un sous-dossier correspondant au type de compilation. Des options adaptées sont ajoutées dans `CFLAGS`. La macro `NDEBUG` permet de supprimer les assertions faites avec `assert()`. Après ces conditions, le dossier de sortie étant défini, les chemins vers les fichiers objets `.o` sont générés en fonctions des chemins des fichiers sources `.c`.

L'utilisation de `:=` plutôt que `=` pour définir les variables est un peu superflue : elle conduit à avoir directement l'expansion des variables au moment de l'affectation. Si on utilise `=`, cette expansion ne se produit qu'au moment où la variable est invoquée — et donc à chaque fois où elle est invoquée —, ce qui est nécessaire pour `DEPFLAGS`.

La cible par défaut est comme d'habitude appelée `all` et compile le programme en mode `release`, le manuel en HTML et la complétion :

gengraph-repo/Makefile (extrait)

```
45  all: release doc-html _build/tools/clih/bash-completion
46
47  release debug: bin
48  bin: $(BINDIR)/gengraph $(MANUAL:%=$(BINDIR)/%)
```

gengraph-repo/Makefile (extrait)

```
49
50  doc: doc-html doc-pdf
51  doc-html: _build/gengraph.html
52  doc-pdf: _build/gengraph.pdf
53
54  clean:
55      echo 'Removing directory _build/...'
56      $(RM) -r _build
```

La cible classique `clean` supprime tout simplement le dossier `_build`, sans poser de questions.

Normalement, Make affiche chaque ligne de commande exécutée. Étant donné tous les avertissements utilisés, les commandes de compilation sont très longues, et dans le terminal elles étaient ainsi très désagréables à avoir. J'ai donc utilisé la cible spéciale `.SILENT` qui inhibe ces affichages, et j'ai compensé en faisant des affichages avec `echo`. J'ai vu sur le net des recommandations de donner un moyen d'outrepasser `.SILENT`, cependant Make en offre déjà un par défaut : l'option `--trace`. Aussi, j'ai fait en sorte que la commande de compilation soit affichée la première fois seulement (au lieu d'être affichée pour chaque fichier).

Le reste du code est le suivant :

gengraph-repo/Makefile (extraits)

```
69  $(BINDIR)/gengraph: $(OBJ) $(DEPPROG) | $(BINDIR)/
70      echo 'Linking... Command is: $(LINK.o) $(OBJDIR)/**.o $(LDLIBS) -o $@'
71      $(LINK.o) $(OBJ) $(LDLIBS) -o $@
72
73  $(BINDIR)/%.txt: %.txt | $(BINDIR)/
74      echo 'Copying $<...'
75      cp -f $< $@
76
77  _build/gengraph.html: _build/tools/clih/generate-html-manual tools/clih/manual-template.html $(MANUAL) | _bu
78      echo 'Generating $@...'
79      ./$^ > $@

...  […]

90  $(DEPPROG): CFLAGS := -std=gnu11 -O2 -Wall
91  $(DEPPROG): LDLIBS :=
92  _build/tools/gengraph_c-%: $(DEPPROG)
93  _build/tools/clih/%: tools/clih/%.c $(CLIH_OBJ) $(DEPPROG) | _build/tools/clih/
94      echo 'Compiling $@... Command is: $(LINK.c) $< $(CLIH_OBJ) $(LDLIBS) -o $@'
95      $(LINK.c) $< $(CLIH_OBJ) $(LDLIBS) -o $@
96  _build/tools/%: tools/%.c | _build/tools/
97      echo 'Compiling $@... Command is: $(LINK.c) $< $(LDLIBS) -o $@'
98      $(LINK.c) $< $(LDLIBS) -o $@
99
100 $(BINDIR)/ _build/tools/ _build/tools/clih/ $(OBJDIR)/ _build/obj/deps/ $(SRCSUBDIRS:%=$(OBJDIR)/%/) $(SRCSU
101     echo 'Creating directory $@...'
102     mkdir -p $@

...  […]

106 .SECONDEXPANSION:
107
108 $(OBJDIR)/obj-start: | $(OBJDIR)/
109     mkdir -p $(SRCSUBDIRS:%=$(OBJDIR)/%/) $(SRCSUBDIRS:%=_build/obj/deps/%/)
110     echo 'Compilation command is: $(COMPILE.c) -o $(OBJDIR)/**.o '$(call DEPFLAGS,**)' $(OBJDIR)/**.o'
111     touch $@
112 .INTERMEDIATE: $(OBJDIR)/obj-start
113
114 $(OBJDIR)/%.o: src/%.c $(DEPPROG) | $(OBJDIR)/obj-start
115     echo 'Compiling $<...'
```

Les fichiers `.txt`, sources du manuel, sont placées à côté de l'exécutable. La commande pour générer l'exécutable à partir des `.o` est classique. Pour les exécutables de `tools/`, on ne génère pas de `.o` : on compile directement le fichier `.c` (puisqu'il n'y en a qu'un seul). Pour la génération du manuel en HTML, le programme `generate-html-manual` est invoqué avec les bons arguments, et la modification de chacun de ces fichiers provoque la regénération du manuel. `$^` désigne la liste des dépendances (avant `|`), `$<` désigne la première dépendance seulement, et `$@` désigne la cible.

Chaque fichier est dépendant de la création du dossier dans lequel il doit se trouver. J'utilise pour cela une dépendance placée après `|`, c'est-à-dire une dépendance dont seulement l'existence est vérifiée (et non la date de dernière modification ; je ne sais d'ailleurs pas comment Make gère cela pour les dossiers). C'est cela que GNU Make 3.81 gère incorrectement ; c'est néanmoins

la solution que je trouve la plus propre.

Il y a de nombreux dossiers différents à créer. Pour les dossiers destinés à accueillir les `.o`, j'ai dû utiliser la deuxième expansion activée par la cible spéciale `.SECONDEXPANSION`, car j'obtiens le nom du dossier avec `$(dir ...)`, et vraisemblablement, par défaut, ça ne marche pas si on utilise `%` dedans. J'ai trouvé sur le net (le fameux StackOverflow) que `.SECONDEXPANSION` permet de gérer ce cas. `$$(dir $(OBJDIR)/%.o` devient ainsi par exemple `$(dir _build/obj/release/%.o)` puis `$(dir _build/obj/release/gengraph/graph.o)` puis finalement `_build/obj/release/gengraph/`.

Pour afficher la commande de compilation la première fois seulement, la technique est d'utiliser un fichier intermédiaire, que Make supprime lorsqu'il se termine (qu'il soit allé au bout de ses tâches ou non). J'ai nommé ce fichier `obj-start`.

`$*` est remplacé par ce qui correspond au `%` dans la cible. `$(call DEPFLAGS,$*)` provoque l'expansion de la variable `DEPFLAGS` en remplaçant `$(1)` par `$*`.

Pour la compilation de `dependencies` (contenu dans la variable `DEPPROG`), on le supprime bien sûr des options de compilation. `` `$(DEPPROG) --cflags` `` et `` `$(DEPPROG) --libs` ``, avec les accents graves, permet d'intégrer sa sortie à la commande de compilation.

J'ai également écrit des règles pour générer le manuel au format PDF à l'aide de Chromium et Ghostscript :

gengraph-repo/Makefile (extrait)

```
81 _build/gengraph_base.pdf: _build/gengraph.html
82     echo 'Generating $@ from $<...'
83     chromium --headless --disable-gpu --run-all-compositor-stages-before-draw --print-to-pdf-no-header --prin
84         echo '>> Failed. Is Chromium installed?'
85 _build/gengraph.pdf: _build/gengraph_base.pdf
86     echo 'Compressing $< into $@...'
87     title=`head -1 MANUAL.txt` && title=$${title%% [-(]*}
88     gs -q -sDEVICE=pdfwrite -dNOPAUSE -dBATCH -sOutputFile=$@ -c "[ /Title ($$title) /DOCINFO pdfmark" -f $<
```

Ghostscript (par je ne sais quelles techniques) réduit de moitié la taille du document et lui ajoute un titre. En pratique, je préfère créer `gengraph_base.pdf` avec l'interface de Chromium, car cela permet de paramétrer la sortie, notamment je préfère réduire la taille du texte à 85 %. J'utilise Chromium et non Firefox pour cela, parce que Firefox laisse les liens vers des sections pointer vers l'URL web au lieu de les remplacer par des liens internes au document, et il remplace les textes (sans doute à cause de la police de caractères utilisée) par des images, ce qui conduit à avoir une taille de document énorme. La création du PDF avec Chromium prend une quinzaine de secondes.

J'ai trouvé l'écriture et le test du Makefile bien laborieux, et le code produit ne me semble pas intuitif. Mais il fonctionne et on ne devrait pas avoir de raison d'y faire de grosses modifications désormais. Si jamais il le fallait, je pense que je préfèrerais simplement coder mon petit outil en C, compatible avec POSIX. Pourquoi Make ne crée-t-il pas les dossiers cibles tout seul, par exemple ?

Pour pouvoir compiler et exécuter directement GenGraph, et éviter d'avoir à écrire `_build/gengraph` pour exécuter (note : quand on commence à écrire `_b` dans l'interpréteur de commandes, il ne complète pas avec le nom du dossier), j'ai développé le script `gengraph.sh`. Ce script utilise `gmake` à la place de `make` si disponible, et passe ses arguments à un autre programme en fonction de ce qui se trouve dans le premier (qui est éliminé de la liste des arguments) :

- rien ou `--` : compile et exécute le livrable ;
- `debug` : compile et exécute la version de débogage ;
- `backtrace` : compile la version de débogage, puis l'exécute avec un débogueur (par défaut GDB) pour afficher sa pile d'appels si le programme plante ;
- `valgrind` : compile la version de débogage et l'exécute avec Valgrind.

Des exemples d'utilisation de ce script sont donnés dans le lisez-moi. Valgrind est extrêmement pratique pour trouver toutes les fuites de mémoire et écritures dans un emplacement mémoire non valide. Je m'en suis servi un certain nombre de fois au cours du stage. La pile d'appels du débogueur permet quant à elle de diagnostiquer les problèmes les plus simples, par exemple l'écriture dans un pointeur nul ou non défini. Pour un dépassement de tableau, GDB ne fait généralement pas l'affaire car cela ne fait pas planter le programme tout de suite ; c'est plutôt au moment où `free()` est appelée qu'elle va voir que les informations dont elle a besoin pour faire la désallocation ne sont pas correctes. Valgrind permet de détecter ce genre d'écritures incorrectes. Contrairement aux avertissements, à mes yeux, ce ne sont pas des outils qui permettent d'éviter d'avoir à se concentrer : je me sens vraiment mieux quand je ne produis pas ce genre de bogues. Mais la réalité étant ce qu'elle est, ces logiciels permettent de résoudre les problèmes assez rapidement, en particulier quand on ne comprend pas bien ce qu'on fait.

3.2.3. Les outils pour travailler avec un seul fichier

L'outil est présenté dans le lisez-moi. La commande :

```
make merge
```

crée le fichier `gengraph.c` à la racine du projet. Ce fichier est ignoré par Git (grâce au fichier `.gitignore`). Y sont écrits tous les fichiers de `src/` des quatre sous-projets, ainsi que le manuel et l'historique dans un long commentaire tout à la fin (comme dans le fichier `gengraph.c` de GenGraph v5.4). Chaque fichier commence avec un préfixe `/////--` (cinq barres obliques, deux traits d'union et une espace), ou `/*****--` pour le manuel et l'historique. Par exemple :

```
/////-- src/gengraph/algos/algos.h
```

Cela est fait avec des commandes POSIX, utilisant des boucles `for`, `echo` et `cat`, ainsi que `exec 3> gengraph.c` pour ouvrir le fichier de sortie et `exec 3>& -` pour le fermer. L'ordre n'est pas idéal : pour chaque sous-projet, on a tous les `.h` au début, suivis par tous les `.c`. Je n'ai pas trouvé de moyen (simple) de faire mieux avec des commandes POSIX. S'il fallait améliorer cet ordre, je préfèrerais coder en C la concaténation des fichiers, parce que ça me fatigue finalement beaucoup moins que chercher comment faire les choses avec Bash et les commandes.

Une fois que des modifications ont été effectuées, elles peuvent être réparties dans les fichiers avec :

```
make split
```

Cela crée l'exécutable `gengraph_c-split` à partir du fichier source correspondant, et l'exécute. Ce programme prend en charge les fonctionnalités suivantes :

- la mise à jour des fichiers ;
- la création de nouveau fichier ;
- la suppression de fichier avec demande de confirmation.

Le programme associe les morceaux de `gengraph.c` aux fichiers en fonction du chemin dans l'entête commençant par `/////--` (ou `/*****--`). Pour la suppression, le programme fait un parcours de `src/` avec la bibliothèque POSIX `dirent.h`, afin d'enregistrer les chemins dans un tableau et pouvoir les marquer chaque fois qu'un fichier est trouvé dans `gengraph.c` (les fichiers qui ne sont pas dans `src/` sont ignorés dans ce processus). Pour la mise à jour, il commence par comparer la longueur, puis en cas d'égalité, compare le contenu caractère par caractère. Les lignes vides en début et en fin sont toujours supprimées, et un retour à la ligne est toujours mis en fin. Les éventuelles lignes `--*****/` (`make merge` en place une à la toute fin du fichier) sont un délimiteur de fin de fichier, mais pas de début. Afin d'être robuste, le programme est ainsi capable de gérer des lignes (au tout début et après `--*****/`) qui ne font partie d'aucun fichier.

Le programme écrit sur la sortie standard son journal, de ce type :

```
src/gengraph-formats/new_generation.c created
src/gengraph-cli/main.c updated
Remove src/gengraph/graphs/fixed.c? [Y/n]
src/gengraph/graphs/fixed.c removed
```

Dans le cas où son journal serait vide autrement, le programme compense en écrivant « No file updated ». Pour la confirmation de suppression de fichier, très précisément : le programme lit exactement une ligne et interprète « oui » soit si la ligne ne contient que des espaces (y compris tabulations horizontales), soit si son premier caractère qui n'est pas une espace est un Y ou un O, peu importe la casse. Le fait que le « Y » est en majuscule dans le message est une convention utilisée par d'autres programmes en ligne de commande, signalant qu'il s'agit de la réponse par défaut.

Pour compléter l'outil, j'ai développé un autre programme amusant : `gengraph_c-compile`. Ce programme modifie la sortie du compilateur afin d'y ajouter les numéros de ligne de `gengraph.c`. Le programme fait d'abord un parcours complet de `gengraph.c` pour trouver quel numéro de ligne correspond à la ligne n° 0 de chaque fichier. Puis, avant chaque ligne contenant `src/` puis trois deux-points, il écrit le numéro de ligne décalé, de sorte à rajouter les lignes orange suivantes :

```
Compiling src/gengraph-cli/query_test.c...
src/gengraph-cli/query_test.c: Dans la fonction « GetGraphPair »:
(gengraph.c:26856:3)
src/gengraph-cli/query_test.c:18:3: erreur: expected declaration specifiers before « GenQueryGraph »
   18 |   GenQueryGraph(Q);
      |   ^~~~~~~~~~~~~
(gengraph.c:26860:3)
src/gengraph-cli/query_test.c:22:3: erreur: le paramètre « testData » est initialisé
   22 |   } * testData = data;
      |   ^
(gengraph.c:26860:18)
src/gengraph-cli/query_test.c:22:18: erreur: « data » non déclaré (première utilisation dans cette fonction)
   22 |   } * testData = data;
      |                  ^~~~
```

```
(gengraph.c:26860:18)
src/gengraph-cli/query_test.c:22:18: note: chaque identificateur non déclaré est rapporté une seule fois pour ch
(gengraph.c:27032)
src/gengraph-cli/query_test.c:194: erreur: expected « { » at end of input
(gengraph.c:27031:1)
src/gengraph-cli/query_test.c:193:1: attention: « return » manquant dans une fonction devant retourner une valeu
  193 | }
      | ^
src/gengraph-cli/query_test.c: Au plus haut niveau:
(gengraph.c:26854:21)
src/gengraph-cli/query_test.c:16:21: attention: « GetGraphPair » défini mais pas utilisé [-Wunused-function]
   16 | static graph_pair_t GetGraphPair(query * Q, void * data)
      |                     ^~~~~~~~~~~~
```

Ce programme est invoqué par les quatre commandes suivantes :

```
make split-all
make split-debug
./gengraph.sh split ...
./gengraph.sh split-debug ...
```

Les deux dernières appellent les deux premières, et exécutent GenGraph en relayant les arguments si la compilation réussit.

Je n'ai pas prévu que le fichier `gengraph.c` puisse être compilé directement. En fait, on pourrait penser que c'est presque possible, avec une commande telle que celle-ci :

```
cc -o gengraph gengraph.c -I src/ $(for folder in `find src/* -type d`; do echo -iquote $folder; done) -lm
```

Le principal problème est que les fichiers ne sont pas dans le bon ordre dans le fichier `gengraph.c`, notamment les entêtes, d'où le `-I src/` qui demande d'aller les chercher dans le dossier `src/`. Cela conduirait à utiliser ces fichiers plutôt que le contenu de `gengraph.c`. Notons aussi que l'utilisation de plusieurs fichiers permet de réutiliser certains identifiants, grâce au mot-clé `static`, ou encore dans les macros et les énumérations. C'est pourquoi cette ligne de commande conduit à des erreurs de compilation.

Je me suis pas mal amusé à développer ces outils. Ils permettent à des développeurs de collaborer avec l'interface qu'ils maitrisent ou apprécient le plus, et même de profiter tantôt des intérêts de l'une et des intérêts de l'autre. En fait, je me suis moi-même retrouvé à faire usage du fichier combiné un certain nombre de fois : étant donné que j'ai souvent modifié des interfaces de fonctions, changé les noms des variables globales (ou placé celles-ci dans des structures), ou réorganisé les modules (et donc les `#include`), faire tout cela directement dans `gengraph.c` (parfois avec un simple « Remplacer tout ») était plus simple qu'ouvrir chaque fichier listé par `grep`. Une meilleure solution serait que mon éditeur permette de faire une recherche dans les fichiers d'un dossier comme s'ils étaient concaténés ; je n'ai jamais connu d'éditeur qui le fasse vraiment bien, et je n'ai pas encore eu l'occasion de développer le mien. J'ai quelques fois utilisé `gengraph_c-compile`, et j'ai trouvé qu'effectivement, cela ajoutait un vrai plus.

Notons que l'utilisation parallèle du fichier combiné `gengraph.c` et de l'arborescence de fichiers est un peu risquée. Avant de passer d'une interface à l'autre, il faut bien faire attention à la mettre à jour. Sinon, utiliser `make merge` ou `make split` ne fera qu'effacer les modifications faites d'un côté au profit de celles faites de l'autre côté. La solution consiste alors à fusionner les modifications à la main dans `gengraph.c`, en affichant celles faites dans l'arborescence avec `git diff`. Inverser `make merge` et `make split` peut également être fatal pour les modifications effectuées. `make merge` n'aura pas d'effet si `gengraph.c` est plus récent que tous les fichiers à mettre dedans ; avec `make split`, il n'y a aucune vérification de date (cela pourrait être facilement rajouté avec `stat()`), mais ma recommandation est de simplement supprimer `gengraph.c` si on ne s'en sert pas.

3.2.4. Les procédures d'installation et de désinstallation

Le but de l'installation est de placer les fichiers de GenGraph dans des dossiers standards, afin que l'utilisateur n'ait plus à s'en soucier, et puisse exécuter le programme avec la commande `gengraph` sans créer d'alias. C'est aussi un confort pour les administrateurs souhaitant installer un logiciel pour leurs utilisateurs, et cela peut de plus permettre à d'autres programmes d'utiliser GenGraph s'il est installé.

Je souhaitais offrir une procédure d'installation qui me rappelle un peu les installateurs d'InstallShield que j'avais lorsque j'installais des logiciels (notamment des jeux vidéos) à l'époque de Windows 98. (C'était le truc qui commençait toujours avec une barre de chargement qui montait très très vite jusqu'à 99 %, puis qui s'y arrêtait quelques secondes, avant de monter à 100 %.) GNU a défini un ensemble de variables standards, très communément utilisées (notamment `prefix`), pour définir les chemins dans lesquels installer les logiciels. Cependant, j'ai mis des années d'abord pour les découvrir puis pour me sentir vraiment familier avec ces variables. Ainsi, j'ai souhaité offrir une procédure légèrement plus interactive : elle indique quelle est la configuration qui va être utilisée, demande confirmation, et en cas d'annulation, des exemples de commandes modifiant la configuration sont affichées.

La cible de Make bien connue pour lancer les installations est `install`. Elle est souvent utilisée avec `sudo` (la commande passant le relai au superutilisateur) pour une installation partagée avec les autres utilisateurs. J'ai aussi prévu une installation « portable » de GenGraph, permettant à celui-ci d'être installé par exemple dans un dossier sur clé USB, sans que l'utilisateur ait à sélectionner les bons fichiers dans le dossier `_build` (qui pourrait bien devenir plus complexe qu'il l'est actuellement).

Voici un exemple d'utilisation :

```
$ make install

Installation configuration:
   _build/gengraph --> /usr/local/bin/gengraph
   Data (MANUAL.txt CHANGELOG.txt) --> /usr/local/share/gengraph/
   Documentation (_build/gengraph.html README.fr.md LICENSE.fr.md) --> /usr/local/share/doc/gengraph/
   Completion: Bash (/usr/local/share/bash-completion)
   Use relative paths from program (portable install): no
Proceed? [y/N] n

Please set the directories you want with a command such as one of these:
   make install prefix=~/.local
   make install DESTDIR=gengraph_portable portable_install=yes_please \
      prefix=. completion=bash
   make install DESTDIR=gengraph_portable portable_install=oh_yes_again \
      binpath=gengraph ourdatadir=. docdir=.
   make install DESTDIR=some_other_root completion=
   make install -e

$ make install DESTDIR=gengraph_portable portable_install=oh_yes_again \
      binpath=gengraph ourdatadir=. docdir=.

Installation configuration:
   Actual root: gengraph_portable
   _build/gengraph --> gengraph
   Data (MANUAL.txt CHANGELOG.txt) --> .
   Documentation (_build/gengraph.html README.fr.md LICENSE.fr.md) --> .
   Completion: none
   Use relative paths from program (portable install): yes
Proceed? [y/N] y

Linking the program with the configuration...
Creating directories and copying files...
Done.
```

Le cas par défaut est l'installation non portable. Dans ce cas, les chemins absolus sont préférables, et l'installateur est capable de rendre les chemins absolus lui-même :

```
$ make install prefix=../../../.local

Some of your installation paths seem not to be absolute.
Do you want them to be prepended with $PWD? [y/N] y

Installation configuration:
   _build/gengraph --> /home/sylchiron/.local/bin/gengraph
   Data (MANUAL.txt CHANGELOG.txt) --> /home/sylchiron/.local/share/gengraph
   Documentation (_build/gengraph.html README.fr.md LICENSE.fr.md) --> /home/sylchiron/.local/share/doc/gengraph
   Completion: Bash (../../../.local/share/bash-completion)
   Use relative paths from program (portable install): no
Proceed? [y/N]
```

Ce n'est cependant pas proposé si `DESTDIR` est spécifié. `DESTDIR` est une variable de la convention GNU servant à indiquer la racine d'un autre système de fichiers dans lequel l'installation est effectuée. Cela est utilisé par exemple par le programme `makepkg`, le créateur de paquets de Pacman, pour obtenir les fichiers à installer dans un dossier afin de créer le paquet. La sortie est alors :

```
$ make install DESTDIR=/mnt prefix=.local

Error: DESTDIR is specified and portable_install is not,
but not all installation paths are absolute.
```

La création de cette interface n'a pas posé de problème particulier. Pour que les variables de Make soient appréhendées par Bash, j'ai défini une petite fonction `shEsc` dans le Makefile qui (similaire aux fonctions `quote()` de beaucoup de langages et bibliothèques) se charge de délimiter la valeur avec des apostrophes, et d'échapper les apostrophes qui s'y trouvent en sortant

celles-ci des apostrophes (le langage ne permet pas l'échappement entre apostrophes) :

gengraph-repo/Makefile (extrait)

```
39 shEsc = $(1)='$(subst ','\'',$($(1)))'
```

La coloration syntaxique de mon éditeur n'est pas bonne (il faut encore que j'envoie une amélioration à GtkSourceView), mais dans cette expression, Make voit et ne voit que les `$(...)`. Les `$(1)` sont remplacés par l'argument : le premier, pour donner son nom à une variable de l'interpréteur de commandes ; le second, pour récupérer la variable du Makefile. Dans cette variable, `$(subst find,replace,text)` remplace tous les `'` par `'\''`. Notons que dans `$(...)`, Make voit le premier mot, puis les arguments peuvent contenir n'importe quels caractères, sauf la virgule qui sert à les séparer (pour mettre une virgule dans un argument, il faudrait utiliser une variable). Petit exemple d'expansion sur une variable `shtick` qui contiendrait `ab'cd` :

```
$(call shEsc,shtick)
shtick='$(subst ','\'',$(shtick))'
shtick='ab'\''cd'
```

La procédure d'installation essaie de prendre les fichiers `README.md` et `LICENSE.md` qui correspondent à la langue de la session, dans la variable d'environnement `$LANG`. Dans mon cas, elle vaut `fr_FR.utf8`, donc pour le lisez-moi, l'installateur va chercher `README.fr_FR.md`, `README.fr.md` et `README.md`. Il prend le premier qui existe, en l'occurrence `README.fr.md`. Cela n'est pas encore fait pour le manuel et l'historique, car puisqu'ils n'existent pour l'instant que dans une seule langue, je n'ai pas pris la peine de coder la prise en compte de la variable `$LANG` pour choisir les fichiers à lire dans GenGraph. Il faudrait aussi revoir ce processus pour qu'il permette d'installer plusieurs langues. Il me semble judicieux de ne pas installer simplement le manuel dans toutes les langues, car il a un certain poids : le texte pèse 270 ko, la version HTML pèse 450 ko. L'exécutable `gengraph`, lui, pèse 420 ko. Ne proposer que l'installation de toutes les langues conduirait GenGraph à peser plusieurs mégaoctets dès qu'il y en aurait quelques-unes, ce qui pourrait, je pense, limiter un peu son adoption. Toutefois, si on offre des distributions du logiciel compilé, l'utilisateur ne pourra pas forcément sélectionner ses langues si facilement, j'imagine. Une solution pour le manuel au format texte pourrait être de le compresser à l'installation, et d'ajouter dans Clih la capacité de lire un fichier compressé (les commandes `man` et `less` sont capables de lire des fichiers compressés).

Les variables utilisées pour l'installation sont les suivantes :

gengraph-repo/Makefile (extrait)

```
26 PREFIX := /usr
27 pkgname := gengraph
28 # GNU standard variables
29 prefix := $(PREFIX)/local
30 bindir := $(prefix)/bin
31 datarootdir := $(prefix)/share
32 datadir := $(datarootdir)
33 docdir := $(datarootdir)/doc/$(pkgname)/
34 # Other stuff
35 binpath := $(bindir)/$(pkgname)
36 ourdatadir := $(datadir)/$(pkgname)/
37 completion = $(if $(portable_install),,bash)
```

`PREFIX` en majuscules est une variable que j'ai vue dans Termux sous Android. En exécutant `make install -e`, l'utilisateur peut utiliser la variable d'environnement PREFIX pour définir cette variable. En effet, les affectations dans le Makefile sont ignorées si la variable avait déjà une valeur à l'origine. `pkgname` est une variable du fichier `PKGBUILD`, que j'ai également utilisée ici pour donner un moyen d'utiliser un autre nom que `gengraph` pour le logiciel. `binpath` et `ourdatadir` servent également à cela. J'ai en effet eu un peu peur que les utilisateurs aient déjà un autre programme qui s'appelle GenGraph. Le programme de Cyril Gavoille est bien référencé, mais on trouve aussi un module Python qui s'appelle GenGraph. Et en tapant `gengraph` sous Debian lorsque le programme n'est pas installé, il m'est proposé d'installer un paquet qui contiendrait un programme à ce nom :

```
$ gengraph

Command 'gengraph' not found, but can be installed with:

apt install ncc
Please ask your administrator.
```

La complétion est installée par défaut seulement pour une installation non portable. Et pour elle, il n'est pas possible de choisir un autre nom que `gengraph` ; la complétion ne fonctionnera pas si l'exécutable n'a pas ce nom. Une fois que le GenGraph de Cyril Gavoille sera mieux implanté, je pense qu'on pourrait supprimer ces possibilités de changer le nom.

La complexité a surtout été dans le fait de permettre à GenGraph et à l'outil de complétion de trouver le manuel. Dans certaines bibliothèques telles que wxWidgets et Qt, il existe des classes nommées StandardPaths qui permettent d'obtenir les chemins des dossiers standards. Cependant, cela ne m'a semblé ni tout à fait générique ni tout à fait fiable. En cherchant sur Internet comment

faisaient les programmes en général, j'ai vu que la solution est en fait d'écrire les chemins directement dans l'exécutable. Pour l'installation, le programme refait donc l'édition de liens, avec un code généré à la place de celui du fichier `config.c` (dont les valeurs sont celles d'une installation portable plate, sans niveaux de dossiers) :

gengraph-repo/Makefile (extrait)

```
169         mkdir -p "`dirname "$$DESTDIR$$binpath"`" &&
170             bindir=`dirname "$$binpath"` &&
171             pathToRoot=`realpath --relative-to="$$DESTDIR$$bindir" "$(if $(DESTDIR),$$DESTDIR,.)"` &&
172             esc_pathToRoot=`cEsc "$$pathToRoot"` &&
173             esc_binPath=`cEsc "$$binpath"` esc_dataPath=`cEsc "$$ourdatadir"` esc_docPath=`cEsc "$$docdir"` 
174             echo "#include <gengraph-cli/config.h>
175                 AppPaths APP_PATHS = { .toRoot = \"$$esc_pathToRoot\",
176                 .bin = \"$$esc_binPath\", .data = \"$$esc_dataPath\", .doc = \"$$esc_docPath\",
177                 .relative = $(if $(portable_install),true,false) };" |
```

L'entrée standard est incluse dans l'édition de liens grâce aux arguments `-x c -` (`-` désigne l'entrée standard, `-x c` indique qu'elle est en langage C). Notons que pour obtenir le chemin allant de l'exécutable vers la racine du projet (utile pour une installation portable), j'ai préféré utiliser la commande `realpath`, qui est GNU mais pas POSIX. `cEsc` est une fonction dans le langage de l'interpréteur de commandes qui fait l'échappement pour le C, en utilisant `sed` pour rajouter des `\` devant les `\` et ". Help (le module de GenGraph CLI) utilise donc la variable `APP_PATHS.data` pour trouver le manuel au format de Clih, et `APP_PATHS.doc` pour trouver le manuel au format HTML. Elle utilise de plus `APP_PATHS.toRoot` si `APP_PATHS.relative` vaut vrai.

C'est un peu dommage de devoir refaire l'édition de liens, qui peut être une opération couteuse (pour de très gros projets, c'est une opération qui prend vraiment du temps et c'est celle qui requiert le plus de mémoire). Peut-être que cela pourrait être évité en compilant les fichiers autres que `config.c` dans une bibliothèque (statique ou partagée) avant d'intégrer cette bibliothèque à un exécutable. En l'occurrence, l'édition des liens pour GenGraph est très rapide, donc cela ne m'a pas préoccupé.

Pour que la complétion trouve le manuel, et aussi son programme, des variables sont ajoutées à la fin du script installé.

La procédure de désinstallation ne fait que des `rm -f` et `rm -rf`, ainsi que des `rmdir` pour supprimer les dossiers de la complétion pour Bash s'ils ne contenaient rien d'autre que les fichiers de GenGraph :

```
$ make uninstall prefix=~/.local

Uninstallation configuration:
   Remove /home/frigory/.local/bin/gengraph
   Remove directory /home/frigory/.local/share/gengraph/
   Remove directory /home/frigory/.local/share/doc/gengraph/
   Remove /home/frigory/.local/share/bash-completion/{completions,helpers}/gengraph
Proceed [y/N]? y

Done.
```

Cela n'affiche rien de plus, même si rien n'a été supprimé.

Une autre chose que j'ai voulu faire a été le développement d'un petit script d'installation pour un gestionnaire de paquets. Je l'ai fait pour Pacman, parce que je me suis dit que ce serait plus simple que pour les autres gestionnaires de paquets, et aussi parce que ma distribution utilise Pacman et que j'avais déjà pas mal étudié ce gestionnaire de paquets très transparent. Il suffit en effet simplement de compléter un petit patron qui est un script Bash avec des variables (certaines étant de type tableau) et des fonctions.

Les scripts pour Pacman s'appellent couramment `PKGBUILD`. Pour lancer l'installation, il faut être avec le terminal dans le dossier contenant ce fichier, et exécuter `makepkg`. Une archive du projet est téléchargée, la compilation est effectuée et un paquet est produit (avec l'extension `.pkg.tar.zst`). L'installation peut alors être réalisée avec `pacman -U gengraph-*.pkg.tar.zst`.

```
$ makepkg

[…]
==> Création terminée : gengraph master-1 (jeu. 31 août 2023 21:08:18)

$ sudo pacman -U gengraph-master-1-x86_64.pkg.tar.zst

[sudo] Mot de passe de sylchiron :
chargement des paquets…
résolution des dépendances…
recherche des conflits entre paquets…

Paquets (1) gengraph-master-1

Taille totale installée :  1,17 MiB
```

```
:: Procéder à l'installation ? [O/n]
(1/1) vérification des clés dans le trousseau                      [-----------------------------------
(1/1) vérification de l'intégrité des paquets                      [-----------------------------------
(1/1) chargement des fichiers des paquets                          [-----------------------------------
(1/1) analyse des conflits entre fichiers                          [-----------------------------------
(1/1) vérification de l'espace disque disponible                   [-----------------------------------
:: Traitement des changements du paquet…
(1/1) installation de gengraph                                     [co  o  o  o  o  o  o  o  o  o  o  o  o
(1/1) installation de gengraph                                     [-----------------------------------
Dépendances optionnelles pour gengraph
    less: pager for help within the terminal [installé]
    graphviz: visualization of graphs [installé]
:: Exécution des crochets (« hooks ») de post-transaction…
(1/1) Arming ConditionNeedsUpdate...

$ sudo pacman -R gengraph

vérification des dépendances…

Paquets (1) gengraph-master-1

Taille totale supprimée :  1,17 MiB

:: Voulez-vous désinstaller ces paquets ? [O/n]
:: Traitement des changements du paquet…
(1/1) désinstallation de gengraph                                  [co  o  o  o  o  o  o  o  o  o  o  o  o
(1/1) désinstallation de gengraph                                  [----C  o  o  o  o  o  o  o  o  o  o  o
(1/1) désinstallation de gengraph                                  [-----------------------------------

:: Exécution des crochets (« hooks ») de post-transaction…
(1/1) Arming ConditionNeedsUpdate...
```

Le script du dépôt est fait pour installer la toute dernière version de développement du projet contenue dans la branche « master ». Pour obtenir un script pour une version particulière, il suffit de remplacer la valeur de la variable `pkgver` par l'étiquette de la version. Il faut aussi indiquer la somme de contrôle à la place de `sha256sums=('SKIP')` ; elle peut être obtenue automatiquement en exécutant `makepkg -g` dans le dossier contenant le fichier `PKGBUILD` :

```
$ makepkg -g

==> Récupération des sources…
  -> Téléchargement de gengraph-master.tar.gz…
[…]
==> Génération des sommes de contrôle des sources…
sha256sums=('377f575cff9a899679911d1c88c697daab97f78ec01d248433b82cbf082504a5')
```

Arch Linux est la distribution Linux la plus récente (parue en 2002) à avoir une branche de dérivées un peu imposante (très loin derrière Debian et Red Hat toutefois) : plus d'une vingtaine de dérivées directes, et près d'une vingtaine de dérivées (directes ou non) encore actives. Son gestionnaire de paquets peut être installé sous d'autres distributions (paquet « pacman-package-manager » sous Debian par exemple). Il est également utilisé par MSYS2, plateforme de développement pour Windows offrant un environnement de style Unix.

3.2.5. Les plateformes utilisées pour faire les tests

Elles sont indiquées dans le lisez-moi : le système d'exploitation, son numéro de version majeure, et le type d'architecture. Le fonctionnement sur le système d'exploitation de Cyril Gavoille (macOS Ventura) est peut-être à revérifier.

J'ai principalement travaillé au CREMI sous Debian Buster, et chez moi sous Manjaro, des distributions GNU/Linux, toujours avec un processeur 64 bits. J'ai également fait le test sur mon vieil ordinateur portable en 32 bits ; je n'ai pas réussi à faire fonctionner la version 32 bits de Manjaro (très peu maintenue), j'utilise donc à la place Arch Linux 32 bits. Je me suis amusé à tester (et faire fonctionner) GenGraph sur d'autres plateformes, plus originales : Android utilisant Termux, Windows XP SP3 et Windows 10.

Termux est une application pour Android offrant une distribution Linux très complète, qui fonctionne très bien pour tout ce qui se fait dans le terminal classique. Elle utilise le gestionnaire de paquets APT de Debian, avec une petite surcouche. Comme compilateur C, elle ne fournit pas GCC, seulement Clang. Je l'ai installée sur mon vieux téléphone, un Samsung Galaxy S4 datant de 2014, qui a un processeur ARM 32 bits. Tout a très bien fonctionné, je n'ai pas eu plus de problèmes qu'avec les autres plateformes sous 32 bits. Les seuls soucis particuliers ont été d'étranges avertissements mentionnés plus haut, à désactiver avec `-Wno-cast-function-type-strict`, ainsi que l'ouverture d'un fichier par une application externe (avec `-visu` et `-html`) qui n'a fonctionné qu'en désactivant une option de sécurité sous Termux. J'ai été capable de montrer facilement à des gens des graphes générés par GenGraph sur mon téléphone.

Sous Windows XP, j'ai utilisé Cygwin, un logiciel développé depuis longtemps, qui permet, comme Termux pour Android, d'avoir comme une distribution Linux, à l'aide d'une bibliothèque leur permettant d'être compilés en programmes Windows et de fonctionner sous Windows. Cygwin n'est plus mis à jour que pour les Windows 64 bits à partir de Windows 7. Pour Windows XP 32 bits, il faut d'abord télécharger l'installateur 32 bits (qui demeure mis à jour), puis utiliser une option pour obtenir les paquets de Cygwin pour Windows XP SP3, comme expliqué dans cette section. Sans le Service Pack 3 de Windows XP, il n'est proposé qu'une version bien plus ancienne. J'ai ainsi obtenu GCC 5.4 datant de 2016. Le seul souci particulier que je me rappelle avoir eu avec cet environnement est le refus de GCC de faire quoi que ce soit s'il reçoit une option `-W` qu'il ne connait pas.

Ces deux environnements demandent tous les deux beaucoup d'espace disque, c'est-à-dire plusieurs centaines de mégaoctets. Rien que l'application Termux pèse 130 Mio. De plus, les paquets ne peuvent pas être installés sur carte SD (sauf peut-être en bidouillant, ce que j'ai essayé de faire, sans succès), car Android ne permet pas de la traiter avec un système de fichiers prenant en charge les permissions. J'ai utilisé les paquets fournis par Termux et Cygwin pour installer Graphviz ; je n'ai malheureusement pas réussi à faire fonctionner une distribution officielle de Graphviz sous Windows XP.

On peut noter que, si on peut se passer de Graphviz et d'un bon rendu de l'aide intégrée, on peut obtenir des distributions de GenGraph très légères pour Windows XP et Android :

- Sous Windows XP, il suffit de deux bibliothèques partagées (DLL) de Cygwin pour permettre au programme GenGraph compilé avec Cygwin de fonctionner. L'aide intégrée fonctionne relativement avec le terminal textuel de Windows par défaut : on a bien les couleurs, par contre la plupart des caractères Unicode ne fonctionnent pas, et je n'ai pas non plus réussi à obtenir un vraiment bon fonctionnement de `less` (mais on peut utiliser `more` à la place).
- Bien que je ne l'ai pas expérimenté, le programme GenGraph compilé sous Termux devrait très bien fonctionner sans que Termux soit installé, avec un autre émulateur de terminal. En effet, le programme ne semble pas dépendre de bibliothèques partagées de Termux, mais seulement des bibliothèques C natives d'Android (Bionic). On pourrait également compiler GenGraph avec le NDK d'Android, mais cela ne semble pas être aussi aisé qu'utiliser Termux.

Termux et Cygwin sont tous les deux sous licence GNU GPL, avec quelques exceptions.

Ce que je trouvais très intéressant pour démontrer la portabilité de GenGraph était de le faire fonctionner sur un système d'exploitation ancien. Le notoire Windows XP est sorti en 2001 et son Service Pack 3 a été publié en 2008. Cygwin étant maintenu pour les versions de Windows plus récentes, cela était suffisant pour moi pour penser que le fonctionnement serait le même avec les versions actuelles de Windows. Néanmoins, j'ai voulu faire le test avec MSYS2, un logiciel équivalent à Cygwin, et qui inclut Cygwin en partie, notamment parce que je ne comprenais pas bien s'il prenait en charge POSIX ou pas. Ce n'est en fait le cas qu'avec l'environnement « MSYS » qui est basé sur Cygwin, et je n'ai donc pas pu compiler GenGraph avec les autres environnements car il a quelques dépendances à POSIX. MSYS offre d'autres environnements, avec les mêmes paquets recompilés, pour se baser sur les bibliothèques Windows natives plutôt que sur Cygwin (les anciennes bibliothèques Windows « msvcrt », et les nouvelles « ucrt »).

On peut obtenir la durée des compilations à l'aide de la commande `time`. La compilation du projet complet prend :

- au CREMI : 2 s sur Quirrel, 4,5 s sur Trelawney ;
- sur mon ordinateur fixe personnel : 10 s avec GCC, 9 s avec Clang ;
- sur mon téléphone : 1 min ;
- sur mon vieil ordinateur portable : 1 min 15 s.

Sans GNU Make (avec `cc -o gengraph -I src `find src -name *.c` -lm`), les durées sont :

- au CREMI : 6 s (sur Quirrel et Trelawney) ;
- sur mon ordinateur fixe personnel : 6,5 s ;
- sur mon téléphone : 45 s ;
- sur mon vieil ordinateur portable : 40 s.

La compilation du fichier source unique de GenGraph v5.4 prend en comparaison :

- au CREMI : 1 s sur Quirrel, 3,5 s sur Trelawney ;
- sur mon ordinateur personnel : 2 s avec GCC, 1 s avec Clang ;
- sur mon téléphone : 11 s ;
- sur mon vieil ordinateur portable : 11 s.

La taille du code n'ayant pas tellement augmenté, et l'édition de liens étant une opération très courte, ces chiffres invitent à penser que les compilateurs pourraient vraiment gérer de façon beaucoup plus optimisée la compilation de plusieurs fichiers.

3.3. Travail sur l'aide

Ce travail a abouti à la réalisation de Clih, une bibliothèque qui est pour l'instant un sous-projet de GenGraph mais peut en fait être utilisée pour n'importe quel projet ayant besoin de documenter ses mots clés. Clih se base sur un fichier formaté avec la syn-

taxe propre à Clih (qui est très simple et qui copie souvent des syntaxes existant par ailleurs), et peut également gérer un fichier d'historique (dont le format est le même, mais les titres de section sont gérés un peu différemment). Clih offre les fonctionnalités suivantes :

- l'écriture colorée (avec les codes ANSI) du manuel et de l'historique, avec mots non coupés sur deux lignes ;
- la production d'une version HTML formatée du manuel et de l'historique, avec un sommaire ;
- l'écriture d'une section particulière du manuel, de toutes les sections contenant un texte donné, ou encore de tous les titres de section ;
- le formatage de texte en utilisant la syntaxe de Clih (plutôt que la syntaxe ANSI) ;
- la complétion pour l'interpréteur de commandes Bash en mode interactif.

L'API de Clih est utilisée par GenGraph pour son module Help, dont je vais également parler dans cette section. L'option `-print` utilise aussi Clih pour le formatage.

Cette section couvre aussi la modification des routines de GenGraph CLI pour la lecture de ses arguments pour ce qui concerne l'aide, et les changements et améliorations apportées à la fonctionnalité de rapport d'erreur.

3.3.1. Traitement du code préexistant

Avant de me lancer dans l'écriture de la bibliothèque à partir de zéro, j'ai passé un certain temps à étudier les fonctions existantes que j'ai placées dans le module CLI, et j'en avais modifié le contenu (que j'avais placé dans le module CLI), notamment pour qu'il fonctionne avec le manuel déplacé dans un autre fichier, pour remplacer les nombreux appels à `strcat()` par des appels à `sprintf()`, et pour factoriser les fonctions `Visu()`, `Visuh()` et `MainCC()`. Du fichier accessible à ce lien, seule la fonction `MakeCMD()` a subsisté (renommée en `GetInvocationCmd()` et ne prenant plus aucun paramètre) ; tout ce qui suit a été supprimé, rien n'a été gardé.

L'utilisation de `sprintf()` à la place de `strcat()` a consisté à remplacer ce genre de code :

gengraph-v5.4/gengraph.c (extraits)

```
19270       strncat(fin,ARGV[t],k);
19271       strcat(fin,"[^");
19272       strncat(fin,ARGV[t]+k,1);
19273       strcat(fin,"]|");

...  […]

19293     strcat(strcat(strcat(s,"sed -nE '/^[.]{4}"),mot),"($|[ ])/,/^$/p'|");
```

par celui-là :

```
      pos += sprintf(fin + pos, "%.*s[^%c]|", k, ARGV[t], ARGV[t][k]);
[…]
    pos += sprintf(s + pos, "sed -nE '/^[.]{4}%s($|[ ])/,/^$/p'|", mot);
```

Et pour tout dire, je ne vois aucun intérêt à l'existence de `strcat()`. D'une part, il semble que `strcat(dest, src)` est juste un raccourci pour `strcpy(dest + strlen(dest), src)`, et d'autre part, je n'en ai jamais l'utilité, puisqu'à chaque fois, la longueur de `dest` est en fait connue à l'avance (sinon, il est nécessaire de la recalculer à chaque fois, ce qui ralentit considérablement l'exécution), et donc je me retrouve à écrire `strcpy(dest + destLen, src)` plutôt qu'à utiliser `strcat()` quand j'ai besoin de faire une concaténation. `strcat()` aurait eu plus d'intérêt pour moi si elle faisait une réallocation, mais ce n'est pas le cas : elle écrit dans un tableau qui doit déjà être alloué, et dans tous les contextes où je sais quelle longueur j'ai allouée, je sais toujours aussi combien de caractères j'ai déjà écrits, donc je ne vois aucun intérêt à `strcat()`. Dans la bibliothèque standard du C, il y a ainsi certaines fonctions sur les chaines de caractères qui ne sont vraiment pas utiles, et d'autres qui manquent beaucoup, notamment `has_prefix()` que j'ai développée dans Clih et qui est également utilisée un peu par GenGraph et GenGraph CLI.

`sprintf()` et `sscanf()` en revanche, sont des fonctions très puissantes, qui donnent accès aux capacités de nombreuses autres, sans surcout significatif. Elles permettent souvent de faire beaucoup de choses avec du code concis, et en plus l'essentiel de leur syntaxe est bien connue et utilisée ailleurs qu'en langage C, par exemple avec la commande POSIX `printf`, le formatage avec l'opérateur `%` en Python (syntaxe obsolète), ou encore les modules `Printf` et `Scanf` du langage OCaml. C'est pourquoi j'encourage à les utiliser intensivement. `snprintf()` permet d'obtenir la longueur du texte à écrire pour pouvoir faire l'allocation avant d'écrire effectivement.

En fait, il m'arrive même parfois d'utiliser `sprintf()` seulement pour faire un `strcpy()` et un `strlen()` (POSIX.1-2008 offre `stpcpy()` pour cela) :

```
pos += sprintf(buf + pos, "%s", str);
```

La syntaxe est relativement concise et cela ne fait probablement qu'une seule itération sur `str`. Le surcout engendré par l'utilisation de la chaine de format `%s`, qui est parcouru par `sprintf()`, est probablement très faible, car il ne s'agit que de voir le `%` et de faire le branchement correspondant au `s` avant de voir que le format est terminé. En plus de ça, on se rappelle qu'on fait du C, donc que l'exécution va naturellement bien plus vite qu'avec les autres langages, et donc qu'on peut largement se permettre un surcout de ce genre, qui nous permet de nous satisfaire du C au lieu de nous faire désirer un langage plus nécessiteux.

Concernant l'aide HTML, j'ai essayé de lire le fichier `do_help.awk` dans l'idée d'en reprendre au moins la logique. Mais j'ai eu du mal à la saisir. Le fichier contient essentiellement une grosse boucle `for` itérant sur les lignes à formater, et traitant à la fois le manuel et l'historique (alors qu'ils n'avaient presque aucun formatage en commun). Il y avait une logique étrange d'utiliser constamment `continue` plutôt que `else`. Et le code utilisait également `substr()` avec `RSTART` et `RLENGTH`, au lieu d'utiliser la capacité de `match()` à renvoyer les sous-chaines correspondant aux parenthèses dans l'expression rationnelle (ce qui est une fonctionnalité standard des moteurs d'expressions rationnelles). Après un certain temps passé à essayer de comprendre ce code, j'ai fini par craquer sur la question de l'articulation entre les instructions utilisées par le manuel et celles utilisées par l'historique. J'ai donc tout jeté et écrit mon propre script Awk avant de passer à du code C, jugeant que j'avais suffisamment bien compris le système pour pouvoir en reprendre toutes les fonctionnalités. À en croire l'historique de GenGraph, `do_help.awk` avait été écrit entre 2016 et 2018.

3.3.2. Les fonctions du noyau

Le noyau correspond au module Parsing de Clih.

3.3.2.1. Le type `Clih_ParserBuf` pour les traitements

GenGraph v5.4 utilisait à divers endroits des tableaux de caractères statiques pour renvoyer des chaines de caractères — car, comme on le sait, le C ne peut renvoyer que données de taille statique, ou alors elles seront alloués sur le tas et il faut s'ennuyer à les désallouer après les avoir reçues. J'ai supprimé la plupart des occurrences de ces tableaux, car le fait d'occuper ainsi la mémoire statique pour des fonctionnalités un peu partout et pas forcément utilisées me semble être une mauvaise idée. Je ne souhaitais cependant pas non plus m'ennuyer avec de l'allocation dynamique ; Clih n'en contient pas la moindre trace. Ma solution a été de définir une structure contenant pour seul champ un tableau de caractères :

gengraph-repo/src/clih/clih.h (extrait)

```
 9 #define CLIH_BUF_LEN 4096
10
11 typedef struct {
12   char s[CLIH_BUF_LEN];
13 } Clih_ParserBuf;
```

Cette structure `Clih_ParserBuf` est utilisée un peu partout dans Clih, notamment pour la lecture des fichiers ligne par ligne ainsi que pour la fonction `replace()` présentée dans la section suivante. Dans GenGraph CLI, j'ai utilisé une structure équivalente nommée `cmd_string` (quatre fois moins grosse).

Clih se sert de cette structure pour stocker des paragraphes ; ainsi, les choses vont certainement très mal se passer si un paragraphe se retrouve à contenir au moins 4096 caractères, y compris après conversion. Actuellement, les paragraphes de `MANUAL.txt` semblent toujours faire moins d'une quinzaine de lignes, chacune faisant moins de 80 caractères, et en plus l'indentation (de 9 espaces en général) n'est pas conservée dans le tampon, donc la limite est loin d'être atteinte ; j'ai préféré prendre large.

On peut craindre que l'utilisation de cette structure conduise à des copies fréquentes de 4096 octets. Cependant, d'après mes recherches, si on fait attention à ce que cela puisse être évité, cela n'est pas censé arriver. Je le savais grâce à ma connaissance du C++, où cette question est encore plus importante puisqu'elle est susceptible de faire intervenir des fonctions définies par l'utilisateur du C++ : le constructeur par copie et l'opérateur d'affectation. Ainsi, notamment, on peut initialiser un `Clih_ParserBuf` avec la valeur renvoyée par une fonction. Par contre, plutôt que faire une affectation directe d'un `Clih_ParserBuf` à un autre, il faut utiliser `strcpy()` sur le champ (qui certes fera une copie, mais de beaucoup moins de 4096 octets en général). Par exemple, avec une fictive fonction `format()` :

```
Clih_ParserBuf buf1 = format(text), buf2;
strcpy(buf2.s, buf1.s); // Et pas : buf2 = buf1
```

S'il ne s'agit pas d'initialisation, il vaut mieux utiliser `strcpy()`. Car là, l'espace de la variable est déjà alloué dans la pile (à une certaine profondeur), et le compilateur ne peut pas demander à la fonction appelée d'écrire sa valeur de retour directement dedans. Ainsi, le code C++ suivant :

```
#include <cstdio>
using namespace std;

struct Test {
```

```
   Test() = default;
   Test(Test const & test) {
      printf("+ Copie\n");
   }
   Test & operator=(Test const & test) {
      printf("+ Affectation\n");
      return *this;
   }
};

Test fct()
{ return {}; }

void fct2(Test test)
{}

int main()
{
   Test test1 = fct(), test2;
   printf("Après initialisation\n");
   test2 = test1;
   printf("Entre deux affectations\n");
   test1 = fct();
   printf("Appel avec paramètre\n");
   fct2(fct());
   printf("Témoin\n");
   Test test3 = test2;
}
```

affiche :

```
Après initialisation
+ Affectation
Entre deux affectations
+ Affectation
Appel avec paramètre
Témoin
+ Copie
```

Il n'y a pas de copie à l'initialisation ni au passage d'une valeur de retour à une fonction, mais il y a bien deux affectations. En pratique, je n'ai jamais eu besoin de me préoccuper des cas d'affectation hors initialisation : en effet, le contenu de mes tampons de type `Clih_ParserBuf` n'est jamais écrasé, il ne fait que subir des ajouts, ou simplement je passe directement le tampon renvoyé à une autre fonction (notamment `printf()`).

3.3.2.2. La fonction pour les substitions : `replace()`

Dans le noyau, j'ai notamment voulu écrire une fonction un peu grosse, offrant une fonctionnalité élaborée pour le remplacement de sous-chaines, pour la mise en place des styles au sein des paragraphes. Il s'agit de la fonction `replace(state, data)`. Contrairement à `gsub()` (substitution globale) d'Awk, je voulais que ma fonction fasse les remplacements en une seule itération du texte. Elle ne prend pas non plus en charge les expressions rationnelles. Ainsi, elle est finalement plus proche de `str_replace()` du PHP, mais va tout de même plus loin.

Le second paramètre de `replace()` est une structure de type `ReplacementData`, qui contient essentiellement un tableau de structures indiquant les remplacements à faire. Comme le C ne permet pas facilement de saisir à la fois des données et leur longueur, j'ai défini une macro pour permettre de définir plus facilement des structures de ce type. Celle qui est utilisée pour le format HTML est à la fois complète (elle utilise toutes les fonctionnalités de `replace()`) et parlante :

gengraph-repo/src/clih/manual_html.c (extrait)

```
7  static InitReplacementData(delimiters, (
8      { CLIH_CODE, "<code>", "</code>" }, { CLIH_QUERY, "", "" },
9      { CLIH_SUP, "<sup>", "</sup>" }, { CLIH_SUB, "<sub>", "</sub>" },
10     { CLIH_BOLD, "<strong>", "</strong>" }, { CLIH_ITALIC, "<em>", "</em>" }, { CLIH_BIG, "<big>", "</big>" }
11     { CLIH_UNDERLINE, "<u>", "</u>" }, { CLIH_STRIKE, "<s>", "</s>" },
12     { CLIH_WARNING, "<span class=\"warning\">", "</span>" }, { CLIH_ERROR },
13     { CLIH_LINK_OPEN, "<a href=\"%s\">", "</a>", &escapeHtml }, { CLIH_LINK_CLOSE },
14     { CLIH_ANCHOR_OPEN, "<a href=\"#%s\">", "</a>", &genHtmlId, .escapeAfter = true }, { CLIH_ANCHOR_CLOSE, .
15     { CLIH_UL_LI, CLIH_UL_LI }, { CLIH_OL_LI, CLIH_OL_LI },
16   ), "\\", &escapeHtml);
```

Ce code initialise la variable `delimiters` en la déclarant statique et en lui donnant le type `ReplacementData`. Le premier para-

mètre est un tableau, dont les éléments peuvent contenir jusqu'à trois chaines de caractères, un pointeur vers une fonction et un éventuel `escapeAfter = true` ; le type correspondant s'appelle `ReplUnit` (mais ce nom n'a pas à être utilisé directement). Les éléments de ce tableau sont donnés entre parenthèses et pas entre accolades, à cause d'une limite du système de macros du préprocesseur du C, expliquée dans la section Refonte de l'analyse des instructions.

Comme on peut s'en douter, la première chaine de caractères contient l'élément à remplacer, qui ici est (sauf pour les deux derniers) un délimiteur de la syntaxe de Clih. Les deux chaines suivantes sont les délimiteurs d'ouverture et de fermeture dans le langage cible. Si le délimiteur de fermeture est spécifié, `replace()` utilise un champ booléen de `ReplUnit` pour alterner entre délimiteur d'ouverture et de fermeture chaque fois qu'il voit la chaine à remplacer ; sinon, c'est toujours la même chaine de remplacement qui est spécifiée. Si elle-même est absente, alors le comportement à avoir est le même que pour la chaine précédente (en remontant éventuellement plusieurs fois de cette manière). Ainsi, `]]` n'est rien d'autre qu'un synonyme de `[[`, utilisé pour faire plus joli à la fin des liens.

La fonction éventuellement donnée en quatrième paramètre des `ReplUnit` est appelée pour formater le contenu entre les délimiteurs et le placer à l'endroit du `%s` dans le délimiteur d'ouverture. Cette fonctionnalité un peu bizarre et finalement pas utilisée pour autre chose que les liens demande à `replace()` de se souvenir de la position du délimiteur d'ouverture dans la chaine source et dans la chaine destination, puis de faire l'insertion, dans la chaine destination à l'emplacement du délimiteur, du texte source formaté avec la fonction (dans la chaine destination, ce texte source a déjà potentiellement subi du formatage). Pour faire cela, `replace()` fait deux `memcpy()` et deux `sprintf()`.

Après le tableau de `ReplUnit`, il y a deux autres paramètres, tous les deux optionnels : une chaine utilisée pour l'échappement des textes habituellement remplacés, et une fonction pour appliquer un formatage au texte qui n'est pas substitué. Ainsi, ce texte subit un échappement des caractères HTML spéciaux (comme avec `htmlspecialchars()` de PHP), réalisé par `escapeHtml()`. Le même échappement est réalisé pour les liens entre `[[[` et `]]]`. On peut remarquer que le texte de ces liens et l'adresse du lien (j'entends : dans `<a href="adresse">texte</a>`) ne donneront pas le même résultat si le texte source contient des sous-chaines à remplacer : le remplacement sera fait dans le texte mais pas dans l'adresse. Cela pourrait être corrigé en donnant `&htmlFormat` à la place de `&escapeHtml` pour le format des adresses, mais cela n'aurait aucun intérêt.

Toutes les fonctions à placer dans les objets `ReplacementData` (telles que `escapeHtml()` et `genHtmlId()`) doivent être de type `TextProcessor`, qui est défini comme suit :

gengraph-repo/src/clih/parsing.h (extrait)

```
42 typedef Clih_ParserBuf TextProcessor(char const * text);
```

Les `•` et `#.`, qui en début de paragraphe indiquent un élément de liste à puces, et qui sont ici remplacés par eux-mêmes, sont donnés là pour profiter de la fonctionnalité d'échappement : un `\` présent en début de paragraphe évitera la création d'une puce, et sera supprimé au dernier moment par `replace()`.

Je n'ai pas compris tout de suite qu'il fallait que ce soit le cas, mais l'échappement (avec `\` en général) ne conduit l'analyse à ne sauter que le premier caractère suivant la sous-chaine à ne pas remplacer. Au départ, j'avais programmé que cela saute l'entièreté de la sous-chaine. Cependant, si on veut mettre un lien interne entre crochets par exemple, on écrira : `\[[[ancre]]\]`. En fait, j'ai rencontré le problème quand j'ai voulu placer une apostrophe avant `''`, par exemple dans :

gengraph-repo/MANUAL.txt (extrait)

```
3512 ....  ''whexagon p q''
3513
3514         Comme le graphe ''hexagon p q'' sauf que chaque hexagone est
3515         remplacé par une roue de taille 6 (chaque hexagone possède un
3516         sommet connecté à ses 6 sommets). C'est le dual de l'hexagone.
3517         Il possède p·q sommets de plus que l\'''hexagon p q''.
```

Voyant `\''`, `replace()` saute `\'` en écrivant `'`, de sorte à se trouver juste avant `''` et procéder à son remplacement.

Le cas de `\[[[ancre]]\]` est plus compliqué, parce que la dernière barre oblique ne précède pas une chaine à remplacer. Cela est dû au fait particulier que le délimiteur `]]` est le début du délimiteur `]]]`. Les `\` n'ont pas à être échappés par défaut : ils n'ont besoin d'être échappés que s'ils précèdent une sous-chaine qui serait normalement remplacée. Pour le cas de `]]`, j'ai donc mis en place le champ booléen `escapeAfter`, qui peut être mis à `true` pour sauter la chaine d'échappement placée après le délimiteur. Notons aussi que `[[[` et `]]]` doivent absolument être placés avant `[[` et `]]` dans le tableau de la structure `ReplacementData`, car sinon `replace()` verra toujours `[[` et `]]` à la place de `[[[` et `]]]`.

Le délimiteur `''` utilise une autre fonctionnalité de `replace()` : celle qui le conduit à rendre la main après chaque rencontre de texte à substituer (et après que cette substitution a été faite, sauf dans le cas où il y a une fonction), en enregistrant l'état dans le paramètre `state` pour que la substitution puisse être poursuivie, et en indiquant quelle est la sous-chaine à substituer qui vient d'être rencontrée. La fonction `htmlFormat()` s'en sert pour appliquer un formatage au texte situé entre les `''` (on voit que pour cela, on aurait pu songer à donner une fonction en paramètre, mais bref).

Cette possibilité d'intervenir après chaque rencontre de texte substitué n'est généralement pas utilisée, ainsi il y a une fonction

`replaceAll()`, avec laquelle on peut voir comment utiliser `replace()` :

gengraph-repo/src/clih/parsing.c (extrait)

```
163 Clih_ParserBuf replaceAll(char const * text, ReplacementData const * data)
164 {
165   Clih_ParserBuf buf;
166   ReplacementState state = { text, buf.s };
167   while (replace(&state, data) != NULL) {}
168   return buf;
169 }
```

`replace()` s'attend en principe à ce que les chaines à substituer par un délimiteur de début ou de fin apparaissent un nombre pair de fois dans le texte en entrée, pour que chaque délimiteur de début soit suivi par un délimiteur de fin. Pendant longtemps, le traitement se faisait ligne par ligne, comme précédemment avec Awk. Cela posait bien entendu problème pour les liens : si un texte à transformer en lien était coupé sur deux lignes, la fonction de remplacement ne pourrait pas le créer correctement. Pour cette raison et pour d'autres, le traitement se fait maintenant paragraphe par paragraphe, ce qui est bien plus intéressant pour Clih. Lorsque le traitement se faisait ligne par ligne, `replace()` laissait souvent un délimiteur ouvert à la fin d'une ligne, afin de le fermer à la ligne suivante ; sauf pour le cas des liens afin d'éviter de planter. Maintenant qu'il n'est question que de paragraphes, `replace()` met toujours les `isOpen` à `false` lorsqu'elle a atteint la fin de l'entrée, sans se préoccuper d'écrire le délimiteur de fin. Cela demeure nécessaire pour éviter les plantages avec le cas d'un lien qui ne serait pas fermé à la fin d'un paragraphe, et assure une cohérence avec les autres délimiteurs. Le rendu HTML corolaire contiendra une balise non fermée (sauf pour un lien, puisque le délimiteur d'ouverture n'est alors écrit que lorsque le délimiteur de fin est trouvée), ce qui se verra jusqu'à la fin du document (par exemple, toute la fin du document se trouvera dans une balise `<code>`), et l'erreur pourra ainsi être détectée rapidement.

Cette fonction `replace()` me semble assez intuitive lorsqu'on veut commencer à l'utiliser. On peut par la suite découvrir qu'elle offre des fonctionnalités plus avancées, permettant de gérer davantage de cas. Pour les délimiteurs, que ce soit pour la version ANSI, la version HTML ou simplement le texte brut (donc avec les délimiteurs retirés), elle est vraiment pratique. Ce qui est le plus dommage la concernant, c'est probablement son implémentation qui est un tantinet intimidante. Elle n'est pas non plus catastrophique, et on verra que Clih contient malheureusement au moins deux fonctions pires.

3.3.2.3. Les outils pour l'analyse de la syntaxe

Il y a d'abord des fonctions et macros de support, notamment `has_prefix()`. Cette fonction utilise la casse_serpent et non la casse chatMot comme partout ailleurs dans Clih, car c'est une fonction que je veux faire venir d'une mini bibliothèque que je suis en train de réaliser : HandyMacros, qui consiste essentiellement en un fichier d'entête définissant une longue série de macros permettant d'utiliser le C beaucoup plus comme un langage haut niveau. Je souhaite l'utiliser dans tous mes projets futurs en C, mais elle n'est pas encore assez mure. `has_prefix()` n'utilise pas `string.h` : elle a sa propre boucle `while` pour être plus efficace que si elle utilisait `strncmp()`. C'est une fonction qui me manque vraiment souvent et qui est apportée par HandyMacros ; elle a le prototype désirable :

gengraph-repo/src/clih/parsing.h (extrait)

```
83 bool has_prefix(char const * str, char const * prefix); // Fonction de HandyMacros
```

Une autre petite fonction d'analyse est `matchesOneOf()`, qui indique si le début d'une chaine de caractères correspond à un motif parmi une liste de motifs donnée en paramètres, en utilisant la syntaxe de `scanf()`. Cela n'est utilisé que pour les préfixes d'exemples, notamment parce qu'ils peuvent être numérotés.

La petite fonction `trimLineFeed()` supprime un éventuel retour à la ligne se trouvant à la fin d'une chaine de caractères. Elle est utilisée souvent pour se débarrasser du retour à la ligne renvoyé par `fgets()`.

Ensuite, il y a une série de petites fonctions (la plupart sont `inline`) pour reconnaitre les éléments syntaxiques :

- `getHeadingLevel(line)` renvoie le niveau de titre correspondant à `line` le cas échéant, sinon 0 ;
- `isNonPrefix(crt)` indique si le caractère fourni peut être un début de titre de section (si ce n'est pas le cas, c'est soit une partie de préfixe de titre de section, soit une fin de ligne ou de chaine) ;
- `getHintSuffix(line)` renvoie soit un pointeur sur la fin de `line` soit sur son suffixe indiquant que c'est une astuce (cela permet de tronquer ce suffixe tout en ayant vu qu'il est présent) ;
- `isExampleStart(indent, line, &pos)` et `isPreStart(indent, line, &pos)` indiquent si `line` est un début de bloc préformaté, éventuellement de type « exemple », en plaçant dans `pos` la position de la fin du préfixe de ce type de bloc ;
- `isUlStart(indent, line)` et `isOlStart(indent, line)` indiquent si `line` est un début d'élément de liste à puces.

Le paramètre `indent` est un entier, toujours obtenu avec `strspn(line, " ")` (une fonction de `string.h`). Les fonctions ci-dessus ne calculent pas elles-mêmes cette indentation, pour deux raisons : les fonctions qui l'appellent ont toujours déjà cette information (puisqu'elles font une série de tests d'un coup), et aussi parce que des fois elles sont appelées avec l'indentation du paragraphe précédent (parce que les éléments d'une liste à puces doivent tous avoir la même indentation).

Le module Parsing donne aussi directement accès à la liste des préfixes des titres de section : `headingPrefixes` ; et aussi à leurs longueurs : `headingPrefixesLen`. Le nombre de niveaux de titres pris en charge par Clih (à savoir : 5) est donné dans la variable `headingLevelsNb`. La macro `INSTR_FIRST_LEVEL` indique à partir de quel niveau (ce niveau est malheureusement fixe) les titres sont conçus comme des instructions, ce qui est une information utile surtout pour les accessoires et la complétion.

L'objet global prédéfini `rmDelimsData` de type `ReplacementData` permet de supprimer tous les délimiteurs de la syntaxe de Clih dans une chaine de caractères. Cela est utilisé par les accessoires, pour la complétion, et aussi pour la génération des identifiants HTML.

Il y a enfin deux fonctions pour les blocs indentés :

- `printIndented(inputFile, indent, indentToRemove, processor)` affiche les lignes indentées en les passant à travers une fonction de formatage et en supprimant éventuellement une partie de l'indentation (cela est utilisé pour afficher les blocs préformatés) ;
- `getIndented(inputFile, buf, indent)` rajoute à la fin de `buf` les lignes suivantes de `inputFile` qui sont indentées d'au moins `indent` caractères.

3.3.2.4. La génération de toutes les commandes possibles : `getOptionCase()`

Cette fonction horrible analyse une option décrite avec le format présenté dans la section suivante sur la syntaxe, afin de générer toutes les options concrètes qui en résultent. Elle est utilisée pour générer les identifiants HTML, et surtout pour la complétion.

Pour les identifiants, elle ne génère heureusement pas toutes les options possibles. En fait, elle a deux modes : celle qui inclut les mots optionnels, et celle qui ne met un mot que lorsqu'il en faut nécessairement un. La génération d'identifiants utilise ce second mode (appelé mode « canonique »), qui n'inclut pas non plus les variables. Prenons comme exemple le format d'option suivant :

```
-option [variant <v>] one|two <n>
```

En mode canonique, on aura les résultats suivants : `-option one` et `-option two`. Avec les mots optionnels, on aura deux fois plus de résultats et ils seront plus longs :

- `-option variant <v> one <n>` ;
- `-option variant <v> two <n>` ;
- `-option one <n>` ;
- `-option two <n>`.

Je dis que cette fonction est horrible, pour trois raisons. La première est qu'au départ c'était une gentille fonction pour générer un identifiant, prenant en charge très peu de fonctionnalités ; peu à peu, je l'ai faite grossir, elle est devenue longue et lourde (concrètement : 120 lignes, dont 95 dans une boucle `for`) et s'est vraiment mise à ressembler à un assemblage de bidouilles, au point que j'ai renoncé à vraiment prendre du recul sur son fonctionnement. La deuxième raison est que je me targue d'avoir codé Clih en C en évitant vraiment presque tous les calculs inutiles, pour que cette bibliothèque soit particulièrement efficace même si on n'en a pas tellement besoin ; et là, pour la complétion, ce système n'est absolument pas adapté, et bien qu'il fonctionne, il est bien plus lourd que ce qu'il faudrait (c'est expliqué dans la section sur la complétion, plus bas). La troisième raison est que je n'ai même pas réussi à implémenter dans cette fonction la prise en charge de crochets imbriqués du genre `[html|dot-[pdf|svg]]`.

Clih contient ainsi quelques grosses fonctions comme ça, qui font un assez bon travail, mais qui sont vraiment tristes quand on regarde à l'intérieur. Je vais sans doute assez rapidement coder une meilleure fonction pour la complétion.

On peut voir comment `genHtmlId()` utilise `getOptionCase()` pour générer un simple identifiant :

gengraph-repo/src/clih/manual_html.c (extrait)

```
27 static Clih_ParserBuf genHtmlId(char const * text)
28 {
29   Clih_ParserBuf buf1 = replaceAll(text, &rmDelimsData), buf2;
30   getOptionCase(&(OptionCasesState){ buf1.s, buf2.s, .canonical = true, .spaces2us = true });
31   return escapeHtml(buf2.s);
32 }
```

Pour mieux convenir à la génération d'identifiants HTML, `genHtmlId()` se propose de remplacer toutes les espaces par des tirets bas (drapeau `spaces2us`). Comme on le voit, elle prend un seul paramètre, qui est une structure de type `OptionCasesState`. Si on ne veut que le premier résultat, on peut se contenter de passer l'adresse d'un littéral composé de C99, comme cela est fait ci-dessus. Le premier paramètre est le format à lire, le deuxième est le tampon dans lequel écrire le résultat. Pour obtenir la liste des résultats, il suffit d'initialiser une variable de type `OptionCasesState`, et de donner son adresse à la fonction en appelant celle-ci dans la condition d'une boucle `while` : en effet `getOptionCase()` renvoie vrai si elle a écrit un cas d'option, et renvoie faux après qu'elle les a tous donnés.

3.3.3. La nouvelle syntaxe

3.3.3.1. Simplification du format d'origine

Le manuel d'origine recourait à la syntaxe des quatre points `....` bien plus que nécessaire : de nombreuses lignes vides incluaient ces quatre points afin de manifester la continuité de la section d'une instruction de premier niveau. Bref, il m'est apparu que ces très nombreux `....` pouvaient être éliminés, car le code du manuel contenait d'autre part une indentation très rigoureuse, suffisante pour déterminer sa sémantique. Avant de me mettre à complexifier la syntaxe du manuel, j'ai donc commencé par la simplifier. Au bout du compte, je n'ai gardé les `....` que pour les préfixes des titres de section à partir du niveau 3. Ils sont nécessaires à cet endroit car ces titres ont la même indentation que certains paragraphes. En effet, Cyril Gavoille a eu une logique d'augmenter l'indentation du texte en même temps qu'il augmentait celle des titres.

J'ai aussi normalisé l'indentation. Dans le fichier d'origine, elle était variable : de 0 pour le premier niveau de titre, de 3 pour le deuxième niveau, de 4 (avec `....`) pour le troisième niveau, et de 7 pour le quatrième niveau. Il n'y avait pas de cinquième niveau, que j'ai fait apparaitre pour les options `-check routing`, qui jusque-là étaient placées au quatrième niveau comme le parent. Le texte avait toujours une indentation de 7 espaces, sauf après les titres de quatrième niveau ou l'indentation était de 10. Ce chiffre de 7 semble correspondre au niveau de retrait du texte des pages de `man` bien connues. Tout cela m'a semblé très exotique. En même temps, je n'ai pas souhaité changé la logique d'utiliser la même indentation dans le code source du manuel et dans le rendu en ANSI. L'indentation de 2 espaces utilisée dans le code de GenGraph me semblait alors trop faible, et celle de 4 espaces des scripts Awk de Cyril Gavoille était au contraire trop élevée, alors j'ai pris 3, qui était déjà la différence entre l'indentation de certains niveaux de titre, et qui est aussi la largeur de tabulation que j'utilise maintenant dans tous mes projets (convention piquée aux codes de la plateforme d'entrainement France-IOI). Les niveaux d'indentation sont maintenant tous des multiples de 3, bien qu'ils soient écrits en dur dans le module Parsing. Le code `....` demeure utilisé à partir de 6, et est donc suivi d'au moins deux espaces.

3.3.3.2. Les délimiteurs de format

Voici une présentation des délimiteurs utilisables au sein des paragraphes :

- ``` ``code`` ``` : cela délimite du code. C'est notamment utilisé pour les options. L'utilisation des accents graves à cette fin se retrouve notamment dans le Markdown. On l'a aussi en reStructuredText, bien que les accents graves ont alors d'autres usages.
- `''requête''` : cela délimite une instruction principale, non option, dite aussi requête dans le jargon de Clih. Pour GenGraph, cela correspond à un graphe en général, mais parfois à un groupe de graphes. Ce format offre un lot de fonctionnalités : celles de ``` ``[[requête]]`` ```, mais avec la couleur jaune, en évitant de plus d'inclure les variables dans le lien, et en appliquant également la coloration sur un alias sous la forme `(= alias)`. L'alias est mis en italique et dans une couleur plus sombre, et son premier mot est également coloré en jaune et un lien est créé si possible.
  Ces `''` sont le code utilisé pour l'italique dans la syntaxe de MediaWiki (le moteur des wikis de la Fondation Wikimédia, dont fait notamment partie Wikipédia). MediaWiki utilise également `'''` pour le gras, et ce gras n'est bien souvent utilisé dans Wikipédia que pour mettre en emphase le nom de l'article dans le tout premier paragraphe. C'est pourquoi j'ai eu l'intuition de les utiliser pour les instructions principales.
  J'ai créé ce format dans le but de donner un charme particulier aux graphes dans le manuel, car ils sont l'objet central de GenGraph depuis sa création.
- `[[ancre]]` : cela crée un lien vers la section correspondante. Étant donné que pour les instructions principales, la convention est d'utiliser `''`, les `[[]]` sont utilisés pour les options, et également parfois pour les gros titres tout en majuscules, par exemple « USAGE ».
  Ces `[[]]` sont la syntaxe de MediaWiki pour créer des liens internes (ou interwikis).
- `[[[https://www.leweb.com/]]]` : cela crée un lien. Le texte est laissé, et un lien est créé pour permettre son ouverture automatique. Dans le format ANSI, le texte est également coloré. Le fait que le texte soit laissé tel quel permet aux terminaux de créer le lien par eux-mêmes : dans mon cas, je peux l'ouvrir en cliquant dessus tout en appuyant sur Ctrl. Je n'ai pas souhaité que les liens de ce type soient reconnus automatiquement, car je ne sais jamais trop quels caractères inclure dedans, surtout s'agissant de la fin de l'URL ; à mes yeux, c'est plus simple d'utiliser des délimiteurs. De toute manière, cette syntaxe est utilisée très rarement.
  Il y avait une seule URL dans le manuel d'origine : dans la section sur l'option `-format`, vers visjs.org. J'en ai rajouté quelques autres, notamment dans l'entête au tout début du fichier.
  MediaWiki n'utilise pas cette syntaxe pour les URL. Le moteur reconnait ces URL automatiquement, mais il offre aussi une syntaxe, utilisant simplement une seule paire de crochets, permettant de changer le texte : `[https://graphviz.org/ Graphviz]`. Clih, au contraire, utilise encore plus de crochets pour ces URL.
- `/!\Attention/!\` et `(!)Erreur(!)` : cela sert à colorer en jaune/orange et en rouge le texte, et c'est prévu pour les avertissements et les erreurs. Il n'est pas prévu que des erreurs puissent être écrites dans le manuel ; ce format est utilisé par GenGraph pour signaler ses erreurs. Un triangle avec un point d'exclamation est classique pour dire « attention » ; pour l'erreur, j'ai fait un parallèle avec les panneaux routiers d'interdiction, qui sont de valeur plus forte que les panneaux d'avertissement qui sont en triangle, les panneaux d'interdiction étant ronds. J'ai utilisé le code `/!\` principalement pour colorer les

« Attention » dans le manuel, mais aussi pour colorer les « à finir ».
- `^^123^^` et `,,123,,` : exposant et indice.
- `*'*gras*'*`, `/'/italique/'/`, `+'+gros+'+`, `_'_souligné_'_`, `~'~barré~'~`. Cyril Gavoille n'était apparemment pas très intéressé par ces styles, mais c'était bien peu de travail et une fonctionnalité potentiellement intéressante, donc je les ai rajoutés. J'ai néanmoins vraiment voulu éviter qu'ils interfèrent avec le texte brut, et aussi de les présenter comme des codes à utiliser souvent. C'est pourquoi il y a cette apostrophe `'` à chaque fois au milieu qui les rend longs et compliqués à saisir. Le « gros » est mis en vert en ANSI (la taille ne peut pas être modifiée) ; j'ai créé ce style pour les lignes d'usage au début du manuel, qui étaient un titre de niveau 3 dans le manuel d'origine.
  Les caractères utilisés pour le gras, l'italique et le soulignement sont les mêmes que dans le texte structuré de Thunderbird. Le tilde pour le barré est repris du Markdown.

Et, comme on le sait bien, le `\` pour l'échappement est très courant. Il fonctionne par exemple en Markdown, dans les commandes POSIX, et dans les chaines de caractères en C et plein d'autres langages.

Au départ, je n'avais pas permis d'utiliser ces délimiteurs au sein du texte préformaté, car je pensais que cela était plus susceptible de perturber que d'aider. J'ai fini par faire le choix inverse. Les points ci-dessus ont expliqué mes choix dans la conception de ces formats. Voici maintenant des points sur ce que cela a donné en pratique :

- J'ai utilisé `` ` ` `` pour les options essentiellement, et parfois aussi pour les variables, mais pas lorsqu'elles apparaissaient dans une formule, fut-elle aussi simple que « n<0 ». J'ai même complètement renoncé à l'utiliser pour les variables dans la documentation des graphes (que je n'ai formatée que très tard, après le développement de `''`). De toute façon, pour les variables, le projet est d'utiliser `$` du LaTeX. Il est bien sûr inutile d'utiliser `` ` ` `` dans les blocs préformatés. Je l'ai cependant constamment utilisé dans les titres de section des options ; dès le départ, j'ai utilisé la fonction Rechercher & Remplacer de mon éditeur de code pour formater ainsi tous les titres d'options.
  Je n'ai jamais eu besoin d'échapper ces `` ` ` ``, à part pour montrer leur utilisation dans la documentation de `-print`.
- Le format `''` est le dernier que j'ai codé à cause de sa légère complexité ; avant ça, il a été un projet pendant longtemps, et j'avais codé la coloration des titres de section des graphes en ANSI comme s'ils utilisaient déjà ce format. Contrairement à `` ` ` ``, en HTML il ne place pas son contenu dans un cadre, en en ANSI il ne colore pas les paramètres et n'a donc aucun effet dessus. En HTML, il implique néanmoins l'utilisation d'une police à chasse fixe. L'idée était que les classes de graphes, notamment dans les titres de section, puissent être appréhendés comme des objets assez indépendants du langage de GenGraph, c'est pourquoi leur style est plus léger. Cependant, ce style plus léger fait qu'on ne voit pas aussi bien ce qui est dedans qu'avec `` ` ` ``. C'est notamment problématique pour les listes de paramètres terminées par `.` comme celle de `grid` : le point final ne se voyait pratiquement pas si je n'utilisais que `''`. Donc dans ce cas-là, j'ai toujours utilisé `` ` ` `` en plus de `''` (ce dernier n'encadrant que le premier mot).
  J'ai aussi toujours fait cette imbrication hors de la grande section sur les graphes, et peut-être que ça devrait aussi être fait dans cette section. Parce qu'au final, je trouve que ce format n'a que deux usages pertinents : être équivalent à `[[` `` ` ` `` avec une autre couleur, et donner leur style aux titres de section sur les graphes. Cela passe bien aussi dans l'historique, pour annoncer qu'on ajoute un nouveau synonyme (comme j'ai fait pour `complete`). Ce format a un autre souci : c'est qu'on ne peut pas y imbriquer d'autres liens sans créer des problèmes. Donc si un synonyme utilise des options, il ne faut pas mettre de lien. On peut remarquer que des liens sont mis même dans les titres de section de graphes ; ça n'a pas d'importance, et en fait cela offre aussi un moyen d'obtenir le lien.
- J'ai utilisé `[[` et `''` constamment dans les blocs préformatés (c'est-à-dire surtout les exemples). Cela conduit à avoir de jolies couleurs et des liens. Ces deux formats ont le défaut d'être les plus susceptibles de devoir être échappés ; en cherchant `\` dans le code du manuel, on peut voir qu'ils sont déjà échappés un certain nombre de fois. Pour `''`, comme expliqué précédemment, le problème se pose lorsque l'on veut mettre une apostrophe devant. Personnellement, ça ne me touche pas car j'utilise toujours l'apostrophe courbe `’` (dite aussi apostrophe typographique) pour écrire en français, et je réserve l'apostrophe droite `'` (dite aussi dactylographique) au code (et parfois à l'anglais) ; je recommanderais donc cette solution. `''` me semble avoir bien peu de chance de devoir être précédé d'une apostrophe en anglais. Lorsque MediaWiki voit `'''` avec pour seule correspondance `''`, il fait attention à placer l'apostrophe à l'extérieur de l'italique, mais Clih ne gère pas cette fonctionnalité. En tout cas, je pense que le jeu en vaut la chandelle, car ces formats rendent le manuel vraiment plus sympa.
- Je me suis retrouvé à échapper `,,` une fois : pour un exemple utilisant la commande `sed` afin d'éliminer les codes ANSI de la sortie. Bien que `,,` et `^^` n'ont probablement pas beaucoup d'intérêt, je pense qu'il y a peu de chance qu'il y ait besoin de les échapper, et de temps en temps, ça peut peut-être bien aider (même si on compte surtout sur le LaTeX).
- J'ai utilisé l'italique surtout pour les termes anglophones. Je n'ai pas utilisé le gros pour autre chose que l'usage au tout début, je n'ai quasiment pas utilisé le soulignement, et pas du tout le barré. Je ne pensais pas utiliser beaucoup le gras non plus, mais j'ai finalement fait le choix de l'utiliser pour mettre en emphase les options dans les exemples dans leur propre section. Par exemple, dans les exemples de la section sur `-loop`, tous les `-loop` sont en gras avec leur paramètre (0, 1 ou 2). Au final, ça devient donc un peu gênant d'avoir un format aussi long pour le gras. Notons que les autres langages de balisage léger utilisent souvent le même code pour le gras et l'italique, en utilisant juste plus de caractères pour le gras (cas du Markdown avec `*` et `_`, et de MediaWiki avec `''`) ; je n'ai jamais beaucoup apprécié cette convention. Notons que toutefois, pour les exemples, on va généralement commencer par les écrire sans formatage afin de les tester, et puis finalement on peut utiliser le formatage et donc sélectionner le premier `*'*` écrit pour le copier-coller partout avec un clic milieu de la souris ; c'est ce que j'ai fait pour le rajouter partout.

J'ai relu le manuel en entier pour le garnir de tous ces formats, avec beaucoup d'attention dans la partie sur les options (ma relecture de la partie sur les graphes a été bien plus bâclée). Je songe à rajouter un style pour colorer le mot `gengraph` dans les exemples afin qu'il soit un peu plus transparent, mais je ne suis pas si motivé que ça à le faire parce que je pense que les commandes vont bientôt être présentées sans le mot `gengraph` qui sera devenu capable de traiter des scripts.

3.3.3.3. Les styles de blocs

Voici une présentation des styles de blocs avec des commentaires.

- Le manuel d'origine utilisait `•` pour les listes non ordonnées, et `-` pour chaque élément de l'historique, mis en style préformaté (police à chasse fixe). Quand j'ai activé le formatage dans l'historique, j'ai conservé cette différence pendant longtemps (en affichant `•` dans tous les cas). À un moment où la fonctionnalité s'est complexifiée, j'ai préféré unifier pour éviter la duplication de code, le manuel déterminant le choix fait : `•`. Néanmoins, je pense qu'il faudrait plutôt opter pour `-`, car à chaque fois que je veux écrire une liste à puces dans le manuel, je dois chercher le disque `•` afin de la copier-coller, et `-` est bien plus répandu (utilisé par Markdown). Un autre choix est `*` utilisé par MediaWiki ; en fait, personnellement, j'utilise plus souvent `–` (tiret cadratin), qui est souvent présent sur les dispositions de clavier francophones : en bépo avec AltGr+1, en français azerty sous Linux avec AltGr+Maj+4, en français azerty sous macOS avec Alt+-, ou encore avec Alt+0151 sous Windows.
- Le manuel n'avait pas de style pour les listes ordonnées. J'ai fini par en voir une dans la section sur `rlt`, c'est pourquoi j'ai mis en place `#.`. Cela était devenu nécessaire pour que ces puces ne soient pas mises toutes dans un même paragraphe (c'était d'ailleurs le traitement qu'elles subissaient dans la version HTML). En Markdown et en reStructuredText, la syntaxe correspondante est `1.`, où `1` est en fait un entier naturel quelconque. Cela permet d'avoir éventuellement les numéros dans le fichier source s'il est lu directement. Mais dans le cas de GenGraph, le fichier source n'a pas à être lu directement, ainsi j'ai préféré utiliser simplement `#.`. Il me semble que j'avais vu quelque part que ça pouvait être une syntaxe alternative en Markdown (éventuellement avec une extension), ou peut-être avec un autre langage.
- Les blocs préformatés fonctionnent comme avant, avec `!!!` et `Ex:` ou éventuellement `Ex`*`N`*`:` avec *`N`* un entier naturel. Notons que cette syntaxe des exemples risquerait fort de ne pas convenir si le manuel était traduit dans une autre langue que le français ou l'anglais. En français, il faut normalement une espace devant `:`, il est donc accepté qu'il y ait une espace (insécable ou non) à cet endroit, c'est pourquoi la fonction `matchesOneOf()` est utilisée pour tester six cas. Une espace est requise après `!!!`, `Ex :` et autres. Je pense qu'on finira par mettre en place un certain code à la place de `Ex :`, pour l'internationalisation. `Ex :` est très souvent utilisé dans le manuel, `!!!` n'apparait qu'à quelques endroits, et `Ex`*`N`*` :` n'est en fait utilisé que dans la section de `-vcolor list` (et d'ailleurs, sa typographie, avec « Ex » collé au numéro, ne me satisfait pas vraiment).
  Il y avait une bizarrerie dans l'utilisation de `!!!` : le code préformaté était parfois placé en dessous de `!!!`, avec une ou deux espaces d'indentation seulement, au lieu d'être bien à droite. Cela forçait à avoir une indentation sur les premières lignes, pour correspondre. Je l'ai laissée pendant longtemps, puis je me suis décidé à faire la correction : il est désormais nécessaire que le texte préformaté soit bien à droite de `!!!` et `Ex :`, sinon il n'est pas considéré comme étant dans le bloc, et l'indentation que cela rajoute est supprimée. L'espace après ces préfixes est nécessaire.

La syntaxe des titres de section est expliquée dans la section précédente sur la simplification du format. Tous les titres de section de niveau 1 et 2 (donc autres que ceux des instructions) étaient écrits tout en capitales et c'est toujours le cas ; ce n'est pas imposé par la syntaxe. On peut remarquer qu'ils sont toujours entourés de lignes complètement vides, c'est préférable pour le rendu en ANSI mais Clih ne se base pas du tout là-dessus.

En revanche, les lignes vides terminent les paragraphes et les listes. Il peut aussi y avoir une ligne vide entre deux éléments d'une liste pour faire un espacement ; c'était au départ utilisé à deux endroits, j'ai trouvé que c'était inutile à l'un alors ce n'est plus utilisé que dans la section sur `-check routing cluster`. Contrairement aux autres langages de balisage léger, l'imbrication de listes n'est pas prise en charge, et je la développerai volontiers si cela devient utile. Actuellement, le manuel n'utilise pas énormément de listes ; l'historique en revanche n'est fait presque que de listes.

Les toutes premières lignes du manuel, avant la première ligne vide, sont traitées spécialement : c'est un entête qui sera mis en italique. La toute première ligne est utilisée encore plus spécialement : le programme GenGraph l'affiche avec l'option `-version`, et elle est également utilisée pour intituler les versions HTML et PDF du manuel. Si la ligne contient `  –  ` (un tiret demi-cadratin entouré d'espaces), seul ce qui précède est inclus dans le titre.

Il y a un autre style de bloc auquel j'ai pensé : les tableaux. Cyril Gavoille les a faits avec un style préformaté intéressant, différent de ce qu'on a avec le Markdown et autres, et qui a l'avantage d'être beaucoup plus simple et plus sobre : simplement des colonnes, sans bordure. Ce format se retrouve par exemple dans la présentation des typologies d'`alkane`, dans les listes de `-check routing` et `-norm`, et dans la liste des couleurs de `-vcolor pal`. En particulier concernant les hachages de `-check routing`, on voit que les nouveaux formats font que l'alignement du code source ne coïncide pas avec celui du rendu :

gengraph-repo/MANUAL.txt (extrait)

```
1258            • ``hash prime``   → h(x)=((a·x+b)%p)%k où 0<a,b<p sont aléatoires
1259                              et p=2^31-1 est premier (hachage par défaut)
1260            • ``hash mix``     → h(x)=mix(a,b,x)%k où a,b sont aléatoires sur 32 bits
```

gengraph-repo/MANUAL.txt (extrait)

```
1261                              et mix() est la fonction mélange de Bob Jenkins (2006)
1262            • ``hash shuffle`` → h(x)=π(x)%k où π(x) est une permutation de [0,n[
1263                              basée sur deux entiers aléatoires de [0,n[
1264            • ``hash mod``      → h(x)=(x+a)%k où a est aléatoire dans [0,k[
```

Si Clih reconnaissait les tableaux, on pourrait avoir cet alignement dans le code source, de sorte à le rendre plus simple à saisir. Néanmoins, je n'ai pas du tout travaillé sur cette fonctionnalité car j'avais peur que ça demande pas mal de travail et elle n'aurait pas apporté grand-chose au manuel actuel. Comme j'ai mis beaucoup de temps à savoir pourquoi on appelle ça des textes « préformatés », j'en donne une définition : ces textes sont préformatés car le texte brut contient déjà un format avec ses espacements, et ainsi le moteur de rendu doit préserver ce format plutôt qu'appliquer le sien (ce qu'on demande avec la balise `<pre>` en HTML).

3.3.3.4. Les formats d'instructions

Comme expliqué précédemment, le manuel d'origine comportait déjà des syntaxes pour les formats d'options à quelques endroits… Maintenant que je viens de vérifier, je vois qu'en fait, c'était une barre oblique `/` dans `-filter degmin/degmax`, que j'ai tout de suite décidé de remplacer par une barre verticale `|`. On trouvait cependant les crochets avec `-vcolor deg[r]`, et l'historique contenait le format un peu complexe `[w|u][psl|dis][d]`. Je me suis basé là-dessus pour concevoir un format offrant une certaine généricité.

À l'origine, à la place de `getOptionCase()`, j'avais une fonction `genId()` qui générait simplement le premier cas de l'option, sans les mots optionnels. L'idée était donc de ne prendre que le premier élément des crochets ou de les sauter complètement s'ils ne contenaient pas de barre verticale `|`. Cela était utilisé pour les identifiants HTML et pour l'accessoire permettant d'afficher une section particulière du manuel. Pendant longtemps, j'ai été frustré de ne gérer que ce cas, et j'ai fini par développer la fameuse fonction `getOptionCase()` critiquée plus haut. J'avais aussi la frustration de n'avoir qu'un seul identifiant HTML pour chaque section, alors que certains titres de section contenaient plusieurs lignes parce qu'il y a des alias ou parce que plusieurs options se ressemblant étaient traitées dans la même section (par exemple `-directed` et `-undirected`).

Pour les variables, la plupart des pages de manuel sur les commandes utilisent le soulignement, et enlèvent le gras. Je ne souhaitais faire aucun des deux ; j'ai opté pour une autre syntaxe courante à laquelle je suis plus habitué : les chevrons. Cette convention est utilisée dans les pages de manuel de Git par exemple. J'ai ainsi mis des chevrons autour des variables dans tous les titres de section pour les options. Cela me semblait essentiel pour les options, car parfois elles contenaient plusieurs mots et donc cela rendait difficile de distinguer entre un mot de l'option et un nom de variable. Dans le cas des graphes en revanche, après le premier mot il n'y a toujours que des variables (ou le terminateur `.`) ; je me suis donc dit qu'on pouvait s'en passer, en particulier dans le cas où ces variables ne comportent qu'une lettre, ou une lettre et un indice (utilisant un tiret bas, comme dans `u_i`). Les listes de paramètres (notamment celle de `grid`) peuvent aussi comporter des points de suspension.

L'algorithme de `genId()` pour les variables était le suivant : sauter tout le texte contenu entre `<>` (et l'espace éventuelle précédente), et s'arrêter dès la rencontre d'une variable à une seule lettre minuscule (suivie soit d'une espace, soit d'un tiret bas, soit de la fin de la ligne) ou de trois points de suspension `…` ou `...`. L'algorithme pour `<>` était en fait mêlé à celui utilisé pour `[]` et `()`, et c'est toujours le cas : une seule variable `depth` est utilisée pour gérer l'imbrication de ces délimiteurs, le parenthésage étant supposé toujours correct.

`|` utilisé seul permet de faire un choix entre un mot et un autre, comme dans `degmin|degmax`. Pour plusieurs mots ou un morceau de mot, ou encore parce qu'on est déjà entre crochets, il faut rajouter des crochets : `deg[min|max]`, `[hash [prime|mix|shuffle|mod\]]` (notons qu'il est nécessaire d'échapper l'avant-dernier crochet). Il est possible de ne pas mettre de barre verticale entre crochets, auquel cas c'est équivalent à en mettre une tout au début : `[a]` est équivalent à `[|a]`.

Spécifiquement pour la complétion, j'ai rajouté la possibilité d'écrire des lignes dans les titres de sections qui ne seront pas affichées : le suffixe `....` (une espace et quatre points) l'indique. C'est le suffixe des astuces visées par `getHintSuffix()`. Je m'en suis notamment servi pour offrir une complétion complète pour `-check routing` ou encore les formats de `-format` et les types d'`alkane` :

gengraph-repo/MA

```
1219 ....     ``-check routing [hash <h>] [scenario [nomem] <s>] <schéma> [<paramètre>]...``
1220 ....     -check [variant <v>] routing [hash [prime|mix|shuffle|mod\]] [scenario] [nomem] [none|all|edges|npa

 ... […]

2118 ....  ``-format <type>``
2119 ....  -format auto|simple|standard|default|classic|edges|list|matrix|smatrix|vertex|dot|dot-pdf|dot-ps|dot-

 ... […]

3139 ....  ''alkane <type> n''
3140 ....  alkane normal|cyclo|iso|neo|sec|tert n ....
```

Le fait que `getOptionCase()` ne gère pas bien les imbrications empêche de factoriser les formats `dot-*` de `-format`. Je n'ai pas pris la peine de mettre `''` et `` ` `` sur les lignes terminées par `....`. S'agissant de `-check routing scenario pair`, ce cas où il y a deux formats de variables possibles ne pose pas de problème à `getOptionCase()` mais n'est pas géré par la fonction de complétion.

Les lignes de titre masquées par `....` peuvent également avoir un autre code spécial : un croisillon `#` au début. Cela sert à avoir un identifiant HTML non affiché (`#` est le symbole utilisé pour caractériser les identifiants dans les sélecteurs en CSS). J'ai fini par implémenter cela rapidement, puis je ne m'en suis pas servi. Cela pourrait servir à conserver des liens dans l'historique vers des instructions qui ont changé de nom. Étant donné que de toute façon, avant, il n'y avait pas de lien du tout, je ne les ai simplement pas mis dans ce cas-là ; l'information peut de toute manière être trouvée plus bas dans l'historique, puisque les renommages sont décrits à chaque fois. On pourrait également activer la prise en charge de ces identifiants masqués dans l'aide intégrée, pour donner un moyen de savoir par quoi a été remplacée une instruction qui n'existe plus.

Il y a un ordre imposé lorsqu'il y a des lignes se terminant par `....` ; le non-respect de cet ordre conduirait les dernières lignes à être ignorées dans certains cas :

- les lignes sans `....` terminal doivent être en premier ;
- parmi les lignes avec `....` terminal, celles commençant par `#` doivent être en premier.

3.3.4. Les formats de sortie

On peut dire qu'il y a quatre niveaux de fonctions, avec chaque fois des fonctions communes aux deux formats :

- la fonction `Clih_printManual()` principale (qui inclut l'historique) ;
- les fonctions des grandes parties : `printHeader()`, `printManual()` et `printHistory()` ;
- les fonctions des blocs : `checkAndPrintPre()` vérifie si on est au début de texte préformaté et fait l'affichage le cas échéant, `printHeading()` affiche un titre de section ;
- les fonctions du texte au sein des blocs : `format()` remplace les délimiteurs par la syntaxe du langage cible, `formatQuery()` fait la mise en forme pour les `''`.

On peut remarquer que j'ai malheureusement tendance à dire « manuel » aussi bien lorsque l'historique est inclus que lorsqu'il n'est pas inclus. Je n'ai pas créé de terminologie pour faire la distinction.

Les fonctions `printManual()` et `printHistory()` lisent les lignes du fichier d'entrée. Chaque ligne détermine le type de bloc, et ainsi la fonction appelle la fonction spécialisée dans ce type de bloc. Cette dernière fonction lit les lignes du bloc jusqu'à leur fin ; de cette manière, lorsque la fonction `printManual()` ou `printHistory()` reprend la main, le pointeur de flux du fichier d'entrée se trouve au début du bloc suivant. Les fonctions des blocs lisent chaque fois une ligne de trop (la première ligne qui n'est pas dans le bloc), puis utilisent `fseek()` pour replacer le pointeur de flux juste avant, avant de rendre la main.

Notons que jamais Clih ne stocke plus d'un paragraphe en mémoire. Cela ne m'a pas posé de problème particulier, et peut-être que ça améliore les performances. On peut néanmoins noter que l'affichage dans un navigateur web utilise clairement beaucoup plus de mémoire que la taille du manuel, et l'affichage avec `less` stocke aussi probablement le manuel entier en mémoire. Donc si stocker tout le manuel en mémoire simplifiait les choses, on ne devrait pas hésiter à le faire. Mais à priori, ça ne semble pas utile.

3.3.4.1. ANSI

La fonction `printManual()` est capable de s'arrêter au deuxième titre de section qu'elle rencontre (sans l'afficher). Cela permet aux accessoires de l'utiliser pour n'afficher qu'une seule section.

J'ai fait face à une complexité avec ce format ANSI, due au fait que les codes ANSI ne peuvent pas être annulés un par un : on peut toujours en cumuler plus, mais pour revenir en arrière, la seule solution est apparemment d'utiliser `\e[0m` qui annule tout. Par exemple, pour enlever le gras et ne laisser que l'italique, le seul moyen semble être d'écrire `\e[0;3m` qui annule tout (0) et réactive l'italique (3). Ma solution pour gérer cela a été de placer les codes écrits dans une pile, afin de pouvoir les réécrire constamment après `\e[0;` chaque fois que le code au sommet de la pile est dépilé. Il y a ainsi une pile de délimiteurs : `delimsStack`. Note : en C, `\e` n'est pas standard (il est accepté en tant qu'extension GNU), alors j'ai utilisé `\33`. Mais en principe, on écrit plutôt `\e`.

Autre chose de compliqué avec le code ANSI, est que lorsqu'on remonte dans le texte avec un logiciel comme `less`, il n'est pas forcément capable de déterminer le style du début de la ligne, car elle dépend de codes de la ligne précédente. En fait, pour éviter les problèmes, `less -R` réinitialise le format à chaque début de ligne. `less` tout seul remplace les caractères d'échappement par « ESC » et ne laisse donc pas les codes ANSI faire leur effet ; et `less -r` désactive tout traitement, créant le problème expliqué. L'idée est d'utiliser `less -R` (sollicité par GenGraph CLI, pas directement par Clih), en ayant écrit le contenu de la pile des délimiteurs à chaque début de ligne ; la pile a ainsi un intérêt supplémentaire.

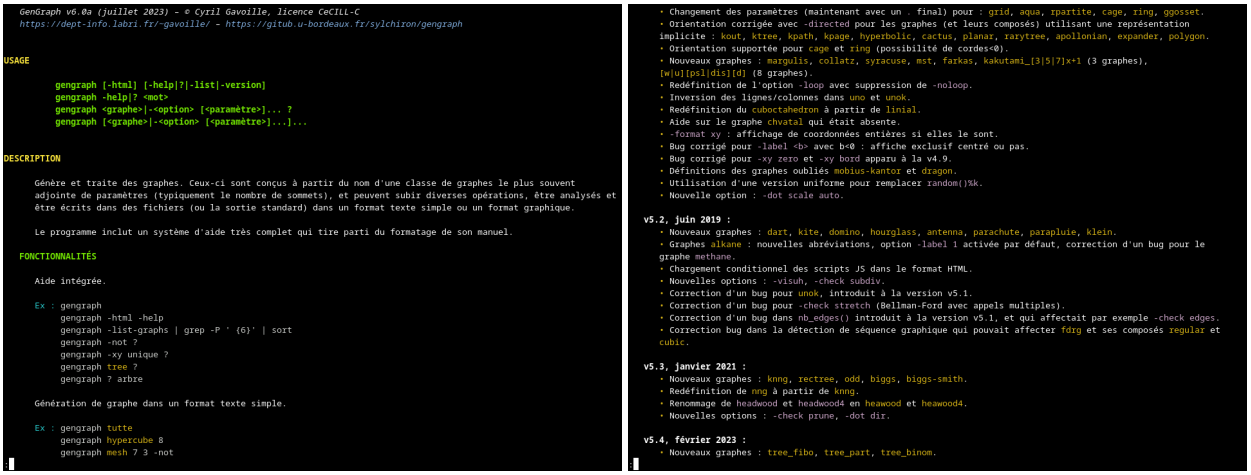
Au départ, Cyril Gavoille ne m'avait donné que les codes couleur commençant par 3 ou 4, offrant chacun les huit mêmes couleurs

(de 0 à 7), pour le texte (3) et le fond (4). J'ai découvert par la suite qu'il y a également 9 et 10, offrant huit couleurs supplémentaires. La plupart du temps, j'ai utilisé 9, car les couleurs qu'il offre sont plus claires et plus lisibles dans mon terminal. Dans les préférences de mon terminal (le terminal de Xfce), on peut voir la palette utilisée :

La palette de mon terminal dans les préférences : c'est la palette Tango du terminal Xfce, légèrement modifiée

La case à cocher en bas, que j'avais l'habitude de laisser cochée parce que je n'y comprenais rien (et maintenant que j'ai compris la technique, je ne comprends toujours pas la motivation), conduit en fait à toujours utiliser les couleurs 9 (à la place des couleurs 3) lorsque le texte est en gras. En dessous, il y a des préréglages proposés, et celui qui me convient le mieux s'appelle Tango. Même pour les couleurs claires, les contrastes sont souvent insatisfaisants (ils ne permettent pas bien de voir les mots), donc je choisis généralement des couleurs un peu plus claires dans mon terminal. Ce système de couleurs est ainsi un peu limité, pas vraiment fiable, mais j'ai quand même fait usage de toutes les couleurs proposées. Par contre, je les ai jugées trop vives pour les utiliser comme fond : je n'ai donc utilisé que 3 et 9, jamais 4 et 10. On peut espérer que les utilisateurs de GenGraph (ou d'un autre programme utilisant Clih) pourront généralement définir des couleurs adaptées pour eux dans une fenêtre de préférences du style de celle ci-dessus. Sinon, il est peut-être préférable de passer à la norme à 256 couleurs (voire 24 bits de couleurs), en espérant qu'elle soit aussi bien supportée par les applications.

Voici ce que donne le rendu chez moi (on voit en bas le curseur de `less`) :



Captures d'écran du manuel et de l'historique affichés dans le terminal

Il y a une dernière fonctionnalité compliquée, l'une des dernières que j'ai développées : le retour à la ligne pour éviter que les mots soit coupés sur deux lignes, ce que fait bien la commande `man`. `man` fait aussi de la justification, mais je ne m'en suis pas préoccupé (elle n'est pas activée non plus dans la version HTML). La fonction `Clih_formatAndWrap()` prend en charge cette tâche, et c'est l'autre fonction laide. En effet, faire ces retours à la ligne étant en principe simple, je m'y suis attelé confiant. Mais au départ, j'ai omis de me souvenir qu'il fallait faire attention à gérer les styles en début de ligne. Pour faire cela, ma technique : appeler la fonction `format()` qui remplace les délimiteurs, puis parcourir le texte pour trouver les espaces en enregistrant les codes ANSI dans une pile. Et quand on voit qu'on a dépassé la fin de la ligne, alors il faut reprendre la pile qu'on avait quand on en était à l'espace auquel on fait la coupure. Le code résultant est assez compliqué (70 lignes, et ce, bien qu'il utilise une macro), ce qui me laisse penser que mon algorithme n'est probablement pas très intelligent.

Le retour à la ligne n'est pas activé pour les blocs préformatés. L'option `-S` de `less` le conduit à laisser déborder les longues lignes sur la droite, et les flèches du clavier peuvent être utilisées pour voir la fin. Je l'ai donc utilisée. Dans les cas où `less` n'est pas utilisé, ces lignes risquent d'être coupées de façon vilaine, ainsi il faut faire attention à laisser courtes les lignes de texte préformaté.

La fonction `format()` fait le remplacement des délimiteurs de Clih par les codes ANSI. Elle est appelée par `Clih_format()` et `Clih_formatAndWrap()`. La première est utilisée pour les blocs préformatés, et ce qu'elle rajoute par rapport à `format()` est un affichage de la pile des délimiteurs au début et l'ajout de `\e[0m` à la fin.

Pour les puces, un code spécial `\e#` est utilisé. Elles sont détectées par la fonction `checkAndFormatList()`, qui rajoute ce code autour de la puce pour la colorer en jaune.

Pour obtenir la taille du terminal, j'ai utilisé la fonction POSIX `ioctl()` :

gengraph-repo/src/clih/manual.c (extrait)

```
72      struct winsize winSize;
73      ioctl(fd, TIOCGWINSZ, &winSize);
74      Clih_winSize = (Clih_WinSize){ winSize.ws_row, winSize.ws_col };
```

Il faut lui donner en paramètre le descripteur du fichier de sortie : soit `STDOUT_FILENO` soit `STDERR_FILENO` en général. La fonction vérifie d'abord si les variables d'environnement `COLUMNS` et `LINES` sont toutes les deux définies et contiennent bien un nombre supérieur à zéro, mais les terminaux semblent ne pas exporter ces variables par défaut.

3.3.4.2. HTML

J'ai très tôt passé beaucoup de temps à bichonner la version HTML du manuel. Je souhaitais qu'elle montre vraiment son intérêt par rapport à l'interface archaïque du terminal. J'ai de très bonnes (et un peu vieillies parfois, parce que je n'ai pas forcément été intéressé par les nouveautés) compétences en HTML, CSS et JavaScript qui ont rendu ce travail facile. Toutefois, je souhaite aller encore plus loin quand j'aurai l'occasion.

Le fichier `manual-template.html` est un gabarit un peu plus complet que le fichier `gengraph.head` d'origine, qui ne contenait que le contenu de la balise `<head>`. J'ai un peu l'habitude d'utiliser ce genre de gabarits (c'est un peu semblable à ce qu'on fait en PHP, ou encore avec les gabarits du cadriciel Django), et d'après moi, l'idée est d'y placer le maximum de contenu statique. Le code de Clih n'affiche que des textes dynamiques (tirés du manuel au format de Clih). Pour déterminer l'endroit où Clih doit écrire, j'ai repris la syntaxe des quatre points `....`. Il faut en placer trois dans le gabarit, pour :

- l'écriture des métadonnées, c'est-à-dire la balise `<title>` ;
- l'écriture du sommaire ;
- l'écriture du contenu.

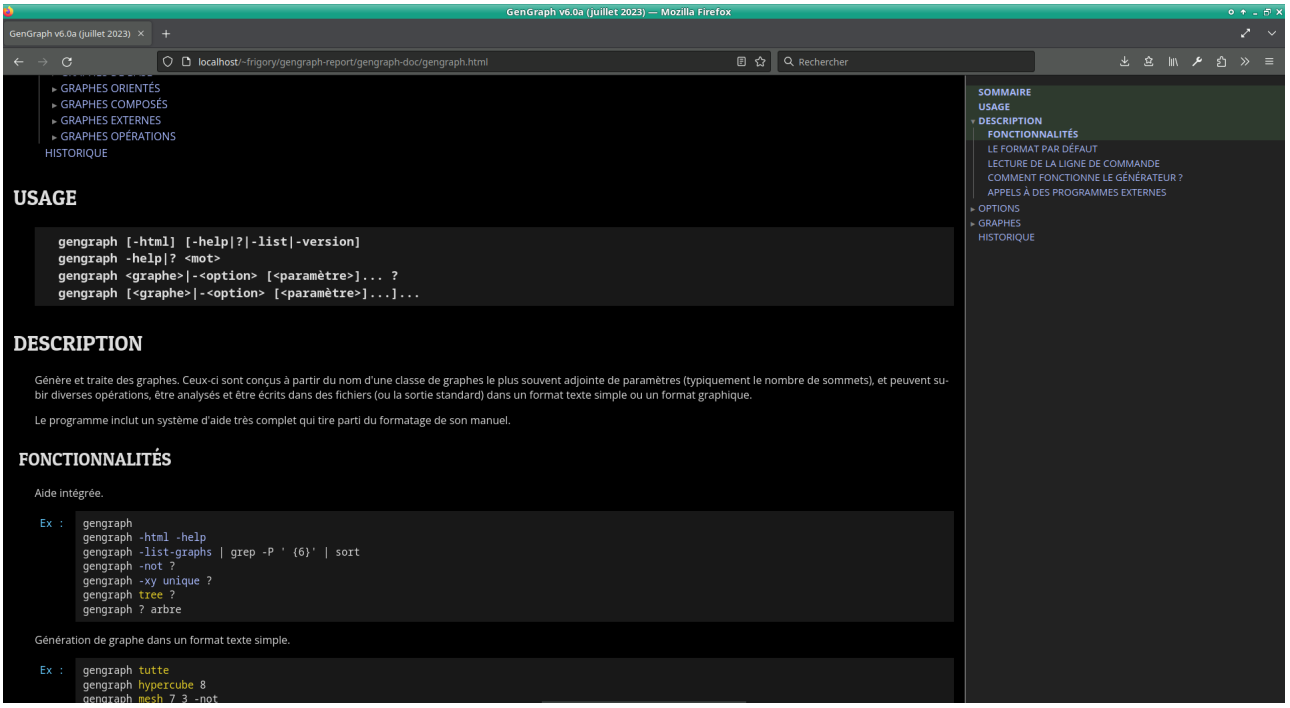
Pour les grandes parties du rendu, il y a trois fonctions de plus : `printTemplatePart()` copie le contenu du gabarit jusqu'à la prochaine ligne contenant `....`, `printMetadata()` affiche les métadonnées, et `printToc()` affiche le sommaire. Cette dernière fait un parcours entier du manuel et de l'historique, après quoi la lecture est reprise depuis le début (ou plutôt juste après l'entête) pour le contenu. J'ai déjà très souvent programmé une génération de sommaire en HTML tellement je trouve ça pratique d'en avoir, alors c'était simple à faire. Le sommaire est une hiérarchie de listes non ordonnées (`<ul>`). Pour savoir quand fermer une liste, il faut comparer le niveau du titre précédent avec l'actuel : on fait autant de fermetures que la différence. Bien que ce n'est plus le cas maintenant, le manuel d'avant passait à un moment (dans la section sur les options) directement du niveau 1 au niveau 3 ; cela augmente un petit peu la difficulté : plutôt qu'un simple compteur, il faut avoir une pile.

S'agissant des blocs :

- Pour l'entête au tout début, j'utilise la balise `<address>` (son style par défaut met le texte en italique).
- Le langage HTML semble ne pas permettre à un élément d'avoir plusieurs identifiants (et je ne comprends pas pourquoi). Pour qu'une section ait plusieurs identifiants, ma technique et de rajouter des `<div>` vides juste avant le titre de section. Le tout premier identifiant est toujours gardé pour le bloc du titre de section (`<hN>`). Le texte source du titre de section est donc parcouru une première fois pour les titres de section, et encore une fois pour afficher chaque ligne.
- La fonction `checkAndPrintList()` s'occupe des listes. Lorsqu'il faut un espacement entre deux éléments, une balise `<div class="space">` est rajoutée.

Je trouve que ce code est vraiment satisfaisant et que ça ne valait pas la peine de s'embêter avec un langage plus haut niveau. La longueur du code est comparable à celle requise par le format ANSI, et le découpage en fonctions permet de s'y retrouver. Je ne trouve pas que l'utilisation d'un langage plus haut niveau apporterait beaucoup.

Une grosse partie du travail a eu lieu dans le gabarit : j'ai ajouté beaucoup de CSS et JavaScript (400 lignes), y compris pour l'impression sur papier ou au format PDF. J'ai par exemple développé un thème sombre, ce qui se fait couramment et est maintenant facilement mis en place avec `@media (prefers-color-scheme: dark) { ... }` :



Capture d’écran du manuel visionné avec Firefox en mode sombre

Mon environnement de bureau (Xfce) est configuré pour utiliser un thème sombre ; par conséquent, Firefox m’affiche automatiquement le manuel en mode sombre. Il est également possible de l’activer dans les paramètres de Firefox. Pour l’obtenir avec Chromium, la solution la plus naturelle est d’installer un thème sombre depuis le Chrome Web Store. La propriété CSS `font-weight: 300;` rend le texte vraiment trop fin avec le mode sombre, donc elle est désactivée dans ce mode, ce qui donne un rendu très bon. Je trouve même que le manuel en mode clair est moins joli, car le texte (dans ses diverses couleurs) y est moins brillant.

Le préfixe `Ex :` des exemples n’est plus placé dans le bloc préformaté : il est maintenant dans un bloc séparé, flottant. Au départ, je l’avais mis dans le bloc `<pre>`, et j’utilisais un dégradé linéaire (qui se cassait brutalement) pour que la couleur de fond ne soit pas dessus ; c’était de la bidouille. Pour déterminer l’indentation du bloc dans la source, l’analyseur prend la longueur de `Ex :` mais fait attention à ne pas compter les octets qui ne sont pas un début de caractère UTF-8, c’est-à-dire ceux qui ont `10` comme bits de poids fort (condition `(octet & 0xC0) != 0x80`).

Il y a quelques styles pour rendre le manuel plus agréable à lire sur petit écran (avec `@media (max-width: Npx) { ... }`) :

- pour une largeur inférieure à 1 000 pixels, la marge à gauche (qui décale le texte par rapport aux titres de section) est supprimée ;
- en dessous de 800 pixels, le sommaire sur la droite est masqué (note : il est quand même généré) ;
- en dessous de 600 pixels, la taille du texte et la marge sur les bords sont réduites.

Généralement, les concepteurs de pages web voient la chose en *mobile first*, soi-disant que cela conduit à un code un peu plus court et à donner intellectuellement la priorité aux mobiles qui sont le type d’appareil de la plupart des utilisateurs du web. Je n’adhère pas à cette logique (de toute façon, je ne souhaite pas passer mon temps à faire du web) ; j’ai plutôt tendance à cibler les écrans de taille moyenne par défaut, puis je mets des requêtes `media` pour les écrans de plus en plus petits, et parfois d’autres pour les écrans plus grands. Dans notre cas, ces requêtes `media` font de toute façon moins d’une trentaine de lignes.

Mes développements ont surtout concerné le sommaire. Le code JavaScript en fait une copie dans la barre latérale sur la droite. Les sections qui sont au moins en partie visibles sont mises en gras sur un fond vert. C’est une fonctionnalité que je n’avais pas encore développée mais que j’avais envie de développer depuis longtemps : actuellement, les visionneuses web (par exemple Wikipédia en français dans son thème récent) ont tendance à ne mettre en gras qu’un seul titre de section, alors qu’il peut y en avoir plusieurs sur la page, ce que je trouve très déconcertant. Je n’ai pas trouvé de moyen de savoir directement quels titres de section sont visibles ; je le détermine en faisant une recherche dichotomique et en comparant les positions.

Le code JavaScript développe automatiquement les titres de niveau 1 dans le sommaire en haut de la page. Le sommaire de la barre latérale ne peut pas être développé avec des clics, parce que je pense que ça risquerait de créer des usages laborieux. Ma logique est que si on veut parcourir le sommaire, il vaut mieux revenir à celui du haut de la page. Ainsi, il faudrait que le lien « Sommaire » en haut de la barre latérale demeure toujours visible (quand on rentre dans de grosses sections, on est obligé de remonter en haut de la barre latérale pour le voir), et que le fait de cliquer dessus nous mène à la section actuelle ; mais je n’ai pas encore développé ces fonctionnalités. Je pense qu’il serait aussi très pratique d’avoir un champ texte en haut de la barre latérale (juste en dessous du lien vers le sommaire) pour filtrer les titres, c’est-à-dire n’afficher que ceux qui contiennent le texte saisi.

Il y a un peu de code JavaScript pour les options `-list`, `-list-options` et `-list-graphs` utilisées avec `-html` : il s'agit de reconnaitre l'ancre spéciale `-html-list` (ou `-html-list-options` ou `-html-list-graphs`), de développer entièrement la section sur les options et/ou la section sur les graphes, et de se déplacer au début de ces sections.

Lors du clic sur un lien interne, l'ancre n'est pas ajoutée à la barre d'adresse et le défilement est codé en JavaScript. J'ai codé cela parce que l'ajout de cette ancre avait tendance à m'ennuyer, car :

- cela rajoute des entrées dans le tableau des pages précédentes ;
- lorsque la barre d'adresse contient une ancre, l'actualisation de la page replace à cette ancre, ce qui est très ennuyeux pour un développeur qui travaille sur la page et qui souhaite constamment voir le résultat de ses modifications.

C'est cependant un peu dommage que les boutons « Précédent » et « Suivant » ne permettent plus de revenir à la position précédente. De toute manière, cela ne fonctionnait déjà plus, car la barre de défilement concernée n'appartient pas à l'ensemble de la page mais au bloc `<main>`. Ce que je trouverais bien plus pratique, c'est d'avoir un tableau des positions précédentes et suivantes en JavaScript, et des boutons dans la barre latérale pour naviguer dedans. Je ne l'ai pas codé parce que le développement de ce genre d'interfaces est parfois vraiment prise de tête, mais cela devrait rendre le manuel vraiment plus agréable pour moi. En version mobile (sans barre latérale), ces deux boutons devraient également apparaitre dans un coin, et il devrait y en avoir un de plus pour remonter au sommaire.

3.3.4.3. PDF

Je n'aime pas du tout le papier et les PDF, mais j'ai été habitué à voir les gens en utiliser beaucoup, et je suis toujours prêt à travailler un peu pour rendre mes produits confortables pour de nouveaux publics. Les documentations sont fréquemment proposées au format PDF ; je me suis dit que cela pourrait être intéressant pour GenGraph, et il y avait une solution toute simple : l'impression du document HTML. Il m'a suffi d'ajouter quelques règles CSS spécifiques à l'impression dans le gabarit, à l'aide de `@media print`.

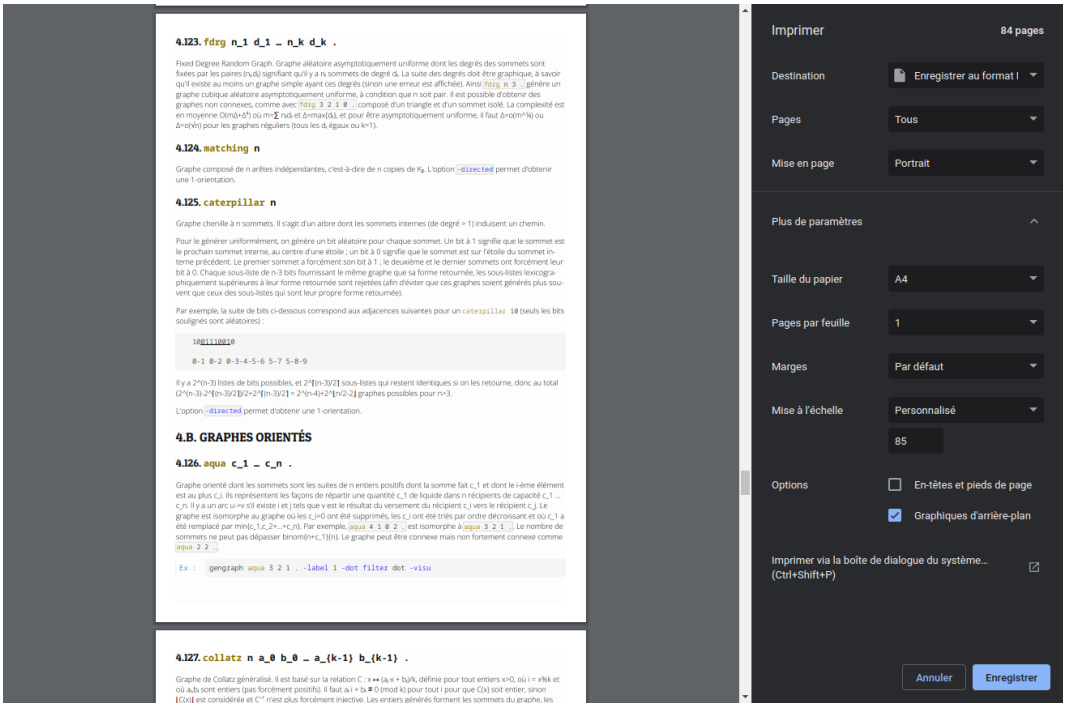
Ces règles ont les effets suivants : l'entête et le titre « Sommaire » sont centrés, le sommaire est tout déplié, il est à deux colonnes à partir des titres de niveau 3, et surtout il numérote les titres.

Je n'ai pas trouvé de moyen d'ajouter les numéros de page en bas des pages. Il me semble que pendant deux jours, j'ai passé l'essentiel de mon temps à faire des recherches sur le sujet tellement ça m'énervait de ne pas pouvoir obtenir quelque chose d'aussi fondamental et simple. Je n'ai trouvé que des solutions insatisfaisantes, ne fonctionnant pas ou trop compliquées. Le CSS offre pourtant des fonctionnalités pour la gestion des pages, mais il semble qu'elles sont peu implémentées par les navigateurs, ce que je trouve bien dommage.

J'ai compensé en m'acharnant sur la numérotation des titres de section. Cela compense grandement, et c'est même mieux, car même si j'avais pu avoir un bon pied de page avec numéro de page, je n'aurais sans doute pas pu ajouter les numéros de page dans le sommaire. De plus, les titres de section sont très nombreux dans le manuel de GenGraph sont toutes courtes : on trouve généralement plusieurs par page. Étant donné qu'il y en a partout, j'ai voulu éviter qu'ils soient lourds avec des numéros trop longs : je ne voulais pas avoir « 3.2.20.24.1 » devant `-check routing cluster` notamment. J'ai donc mis en place un système un peu spécial :

- les titres de niveau 1 sont numérotés avec des chiffres (« 1. », « 2. »…) ;
- les titres de niveau 2 incluent le numéro de niveau 1 et une lettre majuscule (« 1.A. », « 1.B. »…), et ne réinitialisent pas la numérotation pour les titres de niveau 3 ;
- les titres de niveau 3 incluent le numéro de niveau 1 et sont numérotés avec des chiffres ;
- les titres de niveau 4 sont numérotés avec des chiffres (ils n'incluent pas de numéro d'une section supérieure) ;
- les titres de niveau 5 incluent le numéro de niveau 4 et sont numérotés avec des chiffres.

L'idée pour les titres de niveau 4 est qu'ils font partie de grosses sections qu'on peut facilement trouver dans le sommaire, en regardant le premier mot (par exemple `-check` ou `-test`), c'est pourquoi j'ai jugé qu'on peut se passer d'y inclure les numéros des sections supérieures.



La boite de dialogue d'impression de Chromium, montrant le passage de la section « GRAPHES DE BASE » à la section « GRAPHES ORIENTÉS »

L'ajout de ces numéros a une conséquence gênante pour les titres sur plusieurs lignes : les lignes suivant la première se retrouvent en dessous du numéro, elles ne sont pas alignées avec le texte de la première ligne. En fait, j'ai pu résoudre le problème pour la deuxième ligne, en augmentant la hauteur du bloc contenant la numérotation de sorte qu'il occupe le début de la deuxième ligne aussi bien que de la première. Si je l'augmentais davantage, ça serait un problème pour les titres de seulement une ligne. Les quelques titres comportant plus de deux lignes (par exemple `-check ps1` et `-test ps1`).

3.3.5. Le traitement des arguments et des erreurs

Cette section ne traite pas de Clih mais du module Help de GenGraph CLI et du module Errors de GenGraph.

3.3.5.1. La vérification de l'argument suivant

Comme expliqué plus en détail dans la section Refonte de l'analyse des instructions ci-après, GenGraph v5.4 avait trois petites fonctions pour l'analyse du prochain argument : `CheckHelp()`, `GetArgInc()` et `NextArg()`. Les deux dernières étaient redondantes, ainsi je les ai fusionnées en `NextArg()`.

La nouvelle fonction `NextArg()` :

- incrémente de 1 l'itérateur global sur les arguments du programme ;
- déclare une erreur fatale si la fin de la liste des arguments est atteinte ou si l'argument est `?` ;
- convertit éventuellement l'argument avec `sscanf()`, et déclare une erreur fatale si la conversion a échoué ;
- renvoie l'argument sous forme de chaine de caractères.

Selon que le type voulu est une chaine de caractères ou autre chose, la fonction (qui est en fait une macro) sera appelée différemment :

- `NextArg("s")` place la chaine de caractères dans la variable globale `ARGS.cur` et la renvoie (l'appelant utilise soit la variable globale soit la valeur de retour) ;
- `NextArg("x", "%...", &x)` place dans `x` le résultat de la conversion par `sscanf()` avec le format `"%..."`, si la conversion réussit.

Le premier paramètre de `NextArg()` est utilisé pour nommer le paramètre en cas d'erreur. Ce nom doit idéalement correspondre à celui du manuel. Par exemple, pour `-group-n` :

gengraph-repo/MANUAL.txt (extrait)

```
1567 .... ``-group-n <n>``
```

on a l'appel suivant :

gengraph-repo/src/gengraph-cli/args_misc.c (extrait)

gengraph-repo/src/gengraph-cli/args_misc.c (extrait)

```
275     int n; NextArg("n", "%d", &n);
```

En cas d'erreur, l'aide sur l'instruction est affichée.

Si `?` est le premier paramètre, alors il n'est pas lu par `NextArg()` mais par `CheckHelp()` comme expliqué ci-dessous : c'est la syntaxe correcte pour demander l'aide sur une instruction. Le texte de l'erreur lorsque `?` est à une autre position vient du fait que certains paramètres sont des mots clés, par exemple avec `alkane`. Dans ce cas, l'utilisateur pourrait être tenté de demander l'aide spécifique sur ce mot clé (comme si c'était une sous-option, telle que `-check bfs`) ; elle n'existe pas, c'est pourquoi le programme affiche « l'aide d'une section supérieure ».

Pour la vérification des formats avec `sscanf()`, j'ai commencé par essayer de voir comment cela était fait par une autre commande. J'ai regardé avec `head` des coreutils GNU :

```
$ echo OK | head -n 1
OK
$ echo OK | head -n a
head: nombre de lignes incorrect: « a »
$ echo OK | head -n 1a
head: nombre de lignes incorrect: « 1a »
$ echo OK | head -n " 1"
OK
$ echo OK | head -n "1 "
head: nombre de lignes incorrect: « 1  »
```

On voit que le format accepté est : un nombre entier, éventuellement suivi par des espaces, ce qui correspond à une lecture avec `strtol()` (ou `sscanf("%d")`, qui est une surcouche) qui atteint la fin de la chaine de caractères. Bien que cela n'est probablement pas d'intérêt pratique, j'ai souhaité rendre la conversion plus cohérente que cela en acceptant également les espaces en fin d'argument. En recevant `"%d"`, `NextArg()` va ainsi appeler `sscanf(ARGS.cur, "%d %n", ptr, &pos);` et vérifier si `pos` a bien atteint `strlen(ARGS.cur)`. L'entier `pos` est initialisé à `-1` et il semble que la norme du C indique que si le format n'est pas atteint (parce que la lecture a échoué avant), sa valeur n'est pas spécifiée, mais il n'y a vraiment pas de raison que `sscanf()` modifie sa valeur en cas d'échec.

GenGraph v5.4 utilisait `strtol()` et `strtod()` pour faire les conversions, et ne procédait à aucune vérification. Cela conduisait à utiliser la valeur 0 si l'argument ne contenait pas d'entier à son début. Dans l'historique, on pouvait lire :

gengraph-v5.4/gengraph.c (extrait)

```
26525         - remplacement de atof()/atoi() par strtod()/strtol() qui
26526           sont plus standards.
```

Or, `atoi()` et `atof()` sont bien tout aussi standards que `strtol()` et `strtod()` (depuis C89). Ces deux dernières fonctions ont un prototype plus complexe et ont notamment l'intérêt de permettre de vérifier si la conversion a réussi et jusqu'où. Je suppose qu'il y a eu une confusion, due au fait que les commandes standard vérifient le format de leurs arguments avec `strtol()` et `strtod()`, alors que GenGraph ne le faisait pas. Dans ma pratique, il m'arrive d'utiliser `atoi()` et `atof()` par facilité, mais s'agissant de `strtol()` (à part pour sa capacité à lire un nombre dans une base quelconque) et `strtod()`, ma réflexion rejoint celle que j'ai donnée précédemment pour `strcat()` : `sscanf()` offre toutes les fonctionnalités utiles avec une syntaxe concise et puissante, et elle évite d'avoir à s'ennuyer avec d'autres prototypes de fonctions.

Afin de pouvoir recevoir soit un argument soit trois, la technique est la suivante. La fonction `_NextArg()` reçoit en paramètre le premier argument `desc` et une structure `toScan` de type `ScanfElem` contenant les deux autres arguments, ce qui permet de l'appeler comme suit :

```
_NextArg("s", (ScanfElem){});
_NextArg("n", (ScanfElem){ "%d", &n });
```

La macro variadique `NextArg()` permet de se passer d'écrire `(ScanfElem){` et `}`. Ce système est utilisé intensivement et de façon bien plus poussée pour l'analyse des instructions plus complexes (abordée plus loin).

Le dernier paramètre est un pointeur générique, c'est-à-dire de type `void *`. Dans le format, `NextArg()` aurait pu ne pas demander le `%` et l'ajouter par elle-même, mais je me suis dit que le sens du paramètre est bien plus clair s'il a bien un format qui peut être accepté par `scanf()` (`NextArg()` ne prend pas en charge des formats avec plusieurs `%`). Mon éditeur de texte colore les formats acceptés par `printf()` et `scanf()`.

Bien que l'on soit à mon avis un peu loin d'avoir un système de rapport d'erreur vraiment satisfaisant, j'ai souhaité pouvoir afficher le nom des valeurs de paramètres manquantes ou au format incorrect. Cela me semblait minimal pour obtenir un peu de convivialité. À cette fin, `NextArg()` demande ce nom comme premier paramètre. Cela est un peu laborieux et n'est finalement pas appliqué partout : dans certains cas, `NextArg()` est appelée pour plusieurs instructions différentes, et le nom générique `entier` est

alors utilisé. Une meilleure solution serait peut-être de demander à Clih le nom du prochain paramètre d'une instruction. Le nom de l'instruction n'est pas affiché dans le message d'erreur, mais si le chargement de l'aide réussit, on le voit normalement juste en dessous.

3.3.5.2. Le tableau de la section en cours

La fonction `CheckHelp()` est appelée après les mots clés correspondant à des noms d'instructions ayant leur section dans le manuel. Elle vérifie si `?` se trouve après le mot, et si c'est le cas, l'aide sur l'intruction est affichée et le programme est arrêté. C'était déjà le cas dans GenGraph v5.4, et ce test était redondé dans `NextArg()` (et `GetArgInc()`). Ce système m'est apparu comme bancal, à cause des résultats suivants (voir `-check paths`) :

```
$ gengraph-5.4 -check paths 0 ?

Erreur : argument incorrect.
Aide sur -check paths 0 non trouvée.

$ gengraph-5.4 alkane cyclo ?

Erreur : argument incorrect.
Aide sur cyclo non trouvée.

$ gengraph-5.4 -not alkane cyclo ?

Erreur : argument incorrect.
Aide sur -not alkane cyclo non trouvée.
```

L'aide était ainsi cherchée pour les arguments à partir du dernier commençant par `-`, ou pour le tout dernier argument si aucun ne commençait par `-`. L'idée semblait être que GenGraph permettait soit de générer un graphe et d'exécuter éventuellement un algorithme, soit de demander de l'aide ; une exécution ne devait pas servir aux deux en même temps. Cependant, dans ce cas, on peut se demander pourquoi on cherchait le début de l'instruction en cours, au lieu de demander l'aide d'une instruction faite de tous les arguments. En tout cas, afin que GenGraph aille dans le sens d'un langage de script interactif, j'ai voulu rendre ce système plus robuste.

Dans les exemples donnés ci-dessus, le résultat est le même si l'on omet le dernier argument `?`. Ainsi, à l'origine, l'idée était que l'aide était affichée si l'instruction était invoquée sans paramètre. Cela empêchait de demander l'aide sur les instructions sans paramètre telles que `-not` et `tutte`, c'est pourquoi l'opérateur `?` a été rajouté. Afin d'augmenter la cohérence du système, j'ai préféré simplement percevoir l'omission d'arguments comme une erreur ; cependant, l'aide est toujours affichée dans ce cas. Il en est de même pour le placement de `?` à un emplacement incorrect. Il y a en fait toujours une incohérence avec les instructions sans paramètre :

```
$ gengraph tutte ?

(Affiche l'aide sur tutte)

$ gengraph parachute 10 ?

(Affiche l'aide complète)
```

Ma recommandation serait de n'afficher l'aide complète qu'avec `-help`, ou lorsque `?` se trouve au début d'une ligne dans un script (ou en mode interactif).

Afin que `NextArg()` puisse toujours afficher la section de l'instruction en cours, j'ai décidé d'utiliser `CheckHelp()` pour enregistrer les mots du nom de l'instruction dans un tableau. En fait, il est géré comme une pile par `CheckHelp()`, mais en pratique le dépilement n'a toujours lieu que pour tous les éléments en même temps. Le tableau s'appelle néanmoins `ARGS.stack` et sa longueur est stockée dans `ARGS.stackLen`. `CheckHelp()` prend un seul paramètre : le numéro `i` du mot dans le nom de l'instruction. Ainsi, `CheckHelp(i)` enregistre le mot (l'argument en cours d'analyse) dans `ARGS.stack[i]` et affecte `i + 1` à `ARGS.stackLen`. Par exemple, pour `-check routing cluster -1`, le module Main appelle `CheckHelp(0)` pour `-check`, puis CLICheck appelle `CheckHelp(1)` pour `routing` et `CheckHelp(2)` pour `cluster`.

Le tableau `ARGS.stack` est rarement utilisé ailleurs. `GetCurInstrStr().s` permet d'obtenir le nom de l'instruction en cours, c'est-à-dire la concaténation des mots du tableau. Cela est utilisé pour donner le nom de l'instruction pour certaines erreurs. Lorsque ce nom ne contient qu'un seul mot, `ARGS.stack[0]` est généralement directement utilisé à la place.

Note : `?` pour demander l'aide n'est pas courant dans les arguments d'un programme. Les conventions sont plutôt `--help`, `-h` et plus rarement `-?` (sous Windows, le standard est `/?`, quoique cela a peut-être changé depuis PowerShell). Certains programmes (comme `git`) acceptent aussi `help` placé avant les termes. Je crois que `?` est cependant courant dans certaines interfaces interactives de saisie de commandes.

3.3.5.3. Le signalement des erreurs et les avertissements

GenGraph v5.4 a une fonction `Erreur(code)` qui s'appelle avec un paramètre de type entier qui correspond au numéro de l'erreur. Cette fonction sélectionne le message d'erreur avec un `switch`, l'affiche sur `stderr` (précédé par « Erreur : »), puis arrête le programme avec `exit(EXIT_FAILURE)`.

La nouvelle fonction `Error(code, ...)` sélectionne le message d'erreur dans un tableau et l'affiche sur `stderr`, formaté d'abord par Clih puis par `fprintf()`, avec les arguments récupérés avec la technologie `va_list`. Le fichier `errors.c` est dédié à cette fonction qui ne fait rien d'autre que ça. Elle n'arrête plus le programme. Cependant, elle est toujours appelée :

- soit via la macro `FatalError(code, ...)` qui appelle `assert(false)` et `exit(EXIT_FAILURE)` ;
- soit via la macro `FatalErrorSeeHelp(code, ...)` (du module CLI) qui appelle en plus `HelpOnSection()` avant `assert()`.

Je préfère utiliser un tableau plutôt qu'un `switch`, car la syntaxe par élément est plus concise et exprime mieux ce qu'on veut faire. J'utilise une macro (`TAKE_IF_PRESENT(array, index, dflVal)`) pour récupérer le texte de l'erreur dans ce tableau, avec valeur par défaut si l'indice est invalide. La syntaxe de C99 `[indice] = valeur` permet d'indiquer l'indice de la valeur dans le tableau lors de l'initialisation de ce dernier.

L'appel `assert(false)` n'a d'effet que lors de l'exécution de la version de débogage du programme, et produit l'affichage sur la sortie standard de la ligne du code source contenant l'invocation de `FatalError()` (ou `FatalErrorSeeHelp()`). L'exécution avec le débogueur permet d'avoir une pile d'appels plus complète, que l'on n'aurait pas non plus sans `assert(false)`. Lorsqu'un arrêt pour erreur fatale semble être la véritable erreur, cela permet de trouver facilement son origine.

L'entier était donné tel quel dans le code de GenGraph v5.4, avec parfois un commentaire. Par exemple :

gengraph-v5.4/gengraph.c (extraits)

```
 2458     if(f==NULL) Erreur(7); /* on a pas réussit à ouvrir file */
  ...  […]
 7133   if(i>=CHRONOMAX) Erreur(26);
  ...  […]
11776     if(k<1) Erreur(6); // paramètre incorrect
  ...  […]
20302     if(FORMAT<0) Erreur(5); /* le format n'a pas été trouvé */
```

Je trouvais peu satisfaisant d'avoir ces entiers, alors j'ai défini une énumération dans `errors.h`. Cette méthode a le défaut d'imposer la recompilation de tous les fichiers qui incluent `errors.h`, c'est-à-dire tous les fichiers de GenGraph, GenGraph Formats et GenGraph CLI, chaque fois qu'on ajoute ou supprime un message d'erreur (ou qu'on modifie le nom de la constante). L'ajout d'erreur est aussi un peu plus compliqué, mais la conception de l'erreur demeure toujours plus compliquée que l'écriture du code à mon sens. Actuellement, `common.h` dépend d'`errors.h` car elle contient le code de fonctions d'allocation de mémoire qui ont des cas d'erreurs fatales. On pourrait cesser d'avoir une version `inline` de ces fonctions, et ainsi déplacer leur code dans `util/misc.c`. Mais je me suis dit que de toute façon, vu les méthodes de travail de Cyril Gavoille, ça lui ferait sans doute bizarre de devoir inclure un fichier pour signaler une erreur. `FatalError()` est actuellement appelée dans près d'une trentaine de fichiers sources `.c`, sans compter ceux qui incluent `cli_options.h` qui contient `FatalErrorSeeHelp()`.

L'intérêt de l'énumération est qu'elle permet d'une part de savoir sans commentaire (et sans aller voir dans `errors.c`) quelle erreur est signalée, et d'autre part de réorganiser les messages d'erreur dans le module Errors (en évitant de laisser des trous). Les constantes nommées permettent au programmeur d'avoir une description synthétique de l'erreur souvent plus facile à lire que le texte complet (qui contient parfois des informations supplémentaires). Lorsqu'une erreur doit être signalée à un nouvel endroit cependant, il est généralement nécessaire de regarder son texte pour savoir quels sont les paramètres à donner.

Quelquefois, j'ai eu besoin de signaler des avertissements. Pour cela, j'ai juste écrit une petite macro `CustomWarning(format, ...)` qui affiche l'avertissement sur `stderr` précédé de « Avertissement : » mis en jaune gras avec les codes ANSI. C'est utilisé pour signaler les cas particuliers non pris en charge de `-check invlinegraph`, les options ignorées à cause de l'utilisation d'un graphe opération, ou encore la présence d'options après `-visu`, entre autres. Certains de ces avertissements sont voués à disparaitre (à terme, `-visu` ne devrait plus arrêter le programme, `-check invlinegraph` devrait prendre en charge tous les cas et les options annulées par les graphes opérations seront supprimées), c'est pourquoi je n'ai pas pris la peine de faire une fonction propre comme `Error()`. Cela empêche d'avoir la coloration avec Clih (pour l'avoir, il faudrait inclure `clih.h` dans `errors.h`). L'utilisation de Clih pour l'affichage des erreurs permet aussi d'avoir les retours à la ligne adaptés à la largeur du terminal, au lieu de se donner la peine de les écrire à la main. À terme, il sera préférable de créer une fonction `Warning()`. Le fait de grouper les messages dans un fichier est aussi important pour pouvoir ajouter facilement des traductions.

L'utilisation de paramètres dans les messages d'erreur m'a permis de fusionner certains messages existants, notamment :

gengraph-v5.4/gengraph.c (extraits)

```
 846   case  9: s="option -vcolor non reconnue."; break;
  ...  […]
 849   case 12: s="option -check non reconnue."; break;
```

gengraph-v5.4/gengraph.c (extraits)

```
...  […]
851    case 14: s="option -filter non reconnue."; break;
...  […]
876    case 39: s="schéma de routage non reconnu."; break;
877    case 40: s="fonction de hachage non reconnue."; break;
878    case 41: s="scenario non reconnu."; break;
```

Ces messages, ainsi que quelques autres, sont remplacés par un seul :

gengraph-repo/src/gengraph/errors.c (extrait)

```
18      [GGERR_UNKNOWN_SUBOPTION] =
19        "paramètre de ``%s`` non reconnu : ``%s``.",
```

On en trouve des appels notamment dans CLICheck :

gengraph-repo/src/gengraph-cli/args_check.c (extraits)

```
23        FatalError(GGERR_UNKNOWN_SUBOPTION, "-check routing hash", ARGS.cur);
...  […]
41        FatalError(GGERR_UNKNOWN_SUBOPTION, "-check routing scenario", ARGS.cur);
...  […]
70      FatalError(GGERR_UNKNOWN_SUBOPTION, "-check routing", ARGS.cur);
...  […]
236       FatalError(GGERR_UNKNOWN_SUBOPTION, ARGS.argv[ARGS.stack[0]], ARGS.cur);
```

Actuellement, il y a en tout 58 messages d'erreur différents. GenGraph v5.4 en avait 52. Ainsi, j'ai supprimé plusieurs messages mais j'ai rajouté un lot de nouveaux messages : pour l'aide intégrée, pour la lecture des fichiers, pour le nouveau système de pile, pour les nouvelles vérifications de paramètres, pour les nouvelles options… Seulement trois messages requièrent trois paramètres ; les autres en requièrent tous moins. Notons que si les bons paramètres ne sont pas envoyés à `Error()` ou une macro qui lui passe le relai, cela causera probablement un plantage ; c'est l'une des raisons pour lesquelles il est important de tester les cas d'erreurs.

Le module Errors fait partie de la bibliothèque GenGraph et non de GenGraph CLI : il ne sera sans doute pas intéressant que les messages d'erreur (qui utilisent la syntaxe de Clih) soit inclus dans la bibliothèque, cependant certaines constantes doivent faire partie de la bibliothèque. Un travail reste à faire pour répartir correctement ce module dans GenGraph, GenGraph Formats et GenGraph CLI.

3.3.6. L'ouverture des fichiers et les accessoires

Cette section traite le module Accessories de Clih et la façon dont le module Help de GenGraph CLI utilise Clih.

3.3.6.1. L'implémentation des accessoires

Le module Accessories de Clih contient 150 lignes de code offrant trois fonctions assez élémentaires, prenant toutes en premier paramètre le manuel (de type `FILE *`).

- `Clih_printSection(manual, instr)` affiche la section correspondant à l'instruction donnée en mode canonique. Pour cela, elle génère les variantes canoniques des instructions (données par les titres de section du manuel, sans compter les lignes d'astuce), et les compare à `instr`. En cas d'égalité, elle se replace au début du titre de section et affiche la section. GenGraph v5.4 affichait aussi les sous-sections ; j'ai fini par les masquer parce que je trouvais cela très ennuyeux, et à la place j'ai codé l'affichage des titres des sous-sections seulement. Pour `-check`, cela produit quand même un résultat un peu long. On pourrait envisager de n'afficher que les sous-sections directes (et pas les sous-sections des sous-sections).
- `Clih_printSectionsAPropos(manual, text)` affiche les sections d'instructions contenant `text`. L'analyse se fait sur les paragraphes, ainsi `text` peut contenir des espaces. Globalement, seul le texte visible lorsque le manuel est affiché, sans compter l'indentation et en enlevant les codes de formatage, est testé (avec `strstr()`). Il y a encore quelques exceptions : les `!!!` et les `#.` sont conservés dans le texte à tester. Chaque section contenant le texte est affichée, sans les sous-sections. Les titres des sections supérieures, eux, sont affichés (s'ils ne l'avaient pas encore été), principalement pour rendre l'indentation cohérente ; à cette fin, leurs positions dans le fichier sont stockées dans une pile. Le nom de la fonction vient de la commande Unix `apropos`.
- `Clih_printHeadings(manual, accepted)` affiche les titres de sections d'instructions : soit les principales, soit les options, soit les deux, selon les drapeaux levés dans `accepted`. La fonction commence par faire un premier parcours, pour essayer de trouver la position des titres « OPTIONS » et « GRAPHES », dont elle va afficher tous les titres. Cela n'est pas fait de façon très robuste : la fonction suppose que les options sont avant les graphes, et que ces deux grandes sections sont consécutives dans le fichier. L'entête en italique tout en haut est affiché dans tous les cas, puis les titres. Pour rendre la fonction robuste, il faudrait sans doute utiliser une pile comme avec `Clih_printSectionsAPropos()`.

Chacune de ces fonctions renvoie faux si aucune correspondance n'est trouvée. `Clih_printHeading()` renvoie faux si elle ne trouve aucun titre de section d'instruction (qu'il fasse partie de ceux à afficher ou non) ; cela permet de se dire que le fichier du manuel est probablement corrompu).

Le module Help de GenGraph CLI utilise ces fonctions pour fournir les options `-help`, `-list` et cie, ainsi que l'opérateur `?`.

Tout comme pour l'affichage de l'aide complète, le programme essaie de relayer la sortie vers `less`, sauf pour l'affichage d'une section seule (c'est pourquoi j'ai jugé préférable de ne pas afficher les sous-sections dans ce cas). L'idée pour une section seule est que l'on voudra peut-être la voir au-dessus de l'invite de commandes, pendant qu'on tape la commande suivante, c'est pourquoi elle est directement affichée dans le terminal. Dans les autres cas, `less` permet de défiler vers le haut et vers le bas, et a d'autres fonctionnalités, notamment la recherche d'expressions rationnelles. Pour la fonctionnalité de `Clih_printSectionsAPropos()`, j'aurais voulu que les mots soient surlignés dès le départ comme si on avait lancé une recherche dessus, mais je n'ai pas trouvé d'option pour activer cela dans `less`. Les raccourcis claviers sont basés sur les commandes de `vi`, notamment Q permet de quitter, et H affiche un manuel. Lorsqu'on quitte `less`, on retrouve le terminal comme avant le lancement de `less` : il ne laisse pas de trace dans le terminal. Son nom est simplement un jeu de mots avec la commande Unix `more`.

La procédure pour lancer `less` est la suivante : si l'ouverture du fichier du manuel a réussi, la fonction `SetPrintToLess()` est appelée. Un tube vers une nouvelle instance de `less` est créé, avec la technique `pipe/fork/exec/dup2` vue en cours de programmation système cette année. Cette technique est également utilisée par `PipeDot()` de GenGraph Formats. Elle n'est pas idéale, surtout pour Cygwin car `fork()` n'existe pas sous Windows ; une meilleure solution serait probablement d'utiliser `posix_spawn()`, mais je ne me suis pas attelé à son étude. Un commentaire dans la fonction `PipeDot()` d'origine suggérait d'utiliser `popen()` ; je n'ai pas souhaité le faire car cette fonction utilise le même système que `system()`, c'est-à-dire qu'elle exécute la commande avec `/bin/sh -c`, ce qui est plus lourd et moins sûr. De plus, elle renvoie un `FILE *` ; toutes mes fonctions écrivent sur la sortie standard par simplicité, j'aurais dû utiliser `fileno()` pour rediriger… Il était donc plus simple pour moi d'utiliser la méthode expérimentée en TD à la fac.

`less` est exécuté avec `execlp()` ou `execl()` (les deux sont essayés) dans le processus père, afin que ce soit lui que le terminal voie comme tâche d'avant-plan. Avant ce lancement, les fichiers ouverts (transmis en paramètres à `SetPrintToLess()`) sont fermés. La sortie du processus fils est reliée à l'entrée du processus père, grâce à `pipe()` et `dup2()`. Tandis que le processus père a passé la main à `less`, le processus fils quitte `SetPrintToLess()` et appelle les fonctions de Clih qui vont faire l'affichage. Cet affichage est donc envoyé sur le tube, causant l'affichage par `less`. Quand GenGraph a terminé son affichage, c'est la fin de son exécution, mais `less` peut toujours être utilisé (l'affichage complet n'est généralement pas immédiat car `less` ne lit pas de lignes du fichier d'entrée avant d'avoir à les afficher ou à les sauter, et donc GenGraph va être bloqué en attendant que `less` demande plus de données ; on peut aller directement à la fin du fichier avec la touche du clavier Fin). Si le lancement de `less` a échoué, alors le processus père se termine (avec `exit(EXIT_SUCCESS)`, car après tout `less` n'est pas requis), le tube n'est pas créé et l'affichage est fait sur la sortie standard.

Si la fonction de Clih renvoie faux, témoignant un échec, un message d'erreur est affiché. Cependant, si `less` a été lancé, il masquerait l'affichage d'une erreur écrit sur la sortie d'erreur. Par conséquent, les fonctionnalités qui utilisent `SetPrintToLess()` affichent l'erreur en rouge gras sur la sortie standard.

Les lignes de texte préformaté ne sont pas adaptées à la largeur du terminal par Clih. Cependant, `less -S` permet d'éviter de les faire revenir à la ligne, et d'utiliser les flèches pour voir la suite des lignes. Un chevron surligné indique que les lignes ne sont pas terminées. La commande utilisée est donc `less -RS`.

Pour l'option `-version`, GenGraph CLI n'utilise pas Clih : le programme affiche simplement la toute première ligne du manuel. GenGraph v5.4 s'ennuyait à en extraire spécifiquement le numéro ; je n'ai pas eu l'impression qu'il y avait un intérêt à cela, et les options `--version` des autres programmes affichent des informations détaillées ; ainsi, j'ai choisi d'afficher simplement toute la ligne. On pourrait aussi songer à afficher l'entête complet (il ne contient qu'une seule autre ligne, contenant un lien vers le site officiel du projet et un lien vers le dépôt Git). Au cas où le fichier serait corrompu, la fonction vérifie que la ligne à afficher contient bien le texte `GenGraph v`, sinon elle affiche une erreur.

Voici quelques exemples d'exécution :

```
$ gengraph -vcolor ?

      -vcolor <mode> [<paramètre>]...

         Ces options permettent de modifier la couleur des sommets. Elles n'ont d'effet qu'avec les formats
         graphiques : DOT et ses dérivés ainsi que HTML (voir -format). Par défaut, les sommets sont de couleur
         Notez que les attributs par défaut des sommets (couleurs, formes, etc.) peuvent être modifiés directeme
         DOT (voir l'option -N de dot). Cependant l'option -vcolor permet d'individualiser la couleur d'un somme
         fonction de son degré par exemple. Ici le degré est le degré non orienté. Plusieurs options -vcolor peu
         être combinées.

         -vcolor deg[r]
```

```
         -vcolor degm

         -vcolor randg

         -vcolor kcolor <k>

         -vcolor pal <grad>

         -vcolor list

$ gengraph -vcolor degr ?

         -vcolor deg[r]

            La couleur dépend du degré du sommet (deg) ou du rang du degré du sommet (degr). Ainsi, les sommets
            plus petit degré obtiennent la première couleur de la palette, les sommets de plus grand degré la de
            couleur de la palette, et les autres sommets une couleur intermédiaire de la palette. Donc une seule
            couleur est utilisée si le graphe est régulier.

$ gengraph non-graph ?

Erreur : argument incorrect. Aide sur non-graph non trouvée.

$ gengraph ? "n'importe quoi" | tee

   Aucune entrée contenant le texte « n'importe quoi » trouvée.

$ gengraph -version

GenGraph v6.0a (juillet 2023) - © Cyril Gavoille, licence CeCILL-C
```

Il est précisé dans le manuel qu'il suffit de créer un tube vers un programme tel que `tee` (qui ici est équivalent à `cat`) pour échapper à l'interface de `less`. Il est probable que `less` va alors détecter qu'il n'a pas la main sur le terminal, et va donc se contenter de faire comme `cat`. La commande ci-dessus permet de voir que dans le manuel de GenGraph, on ne trouve pas « n'importe quoi », du moins dans les sections d'instructions.

Pour intégrer ce genre de sortie colorée au rapport, j'utilise les programmes `script` et `ansi2html` de cette façon :

```
$ script

... commandes ...

$ [Ctrl+D]

$ ansi2html < typescript
```

J'ai trouvé cette technique sur StackOverflow. `script` écrit dans le fichier `typescript` tout ce qui apparait sur le terminal, et `ansi2html` convertit les codes ANSI en balises HTML.

3.3.6.2. L'ouverture des fichiers texte et HTML

La fonction `GetFilePath(filename, folder)` renvoie le chemin complet supposé pour `filename` se trouvant dans `folder`. Ce dernier doit être l'une des variables globales du module Config : `APP_PATHS.doc` ou `APP_PATHS.data`. La fonction ajoute une barre oblique à la fin de `folder` s'il n'y en avait pas encore, et surtout elle gère le cas de l'installation portable avec chemins relatifs. Pour cela, elle utilise le chemin du programme dans `argv[0]` : elle enlève ce qui suit la dernière `/`, puis ajoute le contenu de `APP_PATHS.toRoot`, avant de mettre ceux de `folder` et de `filename`. Si `argv[0]` ne contient pas de `/`, alors la fonction ne fait aucun traitement et affiche un avertissement ; le fichier ne devrait alors être trouvé que si le répertoire en cours est le dossier principal de GenGraph.

Les fichiers sont toujours ouverts en mode binaire (`fopen(filePath, "rb")`) parce que je trouve que le mode texte ne fait qu'ajouter des complications.

Pour l'ouverture du manuel en HTML avec `-html`, j'ai codé deux fonctions :

- `PathToUrl(path)` rajoute `file://` devant `path`, et également le contenu de la variable d'environnement `PWD` si `path` ne ressemble pas à un chemin absolu.
- `OpenHtmlHelp(anchor)` rajoute `anchor` à l'URL précédé de `#`, et essaie diverses commandes pour essayer d'ouvrir cet URL : d'abord celle de la variable d'environnement `URL_OPENER` si elle est définie, puis `open` (pour macOS), puis `xdg-open` (pour les environnements de bureau libres), puis `cygstart` (pour Cygwin), puis `termux-open` (pour Termux),

puis `start` (pour Windows). La fonction `OpenFile()` du module ArgsOutput, utilisée pour `-visu`, essaie les mêmes commandes, avec une variable d'environnement différente (`FILE_OPENER`).

Toutes ces fonctions ont un peu de code essayant de prendre en charge les conventions de Windows, au cas où GenGraph CLI fonctionnerait un jour sous Windows sans Cygwin.

Sous Android, les navigateurs ne prennent vraisemblablement pas en charge le protocole `file://` pourtant si basique. L'ouverture d'un URL commençant par `file://` n'est pris en charge que par une visionneuse HTML un peu rudimentaire : elle n'exécute pas le code JavaScript, et ne prend pas en compte l'ancre initiale. Il n'est donc pas recommandé de consulter le manuel avec cette méthode sur Termux.

3.3.7. La complétion

À la toute fin du stage (fin juin), après m'être donné sur la compréhension et l'implémentation d'ILIGRA, bien que j'avais l'impression que mon algorithme était très bogué, j'ai voulu me détendre en développant la complétion des instructions de GenGraph avec l'interpréteur Bash. En effet, il m'apparaissait que la complétion serait fortement intéressante pour GenGraph puisqu'il fonctionne avec de très nombreux mots clés, souvent un peu longs. De plus, j'avais envie de voir ce qu'un outil comme Bash proposait pour mettre en place la complétion. Je m'attendais à devoir développer un programme qui recevait en arguments les mots de la ligne de commande à compléter, et qui affichait les mots possibles.

Ce n'était pas tout à fait aussi simple. J'ai effectivement développé ce programme, mais d'abord j'ai eu beaucoup de mal à lire et comprendre la documentation de Bash, que j'ai trouvée guère didactique. J'ai mis un certain temps à trouver ce qu'il fallait écrire pour que la complétion soit générée de façon toute simple par mon programme écrit en C. Ma première version utilisait la ligne de commande suivante :

```
complete -o default -C `realpath "$1"/tools/bash-completion` "${2:-gengraph}"
```

Le premier argument du script devait être le chemin vers le dossier `_build`, et le deuxième argument devait être la commande sur laquelle appliquer la complétion. `-o default` demande à utiliser la complétion par défaut (fichiers, variables…) si rien n'est affiché par mon programme (qui est dans `_build/tools/bash-completion`). Mon programme affiche ses résultats sur la sortie standard.

Ce que semblait principalement offrir Bash à mon programme était deux arguments : le mot à compléter, et le mot précédent. Je me suis au départ basé là-dessus pour offrir la complétion des noms de graphes et des options de premier et deuxième niveau.

J'ai trouvé cet outil simpliste vraiment convaincant : j'avais l'impression qu'il rendait l'utilisation de GenGraph vraiment plus agréable. Cependant, il était pour moi inacceptable de le livrer tel quel, notamment parce qu'il discriminait les options de troisième niveau (concrètement, cela ne concernait que les options `-check routing`). Je n'aurais pas pu tolérer de livrer un programme qui favorise certaines instructions par rapport à d'autres, et j'envisageais donc d'écrire dans le rapport que soit il faudrait finir le développement de la complétion pour qu'elle cesse d'être discriminante, soit il faudrait laisser tomber l'outil. Au lieu de ça, j'ai finalement fait le développement moi-même, et l'outil de complétion est maintenant bien plus puissant et générique, ce n'est plus un petit outil mignon qui regarde juste les deux derniers mots de la ligne de commande.

Par contre, comme expliqué plus haut, sa technique est misérable : plutôt que de vérifier la correspondance entre la ligne de commande et les formats d'instructions du manuel, l'outil utilise `getOptionCase()` pour générer toutes les instructions possibles, et c'est là-dessus que la correspondance est faite. De plus, étant donné que la longueur de l'instruction à compléter n'est pas connue, l'outil commence par la supposer très longue, et parcourt ainsi le fichier jusqu'à une dizaine de fois si jamais il ne trouve aucune instruction correspondante avec plus d'un mot. Cette dizaine de fois n'est même pas suffisante pour détecter complètement les instructions les plus longues, qui sont par exemple : `-check routing hash prime scenario nomem pair <u> <v> dcr <k>`. Les autres fonctions de Clih sont très optimisées et sont utilisées assez ponctuellement, mais la complétion est un outil destiné à être utilisé en permanence, et c'est son exécution qui est la plus lourde. C'est catastrophique. Mais au moins, le service est là, l'outil est quand même super pratique et il ne fait quasiment plus de discrimination. C'était suffisant à mes yeux pour intégrer l'outil à la procédure d'installation de GenGraph, ce que j'ai donc fait. Néanmoins, ça reste une sorte de prototype, plutôt que quelque chose dont je suis vraiment fier. On peut voir, en ajoutant un `fprintf()` dans la fonction générant la complétion, que l'outil génère presque 3 000 lignes (donc cas d'instructions) en un seul parcours du manuel, dont près de 2 400 pour toutes les possibilités pour `-check routing`.

Pour que le script soit installé, la logique de prendre seulement le chemin du dossier `_build` en argument ne convenait pas, puisque la structure de ce dossier n'est présente nulle part lorsque GenGraph est installé. Deux autres arguments sont donc requis à la place : le chemin vers le programme qui génère la complétion, et le chemin vers le manuel.

À la place de `-C`, j'ai utilisé `-F` qui exécute une fonction écrite dans le script. Cette dernière appelle le programme de complétion, mais fait quelques autres traitements. J'ai dû apprendre à utiliser un peu les tableaux de Bash, car lorsque la complétion utilise une fonction, le résultat ne doit pas être écrit sur la sortie standard mais doit être renvoyé sous la forme de tableau. La commande de Bash `compgen` génère des complétions prises en charge par Bash, ce qui permet de l'utiliser pour certains cas ; je m'en sers

pour les noms de fichiers.

La fonction de complétion traite particulièrement le cas où elle trouve `<fichier>` dans un format d'instruction : il la conduit à générer des noms de fichiers. Lorsque le manuel sera traduit en anglais, on pourra songer à utiliser `<file>` à la place.

Ce que j'ai trouvé vraiment bizarre dans la fonctionnalité offerte par Bash pour la complétion, c'est qu'il n'arrange vraiment pas la tâche, c'est-à-dire qu'il ne préformate pas la ligne de commande qu'il demande de compléter, et ne gère aucun cas par lui-même. Il me semble pourtant évident qu'il est inutile de demander à mon programme de faire la complétion si l'utilisateur est en train de saisir un nom de variable commençant par `$`, ou encore s'il s'agit d'une redirection de descripteur de fichier. Pire que ça, les arguments ne sont pas regroupés en fonction des échappements avec guillemets ou contre-obliques `\` : Bash offre une variable `COMP_CWORDS` qui est soi-disant un tableau contenant tous les mots de la ligne de commande, sauf qu'il ne sert à rien puisque le découpage n'est fait qu'en fonction des espaces. Je me suis embêté un moment à essayer d'utiliser `xargs` pour gérer cela, mais il n'est pas fait pour les lignes de commande non terminées. Il y a aussi la fonction POSIX `wordexp()` qui a le même problème.

En plus, Bash ne donne pas juste la partie de la ligne qui se trouve avant le curseur : il donne toute la ligne (variable `COMP_LINE`), ainsi que la position du curseur dedans (`COMP_POINT`). Bref, j'ai développé en C ce qu'il faut pour que ça me satisfasse pendant encore un petit moment. Le découpage de la ligne de commande prend presque autant de lignes de code que la recherche d'instructions correspondantes dans le manuel.

Pour activer la complétion, la norme est de placer le script dans `$prefix/share/bash-completion/completions` où `$prefix` est soit `/usr`, soit `/usr/local`, soit `~/.local`. Bash exécute automatiquement les scripts se trouvant dans ces dossiers. Il y a également le dossier `$prefix/share/bash-completion/helpers` dans lequel on peut placer des programmes accessoires au script principal. L'installation installe donc les fichiers dans ces dossiers. Elle ajoute au script les chemins vers le manuel et vers le programme du dossier `helpers`. Ces variables sont préfixées par `_GENGRAPH_`, et la fonction s'appelle `_gengraph_complete()`. Les préfixes sont importants, car ces identifiants sont intégrés à la session de l'utilisateur ! La convention est vraisemblablement d'utiliser un tiret bas comme préfixe.

Voici ce qu'on peut obtenir une fois que GenGraph est installé et que l'on démarre une nouvelle session avec Bash, avec un terminal de largeur 120 et en utilisant Tab pour activer la complétion :

```
$ gengraph (Tab)(Tab)
Display all 322 possibilities? (y or n) y
?               cross           fritsch         kakutami_5x+1   -norm           regular         treep
accessible*     crown           frucht          kakutami_7x+1   -not            rgraph          tree_part
alkane          cube            gabriel         kautz           not*            rig             triangle
antelope        cubic           gear            kite            not-wl*         ring            triplex
antenna         cuboctahedron   gem             kittell         -n-times        ringarytree     turan
antiprism       cycle           -gen            klein           octahedron      rlt             tutte
-apex           cylinder        -genc           kneser          odd             rng             tutte-coex
apollonian      dart            ggosset         knight          -op             robertson       tw
aqua            debruijn        giraffe         knng            outerplanar     rpartite        udg
arboricity      -dele           goldner-harary  kout            -output         rplg            udis
arytree         del-e*          gosset          kpage           -output-group   sat             udisd
associahedron   deltohedron     gpetersen       kpath           -output-union-d schlafli        -undirecte
banana          -delv           gpstar          kstar           paley           -seed           -ungroup
banner          desargues       gray            ktree           pan             -shift-ids      union*
barbell         diamond         grid            -label          pancake         shuffle         union-d*
bdrg            -directed       grotzsch        ladder          pappus          sierpinski      union-v*
behrend         -discard        -group          line-graph      parachute       soifer          uno
bidiakis        d-octahedron    -group-n        line-graph*     parapluie       -sort           unok
biggs           dodecahedron    haar            linial          pat             -sort-inv       upsl
biggs-smith     doily           hajos           linialc         path            split           upsld
binary          domino          half*           -list           paw             squaregraph     utility
bipartite       -dot            halin           -list-graphs    percolation     squashed        -variant
bpancake        dragon          hanoi           -list-options   permutation     stable          -vcolor
bull            dtheta          harborth        load            -permute        -star           -version
butterfly       -dup            hatzel          load+           petersen        star            -view
cactus          -dup-group      -header         loadc           planar          starfish        -visu
cage            durer           heawood         load-str        plrg            star-polygon    -visu-as
camel           dyck            heawood4        load-str+       point           sunflower       -vsize
-caption        egraph          helm            lollipop        polygon         sunlet          wagner
caterpillar     empty           -help           -loop           poussin         suzuki          wdis
cc*+            errera          herschel        maincc*         prime           syracuse        wdisd
centipede       expander        hexagon         margulis        -print          tadpole         web
-check          -extract        hexahedron      markstrom       -print-prop     td-delaunay     wheel
chess           -extract-all    hgraph          matching        -print-test     -test           whexagon
-chrono         fan             hourglass       mcgee           prism           tetrahedron     -while
-chrono-reset   farkas          house           mesh            -prop           tgraph          -width
chvatal         -fast           -html           mobius          pstar           theta           wiener-ara
```

Pour un mot encore vide, le point d'interrogation est toujours proposé (même si son utilisation causerait en fait une erreur). Cela

évite que la complétion soit faite pour une variable, dont la complétion affiche le nom. Le système de complétion donne beaucoup de liberté au programme et on pourrait imaginer faire des choses plus sophistiquées (en affichant des choses sur la sortie d'erreur notamment), mais je préfère me limiter à un système de complétion classique, car je ne souhaite pas perdre mon temps à développer des quantités de choses pour le terminal textuel, qui selon moi n'a de l'intérêt que tant qu'il est assez rudimentaire justement.

Clih est ainsi garnie d'une fonctionnalité de complétion. De nombreuses commandes fournissent ainsi la complétion pour leurs options.

3.4. Refonte de l'analyse des instructions

Au bout de deux semaines passées à découper le code en fichiers, j'ai fini par atteindre la fonction principale `main()` qui procède à la lecture des arguments du programme, et j'ai donc pu m'attaquer à sa restructuration. J'ai commencé par mettre ailleurs les autres fonctionnalités (c'est-à-dire le générateur et l'exécution des algorithmes avec `-check`), puis j'ai créé des modules pour différentes catégories d'options, et j'ai cherché à améliorer leur syntaxe. J'ai commencé à mettre en place la pile à la toute fin du mois de mai. Par la suite, j'ai fait en sorte qu'elle fonctionne bien avec la plupart des fonctionnalités. À la fin du mois de juillet, j'ai éliminé le système de l'option `-filter` de GenGraph v5.4, pour le remplacer par `-test` et `-prop` qui interagissent avec la pile.

À présent, l'analyse des arguments commence dans le module Args, qui utilise les modules suivants pour les différentes catégories d'options :

- ArgsGraphs : instructions de graphes ;
- ArgsMisc : options de premier niveau diverses ;
- ArgsOutput : options pour l'écriture de graphe ;
- ArgsCheck : options de `-check` et `-op` ;
- ArgsXY : options de `-xy` ;
- ArgsTest : options de `-test`.

Pour tous ces modules, il n'y a qu'un seul entête : `args.h`. Les six modules listés ci-dessus sont ceux qui vérifient à quel cas correspondent les arguments de la ligne de commande, et ce sont les seuls à le faire. Toutefois, les noms des propriétés de l'option `-prop` sont directement dans QueryProp.

J'ai tardivement divisé ArgsMisc en deux, de sorte à créer le module ArgsOutput qui fait la même taille, en même temps que j'ai découpé le module GraphOutput en les différents modules Out de GenGraph Formats. À l'origine, l'écriture de graphe était faite dans une grosse fonction `Out()` contenant deux niveaux d'instructions `switch`, et j'avais placé tout cela dans un module GraphOutput. Par ailleurs, avant la mise en place de `-test` et `-prop`, il y avait, à la place du module ArgsTest, le module CLIFilters (avant la création de GenGraph CLI, le module Args s'appelait CLI), sur lequel j'avais travaillé afin qu'il soit structuré de la même façon que les autres modules analysant des arguments, pour finalement tout effacer. Je n'en parle pas dans cette section, mais son code peut être trouvé dans l'historique du dépôt.

3.4.1. Les fonctions, structures et macros centrales

Comme déjà expliqué précédemment, dans le travail sur l'aide, sous-section La vérification de l'argument suivant, dans le but de mettre en place de nouvelles manières puissantes de lire les instructions de GenGraph, j'ai commencé par revoir ses petites fonctions et structures avançant pas à pas dans les arguments. Dans GenGraph v5.4, trois fonctions travaillaient sur un tableau global `ARGV` : `CheckHelp()`, `NextArg()` et `GetArgInc()`. La première vérifiait si l'aide était demandée via un point d'interrogation dans l'argument suivant, les deux autres servaient à avancer un pointeur global à l'argument suivant. Ces deux dernières étant redondantes, je n'ai gardé que le meilleur nom : `NextArg()` ; mais j'ai gardé la meilleure fonctionnalité qui était celle de `GetArgInc()` : avancer le pointeur, puis analyser l'argument pointé (alors que le `NextArg()` d'origine faisait le contraire).

Conformément à mes conventions, j'ai placé les différentes variables globales dans une structure commune : `ARGS`. Son contenu est le suivant :

gengraph-repo/src/gengraph-cli/args.h (extrait)

```
18 typedef struct {
19   int argc; // accès global à argc et argv
20   cstring const * argv;
21   cstring cur; // argument en cours (raccourci pour argv[argi])
22   int argi, caseI; // indices de l'argument en cours et du cas trouvé avec CASE_ARG_IN()
23 
24   int stackLen, stack[5]; // indices des paramètres désignant la section de l'aide en cours de traitement (Ch
25 
26   int lastInstrPos; // position du début de l'intruction précédente
27   int labelsPos[32]; // positions des étiquettes dans argv, pour -n-times
28 } args_data;
```

gengraph-repo/src/gengraph-cli/args.h (extrait)

```
29
30 extern args_data ARGS;
```

Comme expliqué précédemment, `CheckHelp()` se sert maintenant d'une pile pour déterminer l'instruction à afficher, et cette pile est également parfois utilisée pour déterminer l'instruction parente (pour distinguer entre `-check` et `-op` notamment).

Le numéro de l'argument en cours est `ARGS.argi`, mais `ARGS.cur` est généralement utilisé pour lire cet argument.

Dans GenGraph v5.4, la macro largement utilisée pour tester l'argument en cours s'appelait `EQUAL` et était utilisée ainsi :

gengraph-v5.4/gengraph.c (extrait)

```
20136     if EQUAL("-help"){ CheckHelp(&i); i--; }
20137     if(EQUAL("-help")||EQUAL("?")) Help(i);
20138     if EQUAL("-list"){ CheckHelp(&i); ListGraphs(); }
20139     if EQUAL("-version"){ CheckHelp(&i); Version(); }
```

Trouvant cette syntaxe un peu exotique, j'ai mis en place deux macros `CASE_ARG` et `CASE_ARG_IN` pour les remplacer, rapprochant l'écriture de celle des instructions `switch`/`case`. La différence entre `CASE_ARG` et le précédent `EQUAL` est qu'il ne faut plus écrire `if` devant ; et `CASE_ARG_IN` sert à appeler une petite fonction pour vérifier si l'argument en cours ne se trouverait pas parmi un tableau de plusieurs chaines de caractères. Chaque suite de `CASE_ARG` et `CASE_ARG_IN` doit être précédée d'un `if (false) {}` pour le bon fonctionnement des `else if`. C'est donc ce que j'ai fait à chaque fois.

GenGraph utilise le style K&R pour le placement des accolades, consistant à placer les accolades fermantes devant les `else`. Lorsque je sépare des groupes de `CASE_ARG`, cela donne un résultat un peu spécial mais charmant, par exemple :

gengraph-repo/src/gengraph-cli/args_misc.c (extrait)

```
41   } CASE_ARG_IN("-list", "-list-options", "-list-graphs") {
42     ListInstructions(ARGS.caseI);
43   } CASE_ARG("-version") {
44     Version();
45   } CASE_ARG("-html") {
46     SETT.htmlHelp = true;
47
48   // Contrôle du flux d'instructions
49   } CASE_ARG("-while") {
```

On voit que l'accolade fermante pour l'instruction `-html` se retrouve devant le `CASE_ARG("while")`, ce qui demande un peu d'attention mais donne un code agréable à lire et qui est aussi facile à manipuler lors de l'écriture. En effet, il suffit d'avoir bien une accolade fermante devant chaque `CASE_ARG`, et d'en avoir une toute à la fin. Et d'ailleurs, tout à la fin, il y a généralement un `else` pour signaler le fait que l'argument n'a pas été trouvé dans la liste testée.

gengraph-repo/src/gengraph-cli/args_misc.c (extraits)

```
32   if (false) {
33
34   // Aide
35   } CASE_ARG_IN("-help", "?") {

...  […]

122  } else {
123    return false; // aucune option correspondante
124  }
```

L'exécution des instructions se fait dans la fonction `ParseArgs(begin, end)`, appelée au milieu de la fonction `main()`, et constituant le cœur de GenGraph CLI. Elle se trouve dans `args.c` et utilise trois fonctions auxiliaires pour respectivement les options, les étiquettes et les instructions de graphes. Les étiquettes sont utiles aux boucles, une fonctionnalité nouvelle, présentée près de la fin de ce rapport.

3.4.2. Les macros, structures et fonctions pour les instructions de graphes

Les instructions de graphes sont les plus nombreuses. J'ai mis en place des macros, structures et fonctions rendant leur analyse concise au point que pour la grosse majorité, il suffit d'une ligne très concise. Cependant, les graphes prenant des paramètres très particuliers peuvent toujours nécessiter de nombreuses lignes de code — mais il est toujours possible d'écrire les instructions C que l'on souhaite.

Pour parvenir à ce résultat, mon constat principal a été le suivant : la plupart des instructions de graphes ne requièrent que des entiers, soit constants, soit saisis par l'utilisateur. D'autres choses se retrouvent souvent : des valeurs à virgule constantes, l'utili-

sation du graphe complémentaire (`-not`, devenu `-op not`), le fait que le graphe est orienté ou encore géométrique, et son type de géométrie. Toutes ces données sont écrites dans une structure, et pour faciliter l'instanciation d'un objet de cette structure, une macro est utilisée. Les deux macros centrales sont `GRAPH_CONF` et `GRAPH_ALIAS`. La première associe une instruction à la fonction qui a exactement le même nom, la deuxième associe une instruction à une fonction qui porte un autre nom. Ces macros :

- utilisent d'abord `CASE_ARG`, et s'arrêtent donc si l'instruction n'est pas la bonne ;
- créent une instance de `GraphConfig` avec les arguments donnés à la macro ;
- donnent cette instance à la fonction `PutGraphConfig()`, qui crée la requête qui correspond et la renvoie ;
- exécutent des instructions éventuellement fournies, pouvant utiliser la requête renvoyée.

Les macros `GRAPH_CONF` et `GRAPH_ALIAS` prennent trois grandes catégories de paramètres :

- le nom de l'instruction, suivi (dans le cas de `GRAPH_ALIAS`) du nom de la fonction C correspondante ;
- les valeurs de l'instance de `GraphConfig`, placées entre parenthèses (pour être mises ensuite entre accolades) ;
- éventuellement : les instructions supplémentaires à exécuter.

Dans le fichier `args_graphs.c`, les graphes sont classés selon les paramètres qu'ils prennent sur la ligne de commande, car c'était déjà plus ou moins le cas dans le code d'origine. Pour des graphes ne prenant aucun paramètre, l'écriture est vraiment concise :

gengraph-repo/src/gengraph-cli/args_graphs.c (extrait)

```
253   // Graphes de base
254     // 0 paramètre utilisateur
255   GRAPH_CONF(tutte, ())
256   GRAPH_CONF(icosahedron, ())
257   GRAPH_CONF(rdodecahedron, ())
```

Ensuite, pour savoir quels paramètres donner aux macros, il faut regarder les exemples existants, ou éventuellement le code de la structure `GraphConfig`. Voici, à titre d'exemple, le code pour les graphes `load`, `loadc` et `load-str` :

gengraph-repo/src/gengraph-cli/args_graphs.c (extrait)

```
245   GRAPH_CONF(load, (PARAMS_LOAD, I_PARAMS(0)))
246   GRAPH_ALIAS(loadc, load, (PARAMS_LOAD, I_PARAMS(0)), {
247       Q->genMode = GENMODE_INITONLY;
248     })
249   GRAPH_ALIAS(load-str, load, (PARAMS_LOAD, I_PARAMS(1)))
```

On y voit que le premier paramètre de la `GraphConfig` est une valeur spéciale pour certains formats particuliers. Ici, il s'agit de `PARAMS_LOAD`, qui indique qu'il faut prendre en argument une chaine de caractères sur la ligne de commande. Ensuite, la macro `I_PARAMS` permet de construire un tableau de constantes de type entier. Cet entier vaut 1 pour `load-str`, 0 pour les deux autres. Dans tous les cas, c'est la même fonction de classe de graphe qui sera utilisée : `load` (qui vient de GenGraph Formats). Pour `loadc`, il y a une instruction spéciale changeant le mode de génération de la requête. Cette instruction est placée entre accolades, et on pourrait en mettre d'autres.

Les autres paramètres de la `GraphConfig` sont soit nommés (dans le style C99), soit placés après la macro `I_PARAMS`. En effet, après les constantes de type entier viennent les saisies utilisateur de type entier, qui sont décrites par un nombre (le nombre d'entiers à saisir) et un indice (celui où placer les entiers saisis au milieu des constantes). Cette possibilité permet de couvrir la majorité des cas, mais si les entiers saisis par l'utilisateur ne doivent pas se suivre ou ne doivent pas être dans l'ordre, alors il est nécessaire d'écrire des instructions supplémentaires. Voici quatre exemples :

gengraph-repo/src/gengraph-cli/args_graphs.c (extraits)

```
296   GRAPH_CONF(gear, (I_PARAMS(), 1))
297   GRAPH_CONF(pstar, (I_PARAMS(2), 1))

...  […]

469   GRAPH_ALIAS(odd, kneser, (I_PARAMS([2] = 0)), {
470       int n; NextArg("n", "%d", &n);
471       Q->param.i[0] = 2 * n - 1;
472       Q->param.i[1] = n - 1;
473     })
474   GRAPH_ALIAS(star, rpartite, (I_PARAMS(2, 1), 1, 2))
```

Le graphe `gear` prend un paramètre de type entier. Le graphe `pstar` aussi, mais il prend en plus la constante 2. L'indice de la saisie n'est pas précisée, c'est donc la valeur par défaut : 0. Donc l'entier saisi par l'utilisateur sera placé avant la constante 2 dans la liste des paramètres entiers (`Q->param.i`).

Le graphe `odd`, basé sur `kneser`, a trois entiers constants valant 0 (il n'est nécessaire de définir que le dernier), mais les deux

premiers sont remplacés par des calculs sur une valeur `n` lue sur la ligne de commande. L'idée est donc d'allouer les valeurs nécessaires comme des constantes, pour les remplacer par la suite. Le graphe `star` quant à lui, basé sur `rpartite`, a deux constantes entières (2 et 1), puis 1 valeur lue sur la ligne de commande, placée à l'indice 2 donc après les deux constantes.

Pour les graphes `alkane`, le code est beaucoup plus long que celui d'origine, mais il est aussi plus robuste, et il pourrait aisément être factorisé pour des fonctionnalités du même genre, en remplaçant juste les tableaux de noms.

Pour parvenir à ce résultat avec ces belles macros bien pensées, j'avais commencé par modifier les `goto` préexistants afin qu'ils aillent vers le bas plutôt que vers le haut, ce qui rendait le code beaucoup plus facile à appréhender. Ensuite, j'ai fait beaucoup de rechercher/remplacer avec des expressions rationnelles, ce qui garantit que les valeurs d'origine ont été correctement conservées, car je ne les ai pas recopiées à la main (ou très rarement).

3.4.3. L'analyse des autres types d'instructions

Les modules ArgsMisc et ArgsOutput ne posent pas de problème particulier : ce ne sont que des instructions impératives utilisant extensivement `CASE_ARG` et `CASE_ARG_IN`. Elles utilisent aussi parfois la fonction `StrPosIn` (qui est la fonction utilisée par `CASE_ARG_IN`) pour associer à une chaine de caractères son indice, par exemple pour les différentes valeurs de l'option `-norm` :

gengraph-repo/src/gengraph-cli/args_misc.c (extrait)

```
197     Q->xy->norm = StrPosIn(NextArg("ℓ"), {
198         [NORM_L1] = "L1", [NORM_L2] = "L2", [NORM_LMAX] = "Lmax", [NORM_LMIN] = "Lmin",
199         [NORM_HYPER] = "hyper", [NORM_POLY] = "poly",
200       });
201     if (Q->xy->norm < 0) {
202       FatalError(GGERR_UNKNOWN_SUBOPTION, ARGS.argv[ARGS.stack[0]], ARGS.cur);
```

Pour les trois autres modules, c'est-à-dire ArgsCheck, ArgsTest et ArgsXY, j'ai utilisé la même méthode que pour ArgsGraphs : une structure, une fonction, et des macros facilitant leur utilisation.

Dans ArgsCheck, une grosse partie est dédiée au traitement des arguments de `-check routing`. On voit qu'il y a deux macros `CHECK_CONF` et `CHECK_CONF_SEVERAL`, associant une ou plusieurs instructions `-check` à un mode `CHECK_*`. En paramètres supplémentaires, il y a juste le nombre d'entiers à demander à l'utilisateur, le nombre de graphes à absorber, et si le graphe généré doit être ou non affiché.

Dans ArgsTest, on peut voir que le principal paramètre est un type de paramètres. Ainsi les tests prennent en paramètres soit d'autres tests, soit un sélecteur d'entiers, soit une constante entière, soit un graphe, soit un groupe de graphes. Pour `minor`, `sub` et `isub`, j'ai mis en place une macro spécifique : `TESTS_HAS_IS`, qui crée deux tests à la fois, par exemple `has-minor` et `is-minor` dans le cas de `minor`, tout en utilisant la fonction `qtest_minor()` dans les deux cas.

Pour prendre une série de paramètres différente dans ArgsTest, il faudrait créer un nouveau cas dans le `switch`, et prévoir une fonction pour la désallocation des données spéciales. En effet, à chaque test s'associe ses données stockées dans une variable de type `void *`.

Le module ArgsXY est le plus particulier, car j'ai souhaité y simplifier la lecture de champs arbitraires de la structure `query_xy`. Pour cela, j'ai créé une macro `PARAM`, et la macro `XY_CONF`, pour ses `XYConfig`, ne prend rien d'autre en paramètre qu'une série de `PARAM`. Cela est documenté dans le fichier. Il est de toute manière toujours possible de mettre les instructions que l'on souhaite après la création initiale de la `XYConfig`.

3.5. Implémentation des nouveaux algorithmes

L'algorithme du graphe chenille est implémenté dans `src/gengraph/graphs/base_caterpillar.c`, tandis que l'inversion de graphe ligne est implémentée dans `src/gengraph/algos/ops_linegraph.c`. Il y a aussi `src/gengraph/graphs/op_linegraph.c` dans lequel est implémenté la version paresseuse de l'opération graphe ligne. Consulter ces fichiers pour voir l'implémentation. Les explications ont été fournies dans la section Sujet : Utilisation de données scientifiques.

3.6. Organisation des contributions

C'est encore un sujet à réfléchir, qui me tient à cœur et j'espère ravira des personnes souhaitant contribuer au projet. Actuellement, le projet est hébergé sur l'instance GitLab privée de l'université de Bordeaux, sur laquelle seules les personnes ayant un compte à l'université peuvent soumettre des contributions.

Les contributeurs du projet sont listés à la fin des lisez-moi (`README.md` et ses traductions). Chaque nouvelle version doit être ajoutée à la fin du fichier `CHANGELOG.txt`.

Les conventions de codage de Cyril Gavoille étant à retravailler, je me suis permis d'adopter un autre style dans mes contribu-

tions, et même de reformater à ma manière les sections de code sur lesquels j'ai travaillés. Il s'avère qu'il n'est pas trop fastidieux de travailler avec les noms de variables à une lettre de Cyril Gavoille tant qu'on emploie toujours les mêmes pour les mêmes choses. En revanche, les instructions doivent être davantage aérées, notamment avec des espaces autour des opérateurs binaires.

Il semble écrit que le projet va utiliser Git, le gestionnaire de versions le plus courant aujourd'hui. Cyril Gavoille n'est pas familier de Git, aussi voici les commandes de base à utiliser pour contribuer :

```
# Faire une copie locale du dépôt distant
git clone git@gitub.u-bordeaux.fr:cygavoil/gengraph.git

# Mettre à jour la copie locale
git pull

# Voir l'état de la copie locale
git status

# Valider des modifications
git add <fichiers>...
git commit -m "Message"
# Envoyer les nouvelles validations sur le dépôt distant
git push
```

3.7. Relecture des fonctionnalités existantes

J'ai apporté un regard critique et de nombreuses améliorations aux fonctionnalités existantes, que ce soient des améliorations sur le plan des fonctionnalités ou sur le plan de l'organisation du code. Voici quelques exemples.

- La fonction de chargement de graphe depuis un fichier est désormais plus robuste, elle repère bien les cas d'erreurs et sait les décrire.
- La fonction de tri de liste d'adjacence est maintenant capable d'utiliser un tri par comptage pour trier les longues listes de voisins. C'est une optimisation.
- J'ai également ajouté une fonction `SortGraphIfNeeded()` que j'utilise parfois. Avant de trier un graphe, elle vérifie s'il n'est pas déjà trié, afin d'économiser du temps.
- J'ai placé les fonctions de comparaison de tri utilisant un tableau sous-jacent derrière des fonctions opaques renvoyant la bonne fonction de tri si on leur donne un tableau en paramètre.
- Mon travail sur la génération aléatoire a débouché sur l'utilisation de la bibliothèque PCG-Random très performante, qui permet de dupliquer les graines et de faire des sauts.

J'ai aussi renommé certaines instructions pour une meilleure organisation, j'ai étendu la portée de certaines instructions, ou encore j'ai fini d'implémenter certaines choses qui étaient commencées dans le code et trainaient. Voici quelques exemples.

- `-pos` est renommé en `-view pos`.
- `-xy grid` et `-xy zero` sont renommés en `-view grid` et `-view zero`.
- J'ai implémenté `-view border`.
- J'ai implémenté la lecture de sucres pour les sommets universels dans le format simple de graphe, voir ici.
- J'ai implémenté l'utilisation de `-vcolor` (sauf `-vcolor list`) par le format HTML.

3.8. Développement de nouvelles instructions

Ma très grande motivation m'a conduit à développer de nombreuses nouvelles instructions pour donner de la puissance à GenGraph.

J'ai présenté tout ce travail à Cyril Gavoille au cours de la première moitié de l'année 2024.

4. Bilan

J'ai accompli un travail colossal sur de nombreux plans pour améliorer le logiciel GenGraph : développement d'un nouveau système d'opérations, structuration du projet, création d'un dépôt, amélioration du format de l'aide, etc. Et ce travail va se poursuivre, plus doucement, dans les années à venir, car je n'en ai pas fini.

5. International part

As requested by the internationalization actions of the University of Bordeaux, in the following section I am discussing the international importance of a technology I used during my internship.

5.1. Graphviz

Graphviz's logo

The name "Graphviz" stands for "graph visualization," and Graphviz indeed is a package of graph visualization software. It was initiated in 1991 by the research labs of AT&T which is a mutinational telecommunications company. It is distributed under the terms of the Common Public License (predecessor of the Eclipse Public License), which is a weak-copyleft license. Therefore Graphviz is free software, with conditions close to those of GenGraph.

Graphviz is a powerful tool for graph drawing. It reads graphs written with a simple language of its own: DOT, with which one can describe a bunch of properties of vertices and edges. It is written in C so it is quite light, it supports many common and open output formats, and it has existed for long as free software. Furthermore, the DOT language essentially works like GenGraph's default format: list of (directed or undirected) edges. Thus it was evidence for GenGraph to use this tool to offer visualization of generated graphs. Graph layout (i.e. 2D-positioning of vertices and determination of edges' curves) is a complex subject, and the Graphviz project does not come without a few research papers.

Graphviz is specialized in static graph visualization, as opposed to other software that allows exploration. It has two main interfaces: a C library, and several command-line programs, the main one being `dot`. `dot` is typically used to convert input files written with the DOT language to other formats (mainly image formats). Graphviz supports several filters which offer various layout algorithms: GenGraph uses `neato` by default; the default `dot` filter is more suited for diagrams whose nodes will fit well on a grid (the steps of a procedure, for instance).

A few documentation software (such as AsciiDoc or Sphinx) support embedding of DOT graphs. vis.js, which is used by GenGraph for its HTML output format, can read networks written in DOT. Even more than that, Doxygen uses Graphviz to automatically generate inheritance diagrams for C++ classes.

The DOT format has become a kind of standard, as it is (at least partially) supported by a few other important graph visualization software other than Graphviz, such as Gephi and Tulip. Syntax highlighting rules for the DOT format have been written for several text editors, e.g. GtkSourceView (the base of GNOME editors), Ace and CodeMirror. The `.dot` extension may be used for DOT files, but `.gv` might be preferred because the `.dot` extension has been used for legacy Microsoft Word templates.

After Graphviz was born, other very powerful programs have come on the market. Gephi and Tulip are also weak-copylefted free software. As opposed to Graphviz, they allow graph exploration, and both are fundamentally French projects: Gephi was started in 2008 in the University of Technology of Compiègne, and Tulip is a project of the LaBRI itself with version 1.0.0 dating from 2001.

Sources:

- Graphviz's homepage: https://graphviz.org/
- DOT on Wikipedia: https://en.wikipedia.org/wiki/DOT_(graph_description_language)
- Graphviz on Wikipedia: https://en.wikipedia.org/wiki/Graphviz
- Doxygen Manual: Graphs and diagrams: https://www.doxygen.nl/manual/diagrams.html
- vis.js, Network documentation: Import data in DOT language: https://visjs.github.io/vis-network/docs/network/#importDot
- Gephi on Wikipedia: https://fr.wikipedia.org/wiki/Gephi
- Tulip's homepage: https://tulip.labri.fr/site/
- *Empirical Comparison of Visualization Tools for Larger-Scale Network Analysis*, on Hindawi, section Advances in Bioinformatics: https://www.hindawi.com/journals/abi/2017/1278932/
- Graphviz online viewer which uses Ace and highlights the DOT syntax: https://dreampuf.github.io/GraphvizOnline/
- DOT language plugin for CodeMirror: https://github.com/yskszk63/cm-lang-dot