Nanterre université

Détection de la complexité algorithmique d'une fonction à partir de son code source

Mémoire

Auteur :
Baptiste Rayer 36003587

Tuteur :
François Delbot

2018-2019

Remerciements

Je tiens à remercier M. François Delbot, mon tuteur, pour tout ce qu'il a apporté à ce mémoire. Sans ses conseils, ce mémoire ne serait pas aussi aboutit.

Je souhaite aussi remercier ma sœur pour toutes les relectures et corrections qu'elle a pu faire malgré son manque de connaissances dans le domaine informatique.

Je souhaite aussi remercier M. Hamouda Raïs et tous les membres des équipes d'Itnovem pour leurs investissem

Sommaire

Chapitre 1

Introduction

Vous retrouverez les fichiers source de ce mémoire sur le dépôt git-hub : `https://github.com/Batrayer/memoire-complexite`

1.1 Motivations

Nous souhaitons avec ce mémoire être capable de comparer les performances de deux codes. Ce mémoire s'inscrit dans le cadre de la formation au développement. Nous souhaitons ensuite appliquer cette comparaison entre le code produit par un étudiant et le code écrit par son enseignant. Nous souhaitons pouvoir comparer la solution qu'un élève pourrait implémenter à la solution du professeur.

1.2 Objectifs du mémoire

Ce mémoire a pour objectif de présenter une solution automatisée, permettant de déterminer la complexité algorithmique d'un code. Pour ce faire il est nécessaire d'expliquer ce qu'est la complexité algorithmique. Ainsi nous commencerons par présenter la machine de Turing et nous continuerons en déterminant une technique permettant de calculer la complexité en pire cas.

Par la suite, nous présenterons différents algorithmes ou code qui peuvent être analysés. Nous montrerons via ce chapitre qu'un algorithme n'est pas forcément un code, et que nous pouvons associer plusieurs implémentations à un algorithme. Nous reprendrons ensuite la méthode de calcul vue dans le premier chapitre afin de calculer la complexité algorithmique en pire cas du tri à bulle.

Dans le quatrième chapitre, nous montrerons qu'il existe des outils capables d'analyser un code C sans pour autant l'exécuter.

Finalement nous présenterons notre implémentation. Nous donnerons à voir les limitations que nous avons du appliquer à notre solution et nous montrerons le résultat que nous avons pu obtenir à ce jour.

Mais tout d'abord : qu'est-ce que la complexité algorithmique ?

Chapitre 2

La complexité algorithmique

Avant de pouvoir comprendre ce que la complexité algorithmique signifie, il est nécessaire d'avoir un idée plus globale du fonctionnement d'un outil informatique. Pour ce faire, Alan Turing a défini, en 1936, un concept qui peut nous permettre de mieux appréhender le fonctionnement d'un ordinateur.

2.1 La machine de Turing

2.1.1 Définition

La machine de Turing est un modèle théorique créé par Alan Turing en 1936. [8] Elle a pour but de définir la calculabilité d'un algorithme.

Une machine de Turing est composée de trois éléments. (Cf : Figure 2.1)

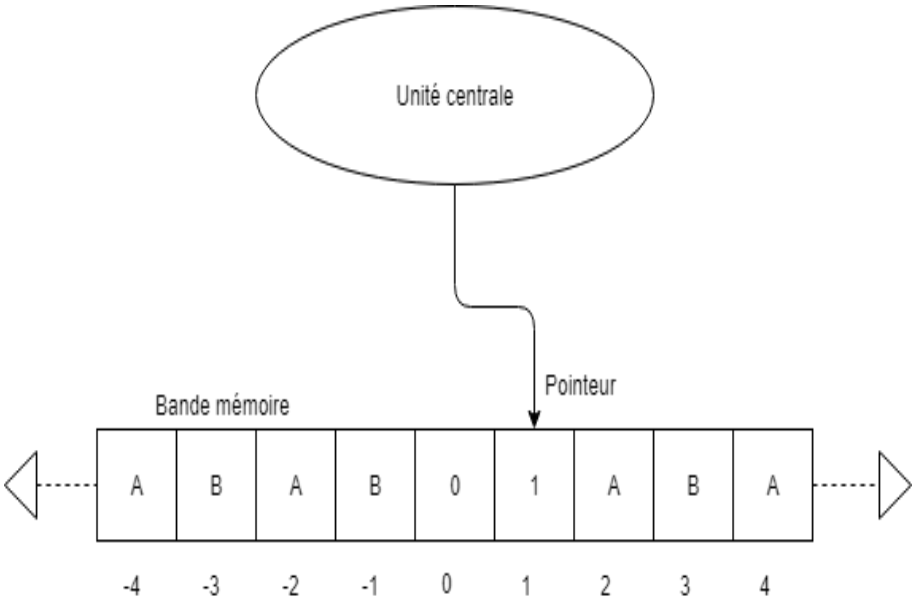

Figure 2.1 – Représentation d'une machine de Turing

1. Une unité centrale capable de dérouler le programme qui doit être exécuté par la machine. Son contenu dépend du programme à exécuter.
2. Une bande mémoire que l'on représente par une bande en papier et qui est d'une taille infinie. C'est d'ailleurs pour cela que la machine de Turing reste une machine abstraite. Un ordinateur, de nos jours, possède des centaines de giga-octets d'espace, mais nous ne sommes jamais à l'abri d'un débordement de mémoire. C'est un problème impossible à rencontrer avec une bande mémoire de taille infinie. Celle-ci est découpée en cases numérotées par des entiers relatifs. C'est sur cette bande mémoire que les résultats et les calculs seront écrits.
3. Un pointeur qui permet la lecture et l'écriture sur la bande mémoire. Ce pointeur agit conformément aux instructions de l'unité centrale. C'est cet élément qui fait le lien entre les deux autres.

La machine de Turing dispose aussi d'un ensemble fini S de symboles. Il s'agit de la liste des caractères qui peuvent d'apparaître dans une case de la bande mémoire. Parmi ces caractères, nous retrouverons un caractère spécial capable de représenter une case vide dans la bande mémoire. Nous disposons par ailleurs d'un ensemble E d'états possibles pour la machine de Turing. Enfin, nous avons une fonction de transition t qui permet de définir ce que la machine doit faire à chaque étape.

Il existe deux types de machine de Turing. D'un côté la machine de Turing déterministe qui, pour chaque état, possède une action à effectuer et une valeur pour faire avancer ou reculer le pointeur. Cette action permet d'affecter une nouvelle variable à la case mémoire pointée par le pointeur. De l'autre nous avons la machine de Turing non déterministe. Pour celle-ci, il est possible d'avoir plusieurs actions disponibles pour la machine et/ou plusieurs valeurs pour déplacer le pointeur. L'action effectuée sera donc choisie aléatoirement par la machine.

Nous nous intéresserons donc au fonctionnement d'une machine déterministe afin de déterminer la calculabilité d'une fonction.

2.1.2 Fonctionnement d'une machine de Turing déterministe

Dans le cas d'une machine de Turing déterministe, la fonction de transition t peut être écrite ainsi :

$$t = E * S * \{e', s', d\}$$

Nous retrouvons donc :
- E, l'état actuel de la machine.
- S, le symbole courant.
- e', l'état après exécution de t.
- s', le symbole qui va remplacer S.
- d, le changement de position du pointeur sur la bande (typiquement 1 ou -1).

Afin d'illustrer cette définition je vais modéliser une machine simple. Elle aura pour but de transformer les caractères A en B et B en A sur les cases paires dans la bande mémoire. Si la machine rencontre deux caractères C à la suite elle s'arrêtera.

Nous avons donc comme dictionnaire de données : {A, B, C}. Nous considérerons qu'il n'y a pas de caractères vides dans la bande.

t	A	B	C
e1	$(e2, B, 1)$	$(e2, A, 1)$	$(e4, C, 1)$
e2	$(e1, A, 1)$	$(e1, B, 1)$	$(e3, C, 1)$
e3	$(e2, B, 1)$	$(e2, A, 1)$	–
e4	$(e1, A, 1)$	$(e1, B, 1)$	–

Table 2.1 – Représentation d'une machine de Turing

— e1, e3, inverse A et B.
— e2, e4, avance le pointeur d'une case dans la bande sans modifier A et B.
— e3, e4, si C est rencontré alors l'algorithme s'arrête, sinon il exécute e1 ou e2.

Maintenant que nous comprenons au mieux le fonctionnement d'une machine de Turing, nous avons une meilleur compréhension du fonctionnement d'un ordinateur et d'un logiciel de manière générale. Dans le point suivant, nous présenterons comment il est possible de comparer des algorithmes avec un calcul de la complexité. Nous nous servirons de cette machine de Turing afin de définir ce que signifie une opération élémentaire.

2.2 Vitesse d'exécution et nombre d'opérations élémentaires

Le but d'une étude sur la complexité algorithmique d'une fonction est de pouvoir comparer deux fonctions différentes ; il existe plusieurs points de comparaison entre deux fonctions. D'un côté nous avons le temps d'exécution de la fonction, à savoir laquelle des deux est la plus rapide pour se terminer. De l'autre, l'espace mémoire utilisé par la machine lors de l'exécution de ces fonctions.

2.2.1 Évolution de la vitesse de calcul des ordinateurs

Dans ce mémoire nous nous intéresserons principalement à l'étude des temps d'exécution de fonction. Une approche naïve pour calculer le temps d'exécution des fonctions pourrait consister à mettre deux points d'arrêt dans le code. Le premier avant l'exécution de la fonction, et le second après. Cependant, il ne s'agit pas d'un moyen fiable pour exécuter une comparaison. Pour démontrer ce fait, nous allons vous présenter la loi de Moore.

2.2.1.1 Loi de Moore

La loi de Moore énoncée par Gordon Moore, annonce que, toutes les deux années, le nombre de transistors présents dans les micro-processeurs des ordinateurs doublerait. Un transistor est un élément primordial des composants électroniques et des circuits logiques. Avec le doublement du nombre de transistors dans les micro-processeurs, la puissance de ceux-ci augmente drastiquement et un plus grand nombre d'opérations peut être effectué. Cette loi est approximativement suivie depuis qu'elle a été énoncée. Cependant avec les dernières versions des transistors, nous arrivons à une taille tellement petite par transistor, qu'il est extrêmement difficile de pouvoir réduire leurs

tailles afin d'augmenter le nombre de transistor. Suite à ces problèmes, le PDG de Intel, un constructeur de micro-processeur, annonce maintenant que cette durée est passée à 2.5 ans.[9]

Avec toutes ces évolutions, comparer le temps d'exécution du même programme ne peut pas être fait entre deux machines différentes. Cependant, les codes peuvent être comparés autrement que par leur durée d'exécution.

Afin de comparer deux codes, il est nécessaire de pouvoir différencier les codes sans les exécuter. C'est donc à partir de ce point qu'intervient la complexité algorithmique.

2.2.1.2 Définition de la complexité

La complexité d'un algorithme est la quantité de ressources nécessaires pour traiter les entrées de cet algorithme.[4]

Au lieu de calculer combien de temps le code va utiliser pour se terminer, nous pouvons chercher combien d'opérations l'ordinateur doit effectuer pour l'achever. Une fois ce nombre déterminé, nous avons l'occasion de calculer combien de temps la machine devrait utiliser ; ceci se calculant en secondes. Et ce pour chaque machine existante, peu importe la configuration de celle-ci.

2.2.2 Définition d'une opération élémentaire

Nous définirons une opération élémentaire comme étant une opération qu'une machine de Turing peut effectuer. Cela regroupe :

— Accès à la mémoire.
— Bouger le pointeur.
— Écrire dans la mémoire.
— Changement d'état.

Dans le cadre de l'étude d'un code C, nous retrouvons ces opérations élémentaires dans les calculs, les comparaisons, l'affectation de variables... Il s'agit de toutes les petites instructions du code.

Nous comprenons maintenant ce qu'est la complexité algorithmique et nous avons comme unité de mesure le nombre d'opérations élémentaires. Cependant, est-il vraiment utile de calculer le nombre exact d'opérations élémentaires ? Nous constaterons dans la partie suivante qu'une approximation permet parfois d'avoir une réponse rapide sur une comparaison de deux algorithmes.

2.3 Évolution asymptotique du nombre d'opérations élémentaires

Lorsque l'on analyse un algorithme dans des cas extrêmes, nous avons tendance à simplifier le nombre d'opérations effectuées par l'algorithme. En effet, l'étude de l'ordre de grandeur de celui-ci est plus pertinente que le fait de calculer le nombre exact d'opérations. Ce calcul, en plus d'être complexe et de consommer beaucoup de ressources, n'apporte que peu d'informations dans le cas où les données sont de taille suffisamment grande. Pour illustrer mes propos, observons le graphique suivant.

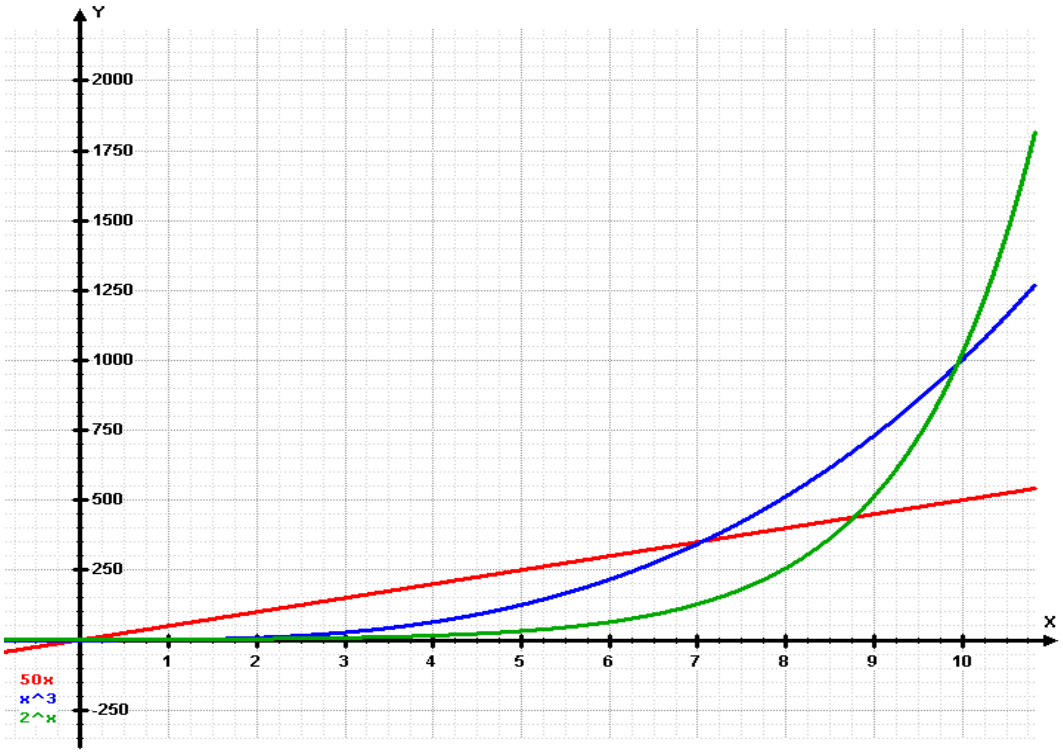

Figure 2.2 – Représentation de trois fonctions

Sur ce graphique, nous pouvons observer trois courbes.

1. En rouge, $f(x) = 50y$
2. En bleu, $f(x) = y^3$
3. En vert, $f(x) = 2^y$

On remarque rapidement que, malgré un nombre très important, pour le facteur de la première fonction, la deuxième devient supérieure à partir de $x > 7$. Et selon la même logique, nous pouvons observer que la troisième fonction est supérieure pour $x >= 10$.

Cette observation nous permet de déterminer que si deux algorithmes ne sont pas du même ordre de grandeur, il n'est pas nécessaire de calculer un nombre précis d'opération pour les comparer.

Dans le tableau suivant nous retrouverons des exemples d'ordre de grandeur d'algorithmes. [1]

Temps	Type	Exemple
$O(1)$	Complexité constante	Accès à une case d'un tableau
$O(log(n))$	Complexité logarithmique	Recherche dichotomique
$O(n)$	Complexité linéaire	Parcours d'une liste
$O(n^2)$	Complexité quadratique	Tri à bulle
$2^{O(n)}$	Complexité exponentielle	Brute force sur le problème du voyageur de commerce

Table 2.2 – Ordre de grandeur du nombre d'opérations exécutées par un algorithme

Maintenant que nous comprenons qu'un ordre de grandeur est utile pour comparer deux algorithmes, comment calculer précisément le nombre d'opérations faites par une machine lors de l'exécution de ces mêmes algorithmes ? Lors de la rencontre d'un if, que doit-on choisir comme parcours ? Doit-on entrer dans le cas où la condition est vraie ? Au contraire, faut-il analyser les deux résultats et faire la moyenne ? Dans la section suivante, nous déterminerons que la réponse à ces interrogations dépend de la complexité qui nous intéresse.

2.4 Les différents types de complexité

2.4.1 En meilleur cas

La complexité en meilleur cas consiste en l'étude d'un algorithme lorsque tous les éléments sont favorables à celui-ci. Par exemple sur un algorithme de tri de tableau, le tableau en entrée sera déjà trié. Afin de calculer le nombre d'opérations effectuées lors d'une étude en meilleur cas, il est nécessaire de suivre toutes les routes de l'algorithme contenant le moins d'opérations.

Cette complexité permet de répondre à la question : "Quelle est la durée minimale d'exécution de mon algorithme ?". Un développeur pourra calculer la durée de réponse en meilleur cas. Mais si à partir de ce calcul, le temps excède le maximum demandé, le développeur devra utiliser un autre algorithme plus performant afin de répondre à la problématique de son application.

2.4.2 En moyenne

La complexité en moyenne peut être considérée comme étant la plus représentative d'un cas réel. Un utilisateur souhaitant trier un tableau ne connaît pas forcément le contenu exact du tableau en entrée. S'il exécute plusieurs fois la fonction de tri avec des tableaux différents, il souhaite pouvoir avoir une idée du temps de réponse de l'algorithme. Cependant, l'étude de cette complexité repose sur des problèmes de distribution.

Un problème de distribution correspond à une paire entre un problème de décision et une collection de distribution. Historiquement, l'étude de la complexité en moyenne analysait des problèmes ayant des chances équiprobables.[7] C'est-à-dire que chaque entrée avait la même probabilité de se produire. Nous ne pouvons pas considérer de nos jours que toutes les entrées d'une fonction soient équiprobables. Il faudrait déterminer la probabilité de chaque cas et, pour ce faire, une étude statisticienne de la fonction est nécessaire. L'objectif de ce mémoire n'est pas d'automatiser une étude des fonctions afin de déterminer la probabilité des valeurs d'entrée mais plutôt de calculer la complexité algorithmique de la fonction.

2.4.3 Pire cas

2.4.3.1 Présentation

L'analyse de la complexité en pire cas correspond au cas où le jeu de données sera le moins favorable à l'algorithme. Cette complexité est importante lorsqu'un système doit répondre en un temps donné. Il s'agit d'une problématique fréquente dans les systèmes embarqués, par exemple dans un outil permettant de calculer les résultats de capteurs. Dans cette optique, analyser la complexité en pire cas permet d'estimer si, avec une machine spécifique, l'algorithme terminera son travail dans les temps.

2.4.3.2 Évaluation du nombre d'opérations en pire cas

Afin de calculer le nombre d'opérations en pire cas il faut tout d'abord déterminer quelles sont les "routes" possibles que les données peuvent emprunter dans le code.

A partir de ces chemins différents il est nécessaire de calculer le nombre d'opérations effectuées pour chacune de ces routes et de choisir celles qui contiennent le plus d'opérations. Pour information, dans le code suivant, nous avons un total de trois opérations.

```
i = i * 2 + 7
```

Listing 2.1 – Code d'une affectation de variable avec trois opérations

La première opération est la multiplication de i par 2. Cette opération est suivie par un ajout de 7 au calcul précédent. Nous avons donc deux opérations. La troisième opération correspond à l'affectation du résultat de ces deux opérations à la variable i.

Dans le cas où une instruction élémentaire aurait un coût i, nous noterons le nombre d'opérations effectuées lors de l'exécution de la fonction

$$T(i) = 1$$

Le coût d'exécution d'une fonction f comprenant deux instructions i, j peut être calculé de la manière suivante :

$$T(i, j) = T(i) + T(j)$$

Jusqu'ici il s'agit de la méthode de calcul simple. La première spécificité du calcul du nombre d'opérations en pire cas vient lors des conditions. Le nombre d'opérations en pire cas d'une condition est égale au plus grand nombre d'opérations entre la fonction dans le if et la fonction dans le else. Considérons une condition simple que nous nommerons COND. Si la condition est valide nous exécutons le code IF() sinon nous exécutons le code ELSE() Le nombre d'opérations de cette condition peut être écrit ainsi :

$$T(COND) = Max(T(IF()), T(ELSE())$$

Le calcul du nombre d'opérations effectuées en pire cas d'une itération est fait par l'addition du nombre d'opération faite dans la boucle multiplié par le nombre de fois que la boucle itère. Prenons une itération et nommons la ITER. Cette itération a pour

maximum de nombre d'itérations n. Dans cette itération nous exécuterons le code de la fonction nommée FUNC.

$$T(ITER) = \sum_{n=0}^{n-1} T(FUNC)$$

A partir de ces quatre modèles de calcul il devient possible de déterminer le nombre d'opérations effectuées en pire cas de la majorité des fonctions.

2.4.4 Apport de ces différentes complexités

Avec les calculs de chaque complexité, le développeur peut apprendre le comportement de son algorithme. Il peut borner ces résultats, définir en moyenne quel est le temps de réponse de son algorithme. Cela permet de décider de manière pratique quelle est la meilleure procédure à implémenter dans son code.

Grâce à ces calculs, il peut montrer des métriques afin de pouvoir comparer deux algorithmes, et convaincre son entourage de l'algorithme qui est le plus favorable dans le cas courant.

Il reste notable qu'en informatique théorique, ces trois types de complexité ne sont pas les seuls qui existent. Nous pouvons notamment retrouver des études sur la complexité paramétrée, qui s'occupe de rechercher la complexité d'un algorithme en fonction des paramètres et non en fonction de la taille de ces paramètres.

Cependant, il existe une grande différence entre un outil qui a été développé et un algorithme. Ce qui intéresse un développeur c'est de savoir si son outil répondra dans les temps. Pour comprendre cela, nous allons présenter les différences entre un algorithme et son implémentation dans le chapitre suivant.

Chapitre 3

Complexité et code source

Afin de montrer qu'un algorithme est différent d'une implémentation je vais tous d'abord présenter un algorithme en pseudo-code. Ensuite je vais présenter différentes implémentations de celui-ci. Pour illustrer mes propos je me servirai de l'algorithme du tri à bulle comme exemple.

Une fois que nous comprendrons les différences entre un algorithme et ses implémentations, nous calculerons la complexité algorithmique en pire cas d'une des implémentations du tri à bulle. Ce calcul sera effectué en suivant les règles établies dans le chapitre précédent.

3.1 Un algorithme en pseudo-code

Il est important de rappeler qu'un algorithme n'est pas une implémentation. Prenons la définition du dictionnaire. "Algorithme : Ensemble de règles opératoires dont l'application permet de résoudre un problème énoncé au moyen d'un nombre fini d'opérations" [1]. Un algorithme peut être traduit, grâce à un langage de programmation, en un programme exécutable par un ordinateur. Cet ensemble de règles ne permet pas de résoudre un cas pratique, un problème particulier, mais donne les indications pour créer un code permettant de résoudre ce cas. Prenons l'algorithme du tri à bulle :

Données : $tab, taille$ Un tableau ainsi que la taille de ce tableau
Résultat : Un tableau trié
pour i allant de $taille - 1$ à 1 faire
  pour j allant de 0 à $i - 1$ faire
    si $tab[j + 1] < tab[j]$ alors
      échanger$(tab[j + 1], tab[j])$
    fin
  fin
fin

Algorithme 1 : Tri à bulle$(tab, taille)$

Ici nous avons l'équivalent d'une recette de cuisine avec les variables comme ingré-

1. Définition extraite du dictionnaire Larousse en ligne `https://www.larousse.fr/dictionnaires/francais/algorithme/2238`

dients et comme recette, la méthode de programmation. Ce pseudo-code correspond à un langage abstrait qu'un humain pourra interpréter et implémenter sous la forme d'un code. Pour conclure, il est important de comprendre qu'un algorithme n'est pas une spécification formelle. Différents développeurs suivant un algorithme obtiendront différents codes en fonction de leurs implémentations.

3.2 Un même algorithme, plusieurs variations

Nous pouvons observer aussi qu'il existe pour un même algorithme plusieurs façons de l'écrire dans un langage commun. Reprenons le tri à bulle. Il s'agit d'un algorithme permettant de trier un tableau. Ce tri lit les éléments du tableau par ordre croissant des indices et inverse les nombres qui ne sont pas ordonnés.

```c
void tri_a_bulle(int tab[], int taille)
{
    int i,j,tmp;
    for(i=0; i<taille; i++)
    {
        for(j=0; j<taille-1; j++)
        {
            if(tab[j]>tab[j+1])
            {
                tmp=tab[j];
                tab[j]=tab[j+1];
                tab[j+1]=tmp;
            }
        }
    }
}
```

Listing 3.1 – Tri à bulle en C

Mais rien n'empêche le développeur de trier le tableau dans le sens inverse. Il peut commencer à lire le tableau par les indices descendants, l'algorithme utilisé reste le même. Cela parait anodin comme changement mais les répercussions sont importantes dans le code. Par exemple, les conditions des itérations changent entièrement.

```c
void tri_a_bulle_inverse(int tab[], int taille)
{
    int i, j, tmp;
    for(i=taille-1, i>0; i--)
    {
        for(j=taille-1; j>1; j--)
        {
            if(tab[j]<tab[j-1])
            {
                tmp=tab[j];
                tab[j]=tab[j-1];
                tab[j-1]=tmp;
            }
        }
    }
```

```
|| }
```

Listing 3.2 – Tri à bulle triant dans le sens inverse

Dans le premier cas, il est facile, en observant la condition, de savoir combien d'itérations la boucle va faire et quelles variables sont impliquées dans le calcul de la complexité. Dans le cas précédent nous pouvons observer que la variable taille est l'élément clé de ces itérations. Dans le second cas, il est impossible de déterminer la variable impliquée dans celles-ci à partir de la condition. Il est nécessaire d'observer les initialisations des variables pour se rendre compte que la variable taille est l'élément clé de ces itérations.

Nous venons de montrer que pour un algorithme, nous pouvions retrouver plusieurs interprétations. Chaque interprétation conduit donc à des codes différents. Mais pour la même interprétation d'un algorithme, il est possible d'avoir un code différent. Dans le tri à bulle, nous retrouvons des boucles qui itèrent sur les élément d'un tableau. Les développeurs ont accès à trois structures de contrôle pour effectuer les itérations en C : le for, le while ou encore le do while. Le code est différent entre ces trois cas bien que l'algorithme soit identique.

3.3 Exemple d'évaluation de la complexité en pire cas : le tri à bulle

Reprenons le code C du tri à bulle. Je vais à présent dérouler la méthode vue en 2.4 afin de calculer le nombre d'opérations élémentaires effectuées en pire cas pour ce code C.

Nous avons dans un premier temps trois déclarations de variables, ce qui correspond à trois opérations élémentaires exécutées peu importe les données envoyées ; la complexité est donc constante

$$T(i) = 3$$

Je vais ensuite calculer le nombre d'opérations élémentaires du code qui se trouvent à l'intérieur de la condition et des deux itérations. A la première ligne de ce code nous avons une affectation de valeur soit une opération élémentaire. A la deuxième et à la troisième ligne, nous retrouvons deux opérations élémentaires. La première, l'affectation de variable et la deuxième le calcul de j+1. Nous obtenons donc un total de cinq opérations élémentaires, soit

$$T(i) = 5$$

Maintenant ajoutons la condition. La condition représente à elle seule deux opérations élémentaires, une pour j+1 et une pour la comparaison entre deux variables. La condition ne possède pas de else, et donc ne donne pas d'instructions supplémentaires si elle n'est pas valide. Comme nous cherchons le pire cas, nous allons considérer que la condition est toujours invalide. Nous obtenons donc pour cette condition en pire cas sept instructions élémentaires pour une complexité toujours constante de

$$T(i) = 7$$

Je vais maintenant ajouter le calcul de la deuxième itération. Cette itération est composée d'une affectation effectuant une opération élémentaire. Cette affectation n'est effectuée qu'une fois, à l'initialisation de la boucle. Nous avons une incrémentation de la variable y qui est exécutée autant de fois que la boucle itère. Ensuite nous avons la condition de l'itération qui elle aussi est exécutée autant de fois que la boucle itère. Et finalement les instructions élémentaires à l'intérieur de la boucle sont, là encore, répétées autant de fois que la boucle itère. Cette condition est composée de deux opérations élémentaires : le calcul de taille - 1 et la comparaison entre j et le résultat de ce calcul. Nous avons donc en nombre d'opérations élémentaires :

$$1 + ((1 + 2 + 7) * nombre\ d'itrations\ de\ la\ boucle)$$

En observant la condition de la boucle ainsi que l'initialisation des variables, nous pouvons en déduire que le nombre d'opérations élémentaires est de $1 + 10 * (taille - 1)$. Pour obtenir la complexité en pire cas, nous considérerons donc que la boucle est exécutée le maximum de fois. Nous observons donc notre premier changement d'ordre de grandeur dans la complexité de la fonction. En effet nous n'avons plus une complexité constante mais une complexité dépendante de la variable taille. La boucle étant répétée taille fois , nous avons une complexité qui est maintenant linéaire :

$$T(i) = (1 + 10 * (taille - 1))$$

Elle est réductible par ordre de grandeur à

$$O(10 * taille) => O(taille)$$

En appliquant le même procédé pour la première boucle, nous observons un nombre d'opérations élémentaires égal à 1 + (2 + 1+nombre d'opérations internes à la boucle)∗ nombre d'itérations de la boucle. A nouveau nous pouvons déterminer le nombre d'itérations de cette boucle : la boucle itère taille fois.

Nous avons maintenant tous les éléments nécessaires pour calculer le nombre d'opérations élémentaires en pire cas, et ensuite déterminer la complexité en pire cas de cette fonction.

$$4 + (3 + (10 * (taille - 1))) * taille$$

Ce qui nous donne un nombre d'opérations en pire cas de

$$T(i) = 4 + (3 + (10 * (taille - 1))) * taille$$

Elle peut être réduite en :

$$O((10 * taille) * taille) => O(taille^2)$$

Il s'agit donc visiblement d'une complexité quadratique.(Cf. : 2.2) La complexité est donc fortement liée à la taille. L'objectif de ce mémoire est d'automatiser ce procédé. Cependant est il vraiment concevable de pouvoir faire une automatisation possible pour tous les cas ? Nous observerons plus tard dans ce mémoire 5.1 que ce n'est pas possible, et qu'il sera nécessaire de limiter le code. Pour cela, et avant de déterminer un sous-langage qui permette de travailler sur un maximum de code, nous allons observer les outils existants.

Chapitre 4

Les outils existants

4.1 Complexité cyclomatique

Il n'existe à ce jour pas d'outils ayant pour but de calculer la complexité algorithmique d'une fonction. Nous retrouvons tout de même certains outils comme Codacy qui détecte des fonctions ayant une "forte complexité". Toutefois, il ne s'agit pas ici de complexité algorithmique à proprement parler, il s'agit d'une mesure permettant de déterminer quelle est la taille de la fonction.

Cette mesure permet de déterminer si une fonction est trop longue. En effet, si la complexité cyclomatique d'une fonction est trop importante, la compréhension, l'entretien et les tests de cette fonction par un développeur sont rendus plus difficiles. Il ne s'agit pas d'une mesure inintéressante à étudier, cependant elle ne fait pas partie de notre centre d'intérêt dans le cadre de ce mémoire.

Je vais donc maintenant m'intéresser aux outils qui pourraient nous aider à analyser du code. Pour commencer je vais présenter un outil capable d'analyser le code sans l'exécuter.

4.2 Linter

Les linters sont des programmes informatiques permettant d'effectuer une analyse statique sur le code. Le nom linter a pour origine l'outil lint sur unix. Cet outil avait pour but, à l'époque, de compléter les compilateurs. En effet, afin d'obtenir une compilation rapide, très peu de vérifications étaient effectuées sur le code du développeur. Ainsi le lint permettait de vérifier plusieurs problèmes récurrents du code : l'initialisation ou l'utilisation de variables, le flot de contrôle, les appels de fonction, etc. [10]

Ces fonctionnalités sont maintenant souvent présentes dans les compilateurs. Par exemple GCC avec l'option –WALL affichera les variables non utilisées. Nous retrouvons même des plugins pour IDE qui implémentent des linters. L'utilisation d'un linter de nos jours est considérée comme une bonne pratique de développement. Ils ne vérifient plus uniquement les erreurs possibles de code, ils prennent maintenant en compte les mauvaises pratiques de développement.

Afin de créer un code durable et entretenable afin de palier aux évolutions des besoins, des outils d'intégration continue ont été créés. Parmi ces outils, certains, notam-

ment Codacy, utilisent des linters afin d'analyser le code et de reporter les problèmes possibles. Avec les résultats de ces analyses, ils peuvent donner une note à un projet qui reflétera la qualité du code.

Nous avons donc un outil qui permet de faire une analyse de code, sans exécuter ce code, ce qui correspond exactement à notre besoin. Cependant, les linters actuels pour le C (OCLint, uncrustify) ne fournissent pas de bibliothèque utilisable afin d'analyser le code. Nous avons cependant noté précédemment qu'avec les évolutions des compilateurs, et afin d'optimiser au mieux les programmes, les compilateurs récents appliquaient une analyse statique sur le code source.

4.3 Compilateur

Avant de définir ce qu'est un compilateur, il est important de rappeler les multiples types de langages informatiques.

4.3.1 Rappel sur les types de langages

Nous pouvons retrouver plusieurs niveaux d'abstraction pour les langages.[2]

— Les langages dédiés (ou Domain Specific Language) – tels que MatLab, SQL, etc.– qui ont pour but de répondre à un besoin applicatif spécifique.
— Les langages de haut niveaux – comme java, C, COBOL, etc. – qui permettent de répondre à de multiples problèmes à l'aide d'une écriture plus proche de l'humain.
— Les langages intermédiaires (Gimple, Bytecode Java),qui correspondent au code de transition entre les langages de haut niveau et les langages machine. Leur code est analysable par une machine abstraite afin de détecter les erreurs et d'optimiser au mieux le code machine.
— Les langages machines (x86, ARM), qui correspondent à une suite de bits qui est comprise par le processeur de la machine.

4.3.2 Définition d'un compilateur

Un compilateur est un programme informatique qui transforme un code source écrit dans un langage, le langage source, en un autre langage, le langage cible. [3] Le compilateur peut donc être associé à un traducteur entre deux langages. [2]

Un compilateur peut fonctionner de trois manières différentes :

1. Compilation : le compilateur traduit le langage en entrée vers le langage en sortie. Dans ce cas le résultat des deux programmes est identique.
2. Interprétation : le compilateur prend un programme en entrée avec des données et calcule le résultat.
3. Machine virtuelle : le compilateur traduit le langage d'entrée vers un langage intermédiaire puis interprète celui-ci.

Il existe deux compilateurs principaux pour le langage C. Nous avons d'un côté Clang, qui à été construit avec comme idée d'être une API [1] [5], et donc un outil

1. Interface de programmation qui fournit un ensemble de fonctions et services au développeur.

réutilisable par des développeurs externes. De l'autre GCC qui est un compilateur plus générique du langage C.

Ces deux compilateurs effectuent le travail de compilation vu auparavant. Ils implémentent tous les deux une étude statique du code à la pré-compilation, comme un linter. Et, point intéressant, ces deux compilateurs permettent d'extraire l'arbre syntaxique abstrait du code C.

Nous allons maintenant définir, à l'aide de cas pratiques, ce qu'est un arbre syntaxique abstrait et comment, grâce à ce graphe, nous pouvons analyser le code sans l'exécuter.

4.4 Arbre syntaxique abstrait

4.4.1 Définition

Un arbre de syntaxe abstraite est une modèle capable de représenter du code sous la forme d'un graphe. Ce graphe possède comme noeud chaque interaction du code. Ce graphe est un élément clé pour la visualisation du code. Il permet de modéliser tous les codes que nous pouvons lui fournir.

Une fois la modélisation obtenue, il devient plus simple de pouvoir analyser un code. Il s'agit du point clé de notre automatisation d'analyse de complexité.

Je vais présenter, à titre d'exemple, deux types nœuds que nous pouvons retrouver dans un code : les nœuds opération binaire et les nœuds conditionnels. Il s'agit de nœuds simples que nous rencontrerons dans la majorité des fonctions en C. D'autres nœuds aurait pu être inclus, comme les nœuds d'itération (for, while, do while).

4.4.2 Les nœuds "opération binaire"

Ces nœuds représentent tous les calculs dans les outils. Ils sont représentés par trois éléments : un côté droit, un côté gauche et un opérateur. Les deux côtés de l'opération peuvent être représentés par une nouvelle opération binaire. L'opérateur est la méthode de calcul entre les éléments gauche et droit, par exemple +, -, /, *, %, etc. Nous pouvons donc écrire le code suivant :

```
i = i * 2 + 7
```

Listing 4.1 – Code d'une affectation de variable

Sous la forme de l'arbre syntaxique suivant :

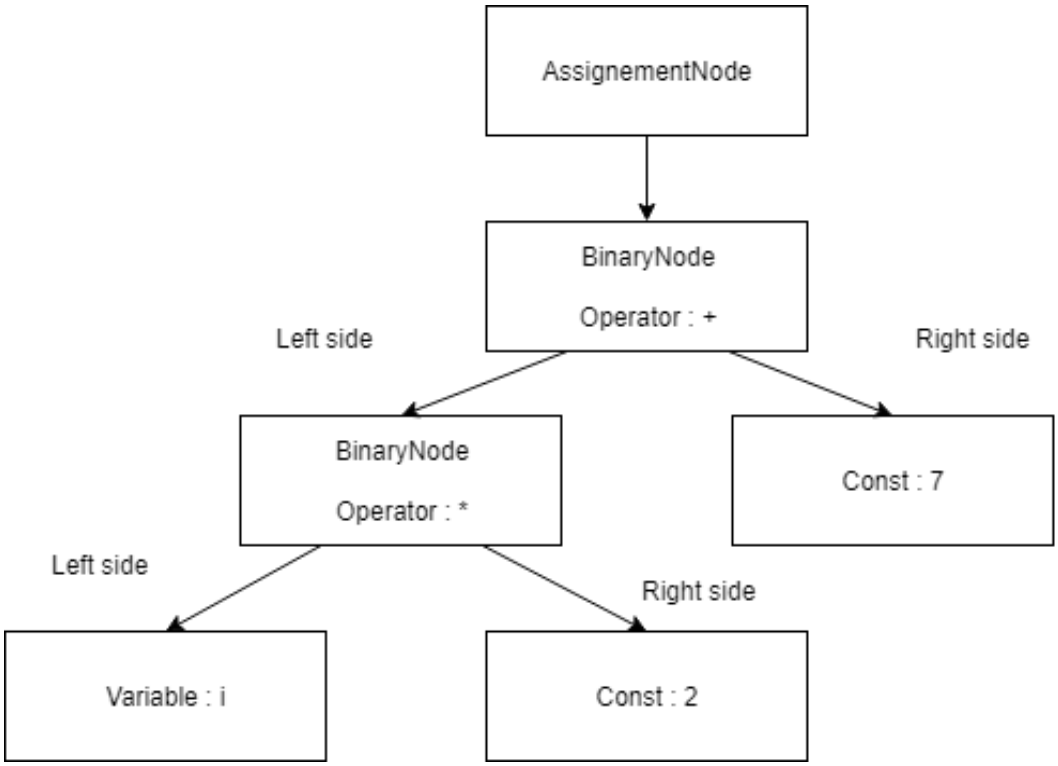

Figure 4.1 – Arbre syntaxique du nœud binaire

4.4.3 Les nœuds "conditionnels"

Ces nœuds correspondent aux conditions ; ils sont composés de plusieurs sous-nœuds. Le premier est un nœud d'opération binaire pour la condition, ensuite il contient deux blocs de nœuds représentant soit le cas où la condition est valide soit le cas où elle n'est pas respectée. Prenons le code suivant comme exemple.

```
if (i >= 0) {
    i++;
} else {
    i = 0;
}
```

Listing 4.2 – Code d'un if

Nous le retrouvons avec comme représentation sous forme d'arbre :

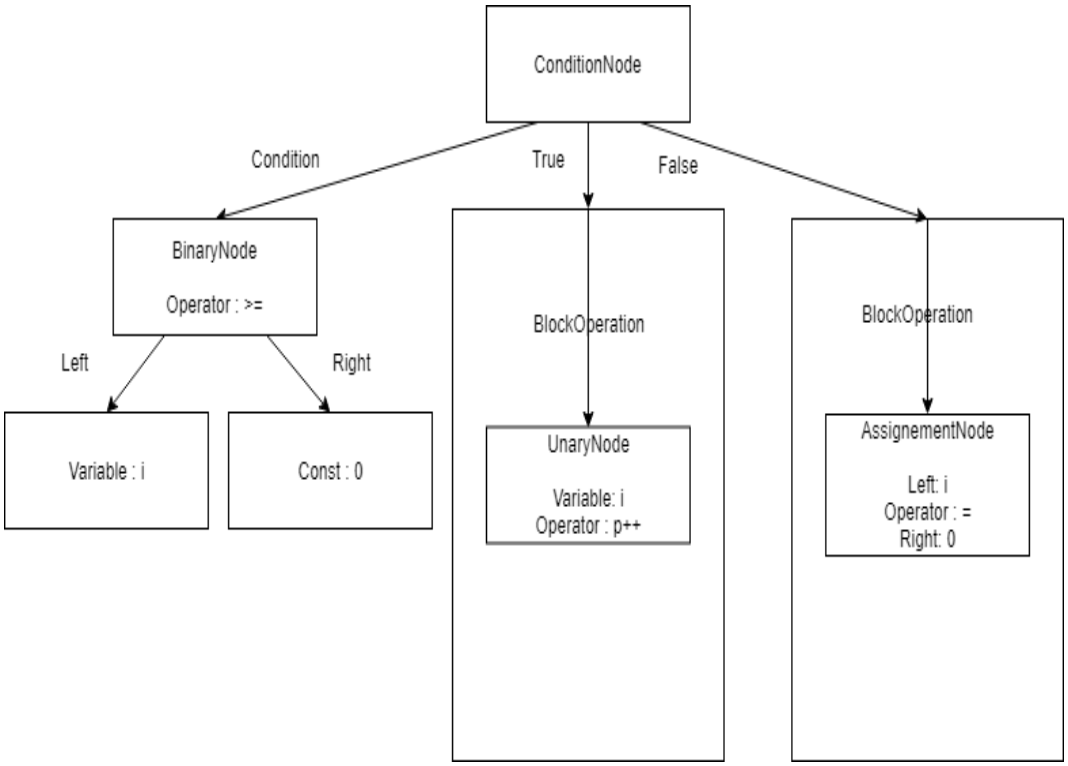

Figure 4.2 – Arbre syntaxique d'un nœud conditionnel

Nous connaissons maintenant les outils capable d'analyser un code C. Nous avons aussi déterminé une méthode capable de calculer la complexité en pire cas d'un algorithme. Nous sommes dés lors en mesure de créer un outil appliquant les règles vu dans le chapitre 2 en utilisant les outils du chapitre 4.

Chapitre 5

Implémentation

Avant de discuter de notre implémentation, nous souhaitons rappeler un fait énoncé par Alan Turing.

5.1 Terminaison d'une fonction : un problème indécidable

En 1936, Alan Turing prouve qu'il n'est pas possible de déterminer à partir d'un programme informatique si un autre programme s'arrête. [12] Il détermine donc qu'il s'agit d'un problème indécidable. Il affirme qu'un outil permettant de répondre au problème de l'arrêt n'est pas possible à réaliser avec les quatre points suivants [11] :

1. Analyse automatique de tous les codes, sans intervention humaine
2. Sans erreur
3. Est capable de répondre à toutes les entrées de la fonction
4. Non limité à des exécutions ou des mémoires bornées

Il s'agit ici d'un problème clé de notre réalisation. Le calcul de la complexité algorithmique n'est qu'une sous-partie du problème de l'arrêt. Cependant, nous ne cherchons pas à pouvoir analyser toutes les fonctions, mais uniquement le code d'étudiants développeurs. Dans un cas aussi spécifique, il reste envisageable de créer notre outil.

5.2 Proposition

Nous avons décidé de planifier notre implémentation en plusieurs étapes. Dans un premier temps, et suite au problème de l'arrêt, nous avons voulu limiter un maximum le langage utilisé par les étudiants. Ensuite nous nous sommes intéressés, pour ce mémoire, aux outils que l'on pourrait utiliser dans le cadre d'une implémentation. Finalement nous avons essayé d'implémenter une solution dans les temps impartis pour ce mémoire. Nous observerons les résultats de notre outil en analysant le code C du tri à bulle.

5.2.1 Restriction du langage

Comme nous l'avons expliqué auparavant, il n'est pas possible de faire un algorithme permettant d'affirmer qu'un code se termine sans exécuter celui-ci. Il paraît

donc évident que créer un code permettant de déterminer la complexité entière d'un algorithme est une tâche non réalisable dans un cas général. Cependant, nous ne cherchons pas à déterminer la complexité de toutes les fonctions qu'un développeur senior pourrait créer. Notre objectif est de créer un outil permettant de calculer la complexité d'un code écrit par un étudiant apprenant à développer. Cet étudiant va donc développer des fonctions relativement courtes. Cela nous permettra de mieux centrer l'objectif de notre analyseur et donc de pouvoir obtenir un résultat, là où il serait impossible d'en déterminer un dans tous les algorithmes.

Pour cela nous avons décidé d'exclure plusieurs pratiques de code et d'ignorer certaines fonctionnalités du C. Par exemple, en C nous pouvons utiliser des labels et des GOTO. Cette fonctionnalité n'est pas recommandée et obfusque le code. Ce type de méthode rend aussi beaucoup plus complexe l'analyse. Durant les enseignements, cette méthode est d'ailleurs fortement déconseillée par les professeurs. Nous allons donc ignorer les codes contenant des GOTO.

Afin de traiter de manière efficace le code, nous nous concentrerons sur des code C utilisant des types de base. Les étudiants seront amenés, en apprenant la programmation, à créer des structures de données avec la méthode struct. L'utilisation de ces structures peut rendre des codes complexes sans pour autant affecter l'algorithme que l'étudiant souhaite implémenter. Nous nous concentrerons donc sur des algorithmes utilisant les types primitifs du C, puisqu'un étudiant capable d'écrire un code fonctionnant avec des types primitifs sera capable de réécrire son code après analyse avec une structure, l'algorithme restant inchangé.

Un autre problème complexe vient avec la gestion des pointeurs en C. Il s'agit ici d'un élément clé de ce langage. Les pointeurs et l'allocation dynamique sont un rite de passage pour tout étudiant apprenant la programmation C. Cependant il est extrêmement difficile d'analyser un code utilisant des pointeurs et une gestion dynamique de mémoire. Obtenir la variable d'un code source via son adresse rend presque impossible l'étude d'un algorithme, ou nécessite de créer un outil bien trop spécifique à ce cas. Toutefois, nous ne pouvons pas nous passer des tableaux dans un langage de programmation, c'est pour cela que nous nous orienterons vers une analyse de code utilisant le sucre syntaxique du C sur les tableaux.

Nous avons aussi énormément élargi la définition d'une opération élémentaire. Par exemple, l'appel à une fonction interne au C comme le scanf ou le printf est considéré comme une seule opération. Ce qui, en réalité, n'est pas forcément précis.

5.2.2 Notre choix d'outil

Nous avons pu observer dans le chapitre précédent qu'il existait plusieurs outils qui pourraient nous intéresser. Les linters que nous avons observé ne bénéficiant pas de bibliothèque ou d'API, nous nous sommes orientés vers les compilateurs.

Nous avons cité deux compilateurs précédemment, clang et GCC. Suite à un projet déjà réalisé avec la bibliothèque clang-llvm nous avons pu remarquer que la réalisation d'un outil via le compilateur clang risquait d'être plus difficile que nécessaire. Dès lors, nous avons décidé de nous orienter sur un outil qui pourrait agir en tant qu'intergiciel [1]

1. Un intergiciel (middleware en anglais) est un outil s'injectant entre deux logiciels.

entre le compilateur et notre implémentation. L'outil que nous avons choisi se nomme PycParser[6].

Il s'agit d'une bibliothèque python utilisant, au choix, le compilateur GCC ou clang. Cette bibliothèque utilise les compilateurs afin d'extraire l'AST du code C.

Un des gros problèmes de cet outil vient de sa documentation quasiment inexistante. Le peu de documentation disponible vient des exemples disponible sur github. Mais, parmi ces exemples, l'un d'entre eux est extrêmement intéressant. Il permet de convertir le code C d'une fonction en AST dans un format JSON. Le format de données JSON est facilement analysable et automatisable. C'est un format compréhensible par les hommes ce qui facilite l'analyse du code C.

Nous avons pu implémenter le code de cet exemple dans notre code, permettant ainsi de parser tous l'AST d'un code C en analysant du JSON plutôt que la sortie naturelle d'un compilateur.

5.2.3 Solution actuelle

Nous avons seulement, à ce jour, une partie de solution. Nous sommes capables de prendre tout code d'une fonction mise en entrée de notre outil. Nous sommes aussi capables, à partir de cette entrée, de pouvoir définir combien d'opérations élémentaires existent dans le code. Par exemple, dans le cas de la fonction du tri à bulle (cf. : 3.1), nous retrouvons dix-sept opérations élémentaires différentes. Ces opérations sont :

- trois instanciations de variables (i, j, tmp)
- deux initialisations de variables à 0 (i=0, j=0)
- deux incrémentations de variables (i++, j++) 7
- quatre calculs simples (trois fois j+1, une fois taille-1)
- trois opérations booléennes (j<taille, j<taille-1 le calcul taille-1 est compté au-dessus, et tab[j]>tab[j+1] le calcul de j+1 est compté au-dessus)
- trois affectations de variables (tmp=tab[j], tab[j]=tab[j+1] j+1 le calcul de j+1 est compté au-dessus, tab[j+1]=tmp le calcul de j+1 est compté au-dessus)

Nous sommes capables de déterminer quelles sont les variables qui sont instances dans le code, permettant de déterminer si une variable reste fixe ou non. Ainsi, dans notre exemple du tri à bulle, i, j et tmp sont instanciées dans le code. Aucune de ces valeurs n'est responsable d'un changement du nombre d'opérations effectuées dans le code. Elles peuvent cependant être affectées à une valeur d'entrée du code (taille ou tab ici). Il nous faudra être capable de déterminer ces cas pour approfondir notre automatisation.

Nous pouvons aussi déterminer à quelle profondeur le nœud de l'AST se trouve lorsque l'on analyse le code. Lorsque l'on parle de profondeur, il s'agit du nombre de boucle imbriqués. Par exemple ici, nous avons le retour suivant :

```
['Decl 0', 'Decl 0', 'Decl 0', 'For 0', 'Assignment 0', 'Constant 0
    ', 'UnaryOp 0', 'For 1', 'Assignment 1', 'Constant 1', 'UnaryOp
     1', 'If 2', 'BinaryOp 2', 'ArrayRef 2', 'ArrayRef 2', '
    Assignment 2', 'ArrayRef 2', 'Assignment 2', 'ArrayRef 2', '
    Assignment 2', 'ID 2']
```

Listing 5.1 – Profondeur de chaque nœud

Nous remarquerons d'ailleurs dans cet exemple une grande variété de nœuds que l'on peut rencontrer dans le code.

5.2.4 Solution future

Il reste encore beaucoup à faire pour finaliser notre outil, et malheureusement le temps restant n'est pas suffisant pour terminer l'implémentassions. L'un des travaux manquant est la détection des variables responsables des itérations des boucles. A ce jour nous connaissons toutes les variables présentes dans le code. Nous pouvons déterminer (Travail en cours) quelles sont les variables présentes dans les itérations.

A partir du moment ou nous pourrons déterminer quelles sont les variables responsables de l'itération de la boucle, nous pourrons déterminer une approximation de la complexité de la fonction. Nous pourrons aussi espérer, en continuant à travailler sur notre outil, pouvoir calculer le nombre d'opérations précises sous la forme d'une formule. Cette formule contiendrait les variables desquelles dépendent le nombre d'opérations.

Chapitre 6

Conclusion

Nous avions dans l'intention pour ce mémoire d'automatiser le calcul de la complexité d'un code C. Pour cela nous avons du définir ce qu'était la complexité algorithmique. Pour ce faire nous avons utilisé la machine de Turing.

Nous avons ensuite montré que les algorithmes n'étaient pas une implémentation, et à partir de ce constat nous avons montré qu'il pouvait exister des différences entre deux implémentations d'un même algorithme. Nous avons ensuite appliqué la méthode de calcul de la complexité algorithmique en pire cas au tri à bulle.

Dans le chapitre 4, nous avons pu observer que certains outils pourraient nous aider dans notre automatisation. Nous avons vu que les linters et les compilateurs pourraient nous permettre d'analyser l'AST d'un code.

Dans la dernière partie, nous avons essayé d'implémenter notre outil. Nous avons aussi essayé d'utiliser les outils vus dans le chapitre précédent afin de répondre à notre problème. Nous n'avons à ce jour qu'une ébauche d'implémentation, et il reste beaucoup a faire pour avoir un outil fini.

Table des matières

Bibliographie

[1] Analyse de la complexité des algorithmes. https ://fr.wikipedia.org/wiki/Analyse_de_la_comp

[2] Cédric Bastoul. (Ré)introduction à la compilation. `http://icps.u-strasbg.fr/~bastoul/teaching/compilation/bastoul_introduction_compilation.pdf`.

[3] Compilateur. `https://fr.wikipedia.org/wiki/Compilateur`.

[4] Complexité d'un algorithme. http ://igm.univ-mlv.fr/ñicaud/poly/L1_5.pdf. (Visité le 10/06/2019).

[5] Documentation de clang llvm. https ://clang.llvm.org/comparison.html. (Visité le 13/06/2019).

[6] Documentation de PycParser. https ://github.com/eliben/pycparser. (Visité le 15/06/2019).

[7] Oded Goldreich. Notes on Levins Theory of Average-Case Complexity. `http://www.wisdom.weizmann.ac.il/~/oded/COL/lnd.pdf`.

[8] Olivier Hudry. Machines de Turing et complexité algorithmique.

[9] Intel Rechisels the Tablet on Moores Law. https ://blogs.wsj.com/digits/2015/07/16/intel-rechisels-the-tablet-on-moores-law/. 16 juil. 2015.

[10] Lint, a C Program Checker. `http://tack.sourceforge.net/olddocs/lint.pdf`. 1988.

[11] David Monniaux. Analyse statique : de la théorie à la pratique, Analyse statique de code embarqué de grande taille, génération de domaines abstraits. `http://www-verimag.imag.fr/~monniaux/biblio/Monniaux_HDR.pdf`.

[12] On computable numbers, with an application to the entscheidungsproblem. https ://www.cs.virg (Visité le 13/06/2019).