Rapport de Stage

Bendi-Ouis Yannis

25 avril 2016

Table des matières

Chapitre 1

Présentation

1 L'Objectif

Quel était-il ?

Il existe plusieurs moyens de compresser des données. L'un d'eux est le Wavelet Tree, qui peut lui même être fait de deux manière différentes : par pointeur ou par tableau, chacune ayant ses avantages et ses inconvénients.

Cependant, un Wavelet Tree normal ne permet pas de faire des modifications dans le temps. De ce fait, nous allons nous intéresser à un Wavelet Tree Mutable. Et c'est notamment là que les avantages et inconvénients des deux différentes implémentations vont apparaître.

En effet, si l'implémentation par pointeur permet une modification simple en complexité, car il suffit juste de modifier les quelques zones touchées, celle par tableau nécessite une reconstruction totale du Wavelet Tree Mutable.

L'objectif de ce stage était alors de déterminer s'il était possible et intéressant de créer un Wavelet Tree Mutable hybride, alliant les avantages de chacune de ces deux implémentations.

Comment faire ?

L'idée première était d'utiliser une bibliothèque sur les Wavelet Tree fournit par Guillaume Blin. Cette bibliothèque est trouvable sur Github, et elle est en réalité le produit du travail de Francisco Claude (pour ne citer que lui).

Ce dernier a produit une bibliothèque implémentant les Wavelet Tree (que l'on va à partir de maintenant appelé WT, et WTM pour les Mutables) par les deux manières citées ci dessus. Il est alors de notre devoir de choisir laquelle est la plus appropriée pour notre besoin lorsque l'on souhaite l'utiliser.

Mon objectif final devait alors être de modifier cette bibliothèque,

de manière à la rendre hybride. Et si possible, qu'elle calcule d'elle même à quel moment il était préférable d'utiliser la méthode des pointeurs plutôt que celle des tableaux, et vice-versa.

Cependant, pour un simple étudiant en L3 tel que moi, cela est bien trop compliqué. En réalité, je ne maîtrise même pas le C++ (or c'est le langage utilisé dans la bibliothèque), et peine pas mal à le comprendre. Même si au début, je voulais bien être optimiste en me disant qu'il me suffisait d'apprendre les bases de ce langage pour comprendre, la réalité fut toute autre.

Un autre problème fut que pour la première fois, je devais me confronter face à une bibliothèque relativement grosse. En effet, celle-ci possède plus de 65.000 lignes de code. Inutile de préciser à quel point pour un étudiant en 3ème année de licence, arriver face à un projet de plus de 65.000 lignes dans un langage qu'il ne maîtrise pas (pour ne pas dire qu'il n'a jamais rencontré) n'est pas chose aisée.

La solution

Face à ce travail bien trop conséquent, il fallait donc trouver un autre moyen d'y parvenir. D'un commun accord avec Guillaume Blin, nous avons convenu que mon travail serait d'implémenter en C une bibliothèque codant les WTM, pour pouvoir parvenir à nos fins.

Évidemment, nous n'envisagions pas de tout ré-implémenter. La bibliothèque de Franscico Claude est fournie avec beaucoup d'optimisation que je n'allais pas refaire. Cela aurait pris bien trop de temps, et je ne sais pas si j'en étais capable. Non, l'idée était de ré-implanter seulement le minimum. Pourquoi cela aurait-il suffit alors que l'on cherche une comparaison des compétences entre une implémentation hybride et une normale ?

Et bien parce que les deux se basent sur la même implémentation. De ce fait, si l'une a des lacunes, l'autre aura les mêmes. Aucune des deux ne sera donc avantagée.

Cependant, même si ici, le travail était réduit, il restait tout de même trop important pour un stage qui devait avoir une durée de un mois. Je ne pouvais pas et implémenter la bibliothèque en C, et la rendre hybride.

Guillaume Blin m'a alors conseillé de m'occuper seulement de l'implémentation de celle-ci, l'hybridation pouvant alors être faite

plus tard, par un autre étudiant de L3. L'idée de ce stage était donc de préparer le travail à un futur étudiant de L3 pour qu'il puisse procéder à l'hybridation des WTM. Car, si celui-ci vient aussi du cursus universitaire (L1, L2, L3), il n'a alors pas eu l'occasion d'apprendre le C++, mais saura utiliser le C.

2 Fonctionnement d'un Wavelet Tree

Le Wavelet Tree, comme son nom l'indique, est un arbre qui va représenter les données que l'on souhaite compresser. Pour faire simple, dans la suite de nos exemples, nous ne nous occuperons que de textes, ou plus exactement de mots. Mais il est aussi possible de compresser des nombres ou tout autre sorte d'objet.
Son fonctionnement nécessite deux étapes.

Première étape : l'alphabet

La première étape est la création de l'alphabet qui correspond à nos données. Ici, l'alphabet correspond à chacune des lettres utilisées dans notre texte.
Il faut alors, pour chacune des lettres de notre alphabet, lui attribuer un "code". Celui sera en réalité un nombre, que l'on codera en binaire.

Exemple :

Le mot choisi pour cet exemple sera **Tartuffe**. L'alphabet obtenu est alors :

Lettres :	A	E	F	R	T	U
Code :	000	001	010	011	100	101

Deuxième étape : Création de l'arbre

Pour la création de l'arbre, il faut se baser sur le code donné à chacune des lettres dans l'alphabet, et à l'ordre de celle-ci dans notre mot.

Nous allons répartir dans notre arbre chacun des bits codant nos lettres suivant un algorithme détaillé ci-dessous. Ici, nous lirons nos codes binaires de gauche à droite. Le bit de poids fort (celui de gauche), est donc le premier bit de nos codes.

Voici un algorithme permettant de comprendre la création de l'arbre. Nous l'expliquerons et le détaillerons sur un exemple ensuite :

Algorithm 1 Création Wavelet Tree

```
1: procedure Création Wavelet Tree(mot)
2:     for chaque lettre l du mot do
3:         Mettre le premier bit du code de l dans la racine
4:         for chaque bit suivant b do
5:             if bit precedent == 0 then
6:                 Mettre b dans le fils gauche
7:             else if bit precedent == 1 then
8:                 Mettre b dans le fils droit
9:             end if
10:        end for
11:    end for
12: end procedure
```

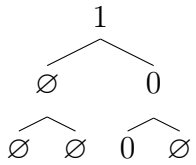
Pour chacune des lettres présentes dans notre mot, il faut commencer par mettre dans la racine de notre arbre le premier bit qui la code.

Ainsi, la première lettre qui code notre mot étant un **T**, on va mettre dans notre racine le premier bit codant le **T** dans notre alphabet. Son code étant **100**, le premier bit est donc **1**.

Le premier bit étant un **1**, on va placer le second bit : **0**, dans le fils droit de la racine.

Le bit précédent étant un **0**, on va donc placer le bit suivant (et le dernier, nos codes étant de taille trois) : **0**, dans le fils gauche.

Voici l'arbre obtenu après y avoir placé la lettre **T** :



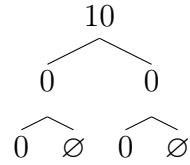
On va donc maintenant s'intéresser à y placer la deuxième lettre. Celle-ci est un **A**. Selon l'alphabet décrit plus haut, son code est **000**.

Par conséquent, on place dans la racine le premier bit du code, soit un **0**. Etant donné que le premier bit était un **0**, on se place dans le fils gauche de la racine.

Le deuxième bit étant lui aussi un **0**, on place **0** dans le noeud, et on se place dans le fils gauche pour y mettre le troisième bit.

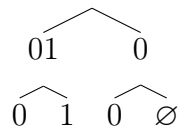
Le troisième bit (et donc le dernier) est lui aussi un **0**. On place donc un **d**ans le noeud actuel.

Après le placement de la deuxième lettre de notre mot (**TARTUFFE**), on obtient cet arbre :

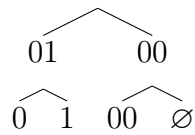


Et ainsi de suite. Ci-dessous la construction de l'arbre après chaque lettre :

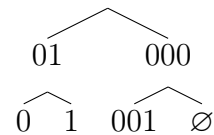
R :



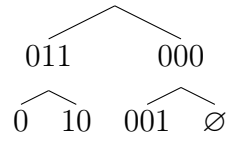
T :



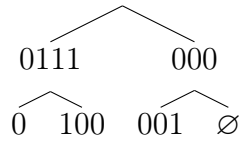
U :



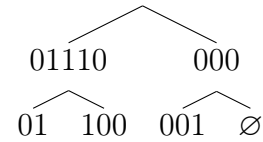
F :



F :



E :



Cette représentation ci-dessus de l'arbre nous donne une parfaite idée de ce que donne l'implémentation par pointeur.

Par tableau, au lieu de stocker une plage de bits dans chaque noeud comme on le fait en utilisant les pointeurs, on ne créée qu'une

seule et unique plage de bits. On va ensuite soi-même la décomposer en connaissant le nombre d'éléments présents.

En effet, dans notre exemple, notre mot fait huit lettres. La racine est donc ainsi composée de huit bits (les huit premiers de chacune des lettres).

Le fils gauche de la racine est juxtaposé à celle-ci et, étant donné que la racine possède cinq bits qui valent **0**, le fils gauche possède donc cinq bits. Par conséquent, il commence en position 8 (inclus) du tableau, et se finit en position 12 (inclus).

Le fils droit de la racine quant à lui commence donc en position 13 (inclus), possède trois éléments car la racine possède trois bits à **1**. Et se finit en position 15 (inclus).

Et ainsi de suite...

Voici la représentation de ce même arbre, mais par tableau :

10011000
01110000
01100001

→

10011000			
01110		000	
01	100	001	

Les petits plus d'un Wavelet Tree Mutable ?

Un WTM ne change pas tellement d'un WT classique. Ce dernier doit juste pouvoir profiter de trois opérations supplémentaires :

— L'ajout
— La suppression
— La modification

Chapitre 2

Implementation

1 Fichiers dans l'archive

Le projet se présente donc sous forme d'une archive dans laquelle sont présents quatre fichiers .c ainsi que leur .h respectifs.

- **bitmap** : Correspond à l'implémentation de la plage de bits ainsi que les fonctions qui lui sont associées (rank, select, access) que nous détaillerons plus tard.
- **dict** : Correspond à l'implémentation des dictionnaires, qui nous serviront d'alphabet ici.
- **wtpt** : Correspond à l'implémentation des WT par pointeurs.
- **wtArray** : Correspond à l'implémentation des WT par tableau.

Avec ces fichiers est aussi présent un Makefile pour permettre de compiler le tout et un README permettant d'expliquer des détails éventuels.

2 Bitmap

Cette bibliothèque se doit donc comme dit précédemment d'implémenter les plages de bits.

Elle doit donc posséder trois opérations indispensables pour travailler correctement : rank, select et access (expliquées plus bas).

De plus, elle doit préparer la mise en place d'un WTM, et donc posséder les fonctions suivantes : insert, remove, modify (expliquées plus bas).

Choix d'implémentation

Pour stocker notre plage de bits, j'ai choisi de faire un simple tableau d'entiers (int). Chaque entier stockant 32 bits.

Pour se repérer dans le tableau, il suffit alors compter le nombre de bits que l'on a stocké.

Ainsi, si l'on a stocké 100 bits, on a donc rempli les 3 premiers entiers de notre tableau, et les 4 premiers bits de notre 4^{eme} entier *(3*32 + 4 = 100)*.

Exemple :

Ici on stocke 37 bits, le premier int (celui de droite) est alors rempli, le deuxième n'a alors que ses 5 (37 - 32) premiers bits d'utilisés.

2^{eme} int	1^{er} int
10101	10101010101010101010101010101010

Les opérations Rank, Select, Access

Ce sont des fonctions "élémentaires" pour un bitmap. Leur utilisation est assez fréquente (notamment rank, qui est très souvent utilisée dans nos bibliothèques), de ce fait, elles doivent avoir une complexité optimale. Elles sont donc en **O(1)**.

— **Rank(i, b)**
b vaut ici soit **0**, soit **1**, et **i** est un entier positif.
Rank retourne le nombre de **b** présents dans les **i** premiers bits de notre bitmap.
Exemple :

Soit le bitmap **01000101**, alors **rank(5, 1) = 2** et **rank(7, 0) = 4**. (On lit un bitmap de droite à gauche.)

— **Select(i, b)**
b vaut ici soit **0**, soit **1**, et **i** est un entier positif.
Select retourne la position du $\mathbf{i}^{eme}$ bit **b**.
Exemple :
Soit le bitmap **01000101**, alors **select(3, 1) = 6** et **select(1, 0) = 1**. (La position du bit tout à droite étant 0.)

— **Access(i)**
i est un entier positif.
Access retourne la valeur du $\mathbf{i}^{eme}$ bit.
Exemple :
Soit le bitmap **01000101**, alors **Access(3) = 1** et **Access(8) = 0**.

Les opérations Insert, Remove, Modify

Ces dernières ne sont en réalité pas obligatoires pour implémenter les WTM, car on pourrait choisir de reconstruire à chaque fois les bitmaps. Cependant, pour un gain de complexité et de facilité dans certains cas, il est plus rentable de les implémenter.

Leur but vise donc à simplifier la mise en place des modifications sur les WT. Leur complexité est en **O(n)** pour la suppression et l'insertion car il faut déplacer les éléments déjà présents, et en **O(1)** pour la modification d'un élément.

— **Insert(i, b)**
b vaut soit **0**, soit **1**, et **i** est un entier positif.
Insert insère alors le bit **b** en position **i**. Si **i** est inférieur à la taille du mot, tous les bits qui suivent sont déplacés. Si **i** est trop grand, alors le bit **b** sera placé à la dernière position.

— **Remove(i)**
i est un entier positif. Remove supprime alors le bit en position **i**.
Si la valeur de **i** n'est pas correct, remove retourne une assertion.

— **Modify(i, b)**
b vaut soit **0**, soit **1**, et **i** est un entier positif.
Modify modifie la valeur du bit en position **i** et lui donne la valeur **b**. Si la valeur de **i** n'est pas correcte, modify retourne une assertion.

3 Dict

Cette bibliothèque a donc pour objectif d'implémenter les alphabets de notre WT. Pour ce, j'ai choisi de les implémenter sous la forme d'un dictionnaire.

Etant donné que l'on cherche à trouver un code pour chacune de nos lettres, et que ces codes ont tout intérêt à être croissants et positifs (donc ≥ 0), j'ai choisi d'utiliser un simple tableau (une dimension) pour créer notre dictionnaire.

Explications :
Les codes que l'on va utiliser vont être des nombres supérieurs ou égaux à zéro. Ces nombres, pour gagner au maximum en mémoire par la suite, ont tout intérêt à être le plus petit possible. Par conséquent, si l'on stocke quatre caractères dans notre alphabet, les codes attribués ont tout intérêt à être : {0, 1, 2, 3} qui par la suite pourront être utilisés en les codant sur deux bits : {00, 01, 10, 11}.

Or, dans un tableau, cela correspond parfaitement aux indices des quatre premiers éléments. Ainsi le code que l'on va attribuer à chaque caractère sera simplement son indice dans le tableau. Son indice étant un entier décimal, il faudra juste ensuite le convertir en binaire pour l'utiliser dans les WT.

Exemple :

indice	0	1	2	3
lettre	a	b	c	d

→

code	00	01	10	11
lettre	a	b	c	d

En pratique
L'exemple ci dessus montre l'utilisation d'un alphabet qui peut attribuer un code à des lettres, en pratique, pour des questions de facilités, nous ne coderons que des entiers (int).

En effet, la finalité de ce projet étant juste comparer des performances, nous n'avons pas besoin de nous casser la tête à instaurer la possibilité de mettre plusieurs types dans notre alphabet.

Cependant, on peut remarquer qu'il ne serait pas non plus très

compliqué de rajouter un type union dans lequel on mettrait nos différents types. Mais cela reviendrait à une perte de temps.

Et si on souhaite vraiment mettre des lettres (de type char), on peut toujours utiliser leur code ASCII, et n'utiliser que des int.

Différence alphabet/dictionnaire

Finalement, notre alphabet ressemble tout de même beaucoup à un simple dictionnaire. Les seules différences sont donc que les fonctions ont été implémentées de manière à ce qu'il n'y ait pas deux fois la même lettre dans notre alphabet (il suffit alors de vérifier toutes les lettres présentes lorsque l'on en rajoute une, en O(n) donc).

A la différence d'un dictionnaire, on ne peut pas choisir le code qui est attribué à une lettre. En effet, celui-ci est attribué selon l'ordre d'arrivée.

Il n'est aussi pas possible de supprimer une lettre que l'on a ajoutée. Malgré tout si on le souhaite vraiment, une telle fonction n'est pas très compliquée à ajouter : il suffirait juste d'appliquer une suppression dans un tableau.

Et pour finir, la bibliothèque possède une fonction qui nous retourne la taille des codes, soit le nombre de bits utilisés pour les écrire en binaire. Finalement cette fonction ne fait que retourner l'arrondi supérieur du logarithme en base deux du nombre d'éléments dans l'alphabet.

4 Wtpt

Cette bibliothèque se doit donc d'implémenté dans un premier temps les WT par pointeur, puis les WTM. Pour passer d'un WT à un WTM, la bibliothèque doit donc posséder trois opérations supplémentaires que nous décrirons plus bas : Insert, Remove, et Modify (qui cette fois-ci s'applique à des WT).

Implémentation des WT par pointeur

Pour implémenter un WT, j'ai bêtement implémenter un arbre par pointeur. A savoir, une structure noeud (struct Wtpt) qui va possèder un pointeur vers le fils droit, et un vers le fils gauche. Qui sont eux aussi deux noeuds (struct Wtpt).

A chaque noeud, est aussi stockée la plage de bit correspondant, à savoir un bitmap. Chacun des noeuds possède aussi un pointeur vers l'alphabet du WT, un compteur du nombre d'éléments présents dans le noeud (qui correspond la longueur du bitmap du noeud), ainsi sa hauteur dans l'arbre (les feuilles sont de hauteur 1, et la racine de hauteur *max(hauteur(filsGauche), hauteur(filsDroit))*).

La suite de l'implémentation du WT par pointeur correspond à la description des WT faites précédemment.

Implémentation des WTM par pointeur

Comme dit précédemment, pour passer d'un simple WT à un WTM, il faut ajouter trois fonctions. Ce sont les suivantes :

— **InsertWtptMutable (Wtpt w, int c, int pos)**
Insert, comme son nom l'indique, va insérer un élément **c**, dans le WTM **w**, à la position **pos**.
Cette fonction va donc dans un premier temps ajouter **c** à l'alphabet du WTM, puis va, itérativement pour chacun des bits du code de **c**, insérer le bit dans le bitmap grâce à la fonction insert sur les Bitmap.
Ensuite, en fonction de la valeur du bit qu'elle vient d'insérer, elle va insérer le bit suivant dans le fils droit ou fils gauche.

— **RemoveWtptMutable (Wtpt w, int pos)**
Remove va quant à elle supprimer le caractère présent à la

position **pos** dans le WTM **w**.
Elle fonctionne de la même manière que fonctionne Insert, à la différence qu'elle supprime un élément au lieu d'en insérer un.
Si jamais après la suppression un noeud voit son bitmap vide, elle ne le supprime pas.

— **ModifyWtptMutable (Wtpt w, int c, int pos)**
Modify a donc pour rôle de remplacer l'élément à la position **pos** du WTM **w** par l'élément **c**.
Pour se faire, elle va donc appeler consécutivement **RemoveWtptMutable** puis **InsertWtptMutable**.
En effet, il n'est pas dit que le code du nouvel élément mène à la même feuille, par conséquent. Nous ne pouvons pas juste utiliser la fonction **modify** de Bitmap.

5 WtArray

De la même manière que Wtpt, WtArray doit dans un premier temps implémenter les WT par tableau, puis ensuite les WTM. A savoir donc rajouter les trois opérations (insert, delete, modify).

Implémentation des WT par tableau

L'idée est donc de créer une structure contenant un arbre par tableau ne pouvant contenir que des **0** ou des **1**, ainsi qu'un pointeur vers l'alphabet, le nombre d'éléments stockés et la hauteur de notre arbre.

Ici, à l'inverse du Wtpt où l'on créée un bitmap à chaque noeud, nous n'utiliserons qu'un seul bitmap que nous traiterons comme un arbre par tableau.

L'implémentation suit ensuite la description faite précédemment des WT par tableau.

Implémentation des WTM par tableau

Ici, nous devons donc ajouter les trois opérations. Cependant, alors que sur un WTM par pointeur ces trois fonctions ne modifiaient qu'une petite partie de la structure ; par tableau, elles modifieront tout le bitmap. En effet, l'ajout ou la suppression d'un simple bit provoquera le décalage des bits qui le suivent.

Par conséquent, il est plus rentable que chacune de ces fonctions reconstruisent entièrement le bitmap.

— **InsertWtArrayMutable(WtArray w, int c, int pos)**
Ici, dans un premier temps, **c** est ajouté à l'alphabet. Puis sont pré-calculés tous les placements de chacun des bits composant le code de **c**. On reconstruit ensuite le bitmap contenu dans **w** en copiant l'ancien et en ajoutant les modifications nécessaires.

— **DeleteWtArrayMutable(WtArray w, int pos)**
Ici, on va procéder de la même manière que pour Insert. A la différence que l'on va dans un premier temps retenir les positions de tous les bits du code de l'élément à supprimer. Puis reconstruire le bitmap en ne réécrivant pas les bits présents

aux positions retenues.

— **ModifyWtArrayMutable(WtArray w, int c, int pos)**
Ici, comme pour les Wtpt, on applique la fonction **DeleteWtArrayMutable(w, pos)** puis **InsertWtptMutable(w, c, pos)**.
Si l'on souhaite optimiser les performances, il faudrait alors implémenter cette fonction de manière à ne recréer le bitmap qu'une seule fois. Car l'utilisation de Delete puis Insert implique qu'on recréée le bitmap deux fois.

Chapitre 3

Conclusion

1 Fruit de mon travail

Au final, les WTM fonctionnent. Je ne peux cependant garantir qu'aucun bug n'est présent, étant donné que je n'ai pu qu'effectuer que des tests et non prouver leur bon fonctionnement.

L'implémentation de ceux-ci m'a pris environ cinq mois (pas à plein temps évidemment). Premier commit effectué le 13 Octobre 2015, dernier commit le 12 Mars 2016. Cette longueur de temps se justifie de par le fait que j'ai fait ce stage en parallèle avec les cours. Vous pouvez d'ailleurs si vous le souhaitez voir l'avancement du projet dans le temps sur le dépôt Github que je fournirai plus bas.

2 Les plus / Les moins

Les plus :

— Ca fonctionne
— Inutile de s'intéresser à comprendre .c si l'on souhaite utiliser l'une des bibliothèques. Comprendre les .h seuls devrait suffire.
— Les tests sont fournis avec en cas de besoin.
— Un fichier README peut fournir de l'aide pour créer un WTM à partir d'un fichier (ne contenant que des nombres entiers sous format texte, utile pour le debug).

Les moins :

— N'ayant pas l'habitude, je n'ai pas pensé à utiliser Valgrind. Et le moins que l'on puisse dire est que les résultats ne sont pas satisfaisants.

— Je n'ai pas fait une implémentation générique des WTM. En effet, il existe deux bibliothèques différentes : par pointeur (Wtpt et par tableau (WtArray). Malgré tout comme expliquer précédemment, ce dernier point n'est pas très compliqué à résoudre.

3 Ce que le stage m'a apporté

Malgré que ce fut long et compliqué de le faire en parallèle des études, j'y ai pris beaucoup de plaisir.

Je suis un grand fan du langage C (mon premier langage), et par conséquent j'ai pas mal apprécié de faire un projet relativement conséquent dessus. Qui plus ai, je pense m'être amélioré dessus à force d'exercice grâce à ce stage.

J'ai aussi compris que le travail seul n'était certainement pas quelque chose d'intéressant pour moi. En effet, j'ai pas mal regretté pendant tout le long du stage le manque de compagnie. J'aurais vraiment apprécié le faire avec quelqu'un. Même si je pensais le contraire au début.

Le fait de travailler seul fut d'ailleurs certainement la chose la plus dur durant ce stage. En effet, trouver la motivation pour se lancer plusieurs heures dans un projet, seul, n'est pas chose aisé (du moins, pour moi). Je le remarque avec les projets de la fac, ou même ceux que j'ai mené personnellement, je prends beaucoup plus de plaisir à faire partie d'une équipe qu'à faire mon projet seul. Et ce, même si cela implique des restrictions ou des divergences de point de vu.

En bref, en plus de m'apporter de l'exercice et quelques connaissances, ce stage m'a notamment permis de savoir dans quelles conditions je suis plus productif.

4 Adresses Utiles

GitHub :
Voici le lien GitHub où est hébergé le projet :

https ://github.com/Naowak/myWtm

Adresse e-mail :

Je risque de ne plus être dans l'université de Bordeaux l'année prochaine. Par conséquent, si un étudiant reprend la suite de ce sujet, voici une adresse e-mail sur laquelle il pourra me contacter si besoin est :

yannis.bendiouis@gmail.com