

Rapport de Projet TER

Calcul booléen sur séquence génétique

Guillaume PÉRUTION-KIHLI

Judicaël RUSSO

Paul MONTI

5 mai 2017



UNIVERSITÉ
DE MONTPELLIER

Table des matières

Introduction	3
I Formalisation du langage bio-logique	3
1 Définitions	3
1.1 Alphabet	3
1.2 Mots bien formés	3
2 Propriétés des mots bien formés	4
2.1 Excision et Inversion	4
2.2 Non-croisement des sites	4
2.3 Confluence des activations	4
II Passer d'un mot à une fonction logique	4
3 Logique des mots	4
3.1 Activation de gène	4
3.2 Mots intègres	5
3.3 Table de vérité d'un mot intègre	5
3.4 Mots symétriques	5
4 Simplification de fonctions logiques	5
4.1 Description de la méthode Quine-McCluskey	5
4.2 Simplifications grâce au principe du tiers-exclu	5
4.3 Simplifications par théorème du consensus	6
5 Sémantique des séquences	7
5.1 Définition de la sémantique	7
5.2 Equivalences sémantiques	7
5.3 Classes d'équivalence représentées	7
III Passer d'une fonction logique à un mot	8

6	Procédé et algorithmes	8
6.1	Générer un maximum de mots représentatifs	8
6.2	Trouver les fonctions logiques simplifiables	8
6.3	Créer le langage de Dyck	9
6.4	Créer les mots de sites	10
6.5	Remplir les inter-sites	11
7	Résultats	12
7.1	Résultats bruts des générations	12
7.2	Interprétations	12
IV	Aspect technique	13
8	Base de données	13
8.1	Notre choix de base de données	13
8.2	Architecture de la base de données	13
8.3	Utilisation de la base de données	13
9	Le programme Genetix	14
9.1	Nos choix techniques	14
9.2	Utilisation du programme Genetix	14
	Conclusion	14
	10 État et ouverture du projet	14
	11 Retours d'expérience	14
	Annexes	15
	A Sémantiques et représentants des classes d'équivalences	15
	B Liste des options du programme Genetix	16
	C Dépôt Git du programme Genetix	16
	D Site web de parcours de la BDD	16

Références

- [1] Frédéric Brouard. Base de données et performances... petites tables et tables obèses!, juin 2012. [Lien](#).
- [2] Tai-Yin Chiu and Jie-Hong R. Jiang. Logic synthesis of recombinase-based genetic circuits, novembre 2016. [Lien](#).
- [3] Jr. E. J. McCluskey. Minimization of boolean functions. *The Bell System Technical Journal*, juin 1956.
- [4] Sarah Guiziou. Thesis committee report. septembre 2016.
- [5] Tarun Kumar Jain, D. S. Kushwaha, and A. K. Misra. Optimization of the quine-mccluskey method for the minimization of the boolean expressions. Fourth International Conference on Autonomic and Autonomous Systems, 2008.
- [6] Michel Leclerc. Notes de travail, 2016.
- [7] W. V. Quine. The problem of simplifying truth functions. *The American Mathematical Monthly*, Vol. 59, No. 8, pp. 521-531, Octobre 1952.

Introduction

Ce projet s'inscrit dans le cadre d'une demande de MME. GUIZIOU, doctorante en biologie, recherchant l'aide de l'informatique afin d'approfondir ses recherches en matière d'approche logique de l'ADN.

L'ADN est composé de bases, briques élémentaires du vivant, dont certains groupements correspondent à une fonction particulière : promouvoir un gène, l'activer, etc. Et plusieurs fonctions à la suite forment ainsi des mots, grandes portions de l'ADN, ayant une sémantique précise.

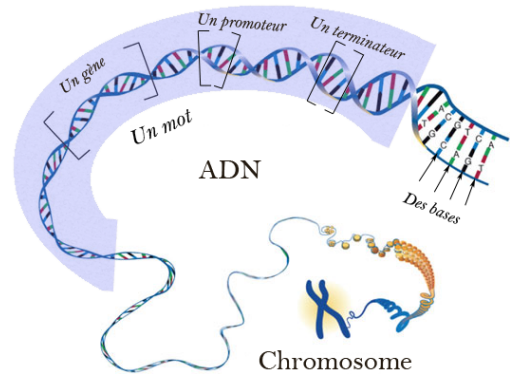
L'information véhiculée par la sémantique de ces mots, selon leur environnement, entraîne l'exécution de telle ou telle tâche. Cette analogie entre code génétique et logique est le sujet de nombreuses recherches, néanmoins il n'existe pas encore de formalisme complet dans ce domaine.

L'étude de cette logique de l'ADN nécessite la synthèse de matériel génétique. Cependant, celle-ci est coûteuse en temps et en argent. Tout le problème réside donc dans la recherche de mots simples et optimaux.

Dans le cadre du projet, il fut dans un premier temps nécessaire de trouver quelle fonction logique implémentait un mot donné.

Dès lors, la suite fut de s'intéresser à la réciproque : quels étaient les mots implémentant une fonction logique donnée ? Notre approche fut celle de la génération de masse passant par la création d'un très grand nombre de mots que l'on savait représentatif de l'ensemble des mots possibles.

La création puis le parcours d'une base de donnée de mots a été notre dernière contribution au projet et nous a permis de répondre à la question dans une certaine mesure.



Première partie

Formalisation du langage bio-logique

1 Définitions

1.1 Alphabet

Soit S l'ensemble des **Symboles** formant, avec le **mot vide**, l'**Alphabet**. Ils sont qualifiés de **Forward** ou **Reverse** : c'est leur sens.

Les différentes couleurs de site représentent les différents **Inputs**, au nombre de n .

Un **Mot du langage** est le résultat de la **concaténation** de plusieurs symboles ou mots du langage.

On parlera de **Séquence** pour désigner des mots composés uniquement de promoteurs, terminateurs et gènes.

1.2 Mots bien formés

On associe à chaque symbole un attribut de **Taille**, et on définit la **distance entre deux symboles** par la somme des tailles des symboles entre les deux.




	Forward	Reverse	Taille (kb ¹)
Promoteurs			0,04
Terminateurs			0,10
Gènes			1,00
Sites			0,04
Sites utilisés			0,04

FIGURE 1 – Alphabet du langage

On attribue aux promoteurs une information de **Portée**, qui nous permet de définir la notion de **croisement des promoteurs** : deux promoteurs se croisent s'ils sont de sens opposés en vis-à-vis (*) et séparés d'une distance inférieure à la portée d'un promoteur uniquement par le mot vide ou des sites. On fixe cette portée à **1,5 kilobases**.

Un mot est dit **bien formé** si pour chaque input, il contient exactement deux sites, et s'il ne contient aucun croisement de promoteurs.

1. kb : kilobases ; en milliers de bases





Exemple :    n'est pas un mot bien formé car il y a croisement de promoteurs.

Exemple :     est un mot bien formé.





2 Propriétés des mots bien formés

2.1 Excision et Inversion

Un mot bien formé de n inputs possède $2 * n$ sites. Ces inputs peuvent être ou non **activées**, ce qui applique une transformation aux symboles situés entre les deux sites de l'input, et qui transforme aussi ces sites en sites utilisés.

—  *  et  *  appliquent une **Excision** à * si l'input est activée, c'est-à-dire que l'on supprime le sous-mot *.





Exemple :     devient  

—  *  et  *  appliquent une **Inversion** à * si l'input est activée, c'est-à-dire que l'on effectue une rotation du sous-mot *, puis que l'on change le sens de tous ses symboles.

Exemple :     devient    

2.2 Non-croisement des sites

Un mot bien formé respecte le **non-croisement des sites**, c'est-à-dire que les paires de sites se terminent dans l'ordre dans lequel elles ont commencé.

Exemple :     n'est pas un mot bien formé car il y a croisement des sites.

Exemple :      et      sont des mots bien formés.

En assimilant chaque paire de site à une paire de parenthèse, on peut ainsi dire que l'ensemble des mots bien formés composés uniquement de sites forme un **langage de Dyck**².

2.3 Confluence des activations

Dans un mot bien formé, l'**ordre d'activation** des inputs ne change pas le mot obtenu.

Exemple :     peut devenir     ou      qui deviennent tous deux    

Dans la suite du document et sauf mention contraire, les mots sont considérés comme bien formés.

Deuxième partie

Passer d'un mot à une fonction logique

3 Logique des mots

3.1 Activation de gène

Dans un mot, un **gène est considéré comme activé**, s'il est à portée d'un promoteur de même sens que lui, sans que les deux soient séparés par un terminateur du même sens.

2. Le langage de Dyck est l'ensemble des mots bien parenthésés, sur un alphabet composé des parenthèses ouvrante et fermante

En d'autres termes, si le mot peut s'écrire $* \textcolor{blue}{\rightarrow} * \textcolor{green}{G} *$ avec aucun $\textcolor{red}{T}$ dans $*$ (ou dans l'autre sens). On dit d'un mot avec au moins un gène activé que son **Output est activée**.

Exemple : $\textcolor{blue}{\rightarrow} \textcolor{red}{\perp} \textcolor{red}{T} \textcolor{green}{G}$ n'active pas son gène mais $\textcolor{blue}{\rightarrow} \textcolor{red}{\perp} \textcolor{green}{G} \textcolor{red}{T}$ oui.

3.2 Mots intègres

L'activation d'une input d'un mot le transforme, et permet donc d'obtenir un nouveau **mot résultat** de la transformation. Pour un mot contenant n inputs, il y a 2^n combinaisons d'activation des inputs, et donc 2^n mots résultats. Rappelons en effet que l'ordre d'activation des inputs est indifférent (section 2.3).

On définit ainsi un nouveau type de mots, les **mots intègres**. Un mot est dit intègre s'il est bien formé lui, mais aussi tous les mots résultats issus de lui.

Exemple : $\textcolor{blue}{\rightarrow} \textcolor{yellow}{\triangleright} \textcolor{blue}{\rightarrow} \textcolor{yellow}{\triangleleft} \textcolor{green}{G}$ est un mot bien formé, mais pas intègre, car si on active son input, un croisement de promoteurs apparaît $\textcolor{blue}{\rightarrow} \textcolor{yellow}{\triangleright} \textcolor{blue}{\triangleleft} \textcolor{yellow}{\triangleright} \textcolor{green}{G}$

3.3 Table de vérité d'un mot intègre

Considérons un mot intègre. Pour chacun de ses mots résultats, on peut rechercher si l'output est activée ou non. La synthèse de ces informations nous permet alors de dresser la **table de vérité d'un mot intègre**, de laquelle on sait déduire une fonction logique sous forme normale disjonctive.

Inputs A ($\textcolor{yellow}{\triangleright}$) B ($\textcolor{yellow}{\triangleright}$)		Mot obtenu	Output	Clause conjonctive
0	0	$\textcolor{blue}{\rightarrow} \textcolor{yellow}{\triangleright} \textcolor{yellow}{\triangleright} \textcolor{red}{\perp} \textcolor{yellow}{\triangleleft} \textcolor{green}{G}$	$\textcolor{red}{1}$	$\neg A \wedge \neg B$
0	1	$\textcolor{blue}{\rightarrow} \textcolor{yellow}{\triangleright} \textcolor{yellow}{\triangleright} \textcolor{red}{T} \textcolor{yellow}{\triangleleft} \textcolor{green}{G}$	$\textcolor{red}{0}$	$\neg A \wedge B$
1	0	$\textcolor{blue}{\rightarrow} \textcolor{yellow}{\triangleright} \textcolor{yellow}{\triangleright} \textcolor{green}{G}$	$\textcolor{red}{1}$	$A \wedge \neg B$
1	1	$\textcolor{blue}{\rightarrow} \textcolor{yellow}{\triangleright} \textcolor{yellow}{\triangleright} \textcolor{green}{G}$	$\textcolor{red}{1}$	$A \wedge B$

TABLEAU 1 – Exemple de comment passer d'un mot à une fonction logique

Ainsi, on peut déduire d'un mot intègre une **unique fonction logique**, selon qu'il a ou non son output activé quand on active ou non ses inputs. Ici le mot implémente $(\neg A \wedge \neg B) \vee (A \wedge \neg B) \vee (A \wedge B)$.

3.4 Mots symétriques

Le **symétrique** d'un mot, bien formé ou non, est défini comme étant le mot obtenu en appliquant une rotation et une inversion de sens des symboles du mot d'origine.

Exemple : $\textcolor{blue}{\rightarrow} \textcolor{red}{\perp} \textcolor{green}{G}$ devient $\textcolor{green}{\curvearrowright} \textcolor{red}{T} \textcolor{blue}{\triangleleft}$

Un mot bien formé et son symétrique implémentent la même fonction logique.

4 Simplification de fonctions logiques

4.1 Description de la méthode Quine-McCluskey

La **méthode de Quine-McCluskey**, présentée par E. J. MCCLUSKEY JR. [3], propose une amélioration de la méthode Quine, et décrit un algorithme de détermination d'une **forme normale disjonctive minimale** d'une fonction logique.

Elle part d'une forme normale disjonctive (FND), pour y appliquer méthodiquement le **principe du tiers-exclus** puis le **théorème du consensus**, par un procédé simple et intuitif.

4.2 Simplifications grâce au principe du tiers-exclu

Pour cette méthode, on part d'une fonction logique sous FND. Dans le cadre du projet en dressant la table de vérité d'un mot, on déduit d'ailleurs directement la FND de l'étude du mot (section 3.3).

Exemple : Pour cet exemple suivi, on va prendre la FND suivante :

$$\underbrace{(\neg a \wedge \neg b \wedge c)}_1 \vee \underbrace{(\neg a \wedge b \wedge \neg c)}_2 \vee \underbrace{(\neg a \wedge b \wedge c)}_3 \vee \underbrace{(a \wedge \neg b \wedge \neg c)}_4 \vee \underbrace{(a \wedge \neg b \wedge c)}_5 \vee \underbrace{(a \wedge b \wedge \neg c)}_6$$

Appliquer le **principe du tiers-exclu** va consister à identifier des paires de clauses conjonctives différentes d'un seul littéral, pour en tirer la clause conjonctive sans cette différence, qui subsume donc les deux d'origines. On répète cette opération tant que l'on peut appliquer le principe du tiers-exclu sur les clauses conjonctives obtenus.

Exemple : Pour déterminer les clauses conjonctives différentes d'un littéral, on leur attribue un numéro et on les classe par nombre de littéraux positifs.

1 littéral positif		2 littéraux positifs	
1	$\neg a \wedge \neg b \wedge c$	3	$\neg a \wedge b \wedge c$
2	$\neg a \wedge b \wedge \neg c$	5	$a \wedge \neg b \wedge c$
4	$a \wedge \neg b \wedge \neg c$	6	$a \wedge b \wedge \neg c$

Exemple : On cherche alors les différents manières d'appliquer le principe de tiers-exclu entre les clauses conjonctives différentes d'un seul littéral positif, jusqu'à ne plus pouvoir inhiber d'autres clauses conjonctives.

(1,3)	$\neg a \wedge c$	(2,6)	$b \wedge \neg c$
(1,5)	$\neg b \wedge c$	(4,5)	$a \wedge \neg b$
(2,3)	$\neg a \wedge b$	(4,6)	$a \wedge \neg c$

4.3 Simplifications par théorème du consensus

A cette étape, nous avons une liste de clauses conjonctives et l'on ne peut plus appliquer le principe du tiers-exclus. On va donc se servir du **théorème du consensus** pour terminer la simplification de la fonction logique. Ce théorème dit que :

$$(a \wedge b) \vee (\neg a \wedge c) \vee (b \wedge c) = (a \wedge b) \vee (\neg a \wedge c)$$

L'idée est donc de chercher, dans les clauses conjonctives que l'on a obtenu à l'étape d'avant, celles dont les littéraux sont présents avec la même positivité dans d'autres clauses conjonctives, afin de les éliminer pour redondance.

Cependant, en fonction de l'ordre avec lequel on le fait, on peut éliminer des clauses conjonctives différentes tout en ne perdant aucune information. C'est pour cela qu'il n'y a pas unicité de la forme minimale obtenue à la fin.

Exemple : Grâce au numérotage des clauses conjonctives, on sait de quelles clauses conjonctives initiales elles proviennent. Il suffit donc très simplement de repérer celles dont la numérotation se retrouve dans d'autres. Par exemple, (1,3) est redondante car on retrouve 1 et 3 dans (1,5) et (2,3).

(1,3)	$\neg a \wedge c$	(2,6)	$b \wedge \neg c$
(1,5)	$\neg b \wedge c$	(4,5)	$a \wedge \neg b$
(2,3)	$\neg a \wedge b$	(4,6)	$a \wedge \neg c$

Exemple : Comme on le voit ci-contre, l'ordre des clauses conjonctives avec lequel on recherche les redondances amène à en éliminer des différentes.

(1,3)	$\neg a \wedge c$	(2,6)	$b \wedge \neg c$
(1,5)	$\neg b \wedge c$	(4,5)	$a \wedge \neg b$
(2,3)	$\neg a \wedge b$	(4,6)	$a \wedge \neg c$

Au final, chaque clause conjonctive initiale (repérée par son numéro), ne doit être représentée qu'une seule fois pour bien avoir une forme normale disjonctive minimale.

Exemple : La première manière de trouver les redondances donne : $(\neg b \wedge c) \vee (\neg a \wedge b) \vee (a \wedge \neg c)$
 La seconde donne : $(\neg a \wedge c) \vee (b \wedge \neg c) \vee (a \wedge \neg b)$
 Et toutes deux sont équivalentes.

5 Sémantique des séquences

5.1 Définition de la sémantique

Un mot se compose de sites entre lesquels on peut trouver, ou non, des séquences (composés uniquement de promoteurs, terminateurs, et gènes). Il existe une infinité de séquences possibles, mais toutes peuvent être décrites selon plusieurs **attributs** : c'est leur **sémantique**. La sémantique d'une séquence nous permet de décrire son comportement au sein d'un mot.

Connaissant le rôle de chacun des symboles qui composent une séquence, on en déduit la liste des attributs qui caractérisent une séquence :

- Est-ce qu'elle promeut dans un sens (**PFing**) ou l'autre (**PRing**) ?
- Est-ce qu'elle termine dans un sens (**TFing**) ou l'autre (**TRing**) ?
- Est-ce qu'elle peut être activée dans un sens (**GFing**) ou l'autre (**GRing**) ?
- Est-ce qu'elle active elle-même le gène (**Ging**) ?



















Séquence	PFing	PRing	TFing	TRing	GFing	GRing	Ging
 	1	0	0	0	1	0	1
  	0	1	1	0	0	1	0
   	0	1	1	0	0	1	0

FIGURE 2 – Exemples de sémantiques

5.2 Equivalences sémantiques

On pourrait dire que deux séquences sont sémantiquement équivalentes si elles ont la même sémantique. Cependant, l'attribut **Ging** est différent des autres, en ce sens que, une séquence **Ging** mise au sein d'un mot entraînera l'activation de l'output de tout le mot, quelque soit l'interprétation de ses autres attributs.

Exemple :     active le gène en son sein, sauf dans le cas de l'excision. Ce mot représente la fonction $\neg a$.








*Mais      a exactement la même logique. Bien que l'attribut **TFing** de la séquence soit différent du mot d'avant, la fonction représentée est toujours $\neg a$.*

Cette différenciation de **Ging** par rapport aux autres attributs rajoute une condition à la définition d'équivalence sémantique : deux séquences sont **sémantiquement équivalentes** si elles possèdent la même caractérisation, ou si elles activent toutes deux le gène en leur sein.

5.3 Classes d'équivalence représentées

Les ensembles regroupant toutes les séquences équivalentes entre elles sont des **classes d'équivalences**. Une classe d'équivalence est dite **représentée** si elle n'est pas vide, et l'union de toutes les classes d'équivalences forme l'univers des séquences.

Ce qui est intéressant, c'est de remarquer que toutes les classes d'équivalences ne sont pas représentées. En effet, certaines combinaisons d'attributs sont impossibles.

*Exemple : **TFing** et **GFing** sont incompatibles, car si un  se trouve avant un , jamais le  ne sera activé, et si le  se trouve après, jamais il ne servira, car s'il y a  plus tôt, il activera le  avant de rencontrer le .*

Pour savoir quelles classes d'équivalences sont représentées, et pour trouver leur représentant le plus simple, l'algorithme 1 construit l'ensemble des séquences de sémantiques différentes, en commençant par les séquences triviales, et en veillant à éliminer les séquences mal formées.

L'algorithme se termine, avec un ensemble de **26 séquences** qui sont donc les représentants les plus simples des **26 classes d'équivalences possibles**. La liste détaillée des 26 est disponible en annexe A.

Exemple : Les classes d'équivalences triviales, caractérisées par aucun ou un seul attribut, ont pour représentants les plus simples le mot vide ou chaque symbole seul.

Algorithme 1 Création de l'ensemble des représentants les plus simples des classes d'équivalences**Algorithme** CRÉERREPRÉSENTANTS : ensemble des représentants les plus simples $R \leftarrow \{\epsilon, PF, PR, TF, TR, GF, GR\}$: ensemble \triangleright Représentants des classes d'équivalences triviales $estR?$: booléen $nouveauR$: mot**Pour tout** $r \in R$ **Faire****Pour tout** $s \in R$ **Faire** $nouveauR \leftarrow r + s$ \triangleright Pour chaque concaténation de deux représentants $estR? \leftarrow Vrai$ **Pour tout** $t \in R$ **Faire** \triangleright Est-elle représentative d'une sémantique nouvelle ?**Si** $nouveauR.estSemantiquementEquivalent(t)$ **Alors** $estR? \leftarrow Faux$ \triangleright Non, donc ce n'est pas un nouveau représentant**Break****Fin Si****Fin Pour****Si** $estR?$ et $estBienForme(nouveauR)$ **Alors** \triangleright Oui, donc c'est un nouveau représentant $R \leftarrow R \cup \{nouveauR\}$ **Fin Si****Fin Pour****Fin Pour****Renvoyer** R \triangleright Résultat en annexe A**Fin Algorithme**

Troisième partie

Passer d'une fonction logique à un mot

6 Procédé et algorithmes

6.1 Générer un maximum de mots représentatifs

Une fois capable de passer d'un mot à sa fonction logique, nous devons pouvoir faire l'opération inverse. Pour cela nous devons réussir à répondre aux questions suivantes : **Toutes les fonctions logiques sont-elles implémentables ? Peut-on donner une liste de mot implémentant une fonction donnée ?**

Notre idée est de partir de ce que l'on sait sur les classes d'équivalences (section 5.3) afin de construire un ensemble de mot représentatif à partir duquel on peut déduire l'ensemble de tous les mots.

La génération de cet ensemble de mot se divise en plusieurs algorithmes exécutés linéairement. La trame est de **construire le langage de Dyck** (algorithmes 3 et 4), d'abord sans différenciation des paires de parenthèses, puis avec ; de là, de **construire l'ensemble des mots composés uniquement de sites** ; et enfin de **remplir les inter-sites de ces mots** (algorithme 5) avec tous les représentants les plus simples. Puis, on regarde si les mots obtenus implémentent bien des **fonctions logiques non simplifiables** (algorithme 2) en moins d'inputs.

Ce faisant, en prenant les représentants les plus simples (annexe A), on forme les mots les plus simples, à partir desquels, si besoin est, on peut trouver d'autres mots équivalents en prenant des séquences équivalentes.

6.2 Trouver les fonctions logiques simplifiables

Pour un nombre n de variables, il y a 2^{2^n} fonctions logiques différentes : 4 pour 1 variable, 16 pour 2, 256 pour 3, ... Cependant, certaines de ces fonctions sont simplifiables en moins de variables, par exemple $(a \wedge b) \vee (a \wedge \neg b) = a$. Au final, on peut calculer le nombre de fonctions logiques non simplifiables avec :

$$f(0) = 2 \mid f(n) = 2^{2^n} - \sum_{k=0}^{n-1} \binom{n}{k} \cdot f(k) \Rightarrow \begin{cases} f(1) = 2 \\ f(2) = 10 \\ f(3) = 218 \end{cases}$$

Afin de vérifier que les mots construits n'implémentent pas des fonctions logiques simplifiables en moins

de variables, on construit, par induction, l'ensemble F de ces fonctions :

$$\left\{ \begin{array}{ll} (B) & \top, \perp \in F \\ (B) & n > 1 \mid a_1, \dots, a_n : \text{variables} \Rightarrow a_1, \dots, a_n \in F \\ (I) & n > 1 \mid p \in F \Rightarrow \neg p \in F \\ (I) & n > 2 \mid p, q \in F \mid \text{nombreVariables}(p \wedge q) < n \Rightarrow p \wedge q \in F \\ (I) & n > 2 \mid p, q \in F \mid \text{nombreVariables}(p \vee q) < n \Rightarrow p \vee q \in F \end{array} \right.$$

Algorithme 2 Recherche de l'ensemble des fonctions logiques du nombre d'inputs donné simplifiables

Algorithme TROUVERFONCTIONSIMPLIFIABLES($n \in \mathbb{N}^*$: nombre d'inputs) : ensemble des fonctions de n inputs simplifiables

lettres $\leftarrow [a, b, c, \dots]$: tableau de caractères

fonctions $\leftarrow \emptyset$: ensemble

Si $n > 1$ **Alors**

Pour $i \leftarrow 0, i < n$ **Faire**

 ▷ Fonctions à un littéral

fonctions \leftarrow *fonctions* $\cup \{\text{lettres}[i], \neg \text{lettres}[i]\}$

Fin Pour

Si $n > 2$ **Alors**

 ▷ Fonctions à plusieurs littéraux

Pour tout $(l, m) \in \text{fonctions}^2$ **Faire**

Si $\text{nbVariables}(l \vee m) < n$ **Alors**

fonctions \leftarrow *fonctions* $\cup \{l \vee m\}$

 ▷ Création de toutes les disjonctions possibles

Fin Si

Fin Pour

Pour tout $(l, m) \in \text{fonctions}^2$ **Faire**

Si $\text{nbVariables}(l \wedge m) < n$ **Alors**

fonctions \leftarrow *fonctions* $\cup \{l \wedge m\}$

 ▷ Création de toutes les conjonctions possibles

Fin Si

Fin Pour

Fin Si

Fin Si

Renvoyer *fonctions* $\cup \{\top\} \cup \{\perp\}$

Fin Algorithme

6.3 Créer le langage de Dyck

Dans un premier temps, on construit le langage de Dyck D par induction, en remplaçant les parenthèses par des 0 et 1 par commodité d'utilisation :

$$\left\{ \begin{array}{ll} (B) & \epsilon \in D \\ (I) & x \in D \Rightarrow (x) \in D \\ (I) & x, y \in D \Rightarrow xy \in D \end{array} \right.$$

Et on s'arrête dès que l'on atteint des mots de taille voulue. Par propriété de ce langage, l'ensemble des mots de Dyck de n paires de parenthèses est de cardinal $C_n^3 = \frac{1}{n+1} \cdot \binom{2n}{n}$.

Dans un second temps, on parcourt les mots obtenus en nommant chaque paire de parenthèses (ou paire de 0 et 1), pour ainsi avoir des paires d'inputs.

Exemple : Pour $n = 2$: 0101 et 0011 sont les deux mots de Dyck :

$$\{ (0.) \ \epsilon \in D \Rightarrow \left\{ \begin{array}{ll} (0.) \ \epsilon \in D \\ (1.) \ 01 \in D \end{array} \right. \Rightarrow \left\{ \begin{array}{ll} (0.) \ \epsilon \in D \\ (1.) \ 01 \in D \\ (2.) \ 0101 \in D \\ (2.) \ 0011 \in D \end{array} \right.$$

Ils deviennent ensuite AABB et ABBA.

3. C_n : nombre de Catalan ; $C_1 = 1$; $C_2 = 2$; $C_3 = 5$; $C_4 = 14$

Algorithme 3 Création de l'ensemble des mots de Dyck de taille donnée

Algorithme CRÉERLANGAGEDEDYCK($n \in \mathbb{N}$: nombre d'inputs) : ensemble des mots de Dyck de n paires
motsDeDyck : tableau de chaînes de caractères
motsDeDyck[0] \leftarrow "01" ▷ Étape (1.) de la construction par induction
nbMotsDeDyck : entier
Pour $i \leftarrow 0, i < n$ **Faire**
 nbMotsDeDyck \leftarrow *taille*(*motsDeDyck*)
 Pour $j \leftarrow 0, j \leq \text{nbMotsDeDyck}$ **Faire**
 motsDeDyck.ajouter("0" + *motsDeDyck*[j] + "1") ▷ Nouveau mot par encapsulation
 Pour $k \leftarrow 0, k \leq \text{taille}(\text{motsDeDyck})$ **Faire** ▷ Nouveaux mots par concaténations
 motsDeDyck.ajouter(*motsDeDyck*[j] + *motsDeDyck*[k])
 motsDeDyck.ajouter(*motsDeDyck*[k] + *motsDeDyck*[j])
 Fin Pour
 Fin Pour
Fin Pour
Pour $i \leftarrow 0, i \leq \text{taille}(\text{motsDeDyck})$ **Faire**
 Si *taille*(*motsDeDyck*[i]) $\neq 2n$ **Alors**
 supprimer(*motsDeDyck*, i) ▷ Suppression des mots trop courts ou longs
 i – –
 Fin Si
Fin Pour
Renvoyer *motsDeDyck*
Fin Algorithme

Algorithme 4 Création de l'ensemble des mots de taille donnée composés uniquement d'inputs

Algorithme CRÉERMOTSDINPUTS($n \in \mathbb{N}$: nombre d'inputs) : ensemble des mots de n paires d'inputs
motsDeDyck, *motsDInputs* : tableaux de chaînes de caractères
motsDeDyck \leftarrow CRÉERLANGAGEDEDYCK(n)
nbZeros, *nbPairesOuvertes*, k : entiers
Pour $i \leftarrow 0, i \leq \text{taille}(\text{motsDeDyck})$ **Faire** ▷ Pour chaque mot de Dyck
 nbZeros \leftarrow 0
 Pour $j \leftarrow 0, j \leq \text{taille}(\text{motsDeDyck}[i])$ **Faire** ▷ On parcourt le mot de Dyck
 motsDInputs[i] \leftarrow ""
 Si *motsDeDyck*[i][j] = "0" **Alors** ▷ Si on trouve une parenthèse ouvrante
 nbZeros + + ▷ On ajoute une nouvelle input
 motsDInputs[i] \leftarrow *motsDInputs*[i] + *nbZeros*
 Sinon ▷ Si on trouve une parenthèse fermante
 nbPairesOuvertes \leftarrow 0 ▷ On recherche sa parenthèse ouvrante
 $k \leftarrow j - 1$ ▷ Parmi les caractères précédents
 Tant que (*motsDeDyck*[i][k] = "1") ou (*nbPairesOuvertes* \neq 0) **Faire**
 Si *motsDeDyck*[i][k] = "0" **Alors**
 nbPairesOuvertes + +
 Sinon
 nbPairesOuvertes – –
 Fin Si
 k – –
 Fin Tant que
 motsDInputs[i] \leftarrow *motsDInputs*[i] + *motsDeDyck*[i][k] ▷ On ajoute l'input correspondante
 Fin Si
 Fin Pour
Fin Pour
Renvoyer *motsDInputs*
Fin Algorithme

6.4 Créer les mots de sites

Les paires de lettres dans les mots obtenus correspondent à des paires de sites. Or, une paire de site possède deux états : excision ou inversion. Pour chacun des mots de Dyck, on crée donc les mots de sites

correspondant à toutes les combinaisons d'états des paires de sites. Le nombre de mots composés uniquement de sites est donc de $2^n \cdot C_n$.

Exemple : Pour $n = 2$: $D = \{AABB, ABBA\}$, les mots de sites sont donc :

$$AABB \Rightarrow \left\{ \begin{array}{c} \blacktriangleright \blacktriangleright \blacktriangleright \blacktriangleright \\ \blacktriangleright \blacktriangleright \blacktriangleright \blacktriangleright \\ \blacktriangleright \blacktriangleright \blacktriangleright \blacktriangleright \\ \blacktriangleright \blacktriangleright \blacktriangleright \blacktriangleright \end{array} \right\} ; ABBA \Rightarrow \left\{ \begin{array}{c} \blacktriangleright \blacktriangleright \blacktriangleright \blacktriangleright \\ \blacktriangleright \blacktriangleright \blacktriangleright \blacktriangleright \\ \blacktriangleright \blacktriangleright \blacktriangleright \blacktriangleright \\ \blacktriangleright \blacktriangleright \blacktriangleright \blacktriangleright \end{array} \right\}$$

6.5 Remplir les inter-sites

Chaque mot de sites comporte deux extrémités et $n - 1$ inter-sites dans lesquels on peut insérer des séquences. Pour obtenir l'ensemble des mots de sémantiques différentes, chacun des $n + 1$ espaces doit donc accueillir les 26 sémantiques différentes, sous la forme du représentant le plus simple de ces classes d'équivalences. Cette méthode porte à $2^n \cdot C_n \cdot 26^{n+1}$ le nombre de mots formés.

Exemple : Le mot de site $\blacktriangleright \blacktriangleright \blacktriangleright \blacktriangleright$ est la racine de plus de 11 millions de mots :
(26 séquences) \blacktriangleright (26 séquences) \blacktriangleright (26 séquences) \blacktriangleright (26 séquences) \blacktriangleright (26 séquences)

Pour réduire ce nombre, et en partant du principe que les mots les plus simples sont les plus intéressants biologiquement parlant, on regarde si chaque séquence insérée change la sémantique de la portion du mot où elle est insérée, c'est-à-dire l'espace en question et le précédent, avec et sans l'input activée. Si cette insertion n'a pas d'influence sur la sémantique, alors on ne la fait pas car elle est équivalente à l'insertion de la séquence vide, plus simple. On élimine ainsi de la génération des mots dont des équivalents ont déjà été générés (en effet la séquence vide fait partie des 26 séquences testées).

Exemple : Soit un espace dans une inversion, après un espace déjà rempli : $* \blacktriangleright ? *$
On forme le mot $\blacktriangleright \blacktriangleright ? \blacktriangleright$ et on remplit $?$ avec chacune des 26 séquences.
Pour chaque mot formé, on compare sa sémantique avec le mot rempli par la séquence vide $\blacktriangleright \blacktriangleright \blacktriangleright \blacktriangleright$, dans les cas où l'input est activée ou non.

Exemple : En prenant $? = \text{T}$, on a :

$$\text{sémantique}(\blacktriangleright \blacktriangleright \text{T} \blacktriangleright) \neq \text{sémantique}(\blacktriangleright \blacktriangleright \blacktriangleright \blacktriangleright)$$

$$\text{sémantique}(\blacktriangleright \blacktriangleright \text{T} \blacktriangleright) \neq \text{sémantique}(\blacktriangleright \blacktriangleright \blacktriangleright \blacktriangleright)$$

Donc T est une séquence que l'on peut placer dans l'espace.

	PFing	PRing	TFing	TRing	GFing	GRing	Ging
$\blacktriangleright \blacktriangleright \text{T} \blacktriangleright$	0	0	1	0	0	0	0
$\blacktriangleright \blacktriangleright \blacktriangleright \blacktriangleright$	1	0	0	0	0	0	0
$\blacktriangleright \blacktriangleright \text{T} \blacktriangleright$	1	0	0	1	0	0	0
$\blacktriangleright \blacktriangleright \blacktriangleright \blacktriangleright$	1	0	0	0	0	0	0

Sur cet algorithme (5), à l'étape où pour chaque mot de sites on commence à générer des mots (**Pour tout repr** \in **representants Faire**), sachant que le nombre de branches est connu (les 26 représentants), on applique une **parallélisation des tâches** sur plusieurs fils (un fil pour un mot de site) afin d'aller d'autant plus vite qu'il y a de cœurs dans le processeur.

Algorithme 5 Remplissage des mots de taille donnée composés uniquement de sites par des séquences**Algorithme** GÉNÉRERMOTS($n \in \mathbb{N}^*$) : ensemble des mots générés $motsGeneres \leftarrow \emptyset$: ensemble $motsDeSites \leftarrow \text{CRÉERMOTSDESITES}(n)$: tableau de mots $representants \leftarrow$ les 26 représentants de l'annexe A : tableau de séquences $fonctionsSimplifiables \leftarrow \text{TROUVERFONCTIONSIMPLIFIABLES}(n)$: ensemble**Fonction** REMPLIRINTERSITE(mot : mot, seq : séquence, $intersite \in \mathbb{N}$) $nouveauMot \leftarrow \text{insérer}(mot, seq, 2 \cdot intersite + 1)$: mot \triangleright On insère seq dans mot à $intersite$ **Tant que** $2 \cdot intersite + 1 \neq 2 \cdot n + 1$ **Faire** \triangleright Tant qu'on n'est pas à l'extrémité du mot**Pour tout** $repr \in representants$ **Faire** \triangleright On remplit récursivement le mot**Si** $estUtile(repr, nouveauMot)$ **Alors** \triangleright Avec des séquences non-inutiles (section 6.5) $\text{REEMPLIRINTERSITE}(nouveauMot, repr, intersite + 1)$ **Fin Si****Fin Pour****Fin Tant que****Si** $nouveauMot.estIntegre()$ et $nouveauMot.fonctionLogique() \notin fonctionsSimplifiables$ **Alors** $motsGeneres \leftarrow motsGeneres \cup \{nouveauMot\}$ **Fin Si****Fin Fonction****Pour tout** $motDeSite \in motsDeSites$ **Faire** \triangleright Pour chaque mot de sites**Pour tout** $repr \in representants$ **Faire** \triangleright On crée une branche de construction par représentant $\text{REEMPLIRINTERSITE}(motDeSites, repr, 0)$ **Fin Pour****Fin Pour****Renvoyer** $motsGeneres$ **Fin Algorithme**

7 Résultats

7.1 Résultats bruts des générations

La génération a été effectuée avec la configuration suivante :

- **CPU** : AMD FX 8350, 4,7 Ghz, 8 coeurs
- **RAM** : 32 Go, 2133 Mhz, DDR3
- **HDD** : 1 To, 7200 RPM, SATA II

7.2 Interprétations

Le premier résultat à remarquer est aussi le plus important : **pour chaque fonction logique à 1/2/3-input(s), il existe des mots pour les implémenter**. Et après avoir présenté le résultat de la génération à MME. GUIZIOU, on a également eu confirmation qu'il existe pour chaque fonction logique des mots biologiquement possibles. Cette question était encore en suspens avant ce projet, et pour certaines fonctions logiques dont aucun représentant n'était connu, **les mots proposés vont être biologiquement testés** pour confirmer leur adéquation.

La seconde interprétation que l'on peut faire des résultats est d'ordre critique, sur **le nombre impressionnant de mots générés et le temps nécessaire à les créer**. Or dans la pratique, les mots seront classés (voir section 8), et seuls les meilleurs seront retenus, l'immense majorité n'étant pas intéressante. Les complexités temporelle et spatiale pourraient donc être améliorées en ne **générant que les meilleurs des mots**.

L'analyse de la répartition des mots vient ajouter à ce point : certaines fonctions logiques présentent des millions de représentants, quand d'autres n'en présentent que des dizaines. Pour 3 inputs, seulement **17 fonctions sur 218 représentent 87% de tous les mots générés**.

Inputs	1	2	3
Fonctions logiques irréductibles	2	10	218
Fonctions logiques représentées	2	10	218
Mots générés	523	750k	1,36Md
Mots retenus	30,0%	11,4%	4,0%
Temps de calcul effectif	< 1s	22s	68h
Temps de calcul processeur	< 1s	2m34	346h
Espace disque consommé	3 Ko	3,1 Mo	2,9 Go

FIGURE 3 – Résultats des générations

De plus, la part des mots retenus est très faible. L'algorithme gagnerait donc en complexité temporelle s'il évitait de générer ses mots non-retenus, ce qu'il est obligé de faire dans cette version de l'algorithme pour savoir ou non s'il faut les retenir.

Quatrième partie

Aspect technique

8 Base de données

8.1 Notre choix de base de données

Le nombre de mots générés étant très grand, pour faciliter leur exploitation, le choix a été fait d'utiliser un **système de gestion de base de données** (SGBD). Car même si les résultats sont enregistrés dans des fichiers textes, parcourir et classer des centaines de fichiers et des milliers de lignes de texte est fastidieux et demanderait de coder de nombreux algorithmes, que les SGBD intègrent nativement. De plus, en cas d'évolution ou de changements dans les critères de classements des mots, la solution fichier n'offre aucune flexibilité.

Quant au choix du SGBD, il s'est orienté vers **PostgreSQL**, un SGDB relationnel respectant la norme SQL, en raison de sa légèreté, de sa licence libre (PostgreSQL License) et de sa facilité d'installation et d'utilisation avec C++.

8.2 Architecture de la base de données

L'architecture de la base de données a été pensée pour **optimiser le temps de réponse aux requêtes et l'espace occupé**. En effet, on parle de dizaines de millions de mots à enregistrer.

Pour optimiser le temps d'exécution des requêtes, le nombre de colonnes de la table contenant les mots a été réduit au minimum, de sorte qu'il ne reste que les mots et les clés étrangères. Le lien entre les données se fait à l'aide d'unions, l'une des opérations les plus optimisées sur les SGBD relationnels [1].

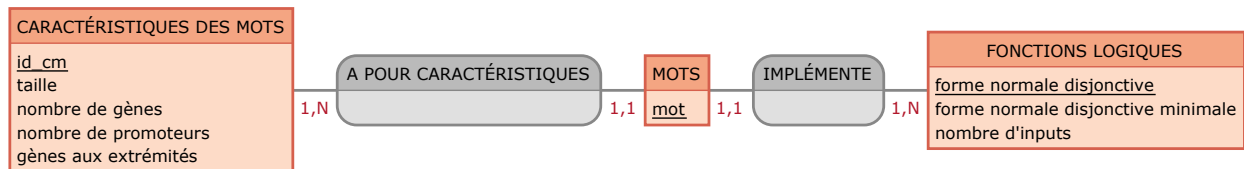


FIGURE 4 – Modèle entité-association de la base de données

Caractéristiques des mots	(<u>id_cm</u> , taille, nombre de gènes, nombre de promoteurs, gènes aux extrémités)
Mots	(<u>mot</u> , id_cm, forme normale disjonctive)
Fonctions logiques	(forme normale disjonctive, forme normale disjonctive minimale, nombre d'inputs)

FIGURE 5 – Modèle relationnel de la base de données

8.3 Utilisation de la base de données

Lors de l'utilisation de notre programme pour trouver les mots qui implémentent une fonction logique, on effectue une requête qui **classe les mots suivant leur nombre de gènes puis leur taille en nombre de bases**, conformément au souhait exprimé lors d'une consultation avec MME. GUIZIOU, la commanditaire du projet.

Exemple : Avec la fonction logique A :

$\text{G} \triangleright \text{P} \triangleleft$ est préférable à $\text{G} \triangleright \text{T} \triangleright \text{P}$ car il est plus court.

$\text{P} \triangleright \text{G} \triangleright \text{T} \triangleright \text{P}$ est préférable à $\text{G} \triangleright \text{G} \triangleright \text{P} \triangleleft$ car il n'a qu'un gène.

De plus, l'utilisation d'une base de donnée permet de la questionner avec des requêtes personnalisées selon les besoins de l'utilisateur.

Le portage d'une partie du code en PHP nous a également permis de mettre en place **un site web** offrant un accès instantané, pratique, et graphique à l'ensemble des mots générés, triés par fonction logique et classés. Le site supporte également la recherche d'informations directement sur les mots ou fonctions logiques voulus. Il est accessible à l'annexe [C](#).

9 Le programme Genetix

9.1 Nos choix techniques

Le programme a été codé en **C++**. C'est un langage objet maîtrisé par les membres du groupe, et sa puissance, sa flexibilité, et sa compatibilité multi-plateforme ont été des atouts au projet. Il fut utilisé dans sa version 11 puis 14 afin de profiter des dernières améliorations en matière de parallélisation, lambdas-fonctions, et pointeurs intelligents.

Nous nous en sommes servis avec les bibliothèques **Boost**, pour sa gestion des options, des ensemble de bits dynamiques et des fichiers, et **Pqxx**, le driver C++ de PostgreSQL. Toutes les bibliothèques utilisées sont **compatibles multi-OS**.

Enfin, notre code source est agrémenté de commentaires formatés pour **Doxygen**, un générateur de documentation.

9.2 Utilisation du programme Genetix

Le programme Genetix s'exécute par ligne de commande et son aide détaillée peut être obtenue par **genetix -h**. Il permet d'effectuer toutes les tâches relatives au projet. On peut obtenir des informations sur des mots : symétrique, table de vérité, fonction logique implémentée sous forme minimale ; sur des fonctions logiques : table de vérité, forme minimale, 10 meilleurs mots qui l'implémentent ; ou encore comparer les deux. Il est également possible de lancer la génération si l'on souhaite se créer la base de données de mots.

Un tableau récapitulatif des options du programme est disponible en annexe [B](#).

Conclusion

10 État et ouverture du projet

Une fois notre programme capable de représenter des mots et les méthodes pour les manipuler, il a été aisé de faire la première partie du projet, la conversion d'un mot en une fonction logique. **La partie centrale et principale du projet a alors été de mettre en place cette génération et de la rendre viable à exécuter.** Une fois fait, à l'aide de la BDD, le passage d'une fonction logique à des mots s'est résumé à l'exécution de simples requêtes.

A mesure des différentes versions de l'algorithme, nous avons réduit le temps de génération, mais celui-ci reste très élevé ; il s'agit donc de le diminuer. On pourrait déjà chercher un moyen d'éliminer les symétries avant l'étape de génération, mais aussi améliorer la parallélisation des tâches, ou encore déterminer des profils de séquences éliminatoires. La génération au-delà de 3 inputs sera alors envisageable.

Une autre ouverture possible serait, à partir des résultats de la génération, d'extrapoler des règles permettant de construire directement les meilleurs mots pour une fonction logique.

11 Retours d'expérience

Ce projet a suscité chez nous trois un grand enthousiasme, et nous avons apprécié travailler en équipe.

Le choix de ce sujet a été motivé par l'envie de répondre à un réel besoin, manifesté par un suivi en continu de notre avancée, et par une évolution des attentes exprimées lors de deux réunions avec l'intéressée.

L'utilisation d'un large panel d'outils techniques nous a également permis d'approfondir nos connaissances sur le C++ et son univers. Se confronter aux limites de la puissance d'un ordinateur nous a aussi poussé à avoir une réflexion sur l'optimisation des algorithmes.

Annexes

A Sémantiques et représentants des classes d'équivalences

Séquence	PFing	PRing	TFing	TRing	GFing	GRing	Ging
	0	0	0	0	0	0	0
\rightarrow	1	0	0	0	0	0	0
\leftarrow	0	1	0	0	0	0	0
\top	0	0	1	0	0	0	0
\perp	0	0	0	1	0	0	0
G	0	0	0	0	1	0	0
\exists	0	0	0	0	0	1	0
$\rightarrow G$	1	0	0	0	1	0	1
$\leftarrow \rightarrow$	1	1	0	0	0	0	0
$\rightarrow \perp$	1	0	0	1	0	0	0
$\top \leftarrow$	0	1	1	0	0	0	0
$\top \perp$	0	0	1	1	0	0	0
$\rightarrow \exists$	1	0	0	0	0	1	0
$G \leftarrow$	0	1	0	0	1	0	0
$\leftarrow \exists$	0	1	0	0	0	1	0
$\top \exists$	0	0	1	0	0	1	0
$G \perp$	0	0	0	1	1	0	0
$G \rightarrow$	1	0	0	0	1	0	0
$\exists G$	0	0	0	0	1	1	0
$\leftarrow \rightarrow \exists$	1	1	0	0	0	1	0
$\top \leftarrow \exists$	0	1	1	0	0	1	0
$G \leftarrow \rightarrow$	1	1	0	0	1	0	0
$G \leftarrow \exists$	0	1	0	0	1	1	0
$G \rightarrow \perp$	1	0	0	1	1	0	0
$G \rightarrow \exists$	1	0	0	0	1	1	0
$G \leftarrow \rightarrow \exists$	1	1	0	0	1	1	0

TABLEAU 2 – Sémantiques et représentants les plus simples des 26 classes d'équivalences représentées

B Liste des options du programme Genetix

Option	Arguments ⁴	Définition
-w	<mot1> ... [motN]	Pour chaque mot, affiche son symétrique, sa table de vérité, et la fonction logique qu'il implémente.
-l	<logique1> ... [logiqueN]	Pour chaque fonction logique, affiche sa forme simplifiée, sa table de vérité, et la liste des dix meilleurs mots l'implémentant.
-c	<mot> <logique>	Affiche les informations respectives du mot et de la fonction logique, et si le mot implémente ou non la fonction logique.
-W	<fichier d'entrée> [fichier de sortie]	Applique -w à tous les mots du fichier d'entrée. Si spécifié, le flux de sortie sera le fichier de sortie.
-L	<fichier d'entrée> [fichier de sortie]	Applique -l à toutes les fonctions logiques du fichier d'entrée. Si spécifié, le flux de sortie sera le fichier de sortie.
-C	<fichier de mots d'entrée> <fichier de logiques d'entrée> [fichier de sortie]	Affiche les mots du fichier indexés par les fonctions logiques du fichier. Si spécifié, le flux de sortie sera le fichier de sortie.
-g	<nombre d'inputs>	Génère les mots utilisés par -l.
-b	<nombre d'inputs>	Enregistre les mots générés dans la base de donnée.

TABLEAU 3 – Liste des options principales de Genetix

C Dépôt Git du programme Genetix

Adresse du dépôt : https://bitbucket.org/Guigui_PL/genetix

D Site web de parcours de la BDD

Adresse du site web : <http://78.193.78.45/resultats/>

4. <...> : argument obligatoire; [...] : argument facultatif