UNIVERSITÉ DE NANTES
U.F.R. Sciences et Techniques
Département d'Informatique

Identification optimisée de protéines par comparaison d'ensembles

Alban Batard Salim Bourougaa
Axel Grimault Glenn Le Cornec

Encadrants

Irena Rusu
Dominique Tessier
Guillaume Fertin

Année 2010 - 2011

Table des matières

1 Introduction **2**

2 Présentation du problème **4**
2.1 Contexte . . . 4
2.2 Problématique . . . 5
2.3 Abstraction du problème . . . 6

3 Structures de données et algorithmes **9**
3.1 Solutions envisagées . . . 9
3.1.1 Comparaison par fréquence . . . 9
3.1.2 Comparaison par signature . . . 10
3.1.3 Comparaison par intersection des contenants . . . 10
3.2 Solution retenue . . . 11
3.2.1 Définition des Objets . . . 11
3.2.2 Fonctions . . . 11
3.2.3 Algorithme d'ajout dans une liste triée . . . 13
3.2.4 Algorithme d'intersection . . . 15
3.2.5 Algorithme d'Inclusion . . . 16
3.2.6 Algorithme d'Appartenance . . . 17
3.2.7 Algorithme Principal . . . 18
3.3 Applications sur des exemples . . . 19
3.3.1 Remplissage des peptides . . . 19
3.3.2 Exemple d'inclusion sur la $\mathcal{P}rotéine_3$ . . . 20

4 Analyse et tests **22**
4.1 Complexité . . . 22
4.1.1 Complexité du remplissage . . . 22
4.1.2 Complexité de l'algorithme d'intersection . . . 23
4.1.3 Complexité de l'inclusion et l'appartenance . . . 23
4.1.4 Complexité de notre algorithme . . . 24
4.2 Tests . . . 25
4.2.1 Comparaison par rapport à la fréquence . . . 26

5 Conclusion **27**

Chapitre 1

Introduction

Ce rapport est le fruit d'un travail e ectué dans le cadre du Ter (Travail d'Etude et de Recherche) de notre Master 1 Oro (Optimisation en Recherche Opérationnelle). Parmi les sujets qui nous étaient proposés, les di érents domaines vus en cours étaient parcourus. Sur notre sélection de trois sujets, il y a eu consensus, au sein de notre groupe, sur des travaux portant sur l'algorithmique ou les graphes. Notre choix premier a été validé pour la proposition portant sur l'identification de protéines par comparaison d'ensembles.

Ce sujet est co-encadré par trois personnes :
- Irena Rusu
- Guillaume Fertin
- Dominique Tessier [1]

Les champs d'études de notre travail portent sur la *protéomique* et l'informatique, même si, nous le verrons plus tard, nous ne nous attarderons pas sur la *protéomique* et les aspects biologiques inhérents qui ne relèvent pas directement de notre champ de compétences. Dans les grandes lignes, notre étude porte sur la mise en place d'un algorithme de comparaison d'ensembles pour identifier des protéines. C'est avant-tout une aide à la décision à destination d'un expert biologiste.

Figure 1.1 – Représentation symbolique d'une protéine

1. Personnel de l'Inra Institut National de la Recherche Agronomique

Dans la suite du rapport, vous trouverez le cheminement de notre réflexion ainsi que le produit de notre travail. Dans un premier temps, nous vous situerons le contexte de ce travail en vous décrivant notre vision et perception du problème. Ensuite, nous vous détaillerons l'algorithme de comparaison que nous avons élaboré en vous expliquant son fonctionnement et les algorithmes que nous avons du implémenter. Enfin, nous analyserons notre algorithme au regard de ses performances (durée d'exécution) et de sa complexité.

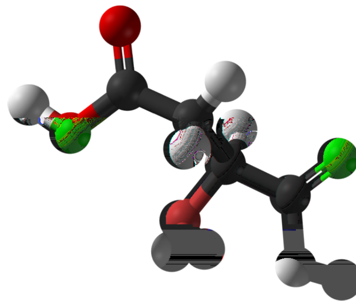


FIGURE 1.2 – Un acide aminé constituant un peptide

Chapitre 2

Présentation du problème

2.1 Contexte

La protéomique est la science qui étudie les *protéomes* (ensemble des protéines contenues dans une cellule). Les techniques d'identifications de protéines se sont développées ces dernières années et celles-ci sont devenues rapides et reproductibles. La spectrométrie de masse, par l'utilisation de gel **2D-E** [1], est devenue une technique courante pour séparer les protéines. L'utilisation de cette technique permet de casser les chaînes de peptides pour mieux les mesurer. Deux méthodes *"historiques"* permettent cette approche :

- **Peptide Mass Fingerprint**
- **Spectrometrie de Masse en Tandem**

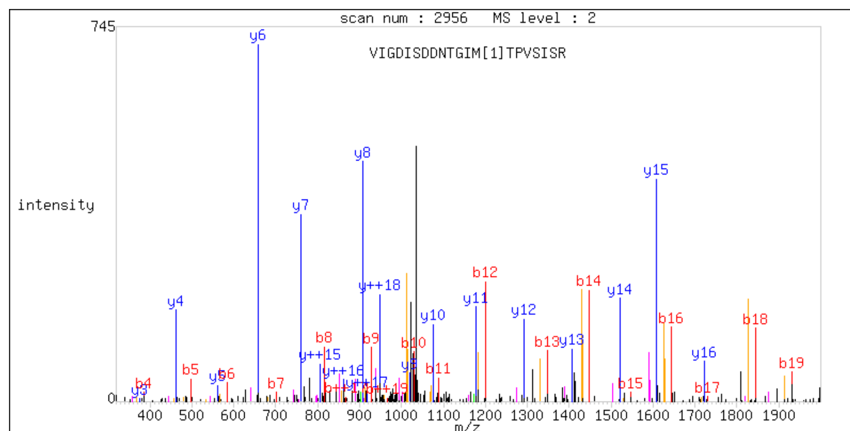


FIGURE 2.1 – EXEMPLE D'UN SPECTRE DE MASSE OBTENU PAR LC-MS/MS

1. C'est l'une des techniques utilisées par l'INRA

La première méthode, Peptide Mass Fingerprint, établit un spectre de masse mesurant le rapport $\frac{masse}{nombre\ de\ charges}$ de la molécule ionisée. La génération de ce spectre par hydrolyse enzymatique, contenant un ensemble de peptides, permet l'identification d'une protéine. La fiabilité de l'identification réside sur le nombre de peptides mesurés et la précision des masses des peptides.

La portée de notre sujet n'ira pas plus loin dans la compréhension de ces techniques d'échantillonnage et d'analyse car elles ne concernent pas le coeur de notre discipline. Cependant, nous allons à présent détailler le passage de la discipline protéomique à informatique.

L'échantillon, sur lequel se porte l'analyse, contient donc un ensemble de peptides identifiés à partir desquels l'expert biologiste souhaite avoir la meilleure connaissance des protéines contenues dans cet échantillon. Le résultat de la Peptide Mass Fingerprintest envoyé à des moteurs de recherche[2], privées ou publiques, qui fournissent un ensemble de protéines supposées couvertes et contenues dans l'échantillon. Ce résultat est associé à un indice de confiance. Le travail de l'expert biologiste est de valider les propositions des moteurs de recherches par confrontation avec les banques de données.

2.2 Problématique

L'expert biologiste utilise un second logiciel qui lui permet d'exploiter les résultats d'identifications. Ce logiciel va présenter de façon astucieuse l'ensemble des propositions de protéines. Il va rendre une hiérarchie d'inclusion entre protéines. Une $\mathcal{P}rot\acute{e}ine_i$ est incluse dans une autre $\mathcal{P}rot\acute{e}ine_j$ si l'ensemble des peptides de $\mathcal{P}rot\acute{e}ine_i$ est également inclus dans celui de $\mathcal{P}rot\acute{e}ine_j$. Cette présentation sous forme d'arbre permet de guider le travail d'identification de l'expert.

Cette analyse est aujourd'hui le rendu d'un algorithme de comparaison qui repose sur un algorithme proche de la *force brute*[3]. Cet algorithme est exécuté à chaque chargement du jeu de données dans le logiciel de présentation des hiérarchies d'inclusions. Le temps nécessaire à l'ouverture et le temps de lancement de cet algorithme est tel qu'il pénalise, aujourd'hui, le travail de l'équipe de recherche. C'est dans ce contexte plus particulier que l'Inra a proposé un travail de design d'un algorithme d'identification de protéines.

Notre problématique est donc de concevoir un algorithme, e cient, de comparaison de protéines en fonction de la liste des peptides permettant d'identifier ces protéines.

2. ***MASCOT*** est un moteur de recherche utilisé par l'Inra
3. L'algorithme effectue toutes les combinaisons possibles

2.3 Abstraction du problème

Notre première étape de recherche fût l'interprétation du problème en langage machine. Nous avons du savoir quelles étaient les variables, les entrées et les sorties de notre algorithme. Il nous a fallu également recenser les fonctions que nous devions créer pour arriver à nos fins. Dans notre cheminement scientifique, nous nous sommes donnés pour exemple des noms formatés de protéines et peptides à savoir $\mathcal{P}rot\acute{e}ine_i$, $\mathcal{P}rot\acute{e}ine_j$, $\mathcal{P}eptide_i$, $\mathcal{P}eptide_j$ ... Mais en réalité, l'appellation des protéines et des peptides se fait par des chaînes de caractères. Nous avons donc utilisé le codage issu des instances de test pour représenter nos protéines et peptides.

Le codage des peptides, tel qu'il est dans les fichiers XML [4], est un identifiant de 40 caractères hexadécimaux. Cette combinaison garantit l'unicité du codage de la peptide au regard de la multitude de peptides existants du monde vivant. Ce codage se trouve dans la balise *ID*.

Il existe cependant un autre identifiant de peptide, que nous trouvons sous la balise *SEQUENCE*. Cette séquence, plus légère de part sa taille, est su sante pour représenter les peptides sur lesquels nous e ectuons notre travail. Nous avons choisi de nous servir de cette séquence de codage qui est composé d'un nombre (nombre qui peut varier d'un peptide à un autre) de caractères codé sur l'alphabet $\mathbb{L}$atin moins les lettres {B,J,X,Z}, soit 22 caractères.

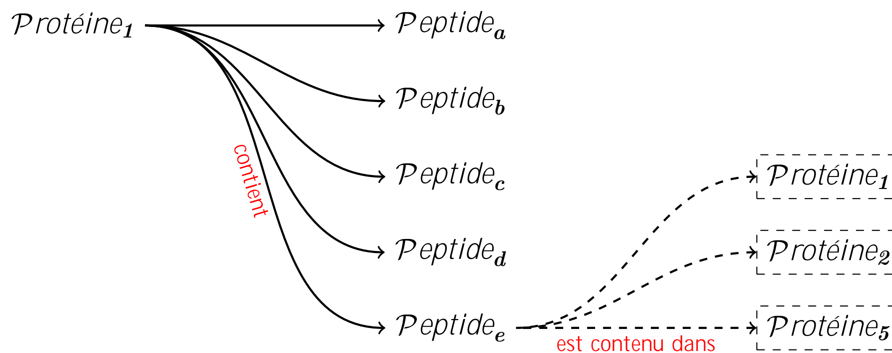
ID	SEQUENCE
f39a2e29125f18bebfea3ba099705264ba6b1efa	AKDFSEGGAVNSQSAR
22ac2ba323db19a5d151f5d13cf49febd5902ca9	LFSAXENR

Table 2.1 – Exemple de codage de peptides

Variables

La compréhension du problème repose sur la compréhension de la protéomique dans sa version la plus simple. Nous devons donc manipuler des protéines et des peptides qui constituent les variables majeures de notre algorithme. Un peptide peut être commun à plusieurs protéines et une protéine peut être incluse dans d'autres protéines. A partir de ce constat, nous savons qu'il y aura non plus une variable mais plutôt un objet (instance de classe) qui représentera le plus fidèlement ces deux éléments.

4. C'est le format du fichier de données issu du moteur de recherche



Entrée

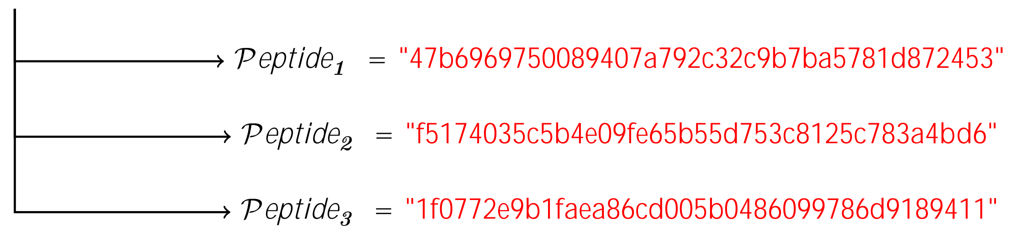
Notre programme doit fonctionner pour les instances d'analyse protéomique fournies par la spectrométrie de masse. Ces instances, normalisées, sont des fichiers XML recensant les protéines et les peptides contenus dans l'échantillon analysé. Les protéines et les peptides sont facilement recensables avec les balises *<PROTEIN>* et *<PEPTIDE>*.

```
...
<PROTEIN ID="C0SQK4" SCORE="154.0">
<SEQUENCE/>
<DESCRIPTION>4425.76,"Glyceraldehyde-3-phosphate dehydrogenase (Fragment)
    OS=Rosa hybrid cultivar GN=RhGAPDH"</DESCRIPTION>
...
<SEQ_MATCH END="181" ID="47b6969750089407a792c32c9b7ba5781d872453" START="
    167">
</SEQ_MATCH>
<SEQ_MATCH END="200" ID="f5174035c5b4e09fe65b55d753c8125c783a4bd6" START="
    194">
</SEQ_MATCH>
<SEQ_MATCH END="214" ID="1f0772e9b1faea86cd005b0486099786d9189411" START="
    201">
</SEQ_MATCH>
</PROTEIN>
<PROTEIN ID="A2XF40" SCORE="98.0">
<SEQUENCE/>
<DESCRIPTION>7585.87,"Putative uncharacterized protein OS=Oryza sativa
    subsp. indica GN=OsI_10982"</DESCRIPTION>
...
```

Listing 2.1 – Extrait d'un fichier XML

Dans l'exemple ci-dessus, nous pouvons voir une *protéine* qui contient 3 *peptides*, ci dessous représentée dans une autre formalisation.

$\mathcal{P}rot\acute{e}ine$ = "C0SQK4"



Sortie

La sortie de ce programme est une organisation par niveaux des protéines. Cette organisation doit permettre de di érencier deux types de protéines :

- Les protéines $\mathcal{P}rot\acute{e}ine_{rang0}$ dont l'identification n'est incluse dans aucune autre
- Les protéines $\mathcal{P}rot\acute{e}ine_{rang1}$ qui sont incluses dans une autre protéine par l'identification des peptides communs

Chapitre 3

Structures de données et algorithmes

Dans cette partie, nous allons vous décrire les pistes de réflexion que nous avons eues lors de notre recherche d'algorithme de comparaisons. Plusieurs directions di érentes se profilaient mais nous avons choisi d'en développer une plus particulièrement par l'exactitude de sa méthode.

Durant la lecture de ce chapitre, vous trouverez des références à des éléments[1] de notre programme en JAVA. L'utilisateur est convié à lire la JAVADOC inhérente à notre application pour obtenir les renseignements qu'il souhaite.

3.1 Solutions envisagées

3.1.1 Comparaison par fréquence

La première chose à laquelle nous avons pensé a été de s'intéresser à la fréquence d'apparition des peptides dans la reconnaissance des protéines. En e et, si une protéine est incluse dans une autre, il est obligatoire d'e ectuer un test sur chacun de ses peptides pour prouver l'inclusion. Pour pallier à la lenteur de cette méthode de test d'inclusion, nous avons chercher à déterminer comment une protéine pouvait ne pas être incluse dans une autre.

L'idée est d'e ectuer un pré-traitement sur notre échantillon pour calculer la fréquence d'apparition d'un peptide qui serait donnée par :

$$\text{fréquence} = \frac{\text{nombre d'apparition de un peptide}}{\text{nombre de peptide}}$$

Il est plus probable de déterminer rapidement qu'une protéine n'est pas incluse dans une autre en comparant, en premier lieu, leurs peptides *"rares"*.

1. Ces éléments seront signalés avec la syntaxe suivante **Elément**

Ce fût notre première piste de travail.

3.1.2 Comparaison par signature

En poussant la réflexion sur la détermination *plus rapide* des non-inclusions et la fréquence d'apparition des peptides, nous avons pensé à a ecter une signature à chaque protéine qui dépendrait de la fréquence de ses peptides.

Pour ce faire, il faut trier les peptides selon leur fréquence calculée au préalable puis classer les peptides en un certain nombre de catégories, catégorie établie en fonction de la rareté.

Soit $\mathcal{C}$ le nombre de catégories, la signature associée à la protéine consiste en $\mathcal{C}$ entiers correspondants. Chaque entier correspond au nombre de peptides dans sa catégorie. Pour déterminer l'inclusion ou non d'une *Protéine*$_A$ dans une *Protéine*$_B$, il su t de comparer les signatures numériques de ces deux protéines, à savoir, comparer les nombres de peptides de chaque catégorie. Si le nombre de peptide de la catégorie i de la *Protéine*$_A$ est supérieur à celui du nombre de peptides de la catégorie i de la *Protéine*$_B$, alors il y a non-inclusion.

Protéine$_A$			*Protéine*$_B$		Inclusion
Catégorie	Quantité		Catégorie	Quantité	
A	2	=	A	2	?
B	3	>	B	1	X
C	4	<	C	6	?

Figure 3.1 – Exemple de non-inclusion

Dans notre figure 3.1, nous pouvons remarquer que, au regard de la catégorie A, le nombre de peptides de *Protéine*$_A$ est supérieur à celui de de *Protéine*$_B$. La *Protéine*$_A$ ne pourra donc pas être incluse dans la protéine *Protéine*$_B$.

Afin d'optimiser cette méthode, les catégories de peptides sont définies selon une échelle logarithmique, c'est à dire que la catégorie de peptides les plus rares concernera très peu de peptides alors que la catégorie des peptides *"communs"* contiendra un grand nombre de peptides.

3.1.3 Comparaison par intersection des contenants

Le principe développé ici repose sur le fait suivant : trouver les protéines dans lesquelles une *Protéine*$_A$ donnée est incluse veut dire trouver les *Protéines*$_i$ qui contiennent tous les peptides de *Protéine*$_A$. L'idée est donc d'e ectuer un prétraitement pour a ecter à chaque peptide la liste des protéines qui le contiennent.

Puis pour chaque protéine, nous calculons l'intersection des protéines en fonction de celles comprises dans chacun des peptides. Le résultat nous donne les $\mathcal{P}roteine_j$ dans lesquelles la $\mathcal{P}roteine_A$ est incluse.

3.2 Solution retenue

Nous avons concentré notre travail de recherche et d'élaboration d'algorithme sur la dernière solution envisagée, à savoir, `comparaison par intersection des contenants`.

3.2.1 Définition des Objets

Dans cette partie, nous vous détaillons les deux principales classes Objet de notre programme pour vous faciliter la lecture des algorithmes qui suivront.

Objet `Proteine`

Nous vous détaillons, ci-dessous, les attributs de notre objet `Proteine` tels que nous les avons définis dans notre programme.

- `id` L'identifiant, une chaîne de caractère, de cette protéine
- `mesPeptides` Une liste d'objets `Peptide` . C'est la liste de toutes les peptides de cette protéine.
- `filleDe` C'est une liste, vide à l'initialisation, d'objets `Proteine` . Elle contient, par la suite, l'ensemble des protéines dans lesquelles celle-ci est incluse.
- `mereDe` Une liste, vide à l'initialisation, d'objets `Proteine` . Elle contiendra l'ensemble des protéines qui sont incluses dans celle-ci.

Objet `Peptide`

Nous vous détaillons, ci-dessous, les attributs de notre objet `Peptide` tels que nous les avons définis dans notre programme.

- `seq` La séquence de notre peptide. Un codage spécifique plus succinct que l'identifiant.
- `id` L'identifiant, une chaîne de 40 caractères, de cette peptide
- `mesProteines` Une liste d'objets `Proteine` . C'est la liste de toutes les protéines qui contiennent cette peptide.

3.2.2 Fonctions

En fonction des variables, des entrées et des sorties, nous avons imaginé plusieurs fonctions qui nous semblent nécessaires à l'exécution de ce programme. Ces fonctions ont un rôle particulier dans le déroulement de l'algorithme et doivent interagir ensemble. Nous vous présentons les fonctions telle que nous les avons imaginées lors du brainstorming de notre projet.

Decoupage Cette fonction permet de traiter le fichier d'instance en extrayant les informations nécessaires à notre algorithme à savoir les peptides et les protéines

Remplissage La fonction de remplissage permet d'a ecter les protéines aux peptides. En e et, le schéma normal consiste à a ecter des peptides à chaque protéine. Mais dans notre algorithme, l'information de savoir quelle protéine contient tel peptide est importante.

Inclusion Une fonction pour trouver les protéines incluses dans les autres en fonction de la comparaison de leur ensemble de peptides.

Appartenance Une fonction qui permet de créer la relation mère $\leftrightarrow$ fille entre deux protéines.

Des fonctions auxiliaires sont présentes dans le programme mais n'ont pas d'utilité dans l'algorithme. Elles nous servent à vérifier que notre algorithme ne fait pas d'erreur. Nous vous les indiquons à titre indicatif :

Output Cette fonction permet de générer un fichier Output hiérarchisant les protéines.

verifierInclusion Plus exactement une méthode de l'objet `Proteine` , elle permet de vérifier que les inclusions sont valides en e ectuant le contrôle des peptides communs entre chaque inclusion.

3.2.3 Algorithme d'ajout dans une liste triée

Afin de pouvoir appliquer l'algorithme d'intersection décrit après, nous avons besoin, en pré-requis, que les listes dont nous cherchons à faire l'intersection soient triées. Etant donné que nous construisons ces listes à partir de leur initialisation, il nous su t d'utiliser la procédure qui permet d'ajouter les éléments dans une liste triée. Voici la procédure que nous utilisons :

Algorithm 1: ALGORITHME D'AJOUT DANS UNE LISTE TRIÉE

```
input  : un peptide, une proteine
output: ce peptide ajouté à la liste triée peptide.pro[]
1  début
2    si peptide.pro.taille ≠ 0 alors
3      entier inf ← 0 ;
4      entier sup ← peptide.pro.taille-1 ;
5      entier indice ;
       /* Nous vérifions si la protéine est au début ou à la fin   */
6      si proteine ≤ peptide.pro[inf ] alors
7        ajouter (peptide.pro, proteine, inf);
         /* La procédure ajouter(liste,élément,indice) ajoute l'
            élément à la liste à l'indice donné et décale le reste
            vers la droite                                          */
8      sinon si proteine ≥ peptide.pro[sup ] alors
9        ajouter (peptide.pro,proteine,sup +1);
10     sinon
         /* Nous effectuons une recherche dichotomique pour trouver
            les deux éléments entre lesquels insérer le nouvel      */
11       répéter
12         indice = |inf+sup| / 2 ;
13         si proteine ≥ peptide.pro[indice ] alors
14           inf = indice ;
15         fin
16         si proteine ≤ peptide.pro[indice ] alors
17           sup = indice ;
18         fin
19       jusqu'à |sup − inf| < 1;
20       ajouter (peptide.pro,proteine,sup);
21     fin
22   sinon
       /* si la liste est vide, on ajoute simplement le protéine a la
          liste.                                                     */
23     ajouter (peptide.pro,proteine);
24   fin
25 fin
```

Nous utilisons une fonction similaire, dans la classe **Decoupage** au moment de la création de la liste globale des peptides. Cette liste est de très grande taille. Néanmoins, cette fonction nous a permis d'obtenir un gain de temps non-négligeable.
L'accès à cette liste se fait également par recherche dichotomique. L'utilisation de cette fonction pour le parsing nous a permis de faire passer le temps d'exécution de la fonction

Découpage d'environ 40 secondes à moins d'une seconde.

3.2.4 Algorithme d'intersection

L'algorithme naïf d'intersection de deux listes $\{\mathbf{L}_1, \mathbf{L}_2\}$ consiste à prendre chaque élément de la première liste $\mathbf{L}_1$ pour regarder s'il est présent dans la deuxième liste $\mathbf{L}_2$. Cet algorithme ne nous satisfaisant pas, nous avons décidé de l'optimiser.

Comme nos listes sont triées (grâce à l'élément **ArrayList**),nous pouvons utiliser l'algorithme suivant qui donne non seulement l'intersection de deux listes, mais nous garantie que cette intersection sera également triée de manière à pouvoir appliquer une nouvelle fois cet algorithme.

Nous ne chercherons pas à prouver la correction de cet algorithme et nous considérons que la liste retournée est effectivement bien triée.

Algorithm 2: ALGORITHME D'INTERSECTION

input : liste1 [],liste2 [] : deux listes triées d'éléments comparables
output: l'intersection de ces deux listes inter[]

```
1  début
2      entier i ← 0 ;
3      entier j ← 0 ;
4      inter ← liste vide ;
5      tant que i <liste1.taille and j <liste2.taille faire
6          si liste1 [i] < liste2 [j] alors
7              i ← i + 1 ;
8          fin
9          si liste1 [i] > liste2 [j] alors
10             j ← j + 1 ;
11         fin
12         si liste1 [i] = liste2 [j] alors
13             i ← i + 1 ;
14             j ← j + 1 ;
15             ajouter (inter,liste1 [i]);
16         fin
17     fin
18 fin
```

3.2.5 Algorithme d'Inclusion

L'algorithme d'**Inclusion** est un algorithme d'Intersection d'ensemble de toutes les protéines dans lesquelles l'ensemble de ses peptides est commun. Cet algorithme fait appel à l'algorithme d'intersection développé ci-dessus.

A la fin de l'appel de l'algorithme d'intersection, nous prenons le soin d'enlever la protéine traitée *(non précisé dans le pseudo-code)*.

Algorithm 3: ALGORITHME D'INCLUSION

input : une proteine
output: un attribut liste filleDe pour chaque proteine qui contiendra les protéines dans lesquelles elle est incluse

```
1  début
       /* On regarde si la proteine contient des peptides                */
2      si prot.pep[].taille ≠ ∅ alors
3          filleDe = prot.pep[0].prots[]-{this};
4          entier i ← 1 ;
5          si prot.pep[].taille > 1 alors
               /* On ajoute toutes les protéines de cette peptide à la
                  liste filleDe, sauf cette protéine                     */
6              tant que i < prot.pep[].taille and filleDe ≠ ∅ faire
7                  filleDe = intersection(filleDe,pep[i].prots[]);
8                  i = i + 1 ;
9              fin
10         fin
11     fin
12 fin
```

3.2.6 Algorithme d'Appartenance

Cette méthode de la classe **Proteine** permet d'ajouter à la liste *mereDe* de chaque protéine l'ensemble des protéines incluses dans celle-ci.

Algorithm 4: Algorithme d'Appartenance

```
   input: une proteine prot*
1  début
2  |  pour chaque proteine pro de prot*.filleDe faire
   |  |  /* Nous ajoutons cette protéine à la liste mereDe des
   |  |     protéines de la liste filleDe                          */
3  |  |  pro.mereDe.ajouter (prot*);
4  |  fin
5  fin
```

3.2.7 Algorithme Principal

Avec l'ensemble des algorithmes que nous avons détaillés précédemment, il ne nous reste plus qu'à les appliquer.

Cet algorithme est en deux parties :

1. Le Prétraitement qui consiste à lister pour chaque peptide les protéines dans lesquelles elles apparaissent. C'est le travail de la fonction **Remplissage**.
2. Le Traitement qui correspond au calcul, pour chaque $\mathcal{P}rot\acute{e}ine_i$ l'intersection des listes de protéines de tous ses peptides. C'est le travail de la fonction **Inclusion**. Cette intersection correspond à la liste des protéines dans lesquelles $\mathcal{P}rot\acute{e}ine_i$ est incluse.
3. La fonction **Appartenance** pour avoir la relation mère↔fille dans le bon sens.

Algorithm 5: ALGORITHME PRINCIPAL

input : une liste de proteine prot[] avec pour chacune leur liste de peptide prot.pep[]
output: un attribut liste included pour chaque proteine qui contiendra les protéines dans lesquelles elle est incluse

```
1  début
       /* On assigne à chaque peptide la liste triée des protéines qui
          le contiennent                                              */
2      pour chaque proteine pro de prot[] faire
3          pour chaque peptide pep de prot.pep[] faire
4              ajouter (pep,pro);
5          fin
6      fin
7      pour chaque proteine pro de prot[] faire
8          pro.inclusion();
9          pro.appartenance();
10     fin
11 fin
```

3.3 Applications sur des exemples

Dans cette partie, nous allons vous détailler le principe de fonctionnement de notre algorithme sur un exemple.

$$
\begin{array}{lcllllllll}
\mathcal{P}rot\acute{e}ine_1 & : & [& a & b & c & d & e & &] \\
\mathcal{P}rot\acute{e}ine_2 & : & [& a & b & & & e & f &] \\
\mathcal{P}rot\acute{e}ine_3 & : & [& a & & c & d & & &] \\
\mathcal{P}rot\acute{e}ine_4 & : & [& a & & c & & & &] \\
\mathcal{P}rot\acute{e}ine_5 & : & [& a & & & & e & &] \\
\mathcal{P}rot\acute{e}ine_6 & : & [& & b & c & d & & &]
\end{array}
$$

FIGURE 3.2 – EXEMPLE

Dans la figure 3.2, vous trouvez un ensemble de 6 protéines comprenant chacune des peptides. Par exemple, la $\mathcal{P}rot\acute{e}ine_5$ contient les peptides $\{a, e\}$.

3.3.1 Remplissage des peptides

En exécutant la procédure **Remplissage**, les peptides se voient a ecter des protéines pour lesquelles ils sont inclus.

$$
\begin{array}{lcllllllll}
\mathcal{P}rot\acute{e}ine_1 & : & [& a & b & c & d & e & &] \\
\mathcal{P}rot\acute{e}ine_2 & : & [& a & b & & & e & f &] \\
\mathcal{P}rot\acute{e}ine_3 & : & [& a & & c & d & & &] \\
\mathcal{P}rot\acute{e}ine_4 & : & [& a & & c & & & &] \\
\mathcal{P}rot\acute{e}ine_5 & : & [& a & & & & e & &] \\
\mathcal{P}rot\acute{e}ine_6 & : & [& & b & c & d & & &]
\end{array}
\quad
\begin{array}{lcllllllll}
\mathcal{P}eptide_a & \in & [& 1 & 2 & 3 & 4 & 5 & &] \\
\mathcal{P}eptide_b & \in & [& 1 & 2 & & & & 6 &] \\
\mathcal{P}eptide_c & \in & [& 1 & & 3 & 4 & & 6 &] \\
\mathcal{P}eptide_d & \in & [& 1 & & 3 & & & 6 &] \\
\mathcal{P}eptide_e & \in & [& 1 & 2 & & & 5 & &] \\
\mathcal{P}eptide_f & \in & [& & 2 & & & & &]
\end{array}
$$

FIGURE 3.3 – EXEMPLE APRÈS REMPLISSAGE

La figure 3.3 nous montre le remplissage de l'attribut **mesProteines** après l'exécution de la méthode. A la lecture de ce tableau, nous nous apercevons que la $\mathcal{P}eptide_b$ est incluse dans les protéines

- $\mathcal{P}rot\acute{e}ine_1$
- $\mathcal{P}rot\acute{e}ine_2$
- $\mathcal{P}rot\acute{e}ine_6$

3.3.2 Exemple d'inclusion sur la $\mathcal{P}rot\acute{e}ine_3$

$\mathcal{P}rotéine_3$	:	[	a		c	d			]
$\mathcal{P}eptide_a$	$\in$	[	1	2	3	4	5		]
$\mathcal{P}eptide_c$	$\in$	[	1		3	4		6	]
$\mathcal{P}eptide_d$	$\in$	[	1		3			6	]

FIGURE 3.4 – DESCRIPTION DE LA $\mathcal{P}rotéine_3$

Première Etape

A l'initialisation de notre inclusion, la liste d'intersection **inter** contient les protéines du premier peptide moins la protéine dans laquelle nous nous trouvons. Dans notre exemple, **inter** contient

$$\{1, 2, 3, 4, 5\} - \{3\} \text{ soit } \{1, 2, 4, 5\}$$

inter =	[	1	2	~~3~~	4	5	]

FIGURE 3.5 – PREMIÈRE ETAPE

Deuxième Etape

inter =	[	1	2		4	5		]
$\cap$ $\mathcal{P}eptide_c$ =	[	1		3	4		6	]
inter =	[	1	~~2~~		4	~~5~~		]

FIGURE 3.6 – INTERSECTION AVEC LES PROTÉINES DE LA $\mathcal{P}eptide_c$

Troisième Etape

inter =	[	1		4		]
$\cap$ $\mathcal{P}$***eptide***$_d$ =	[	1	3		6	]
inter =	[	1		~~4~~		]

FIGURE 3.7 – INTERSECTION AVEC LES PROTÉINES DE LA $\mathcal{P}eptide_d$

Résultat

Finalement, nous avons la liste **inter** qui vaut ce qui est présenté dans la figure 3.8. L'interprétation dans notre problème revient à dire que $\mathcal{P}rot\acute{e}ine_3$ est incluse dans $\mathcal{P}rot\acute{e}ine_1$.

inter = [1]

FIGURE 3.8 – LISTE **inter** À LA FIN

Chapitre 4

Analyse et tests

Dans cette partie, nous allons caractériser notre programme du point de vue de sa complexité, dans un premier temps, et du point de vue de sa réaction, dans un second temps.

4.1 Complexité

Nous nous intéressons à la complexité de certaines fonctions de notre algorithme. Nous avons exclues volontairement les fonctions auxiliaires de notre analyse (**Output**, **verifierInclusion**) ainsi que la fonction **Decoupage**. Ce choix arbitraire nous a paru cohérent puisque ces fonctions ne concernent pas directement l'algorithme et seront ré-implémentées par l'équipe de recherche dans l'intégration logicielle.

Nous nous intéressons, ici, à la complexité *au pire* des algorithmes décrits. Pour cela, nous définissons quelques paramètres :

- $\mathcal{N}$: Le nombre de protéines de l'instance
- $\mathcal{M}$: Le nombre de peptides de l'instance
- $\mathcal{P}ep_{max}$: Le nombre de peptides maximum par protéine
- $\mathcal{P}ro_{max}$: Le nombre de protéines maximum présentes par peptide

Egalement, nous avons considéré que les comparaisons entre deux objets (peptides, protéines) sont en $\mathcal{O}(1)$

4.1.1 Complexité du remplissage

Cette méthode est une double boucle faisant appel à une fonction d'ajout.

La boucle principale fait exactement $\mathcal{N}$ opérations et la boucle imbriquée fait au maximum $\mathcal{P}ep_{max}$ itérations.

La fonction d'ajout appelée est une fonction d'ajout dans une liste triée dont la taille est *au pire*, le nombre maximum de protéines dans laquelle est comprise un peptide, soit

$\mathcal{P}ro_{max}$. Cette fonction a été décrite dans l'algorithme 1. L'ajout se faisant de manière dichotomique, sa complexité est clairement en $\mathcal{O}(log(\mathcal{P}ro_{max})$.

Ce qui nous donne une complexité *au pire* pour la fonction **Remplissage** en

$$\mathcal{O}(\mathcal{N} \times \mathcal{P}ep_{max} \times log(\mathcal{P}ro_{max}))$$

4.1.2 Complexité de l'algorithme d'intersection

Cet algorithme met en jeu deux itérateurs sur une liste et s'arrête quand l'un des deux à atteint le bout de sa liste respective. A chaque étape, au moins un des deux itérateurs est incrémenté et un seul test est e ectué. Les itérateurs n'étant jamais décrémentés, nous pouvons a rmer qu'*au pire*, les deux listes seront parcourues en entier, c'est à dire que tous les éléments seront visités.

La longueur de ces listes étant *au pire* $\mathcal{P}ro_{max}$, nous avons au maximum $2 \times \mathcal{P}ro_{max}$ itérations.

Cet algorithme a donc une complexité, *au pire*, en

$$\mathcal{O}(\mathcal{P}ro_{max})$$

4.1.3 Complexité de l'inclusion et l'appartenance

Une approximation, *au pire*, de la complexité du calcul des inclusions pour une protéine est la complexité de l'algorithme d'intersection multiplié par son nombre de peptides ($\mathcal{P}ep_{max}$ dans le pire des cas).

Ceci nous donne une complexité de

$$\mathcal{O}(\mathcal{P}ro_{max} \times \mathcal{P}ep_{max})$$

Pour la complexité de l'appartenance, nous e ectuons un changement mère$\leftrightarrow$fille sur, *au pire* $\mathcal{P}ro_{max}$ protéines.

4.1.4 Complexité de notre algorithme

L'opération étant répétée pour toutes les protéines, nous devons multiplier cette complexité par $\mathcal{N}$. Cela nous donne, au total, pour le remplissage puis le calcul des inclusions, une complexité théorique *au pire* en

$$\mathcal{O}(\mathcal{N} \times \mathcal{P}ep_{max} \times log(\mathcal{P}ro_{max}) + \mathcal{N} \times (\mathcal{P}ro_{max} \times \mathcal{P}ep_{max} + \mathcal{P}ro_{max})$$

Or, se basant sur cette propriété

$$\mathcal{O}(x \times log(x)) < \mathcal{O}(x^2)$$

notre algorithme est en complexité, *au pire*, en

$$\mathcal{O}(\mathcal{N} \times \mathcal{P}ro_{max} \times \mathcal{P}ep_{max})$$

Dans le pire des cas, nous avons

$$\mathcal{P}ep_{max} \approx \mathcal{M}$$

$$\mathcal{P}ro_{max} \approx \mathcal{N}$$

ce qui donne pour la complexité

$$\mathcal{O}(\mathcal{N}^2 \times \mathcal{M})$$

Ce cas de figures n'arrivera jamais dans la réalité, il correspond à ce que toutes les protéines contiennent tous les peptides.

4.2 Tests

L'Inra nous a transmis des instances de test réelles sur lesquelles nous avons testé notre logiciel. Les résultats semblent convaincants après confrontation avec l'existant. En e et, les temps de calculs étaient "proches" de la minute sur les instances existantes.

A la lecture du tableau, ci-contre, vous observerez que nos temps de calcul sont de l'ordre de la seconde sur l'ensemble de notre logiciel. Ceci ramené au simple traitement, nous somme de l'ordre de la centaine de millisecondes.

Nombre de protéines		6 816	4 925	13 026	7 703
Nombre de peptides		11 127	10 013	38 768	11 234
Nombre d'inclusions		840 690	563 857	798 075	1 165 587
Partie de l'algorithme		Temps d'exécution en millisecondes			
Parsing					
	Decoupage	236	193	643	245
Traitement		**168**	**98**	**226**	**378**
	Remplissage	33	18	59	37
	Inclusions	135	80	167	341
Sorties					
	Génération Output	406	300	548	529
Total		810	591	1417	1152

Table 4.1 – Résultats des tests

4.2.1 Comparaison par rapport à la fréquence

Nous avons implémenté un autre algorithme où nous nous appuyons sur la taille et la fréquence des peptides pour calculer les inclusions. Les résultats vous sont exposés dans le tableau ci-dessous. Nous remarquons qu'il y a peu de di érences (négligeables) par rapport à notre première version. Nous restons dans les mêmes ordres de grandeur de temps de calcul par rapport aux instances qui nous ont été fournies.

Nombre de protéines	6 816	4 925	13 026	7 703
Nombre de peptides	11 127	10 013	38 768	11 234
Nombre d'inclusions	840 690	563 857	798 075	1 165 587
Partie de l'algorithme	Temps d'exécution en millisecondes			
Parsing				
Decoupage	235	197	630	224
Traitement	**184**	**127**	**221**	**374**
Remplissage	32	18	55	37
Inclusions	152	109	166	337
Sorties				
Génération Output	418	294	5415	531
Total	877	652	1420	1190

TABLE 4.2 – Résultats des tests par fréquence

Chapitre 5

Conclusion

Malgré une complexité théorique *au pire* relativement mauvaise (quadratique), nous obtenons, en pratique, des résultats assez satisfaisants. Ceci s'explique par le fait que la variété des peptides et protéines, dans un échantillon d'analyse, est telle qu'il est rare, et peu probable, que toutes les protéines contiennent exactement tous les peptides.

Le résultat, qui nous a été demandé, est une arborescence à deux niveaux où les protéines *filles* du deuxième niveau doivent être incluses dans une et une seul protéine *mère*. Or notre algorithme calcule toutes les inclusions possibles. A partir de ce constat, tous les niveaux possibles sont imaginables et calculables.

Nous avons essayé de réfléchir à un moyen de gagner plus de temps de calcul en faisant des coupes pour calculer uniquement les informations utiles. Cependant, nous n'avons pas trouvé de solution concordante avec notre algorithme. En revanche, il existe surement d'autres méthodes de calcul mais qui reviendraient à utiliser un autre algorithme di érent de celui que nous avons implémenté.

Sur les instances qui nous ont été fournies, notre algorithme réagit correctement en espace temps et espace mémoire. Nous ne pouvons pas prédire la réaction de notre algorithme sur des instances plus conséquentes en terme de nombre de protéines, nombre de peptides et inclusions. Pour finaliser notre travail de recherche, il nous faudrait explorer cette phase de tests de performance.