UNIVERSITÉ PARIS-EST MARNE-LA-VALLÉE

INSTITUT GASPARD MONGE

MÉMOIRE DE MASTER RECHERCHE

présenté par

Florian SIKORA

sur **La Comparaison de Réseaux d'intéractions Protéiques**

Soutenue publiquement le 05/09/2008

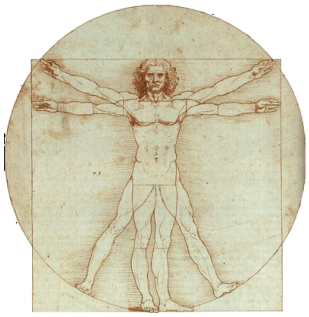
Encadrants :

Guillaume BLIN

Stéphane VIALETTE

Table des matières

Introduction



La seconde année du Master Recherche Informatique de l'université de Marne-la-Vallée propose un stage à effectuer lors du second semestre au sein d'un laboratoire d'informatique, ceci afin de commencer à intégrer une démarche de chercheur.

Le sujet qui m'a été proposé porte sur l'étude algorithmique des réseaux d'interactions protéiques, thème relativement récent mais en plein essor dans la bio-informatique moderne. J'ai choisi cette étude car elle correspond parfaitement à l'orientation que je souhaite donner à mon Master : effectuer de la recherche et l'appliquer à une autre discipline. De plus, ce sujet se place dans la continuité de la filière que j'ai choisie au cours de cette année, l'algorithmique, qui est mon domaine de prédilection en informatique.

Ce stage se déroule au sein du laboratoire d'informatique de l'Institut Gaspard Monge (IGM) [1], un des 23 laboratoires de l'Université de Marne-la-Vallée [2]. Il est encadré par Messieurs Guillaume Blin et Stéphane Vialette, respectivement Maître de Conférences et Chargé de Recherche dans ce laboratoire.

Contrairement à ce qui était prédit il y a quelques années, l'achèvement du séquencage du génome humain montre que l'homme n'aurait qu'environ 25 000 gènes[3] (portion d'une séquence d'ADN), soit moins que la souris domestique (30 000) ou le peuplier (45 000). Cette constatation a amené les chercheurs à se demander si la complexité de l'homme ne résiderait pas en partie dans les protéines. Le nombre et les fonctions biologiques de la plupart de ces molécules codées par les gènes sont encore méconnues, d'où l'intérêt de leur étude, la *protéomique*. Le nombre d'interactions existant entre ces protéines est considérable et forme des réseaux complexes : leur exploration nécessite des méthodologies adaptées ainsi que des outils bio-informatiques puissants pour les analyser. C'est dans cette optique que ce stage m'a été proposé.

Nous introduirons les quelques notions d'algorithmique et de biologie utiles à la compréhension de ce rapport (Chapitres 1 et 2) pour ensuite présenter les problèmes étudiés durant ce stage (Chapitre 3) afin de faire un état de l'art sur le sujet (Chapitre 4). Ensuite, nous présentons notre contribution, la recherche approchée de motifs au sein d'un réseau, en proposant une alternative à l'existant lorsque le motif est un graphe, dans le Chapitre 5.

[1] http://www-igm.univ-mlv.fr/LabInfo/

[2] http://www.univ-mlv.fr/

[3] d'après le site du génoscope http://www.genoscope.cns.fr/spip/Le-projet-Genome-humain.html?artsuite=2#FAQ3

Chapitre 1

Algorithmique et complexité

Contenu

Ce chapitre donne une brève introduction aux notions et notations algorithmiques utilisées par la suite dans ce rapport. Nous décrirons les structures combinatoires utilisées, puis nous introduirons les notions de *problème*, d'*algorithme* et de *complexité*. Nous verrons également la notion de NP-Complétude ainsi que les méthodes utilisées pour la contourner.

1.1 Les graphes

Avant de définir formellement les notions d'algorithmique et de complexité, nous présentons une courte introduction sur la notion de graphe. Plus d'informations sur cette dernière pourront être trouvées dans de nombreux ouvrages, dont [12, 34, 36].

Le sujet de ce rapport porte essentiellement sur les réseaux biologiques, plus précisément les réseaux d'interactions entre les protéines. Par la suite, nous adoptons le terme de "réseau" lorsque l'on se rapporte aux aspects biologiques et celui de "graphe" lorsque l'on se rapporte à l'algorithmique.

On définit un *graphe* par un couple $G = (V, E)$ où V (pour *Vertex*) est l'ensemble de ses sommets (ou noeuds) et E (pour *Edges*) est l'ensemble de ses arcs. Dans le cadre de l'étude des réseaux biologiques, ceux-ci sont représentés par des graphes où les noeuds symbolisent les protéines et les arcs les interactions entre les protéines. Un graphe pourra être *orienté* (les arcs sont des flèches, menant d'un noeud à un autre) ou *non-orienté*.

Un *sous-graphe* $G' = (V', E')$ d'un graphe $G = (V, E)$ est un graphe dont les sommets et les arcs forment respectivement un sous-ensemble $V' \subseteq V$ et $E' \subseteq E$, tel que E' est défini sur $V' \times V'$.

Dans certains cas, un graphe peut être *pondéré*, c'est à dire qu'il existe une fonction $w : E \to \mathbb{R}$ qui assigne à tout arc e de E un réel $w(e)$ symbolisant son poids. On peut également définir un poids sur les sommets du graphe. Dans le cadre des réseaux biologiques, ce poids pourra donner, par exemple, la probabilité de l'interaction entre deux protéines, en considérant une fonction $w : E \to \mathbb{R}^+$.

Un *chemin* de longueur l dans un graphe est une séquence $u_1, ..., u_l$ de l sommets telle que deux sommets successifs soient adjacents (i.e. reliés par un arc). Un chemin sera appelé *simple* si ses sommets sont tous différents (i.e. sans boucle).

Un *arbre* $T = (V, E)$ est un graphe connexe (i.e. il existe pour tout couple (v, v') de sommets de V un chemin menant de v à v') sans cycle (i.e. sans chemin simple de longueur non nulle menant d'un sommet à lui même). Une *forêt* est un graphe acyclique (une forêt est donc un ensemble d'arbres disjoints). Un *arbre couvrant* de G est un sous-graphe connexe et sans cycle d'un graphe G.

1.2 Problèmes et algorithmes

FIG. 1.1 – Muhammad Al-Khwarizmi

La théorie de la complexité est l'étude de l'efficacité des algorithmes ainsi que de la difficulté inhérente aux problèmes. Un *problème* est formalisé de la manière suivante : étant donné un ensemble de données $\mathcal{D}$ en entrée, un problème est une question posée sur $\mathcal{D}$ et dont la réponse peut éventuellement demander des calculs. Il existe généralement deux types de problèmes : les *problèmes de décision* où la réponse est "oui" ou "non", et les *problèmes d'optimisation*. La résolution d'un problème s'effectue par la définition d'un *algorithme*, c'est à dire une suite d'opérations à effectuer. C'est un terme latinisé du nom du mathématicien persan *Al-Khwarizmi* (783-850) (voir Figure 1.1). Voyons quels sont les critères pour classer ces algorithmes et problèmes, et introduisons les notions d'efficacité et de complexité.

1.2.1 Classification d'algorithmes

On différencie deux types d'algorithmes :
- les algorithmes *déterministes*, qui pour une entrée fixée effectueront toujours la même suite d'opérations.
- les algorithmes *non-déterministes*, qui pour une entrée fixée peuvent exécuter des suites d'opérations différentes, car des choix aléatoires sont effectués.

Pour étudier l'efficacité d'un algorithme, on s'intéresse principalement à deux critères :
- sa durée d'exécution. Plus l'algorithme s'exécute rapidement, plus il est considéré comme efficace.
- l'espace mémoire utilisé. Moins l'algorithme occupera de mémoire, plus il est considéré comme efficace.

Il est à noter que ces deux critères sont souvent antinomiques : on optimise généralement le temps d'exécution par de l'utilisation mémoire supplémentaire (et inversement).

Définition 1 (Complexité d'un algorithme)**.** *La complexité d'un algorithme (en temps) est la mesure d'un nombre d'opérations atomiques (i.e. indivisibles) effectuées sur les entrées du problème. Elle s'exprime généralement en fonction de la taille de l'entrée du problème.*

On observe trois types de complexité :

1. *complexité au mieux* : nombre minimal d'opérations à effectuer pour toute instance du problème. Cette complexité peut être atteinte, mais elle ne reflète pas le comportement usuel de l'algorithme.
2. *complexité au pire* : nombre maximal d'opérations pouvant être effectuées pour toute instance du problème (i.e. pire cas). Si ce n'est pas précisé, par la suite on se référera par défaut à la complexité au pire lorsqu'on utilisera le terme de complexité.
3. *complexité en moyenne* : espérance mathématique des complexités de l'algorithme. Elle reflète le comportement en moyenne de l'algorithme. Cette complexité est non-triviale à calculer.

Définition 2 (Efficacité)**.** *Soient deux algorithmes A et B. On dit que A est plus efficace que B si sa complexité est inférieure à celle de B*

Lors de l'étude de la complexité d'un algorithme, on recherche un ordre de grandeur plutôt que le nombre exact d'opérations. Ainsi, on observe la rapidité de croissance de la complexité d'un algorithme en fonction de la taille de ses instances. On utilise pour ce faire la *notation de Landau*, et plus précisément la notation O. On retiendra principalement que si $f = O(g)$, alors g *domine* f. Pour plus de détails sur cette notation, se reporter au Chapitre 9 de [23].

Principaux types de complexité En algorithmique, les principaux types de complexité rencontrés sont les suivants, avec n étant la taille de l'instance :

- la *complexité constante*, notée $O(1)$. Quelque soit la taille de l'instance, il y a un nombre constant d'opérations.
- la *complexité logarithmique*, notée $O(\log(n))$, pour laquelle la complexité croît légèrement avec n. On rencontre généralement ce type de complexité lorsqu'on divise la taille de l'instance à chaque itération.
- la *complexité linéaire*, notée $O(n)$, correspond au cas d'un algorithme comportant une boucle effectuant n itérations, et où le corps de la boucle effectue un travail d'une durée indépendante de la taille de l'instance.
- la *complexité polynomiale*, notée $O(n^c)$, avec c étant une constante.
- la *complexité exponentielle*, notée $O(2^n)$.
- la *complexité factorielle*, notée $O(n!)$.

La Figure 1.2 illustre la croissance de ces fonctions.

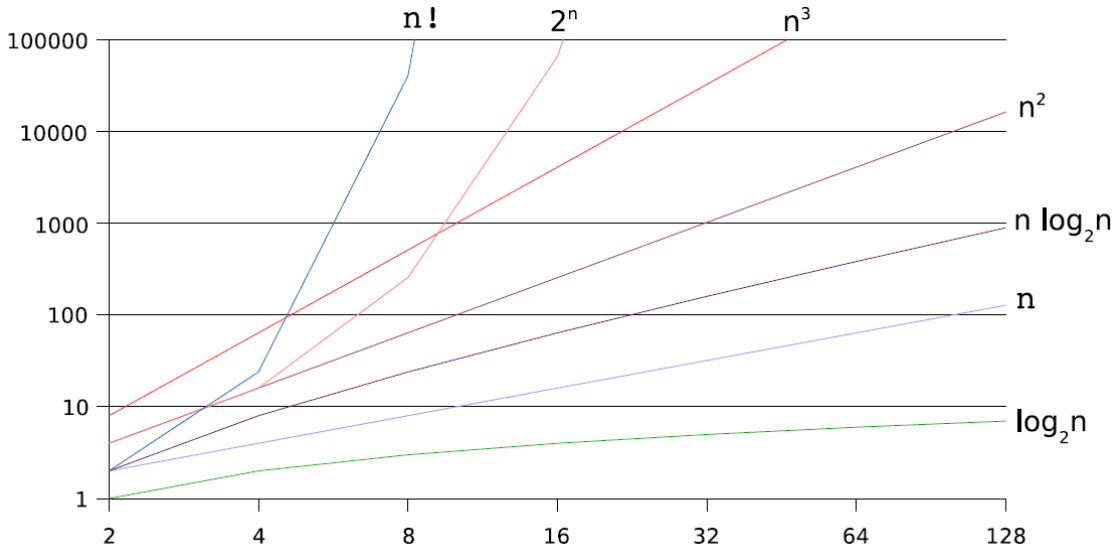


FIG. 1.2 – Rapidité de croissance de fonctions usuelles. Attention, l'échelle de l'axe des ordonnées est logarithmique.

L'intérêt d'un algorithme polynomial est qu'il est considéré comme "efficace en pratique". Ainsi, l'augmentation des performances technologiques a un effet multiplicatif sur la taille des instances que l'algorithme peut résoudre, là où elle a un effet additif pour les algorithmes exponentiels.

1.2.2 Classification de problèmes

La classification usuelle des problèmes se fait en considérant leurs complexités. Celle-ci correspond à la complexité de l'algorithme le plus efficace pour résoudre ce problème. Un tel algorithme peut ne pas être connu.

Classes P et NP

La classe P contient tous les problèmes pouvant être résolus par un algorithme déterministe, en un temps polynomial. La classe NP contient tous les problèmes pour lesquels, étant donné une instance du problème, on peut déterminer en un temps polynomial, si la réponse à cette instance particulière est "oui". Par définition, les problèmes de P se trouvent également dans NP. Intuitivement, les problèmes dans NP peuvent être résolus en énumérant l'ensemble des solutions possibles et en les testant à l'aide d'un algorithme polynomial. Une définition alternative consiste à dire que la classe NP contient tous les problèmes pouvant être résolus par un algorithme non-déterministe en un temps polynomial.

Pour autant, cela n'est pas toujours possible en pratique (on ne sait pas construire de machines non-déterministes). Par exemple, si on cherche à optimiser la durée d'une compilation sur un CD (i.e. avoir une durée totale la plus proche possible de la limite de capacité du CD) en choisissant parmi 90 chansons, alors, pour obtenir la solution exacte, il n'y a pas d'autre alternative que de tester chaque sous-ensemble de chansons (2^{90} choix). Alors, l'algorithme résolvant ce problème nécessite un ordinateur avec un processeur 2Ghz effectuant ce calcul depuis la création du système solaire (6×10^9 années).

Les problèmes NP-Complets

Avant de définir ce qu'est un problème NP-Complet, nous devons définir ce qu'est une transformation polynomiale.

La *transformation polynomiale* d'un problème de décision $\mathcal{P}_1$ en un problème de décision $\mathcal{P}_2$ existe si, étant donnée **toute** instance I_1 de $\mathcal{P}_1$, il est possible de construire **une** instance I_2 de $\mathcal{P}_2$ en un temps polynomial (par rapport à la taille de I_1), et tel que la réponse à I_1 est "oui" si et seulement si la réponse à I_2 est "oui" également.

Un problème $\mathcal{P}$ est *NP-Complet* s'il est parmi les plus difficiles de la classe NP. On prouve qu'un problème est NP-Complet s'il remplit les deux conditions suivantes :

1. $\mathcal{P} \in$ NP,
2. tous les problèmes appartenant à NP se transforment polynomialement en $\mathcal{P}$

En pratique, le second point de la preuve consiste à démontrer qu'il existe une transformation polynomiale d'un problème NP-Complet $\mathcal{P}_2$ en $\mathcal{P}$. Une conséquence directe de la définition d'un problème NP-Complet est que s'il existe un algorithme polynomial pour résoudre **un** problème NP-Complet, alors il est possible de résoudre **tous** les problèmes de NP en temps polynomial.

Il peut sembler surprenant que des problèmes NP-Complets existent, mais Cook [13] et Levin [37] ont prouvé séparément qu'un problème (3-SAT) était NP-Complet. Beaucoup de problèmes NP-Complets sont répertoriés dans le livre de Garey et Johnson [21].

1.2.3 Contourner la NP-Complétude

Durant la partie précédente, nous avons vu que les problèmes NP-Complets sont intraitables en pratique, pour des instances de tailles importantes, de part la complexité exponentielle des algorithmes qui les résolvent. En effet, trouver une solution polynomiale à un de ces problèmes reviendrait à trouver une solution pour tous les problèmes de NP. Cependant, ces problèmes sont souvent très appliqués (entre autres en bio-informatique) et méritent d'être résolus pour toute instance non-triviale. Afin de contourner la NP-Complétude, il existe différentes approches envisageables.

Par des heuristiques

Une première approche pour résoudre des problèmes NP-Complets consiste à faire un compromis sur la qualité de la solution recherchée afin d'accélérer le calcul. Une heuristique (du terme la-

tin *heuriskêin*, "trouver"), est un algorithme qui fournit, en temps polynomial, une solution pas nécessairement optimale au problème. Pour des informations plus détaillées, on pourra par exemple se reporter au Chapitre 10 de [5].

La contrepartie de la rapidité d'exécution d'une heuristique est l'absence de garanties sur la qualité de la solution renvoyée. On ne peut pas évaluer l'écart, même maximum, à l'optimal de la solution renvoyée. C'est pourquoi une heuristique est généralement validée par des tests pratiques sur des données pour lesquelles on connaît les solutions optimales.

Par des algorithmes d'approximation

A l'instar des heuristiques, ce type d'algorithmes effectue un compromis sur l'exactitude de la solution renvoyée. Mais, pour ce faire, il fournit des solutions dont la qualité peut être évaluée. On calcule un éloignement relatif à la solution optimale, qui est quantifié à l'aide d'un *ratio de performance* (quotient entre la solution de l'algorithme d'approximation et la solution optimale). Ce ratio est borné par une constante. Pour des informations plus détaillées, on pourra se reporter à [53].

Par des algorithmes de complexité paramétrée

Dans cette approche, on cherche à faire un compromis sur le temps d'exécution mais pas sur l'exactitude de la solution. Obtenir des solutions exactes, tout en restant efficace, dans le cadre de problèmes NP-Complets peut sembler paradoxal. L'idée principale est de restreindre "l'explosion combinatoire" à une partie de l'entrée : le *paramètre*. Un algorithme exponentiel classique est généralement exponentiel en n, la taille de l'instance. Il est cependant parfois possible de limiter cette partie exponentielle à un autre paramètre, de taille inférieure à n, et indépendant de la taille de l'instance.

Cette complexité introduite par Downey et Fellow [16] mène à la définition suivante :

Définition 3. *Un problème paramétré avec un paramètre k est dans la classe FPT (Fixed-Parameter Tractable) s'il existe un algorithme qui le résout pour n'importe quelle taille n en temps $O(f(k).n^c)$, avec f une fonction quelconque (donc pouvant être exponentielle) et c une constante.*

En pratique, un problème dans FPT peut être résolu efficacement si les deux conditions suivantes sont satisfaites :

- la partie exponentielle de la complexité dépendant de k n'est pas trop élevée. (En principe, la Définition 3 autorise des complexités en 1000^{k^k}).
- le paramètre k est petit en pratique.

Malheureusement, tous les problèmes NP-Complets ne sont pas dans FPT.

Chapitre 2

Concepts biologiques

Contenu

2.1 Les protéines

Les *protéines* (voir Figure 2.1) sont les molécules les plus complexes et les plus variées des êtres vivants, indispensables à leur survie. On estime que tout être humain fabrique à peu près 100 000 types de protéines différentes, chaque cellule en fabriquant en moyenne 15 000 types différents. Le terme protéine, du grec ancien *prôtos* signifiant premier ou essentiel, désigne une longue molécule (macromolécule) formée d'acides aminés.

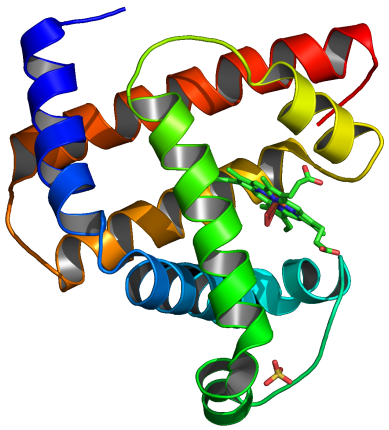


FIG. 2.1 – Représentation 3D de la protéine myoglobin. Son rôle est de transporter l'oxygène dans les tissus musculaires. Source : [1]

Les relations entre ADN, ARN et protéines sont définies dans le *dogme central de la biologie moléculaire* ; qui n'est autre que le modèle schématique de la conservation et de l'utilisation de l'information génétique. L'un des acteurs principaux de la transmission de l'information génétique est l'Acide DésoxyriboNucléique (ADN). L'ensemble de l'information encodée par l'ADN définit le *génome*, ensemble des *gènes* (i.e. séquence d'ADN). L' ADN est le support stable et transmissible de l'information génétique qui définit les fonctions biologiques d'un organisme. Lors de la méiose, l'ADN se réplique. Il est alors transcrit en Acide RiboNucléique (ARN). L'ARN peut alors être traduit en protéine à l'aide d'une entité biologique nommée *ribosome*. On parle alors de traduction de l'ARN en protéine. Le dogme central (cf. Figure 2.2) se résume ainsi : l'ADN se réplique, se transcrit en ARN qui est éventuellement traduit en protéine.

Cette traduction, illustrée par la Figure 2.3 est effectuée par le ribosome qui se fixe sur l'ARN transcrit, appelé ARNmessager (ARNm). Le ribosome va produire une séquence d'acides aminés en parcourant le brin d'ARNm codon par codon (un codon est un groupe de trois nucléotides de l'ARNm, correspondant soit à un acide aminé soit à un signal de fin de transcription). Pour ce faire, le ribosome va faire appel à l'ARN de transfert (ARNt) qui, en fixant un acide aminé et en le

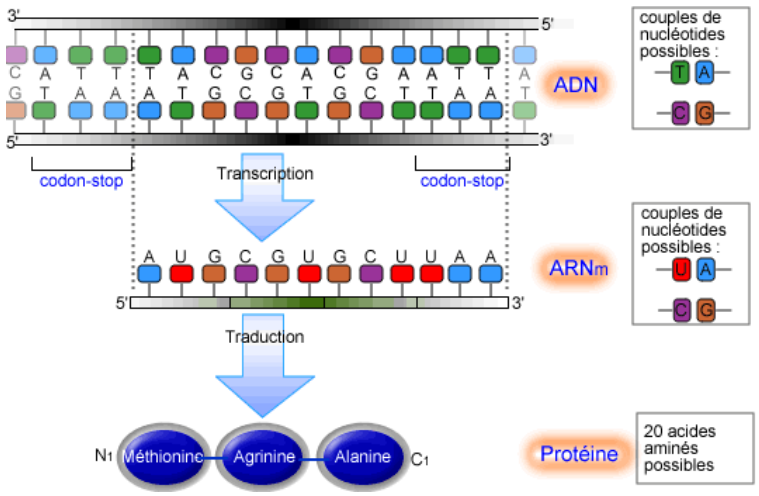


FIG. 2.2 – Dogme central : la séquence d'ADN d'un gène encode la séquence d'acides aminés d'une protéine. Source : wikipedia

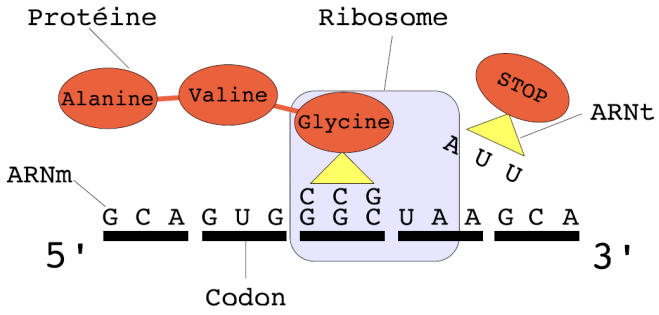


FIG. 2.3 – Illustration de la traduction de l'ARNm en protéine.

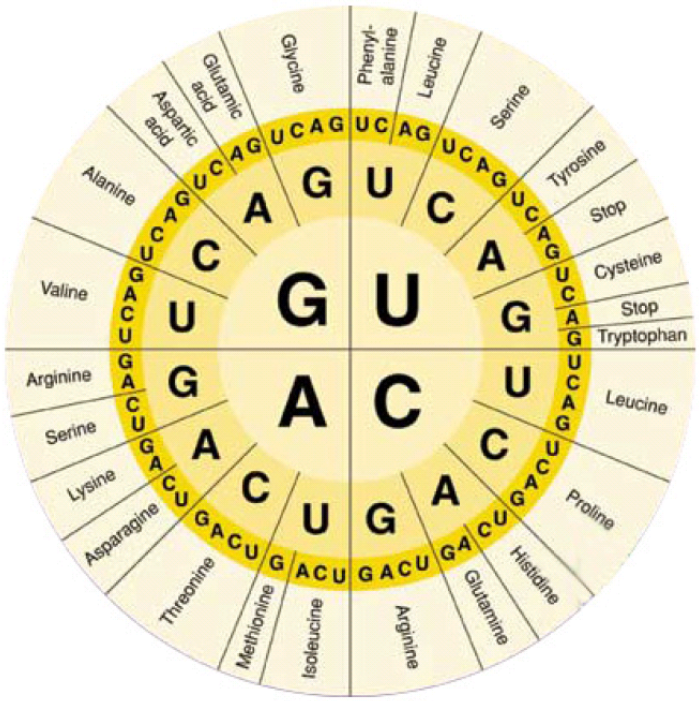
véhiculant jusqu'à l'ARNm, va faire le lien entre les acides aminés et le message porté par l'ARNm. Le ribosome va donc parcourir le brin d'ARNm codon par codon et va, par l'intermédiaire d'un ARNt, ajouter un acide aminé à la protéine en cours de fabrication selon le codon lu. La traduction s'achève lorsque le ribosome rencontre un codon-stop, par la séparation de la protéine, du ribosome et du brin d'ARNm. Avant d'être détruit, un ARNm peut servir à la fabrication, simultanée ou non, de 10 à 20 protéines. Le code génétique utilisé pour le décodage des codons est illustré en Figure 2.4.

Chaque protéine peut assumer plusieurs fonctions, dont :

- *catalyse*. Les enzymes sont des protéines qui accélèrent des réactions chimiques.
- *transport*. L'hémoglobine est une protéine chargée de transporter l'oxygène des poumons aux organes.
- *communication*. Des hormones, comme l'insuline, sont des protéines pouvant transporter des messages à travers tout l'organisme.
- *reconnaissance*. Dans le système immunitaire, l'immunoglobuline permet de reconnaître des corps étrangers.

Homologies entre deux protéines En toute généralité, une *homologie* désigne une similarité entre deux traits observés chez deux espèces, souvent due à l'existence d'un ancêtre commun. Ces traits, dits *homologues*, peuvent être de caractère anatomique, ou, ce qui nous intéresse dans cette étude, moléculaire.

On parle de protéines homologues si les gènes qui les codent ont une origine commune. On les reconnaît de part leurs structures spatiales proches, et les similarités présentées par leurs sé-



Code 1 lettre	Acide Aminé	Code 3 lettres	Code 1 lettre	Acide Aminé	Code 3 lettres
A	Alanine	Ala	O	Asparagine	
B	Alanine		P	Proline	Pro
C	Cystéine	Cys	Q	Glutamine	Glu
D	Acide aspartique	Asp	R	Arginine	Arg
E	Acide glutamique	Glu	S	Serine	Ser
F	Phénylalanine	Phe	T	Thréonine	Thr
G	Glycine	Gly	U	Thréonine	
H	Histidine	His	V	Valine	Val
I	Isoleucine	Ile	W	Tryptophane	Trp
J	Isoleucine		X	Valine	
K	Lysine	Lys	Y	Tyrosine	Tyr
L	Leucine	Leu	Z	Tyrosine	
M	Méthionine	Met	?	Selenocystéine	Sec
N	Asparagine	Asp	?	Pyrrolysine	Pyr

FIG. 2.4 – Le code génétique

quences d'acides aminés. Dans la plupart des cas, les fonctions de ces protéines sont plus ou moins semblables.

On peut trouver des protéines homologues chez des espèces différentes, ce qui est une des conséquences d'ancêtres communs à des espèces différentes, donc de l'évolution. On en trouve aussi au sein d'une même espèce, ce qui montre que l'évolution est aussi due à la complexification des génomes.

Deux protéines sont *orthologues* si elles proviennent d'espèces différentes, *paralogues* si elles proviennent de la même espèce. Pour plus de détails sur les homologies, se reporter à [18]. On détecte informatiquement ces homologies par des algorithmes d'alignement (comme FASTA [41] ou BLAST [3]), donnant un score de similarité.

2.2 Les réseaux de protéines

Ces réseaux peuvent être représentés par des graphes, où les noeuds et les arcs représentent respectivement les protéines et les interactions entre elles. Plus précisément, on pourra utiliser un arc non-orienté pour décrire une interaction mutuelle et une flèche pour une interaction non symétrique. La probabilité d'une interaction est parfois inscrite comme un poids sur l'arc. A titre d'exemple, le réseau de la levure comporte environ 6 000 protéines et 15 000 interactions. Le degré (i.e. nombre d'arcs entrant ou sortant d'une protéine) moyen est de 6, le degré maximum de 200. La Figure 2.5 présente une partie de ce réseau d'interactions.

FIG. 2.5 – Principal cluster du réseau d'interactions de la levure. Figure provenant de [28].

2.2.1 Interactions

Biologiquement, une interaction entre deux protéines représente une proximité physique, une complémentaritée spaciale. Les liaisons peuvent être de différentes natures : *ioniques* (i.e. interaction reliant deux atomes de charges opposées), *hydrogènes* (i.e. un atome d'hydrogène est partagé entre les deux protéines), *hydrophobes* (les poches hydrophobes à l'abri de l'eau ont tendance à se regrouper entre elles), ou dues aux *forces de Van Der Waals* (i.e. interaction électrostatique, sans interêts biologiques car faible et non spécifique). Plusieurs types d'interactions peuvent rentrer en jeu lors d'une interaction protéine-protéine. On peut également parler d'interaction lorsqu'une protéine envoie un signal à une autre, par exemple de régulation (activation/inhibition). Un exemple d'interaction entre protéine est illustrée avec la Figure 2.6.

Pour sa régulation, l'organisme a besoin de faire circuler des informations, d'un lieu à un autre, jusqu'aux cellules visées. L'information doit passer du milieu extracellulaire au milieu intracellulaire. Un moyen utilisé par l'organisme consiste en l'activation de protéines localisées à l'interieur de la cellule et impliquant une réponse biologique (e.g. la protéine kinase qui permet la stimulation d'autres protéines). De plus, une cellule doit adapter ses processus internes à son environnement extérieur (i.g. présence de certains nutriments, de facteurs de croissance, d'hormones, ou des conditions de température). Cela demande une coordination des réactions internes, que l'on étudie au travers des réseaux d'interactions de protéines. Ceux-ci jouent un rôle important dans la compréhension du fonctionnement de la cellule.

Généralement, on s'intéresse à deux types de structures d'interactions, que l'on va chercher à mettre en évidence, et qui vont aider à l'analyse des fonctions des protéines et des dynamiques intracellulaires [42] :

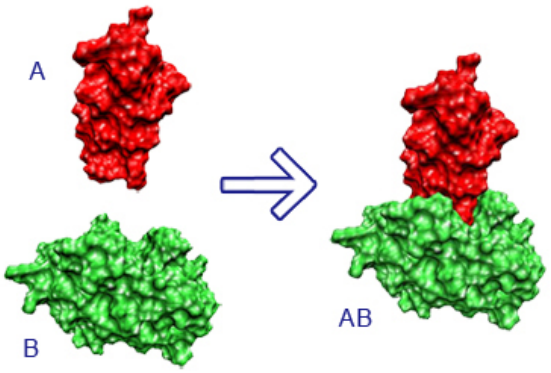


FIG. 2.6 – Exemple d'interaction entre deux protéines. Schéma provenant de wikibooks.org

1. *les chemins de signaux en cascades*, qui sont des chaînes linéaires de protéines qui interagissent. Chaque protéine interagie avec sa protéine voisine, ce qui provoque des réactions en chaîne. Ces structures sont représentées par des chemins dans le graphe. [32, 46, 50]

2. *les complexes protéiques*, qui sont des assemblements de protéines très connectées. Certaines interprétations tendent à dire que les protéines composant ce type de groupe ont généralement des fonctions communes. Ils sont représentés par des sous-réseaux denses (i.e. clusters). [19, 29, 35, 49]

2.2.2 Méthodes d'acquisition

On peut connaître les interactions entre des protéines à la suite de nombreuses expériences biologiques. Les expériences utilisées pour l'extraction des interactions les plus communes sont :

- *Coimmunoprecipitaion (CoIP)*, considérée comme la méthode de référence. Les interactions sont identifiées par électrophorèse. Les résultats retournés sont considérés comme bons, mais les interactions doivent être suspectées comme vraies : la méthode vérifie des hypothèses, elle ne trouve pas de nouvelles interactions. [22, 45]
- *Yeast Two-Hybrid Assay*, méthode mise en place par Fields et Song, qui consiste à créer des fusions artificielles de protéines à l'intérieur du noyau de la levure. [17]
- *Spectrometrie de masse* [25]
- *Extraction d'information depuis des articles scientifiques*. Certaines interactions ne sont pas construites depuis des expériences mais indirectement, en extrayant automatiquement des interactions connues depuis le vaste ensemble des articles scientifiques traitant ce sujet (plus de 300.000). Une approche est donnée par Peri et al. [43]

Une analyse comparative sur la qualité de ces différentes méthodes peut être trouvée en [54]. Ces expériences se font presque protéine par protéine. Le réseau entier d'une espèce ne s'obtient pas de manière instantanée. Il s'effectue touche par touche, en ajoutant petit à petit des interactions. Ceci est aussi un des facteurs du bruit présent dans les données.

Ces informations sont regroupées dans des bases de données disponibles en ligne. Le premier chapitre de la thèse de Bader [6] en énumère une cinquantaine. Finley [1] tient à jour des liens vers certaines. Les plus importantes sont

- DIP (Database of Interacting Proteins) - `http ://dip.doe-mbi.ucla.edu/`
- BIND (Biomolecular Interaction Network Database) - `http ://www.bind.ca/`

[1] `http ://proteome.wayne.edu/PIDBL.html`

2.2.3 Nombre et qualité des données

Le nombre de données sur ces interactions croit de manière exponentielle. En 2001, les réseaux ne comportaient qu'une centaine d'interactions. De nos jours, on en compte des milliers ; disponibles grâce aux techniques listées plus haut.

Cependant, toutes ces techniques souffrent d'un fort taux de bruit. Il a été estimé que le nombre de faux positifs (interactions inscrites dans la base mais n'existant pas biologiquement) est d'environ 50% [22]. De plus, le nombre de faux-négatifs (interactions biologiques existantes non représentées dans les bases) serait de 65% [44]. Algorithmiquement, on prendra en compte ce bruit par l'ajout de poids sur les arcs du graphe, permettant de donner un "score de confiance" à l'interaction.

Cette grande quantité, souvent fournie en parallèle, mêlée au bruit, force la création d'algorithmes performant afin de traiter et interpréter ces données.

Chapitre 3

Définitions des problèmes

Contenu

Dans ce chapitre, nous allons présenter les deux principaux problèmes auxquels nous nous sommes intéressés au cours de ce stage. Ils sont définis dans le but de participer au filtrage, à l'interprétation et à l'organisation des données des interactions entre les protéines, qui augmentent de manière exponentielle et qui sont très bruitées. Nous présentons dans un premier temps la comparaison de réseaux, utilisée dans le but de mettre en évidence des zones communes à plusieurs espèces, puis nous verrons la recherche de chemins au sein d'un réseau.

3.1 Motivations

Ces problèmes, en plus d'être des challenges algorithmiques intéressants, sont étudiés afin de contribuer à l'étude des enjeux biologiques suivants :

- quelles protéines, interactions ou groupe d'interactions ont des fonctions équivalentes à travers plusieurs espèces ?
- pouvons-nous prédire de nouvelles informations sur les protéines et les interactions d'espèces mal caractérisées ?
- quelles informations pouvons-nous obtenir sur l'évolution des protéines et de leur réseau ?
- lorsqu'un pathogène (i.e. un virus ou une bactérie) rentre dans un organisme, il interagit avec le réseau cellulaire (e.g. l'herpès intègre le réseau des protéines en de multiples points durant l'infection [52]). Mieux comprendre ces réseaux peut permettre d'aider au développement de nouveaux vaccins et médicaments (e.g. contre la malaria [20]), et de mieux comprendre les réactions du corps contre ces inflammations [11].
- en comparant des réseaux de cellules dysfonctionnantes avec ceux de cellules saines, de plus amples connaissances sur ces perturbations peuvent mener à une nouvelle approche dans la médecine préventive, comme par exemple sur les récentes recherches sur le cancer de la prostate [26].

3.2 Alignement de réseaux

Aligner des réseaux consiste à disposer, "superposer" leurs sommets afin d'identifier des structures communes entre ceux-ci. Dans le cadre de notre étude, ce principe va permettre de remarquer des points communs et de mettre en évidence des fonctions biologiques entre, au moins, deux réseaux d'interactions. On alignera des réseaux pouvant provenir d'espèces différentes, ou bien d'une même espèce mais sous différentes conditions ou acquis à différentes dates. Ces alignements peuvent s'effectuer entre deux réseaux ou plus.

3.2.1 Alignement de deux réseaux

Lors de l'alignement de deux réseaux, le but est de trouver des structures communes (i.e. chemins, zones denses,...) apparaissant dans chacun des deux réseaux. Certains algorithmes utilisent pour cela un graphe d'alignement.

Dans un *graphe d'alignement*, les noeuds représentent des couples de protéines. Dans un couple de protéines, une protéine provient du premier réseau, l'autre du second. De plus, on souhaite que ces deux protéines soient homologues, ce qui se caractérise par une valeur de similarité supérieure à un certain seuil (voir Section 2.1). Soient A et B (resp. a et b) deux protéines du premier réseau (resp. du second réseau). Par la suite, on considère A et a (resp. B et b) comme étant homologues. Alors, on regroupe A et a (resp. B et b) dans un noeud <A,a> (resp. <B,b>).

Les arcs du graphe d'alignement reliant ces noeuds représentent les interactions conservées entre les protéines de ces couples. Une interaction entre <A,a> et <B,b> est dite conservée si et seulement si une interaction existe entre A et B ainsi qu'entre a et b : c'est une interaction *directe*.

En général, les algorithmes utilisant ce genre de graphes (cf Section 4.1.1) n'autorisent pas seulement des interactions directes entre les paires de protéines. En effet, afin d'obtenir plus de résultats (des tests montrent qu'il y a seulement 7 interactions directes sur 2036 dans le graphe d'interactions de la levure et d'H. Pylori [32]), ils autorisent des *gaps* et des *mismatchs*, interactions "indirectes" dans un des deux réseaux (ou les deux). Ceci permet de prendre en compte l'évolution des réseaux, mais aussi de contourner les erreurs expérimentales pendant l'acquisition des données.

Il existe un *gap* entre <A,a> et <B,b> s'il y a une interaction entre A et B mais il existe au moins une protéine qui se trouve dans le chemin allant de a à b.

Il y a un *mismatch* entre <A,a> et <B,b> s'il y a au moins une protéine qui se trouve dans le chemin allant de a à b **ainsi** qu'entre A et B.

Ces trois types d'interactions sont représentés sur la Figure 3.1.

Une fois ce graphe construit, la recherche de similarités est facilitée. Par exemple, en utilisant les propriétés de ce graphe, on repère les complexes protéiques conservés entre les deux espèces en déctectant les clusters et les noeuds avec beaucoup d'interactions. On pourra utiliser des algorithmes de recherches "classiques" de théorie des graphes. (voir Figure 3.2).

3.2.2 Alignement multiple de réseaux

Une généralisation du problème d'alignement à plusieurs réseaux existe : on veut pouvoir aligner k réseaux d'interactions. Ces alignements multiples sont effectués afin de mettre en évidence des interactions qui n'auraient pas pu être trouvées en alignant les réseaux deux à deux. De plus, ajouter plusieurs espèces permet de contourner le bruit pouvant empêcher de trouver des régions conservées.

Une première approche consiste à construire le même graphe d'alignement que lors de l'alignement de deux réseaux, mais avec les noeuds regroupant k protéines lorsque l'on veut aligner k réseaux (NetworkBlast [49]). Cependant, cette technique limite le nombre de réseaux pouvant être alignés à trois car le graphe résultant grossit de manière exponentielle avec le nombre de réseaux à aligner.

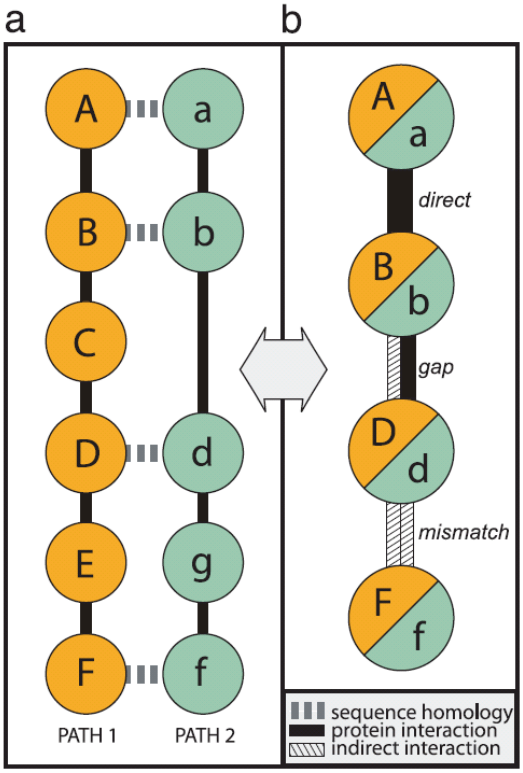


FIG. 3.1 – Exemple de graphe d'alignement. a) Extrait de deux réseaux d'espèces différentes. Les lignes verticales pleines représentent les interactions protéines-protéines d'un chemin simple. Les lignes horizontales en pointillés indiquent une homologie entre les protéines (i.e. une similarité supérieure à une valeur seuil). b) Les deux chemins sont combinés dans un graphe d'alignement. Chaque noeud représente une paire homologue et les liens entre ces noeuds peuvent être de trois types : *directs*, *gap* (i.e. ajout d'une protéine là où il n'y en avait pas), *mismatch* (i.e. suppression de protéines dans le graphe d'alignement). Schéma provenant de [32].

Récemment, de nouvelles structures, tels les *graphes à k-couches* ont permis d'augmenter le nombre de réseaux à aligner (NetworkBlast-M [29]), ces notions seront plus détaillées dans le Chapitre 4.

3.3 Recherche de chemins

Dans la section précédente, nous avons vu que l'alignement consiste à comparer plusieurs réseaux, afin de déterminer des structures communes entre-eux. La recherche de chemins va pour sa part tenter de mettre en évidence la présence de structures significatives (i.e. avec un score élevé) connues dans un réseau donné.

3.3.1 Recherche de meilleurs chemins

La recherche de chemins consiste à mettre en évidence des structures biologiques, dans un réseau donné. Par exemple, les protéines composants les chaînes de signaux en cascades vont avoir tendance à avoir de fortes interactions entre elles. Pour détecter ces phénomènes biologiques, on va algorithmiquement rechercher des chemins simples de poids maximum (i.e. somme des poids des arcs composant le chemin) reliant k sommets du graphe, k étant la longueur du chemin recherché. Lorsque $k = n$, n étant le nombre de sommets dans le graphe, le problème se rapporte au

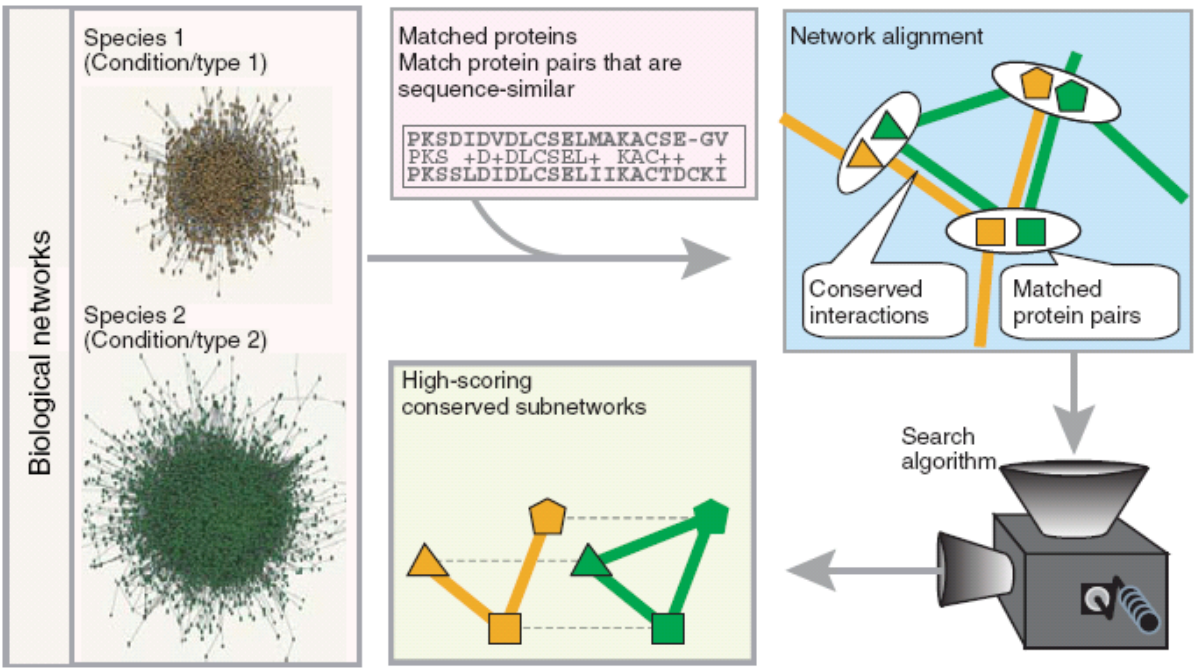


FIG. 3.2 – Alignement de deux réseaux. Après avoir identifié les protéines similaires, on crée le graphe d'alignement regroupant les deux réseaux. On peut ensuite appliquer des algorithmes de recherche, afin d'identifier des motifs avec un score élevé. Schéma provenant de [47].

Problème du Voyageur de Commerce [1] (i.e. relier toutes les villes, en un minimum de temps, en ne passant qu'une seule fois par chaque ville), qui est NP-Complet. Ainsi, la recherche d'un meilleur chemin dans un réseau est NP-Complet. Nous verrons qu'il est possible de rendre la complexité de ce problème paramétrée (*Color-coding*), et d'utiliser des heuristiques pour le résoudre approximativement.

3.3.2 Recherche de motifs

Les algorithmes d'alignement tentent de trouver de nouvelles régions biologiquement significatives. La recherche de motifs suppose quant à elle que des régions intéressantes sont connues au préalable. Ainsi, étant donnés un réseau et un "chemin requête", on désire savoir si ce dernier est présent ou non dans le réseau. De plus, si la réponse est positive, on pourra déterminer à quel degré de similarité la requête est conservée. En effet, comme pour l'alignement, les algorithmes traitant ce problème pourront autoriser une certaine "distance" entre la requête et le résultat, aussi appelée *insertion/délétion* (in/del). On nomme *insertions* les protéines se trouvant dans le résultat mais qui n'étaient pas demandées dans la requête. Symétriquement, on appelle *délétions* les protéines qui étaient demandées dans la requête mais qui ne sont pas dans le résultat. Ceci est résumé avec la Figure 3.3.

Généralement, on cherchera à pénaliser les insertions et les délétions. Pour ce faire, on détermine un *score d'alignement*, entre la requête et le réseau : les protéines de la requête en correspondance avec le réseau augmentent ce score, tandis que les insertions et les délétions le diminuent. On cherchera donc à obtenir les chemins avec un meilleur score, respectant par conséquent au mieux la requête.

Certains algorithmes autorisent des requêtes prenant la forme d'un *chemin* [50]. Plus récemment, les requêtes ont pu prendre des formes plus complexes, comme des *arbres* ou même des *graphes* (sous certaines conditions) [15].

[1] http ://en.wikipedia.org/wiki/Travelling_salesman_problem

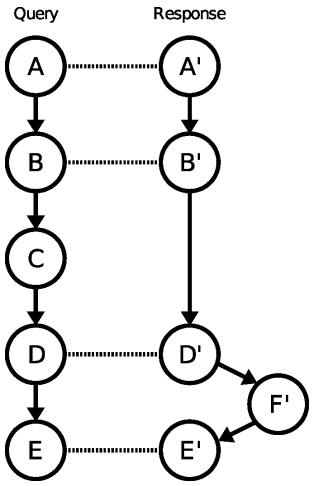


FIG. 3.3 – Figure illustrant un chemin requête sur la gauche, et un chemin résultat extrait du réseau d'interaction. Les lignes horizontales représentent les homologies entre les protéines. La protéine F' est *insérée* dans le résultat, la protéine C est *délétée* par rapport à la requête. Schéma provenant de [50].

Chapitre 4

État de l'art

Contenu

Dans ce chapitre, nous présentons les articles scientifiques traitant des problèmes présentés dans le chapitre précédent. Un état de l'art parfaitement exhaustif est difficile étant donnée la masse d'articles traitant du sujet. Nous avons cependant essayé de sélectionner les articles les plus représentatifs et importants concernant les problèmes auxquels nous nous sommes intéressés.

Ainsi, pour traiter du problème de l'alignement de réseaux, nous avons étudié *PathBLAST*, qui est un des premier à avoir traité ce problème. Nous verrons ensuite *MaWish*, qui, en regard de la complexité de PathBLAST, présente une heuristique pour ce même problème. Dans le cadre de l'alignement multiple, nous présentons *NetworkBLAST*, extension de PathBLAST à trois réseaux, puis *NetworkBLAST-M*, permettant l'alignement d'une dizaine de réseaux en parallèle, au moyen de nouvelles structures.

Dans le cadre de la recherche de chemins, nous présenterons une technique, nommée *color-coding*, mise au point par Alon et al. en 1995, permettant la recherche de chemins avec une complexité paramétrée. Nous verrons l'implémentation effectuée par Scott et al. qui se base sur cette technique, puis l'amélioration effectuée par Hüffner et al. Nous verrons également une heuristique de Lu et al. n'utilisant pas le color-coding mais un algorithme de diviser pour régner. Dans le cadre de la recherche de motif, nous verrons *QPath* et *QNet*, qui se basent tous les deux sur le color-coding, et qui permettent de rechercher respectivement des chemins et des graphes dans un réseau.

4.1 Alignements de réseaux

4.1.1 Alignements de deux réseaux

PathBLAST par Kelley et al. (2003)

PathBLAST est un algorithme mis au point en 2003 par Kelley et al. [32], donnant une première approche de l'alignement de deux réseaux d'interactions de protéines. Il est nommé ainsi en raison de sa proximité conceptuelle avec BLAST [3], son pendant pour l'alignement de séquences (un article de Batzoglou résume ce problème et ses solutions [10]). PathBLAST prend en **entrée** deux

réseaux d'interactions et un entier k, et produit en **sortie** un chemin d'alignement entre les deux réseaux, de longueur k, représentant le chemin avec le plus de similarités entre eux. On note qu'un des deux réseaux peut-être réduit à un chemin simple, ce qui revient au problème de recherche de motifs.

L'algorithme L'algorithme se déroule en deux étapes. Dans un premier temps, l'algorithme construit le graphe d'alignement (voir Section 3.2.1), regroupant les deux réseaux et en autorisant qu'un seul mismatch consécutif. Ensuite, une fois le graphe créé, l'algorithme cherche les chemins de score maximum.

Un chemin peut-être trouvé en temps linéaire, en fonction du nombre d'arcs du graphe, si celui-ci est acyclique (i.e. sans cycles) : la difficulté du problème vient des cycles.

Cependant, dans le cas général, un graphe n'est pas acyclique. Kelley et al. proposent donc de générer $5k!$ sous-graphes acycliques, k étant la longueur du chemin recherché. Ce nombre est considéré comme suffisant ; aucune justification n'est cependant donnée dans l'article sur le choix de ce nombre.

Pour générer un sous-graphe acyclique, les auteurs expliquent qu'ils ordonnent aléatoirement les noeuds du graphe avant de supprimer les "arcs avant" (i.e. les arcs qui ont une source avec un rang inférieur à leur cible) : le graphe résultant est alors sans cycles mais a subit une perte d'information. Les auteurs prouvent qu'un chemin intéressant présent dans le graphe de départ est conservé tous les $\frac{2}{k!}$ sous-graphes, k étant la longueur du chemin recherché. Une fois tous les sous-graphes générés, ils regroupent les résultats des différentes recherches de meilleur chemin (effectuées par programmation dynamique[1]) lancées sur chacun des sous-graphes acycliques. Finalement, la complexité obtenue est de $O(k!n^{O(1)})$ en temps et de $O(m+n)$ en espace, où n et m sont respectivement le nombre de sommets et d'arcs du graphe.

Scores Le score d'un chemin correspond à la somme des probabilités des homologies sur les protéines et des interactions entre les protéines composant le chemin.

Résultats Les auteurs sont plutôt satisfait des résultats obtenus par l'algorithme. La complexité limite la longueur des chemins cherchés à cinq. Toutefois, à partir du réseau de la levure, très connu, obtenu depuis la base de données DIP en novembre 2002, ils arrivent à retrouver des chemins identiques dans le réseau d'une bactérie moins bien caractérisée. Les scores obtenus sont bien meilleurs que lorsque les données sont randomisées (i.e. permutation aléatoire des interactions). Ceci suggère que les deux espèces partagent des chemins d'interactions avec des fonctions biologiques proches.

La seconde expérience proposée consiste à aligner le réseau de la levure avec lui-même. Dans ce cas, l'algorithme retrouve des chemins identiques, à différents emplacements. Ceci suggère une duplication antérieure suivie d'une spéciation. Cependant, dans le cadre de cette expérience, les réseaux sont de tailles trop importantes pour pouvoir lancer l'algorithme en autorisant les gaps et les mismatchs.

Synthèse et discussions PathBLAST est victime de trois inconvénients majeurs. Premièrement, les expériences montrent que sur les 2038 arcs de la levure alignée avec H. Pylori, il y a seulement sept interactions directes, ce qui est peu comparé aux 260 gaps et aux 1769 mismatchs. Ceci confirme l'intérêt de leur autorisation mais suggère qu'augmenter le nombre de mismatchs consécutifs autorisés (PathBLAST n'en autorise qu'un seul) pourrait améliorer les résultats. D'autre part, la complexité empêche expérimentalement d'obtenir des chemins de taille supérieure à cinq et d'autoriser des mismatchs dans l'alignement de la levure avec elle-même. L'utilisation du color-coding pourrait améliorer ceci. Finalement, lors de la création du graphe d'alignement, une même

[1] `http://en.wikipedia.org/wiki/Dynamic_programming`

protéine peut intervenir à plusieurs reprises (e.g. avec une interaction entre le noeud <A,a> et le noeud <B,a>), ce qui n'est pas réaliste biologiquement.

Si PathBLAST introduit les algorithmes d'alignement (notamment avec le graphe d'alignement) , il comporte cependant des inconvénients que d'autres algorithmes tenteront de corriger.

MaWish par Koyutürk et al. (2005)

MaWish est un algorithme de Koyutürk et al. [35] présentant une heuristique pour répondre au problème de l'alignement de deux réseaux. La particularité de cet algorithme réside dans son attachement à prendre en compte au mieux l'*évolution* des réseaux dans le modèle de score. Il prend en **entrée** deux réseaux et produit en **sortie**, au moyen d'un algorithme glouton, un ensemble de sous-graphes très connectés.

L'algorithme Dans un premier temps, à l'instar de PathBLAST, l'algorithme crée le graphe d'alignement des deux réseaux. Ensuite, en sachant que rechercher le sous-graphe de poids maximum est NP-Complet, les auteurs proposent une heuristique. Ils se basent sur les observations suivantes : les modules fonctionnels et les complexes protéiques sont composés de protéines fortement connectées. Ces modules seraient bien reconnaissables car ils interagissent peu entre eux. Ainsi, une protéine d'un module particulier interagit principalement avec des protéines faisant parti du même module [51]. Ceci mène à la création de leur algorithme glouton, qui consiste à construire un sous-graphe grossissant (i.e. de manière gloutonne). A chaque étape, l'algorithme ajoute la protéine qui a beaucoup d'interactions et peu de mismatchs avec ce sous-réseau. Une fois que le sous-réseau ne peut plus s'améliorer, l'algorithme en crée un nouveau. Ainsi, la complexité en temps est de $O(n \times m)$, n et m étant respectivement le nombre de sommets et d'arcs du réseau. En effet, chaque alignement est au pire effectué en $O(m)$, et chaque noeud ne peut être aligné qu'une seule fois.

Scores Les auteurs vont chercher à prendre en compte l'évolution biologique des réseaux. Pour cela, ils vont détecter les duplications (i.e. deux protéines homologues dans le même réseau) ainsi que les créations et suppressions de liens. Ainsi, ils donnent un score négatif aux mismatchs et aux duplications (i.e. deux protéines avec une forte homologie dans le même réseau), et donnent un score positif aux matchs.

Résultats Afin d'illustrer l'intérêt de leur algorithme, les auteurs comparent les réseaux de deux mammifères : l'homme et la souris, provenant de la base de données DIP d'octobre 2004. En quelques millisecondes, ils retrouvent une quinzaine de sous-réseaux communs.

Synthèse et discussions MaWISH présente une heuristique pour l'alignement de deux réseaux. Sa faible complexité permet l'alignement de réseaux de taille conséquente. En revanche, du fait de la nature heuristique de leur approche, ils ne leur est pas possible de prouver théoriquement la qualité du résultat.

4.1.2 Alignements multiples

Dans cette partie, nous présentons les articles proposant des algorithmes pouvant aligner plus de deux réseaux en parallèle. Nous verrons que le graphe d'alignement limite le nombre de réseaux pouvant être alignés. Pour palier à cette limitation, nous présenterons d'autres structures.

NetworkBLAST par Sharan et al. (2005)

NetworkBLAST est un algorithme proposé par Sharan et al. en 2005 [49]. Il permet d'aligner jusqu'à trois réseaux simultanément en **entrée**, ce qui est une avancée par rapport à PathBLAST.

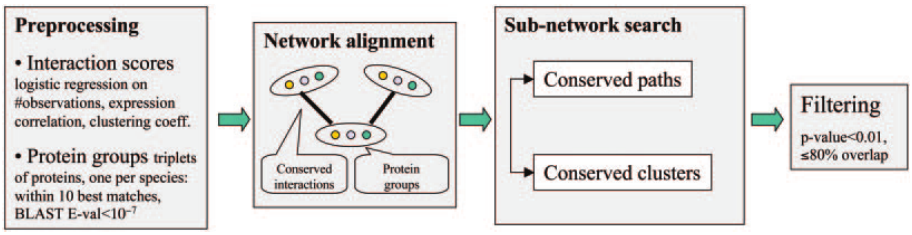


FIG. 4.1 – Illustration de l'alignement multiple de réseaux. Les protéines similaires entre les trois réseaux sont regroupées dans les noeuds du graphe, puis l'algorithme de recherche ressort les structures intéressantes. Schéma provenant de [49].

De plus, il permet de détecter tous les complexes conservés entre les réseaux. Plus précisément, NetworkBLAST propose en **sortie** deux types de structures : les chemins linéaires, comme PathBLAST (les auteurs les définissent comme "court" mais ne donnent pas d'indication de longueur), mais aussi des clusters denses [48] – régions de protéines fortement connexes. La Figure 4.1 illustre ces entrées-sorties.

Algorithme Tout comme PathBLAST, l'algorithme commence par la création d'un graphe d'alignement, non-orienté. Cependant, la construction est plus coûteuse car elle doit prendre en compte plus de deux réseaux en entrée. Dans le graphe, les noeuds correspondent à k protéines similaires, une de chaque espèce, k étant le nombre de réseaux donnés en entrée (en pratique deux ou trois). Deux protéines seront considérées comme similaires si BLAST évalue leur similarité comme étant supérieure à un seuil. Les arcs représentent les interactions conservées entre les noeuds. Ils effectuent une connexion entre deux noeuds $<A_1, ..., A_k>$ et $<B_1, ..., B_k>$ si et seulement si une des conditions suivantes est satisfaite :

- une des paires de protéines interagit de manière directe et toutes les autres paires sont à une distance d'au plus deux (i.e. une protéine est située entre eux).
- toutes les paires de protéines sont à une distance deux.
- il y a au moins $\max\{2, k-1\}$ paires de protéines qui interagissent directement.

La rapidité de l'algorithme de recherche dépend de la taille du graphe d'alignement créé. L'algorithme va donc ensuite y chercher des sous-graphes, correspondant à des conservations de réseaux entre les différentes espèces.

Identifier des sous-réseaux de protéines conservées se réduit au problème d'identification de sous-réseaux à score maximal dans le graphe d'alignement, ce qui est NP-Dur. Les auteurs conçoivent donc une heuristique. Pour cela, ils font une recherche exhaustive de tous les chemins de longueur quatre, cela forme les graines, dont les meilleures (i.e. plus haut score) vont être étendues de manière gloutonne (i.e. on y ajoute des protéines une à une pour augmenter leurs scores).

Scores Ils adaptent une technique de Bader et al. [7] qui assigne des valeurs de confiance aux interactions. La probabilité d'une vraie interaction est dépendante de trois variables expérimentales.

Résultats Ils utilisent leur algorithme pour aligner les réseaux de la levure, du ver et de la mouche. Ils identifient 183 clusters et 240 chemins conservés, couvrant un total de 649 protéines sur les trois réseaux. Afin de valider leur résultats, ils les comparent aux complexes connus du Munich Information Center for Protein Sequences (MIPS) [39], considéré comme une référence. Alors, 94% de leurs complexes sont compatibles. De plus, ils valident des fonctions biologiques de certains complexes avec Gene Ontology [4].

Synthèse et discussion Ce nouvel algorithme apporte plusieurs avantages. Il permet d'aligner plus de deux réseaux, ce qui permet des interprétations biologiques supplémentaires : ainsi, ils

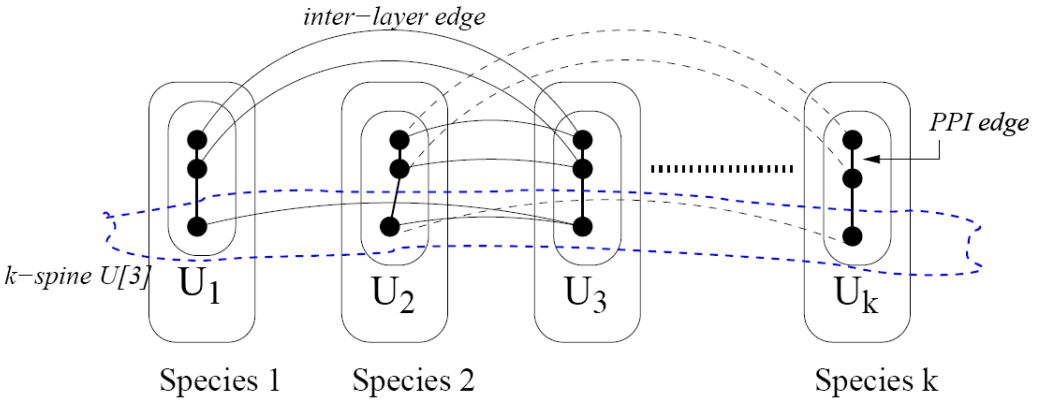


FIG. 4.2 – Schéma d'un graphe à k-couches. Chaque couche, notée U_i, représente le réseau d'une des k espèces. Les arcs inter-couches, liant des protéines de couches différentes, représentent les homologies entre les protéines. La ligne de pointillés bleu englobe un des trois k-spines. Schéma provenant de [29].

trouvent des mécanismes qui n'auraient jamais pu être trouvés sans. Si l'alignement de deux réseaux ne recherche que les *meilleures* structures, aligner plusieurs réseaux permet plus de flexibilité. De plus, leur méthode unifie les recherches de chemins et de clusters. Enfin, les auteurs proposent des solutions de visualisation des structures relevées. Toutefois, le nombre de réseaux pouvant être alignés reste limité à trois, et c'est ce que l' algorithme suivant va chercher à améliorer.

NetworkBLAST-M par Kalaev et al. (2008)

La structure du graphe d'alignement dans NetworkBLAST limite le nombre de réseaux pouvant être alignés à trois. C'est à partir de ce constat que se base NetworkBLAST-M, extension de NetworkBLAST par Kalaev et al. [49], qui augmente le nombre de réseaux alignés en changeant les structures utilisées. De la même manière que son prédécesseur, l'algorithme prend en **entrée** k réseaux, avec k pouvant aller jusqu'à dix, et indique en **sortie** des structures communes.

Algorithme Le but est donc d'aligner k réseaux d'interactions. Les auteurs introduisent une nouvelle structure pour effectuer ceci, un *graphe à k-couches*, qui est la nouvelle structure du graphe d'alignement. Dans celui-ci, chaque couche correspond à une espèce et comporte le réseau de l'espèce en question. Ainsi, la couche i a un ensemble V_i de sommets et E_i d'arcs de l'espèce ($|V_i| = n$ pour tous les i). De plus, ils ajoutent un nouvel ensemble *inter-couches*, noté E_H, représentant les homologies entre les protéines. Ils définissent alors $G_H = (\cup_i V_i, E_H)$, comme étant le graphe limité aux arcs inter-couches. La relation entre l'ancien graphe d'alignement et ce graphe k-couches est claire. Dans le premier, les protéines homologues étaient représentées dans un noeud du graphe. Ici, ce même ensemble est représenté par un sous-graphe de taille k, incluant un noeud de chaque couche. Ce sous-graphe sera appelé *k-spine*. Les k protéines dans cette structure sont connectées (i.e. un arbre recouvrant peut être défini). Un ensemble de k-spines connectés est un possible sous-réseau conservé entre les k espèces. La Figure 4.2 illustre cette représentation.

Une fois cette structure créée, l'algorithme recherche des sous-réseaux de taille m avec un score maximum, pour un m fixé. C'est un problème NP-Dur, même s'il n'y a qu'un seul réseau. Par conséquent, les auteurs proposent une heuristique, consistant à démarrer avec des graines de plus haut score, puis les étendre localement de manière gloutonne. C'est le même principe que dans le graphe d'alignement de NetworkBLAST. Il faut donc l'appliquer à la nouvelle structure.

Il y a donc deux étapes : la recherche de graines, puis leur extension. Pour le calcul des graines, ils calculent tous les sous-réseaux de taille d, avec $d \ll m$, généralement $d = 2$. Cependant, les auteurs ne proposent pas d'autres algorithmes que le naïf, qui doit regarder toutes les paires de

k-spines. Cet algorithme de complexité $O(n^{dk})$ n'est pas réalisable pour les tailles de réseaux habituels. Les auteurs introduisent alors deux hypothèses sur les arcs "inter-couches" afin de réduire la complexité en ratant le moins possible de bonnes graines.

Pour étendre les graines, ils ajoutent à chaque itération le k-spine qui améliore le plus le score.

Scores Les auteurs utilisent le même système de scores que NetworkBLAST.

Résultats Les auteurs testent leur algorithme sur une dizaine de réseaux de microbes et comparent les performances avec NetworkBLAST. En testant les réseaux trois par trois, ils remarquent que leur méthode prend quelques secondes quand NetworkBLAST prend six heures. Ils comparent ensuite leur méthode avec Graemlin [19], qui peut aussi aligner une dizaine de réseaux. Cet algorithme propose d'aligner les réseaux deux par deux, en débutant par les deux réseaux les plus proches dans l'arbre phylogénétique, et de remonter petit à petit : c'est un alignement progressif. Les auteurs ne donnent pas d'ordre d'idées de rapidité des algorithmes, mais indiquent que la spécificité [2] de leur algorithme est meilleure.

Synthèse et discussions NetworkBLAST-M tente de supprimer l'inconvénient de NetworkBLAST qui était limité à l'alignement simultané de trois réseaux. À l'aide d'une nouvelle structure plus adéquate que le graphe d'alignement classique, les auteurs arrivent à proposer la possibilité d'aligner une dizaine de réseaux simultanément de manière rapide. Ceci est particulièrement intéressant étant donné l'augmentation du nombre et des tailles des réseaux d'interactions de protéines. Les auteurs cherchent encore à améliorer les heuristiques de recherches et d'extensions des graines.

4.2 Recherche de chemin

4.2.1 Recherche de meilleur chemin

Les algorithmes de cette section cherchent à détecter les chemins (et des structures plus générales par la suite) de meilleurs scores au sein d'un réseau d'interactions de protéines. On présentera une version naïve, impraticable, puis le color-coding, donnant une complexité paramétrée. Finalement, on présentera des améliorations pour le color-coding ainsi qu'une heuristique pour ce problème.

Scott et al. (2005)

Scott et al. donnent en 2005 [46] une implémentation pour répondre au problème de recherche de meilleur chemin. Ils exposent l'intérêt du *color-coding*, puis présentent quelques extensions selon des contraintes biologiques. Dans tous les cas, un graphe est pris en **entrée**, un "pathway" (pouvant être un chemin, un arbre ou un 2SPG) avec le meilleur score est proposé en **sortie**.

Color-coding Le *color-coding* est une technique proposée par Alon et al. en 1995 [2] pour répondre au problème de recherche de meilleur chemin avec une complexité paramétrée par la taille du chemin, permettant de passer la complexité de $O(n^k)$ à $O(2^k)$, k étant la longueur du chemin recherché et n le nombre de noeuds du réseau. Afin de montrer son intérêt, nous présentons l'algorithme naïf, puis le color-coding afin de pouvoir les comparer.

[2] http://fr.wikipedia.org/wiki/Sensibilit%C3%A9_%28statistique%29

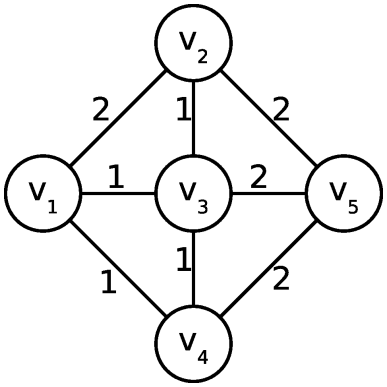
Résolution naïve La résolution de ce problème s'effectue avec un algorithme de programmation dynamique. Soit un graphe pondéré $G = (V, E)$ avec sa fonction de pondération w et un entier positif $i < k$, où k est la longueur du chemin recherché. On suppose que pour n'importe quel noeud $v \in V$ et n'importe quel sous-ensemble de i noeuds $V' \subseteq V$, on connaît le poids minimum $W(v, V')$ d'un chemin de taille i passant par tous les noeuds de V' et se terminant en v. Alors, on peut calculer tous les sous-ensembles de taille $(i + 1)$, car un chemin de taille $(i + 1)$ se terminant en v peut être décomposé en deux parties : un chemin de taille i se terminant sur un *voisin* de v et l'arc liant le voisin à v. La récurrence est directe lorsque le sous-ensemble est de taille 1.

Plus formellement, la relation de récurrence gérant cette programmation dynamique est définie comme suit :

$$W(v, V') = \min_{u \in V' \setminus \{v\}} \left(W(u, V' \setminus \{u\}) + w(u, v) \right), |V'| > 1$$

avec $W(v, \{v\}) = 0$ si $v \in I$ et ∞ sinon, I étant un ensemble de protéines de départ.

Pour illustrer l'application de cet algorithme naïf, cherchons un chemin de poids minimum de taille quatre dans ce graphe :



La table de programmation dynamique est initialisée avec les poids du graphe, avec $W(v, V') = \infty$ si un chemin n'existe pas. Tous les noeuds sont considérés comme étant dans l'ensemble de départ.

$$\begin{array}{llll}
W(v_1, \{v_2\}) = 2, & W(v_1, \{v_3\}) = 1, & W(v_1, \{v_4\}) = 1, & W(v_1, \{v_5\}) = \infty, \\
W(v_2, \{v_1\}) = 2, & W(v_2, \{v_3\}) = 1, & W(v_2, \{v_4\}) = \infty, & W(v_2, \{v_5\}) = 2, \\
W(v_3, \{v_1\}) = 1, & W(v_3, \{v_2\}) = 1, & W(v_3, \{v_4\}) = 1, & W(v_3, \{v_5\}) = 2, \\
W(v_4, \{v_1\}) = 1, & W(v_4, \{v_2\}) = \infty, & W(v_4, \{v_3\}) = 1, & W(v_4, \{v_5\}) = 2, \\
W(v_5, \{v_1\}) = \infty, & W(v_5, \{v_2\}) = 2, & W(v_5, \{v_3\}) = 2, & W(v_5, \{v_4\}) = 2.
\end{array}$$

Après cette initialisation, l'algorithme détermine les $W(v, V')$ pour les sous-ensembles V' de taille deux en utilisant la formule de récurrence :

$$\begin{array}{l}
W(v_1, \{v_2, v_3\}) = \min\{W(v_2, \{v_3\}) + w(\{v_2, v_1\}), W(v_3, \{v_2\}) + w(\{v_3, v_1\})\} = 2, \\
W(v_1, \{v_2, v_5\}) = \min\{W(v_2, \{v_5\}) + w(\{v_2, v_1\}), W(v_5, \{v_2\}) + w(\{v_5, v_1\})\} = 4, \\
\vdots \\
W(v_5, \{v_3, v_4\}) = \min\{W(v_3, \{v_4\}) + w(\{v_3, v_5\}), W(v_4, \{v_3\}) + w(\{v_4, v_5\})\} = 3.
\end{array}$$

Finalement, l'algorithme remplit la table pour les sous-ensembles V' de taille trois, ce qui mène à nos chemins de taille quatre. On obtient :

$$\begin{array}{rcl}
W(v_1, \{v_2, v_3, v_4\}) & = & \min\{W(v_2, \{v_3, v_4\}) + w(\{v_2, v_1\}), W(v_3, \{v_2, v_4\}) + \\
& & w(\{v_3, v_1\}), W(v_4, \{v_2, v_3\}) + w(\{v_4, v_1\})\} = 3, \\
W(v_1, \{v_2, v_3, v_5\}) & = & \min\{W(v_2, \{v_3, v_5\}) + w(\{v_2, v_1\}), W(v_3, \{v_2, v_5\}) + \\
& & w(\{v_3, v_1\}), W(v_5, \{v_2, v_3\}) + w(\{v_5, v_1\})\} = 4,
\end{array}$$

$$\vdots$$
$$W(v_5, \{v_2, v_3, v_4\}) = \min\{W(v_2, \{v_3, v_4\}) + w(\{v_2, v_5\}), W(v_3, \{v_2, v_4\}) + w(\{v_3, v_5\}), W(v_4, \{v_2, v_3\}) + w(\{v_4, v_5\})\} = 4.$$

Lors de la dernière itération, l'algorithme trouve que le chemin de taille quatre de poids minimum dans ce graphe d'exemple est de poids 3. Ce poids est réalisé pour deux solutions, qui sont $W(v_1, \{v_2, v_3, v_4\})$ et $W(v_2, \{v_1, v_3, v_4\})$. Le chemin qui correspond à un certain poids $W(v, V')$ peut être retrouvé par un algorithme de backtracking.

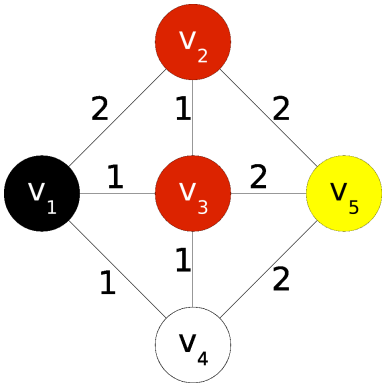
Le problème de cette résolution est son efficacité : dans le pire des cas, elle est bornée par le nombre maximum de $W(v, V')$ différents, qui est de $n \times \binom{n}{k}$. En effet, il y a n possibilités pour choisir le sommet v et $\binom{n}{k}$ possibilités pour former l'ensemble V' (ensemble de taille k formé parmi les n noeuds), ce qui est borné par n^k. On obtient donc une complexité de $O(n^k)$, infaisable en pratique[3]. C'est pour résoudre ce problème que le color coding entre en jeu.

Résolution par color-coding L'idée principale du color-coding est d'assigner *aléatoirement* k couleurs aux sommets du graphe, k étant la longueur du chemin recherché. Ensuite, l'algorithme recherche des chemins *colorful* : on cherche des chemins ou chaque **couleur** est différente. Clairement, un chemin colorful est simple : chaque sommet ne reçoit qu'une seule couleur. L'algorithme est analogue à la solution naïve, mais est plus efficace. La seule différence est qu'au lieu de calculer les valeurs $W(v, V')$, on calcule maintenant les valeurs $W(v, S)$, où S n'est pas un sous-ensemble de noeuds mais un sous-ensemble de k couleurs utilisées pour colorier le graphe. Plus précisément, l'équation est maintenant la suivante :

$$W(v, S) = \min_{color(u) \in S \setminus \{color(v)\}} \left(W(u, S \setminus \{color(u)\}) + w(u, v)\right), |S| > 1$$

avec $W(v, \{color(v)\}) = 0$ si $v \in I$ et ∞ sinon, I ensemble de départ (i.e. un ensemble de protéines de départ biologiquement compatibles).

Pour illustrer l'algorithme de programmation dynamique cherchant un chemin colorful de poids minimum, considérons le même graphe que dans l'exemple précédent, auquel nous avons cette fois assigné aléatoirement quatre couleurs différentes ●, ●, ●, ○ :



L'algorithme de programmation dynamique est initialisé de la manière suivante :

$$W(v_1, \{●\}) = 1, \quad W(v_1, \{○\}) = 1, \quad W(v_2, \{●\}) = 2, \quad W(v_2, \{●\}) = 2,$$
$$W(v_3, \{●\}) = 1, \quad W(v_3, \{●\}) = 2, \quad W(v_3, \{○\}) = 1, \quad ...,$$

De manière analogue à l'algorithme naïf, l'algorithme détermine les valeurs de $W(v, S)$ pour les ensembles S à deux couleurs

[3]Pour s'en convaincre, les applications utilisent des valeurs en pratique de $k = 10$ et $n = 6000$, ce qui donne $6000^{10} \simeq 6 \times 10^{37}$

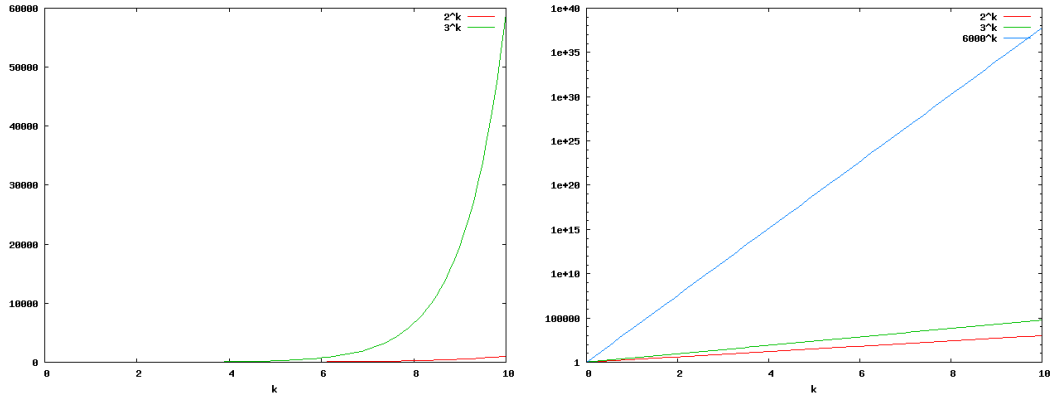


FIG. 4.3 – Illustration du passage de complexité de n^k (version naïve) à 2^k (version color-coding). La figure de gauche montre les courbes de 2^k et 3^k, illustrant l'explosion rapide. Sur la figure de droite, l'échelle est logarithmique. On montre la courbe de 2^k ainsi que 6000^k, valeur classique pour n dans nos réseaux. On remarque que pour $k = 10$ l'explosion est significative.

$$W(v_1, \{\bullet, \bullet\}) = 3, \quad W(v_1, \{\bullet, \circ\}) = 3, \quad W(v_1, \{\bullet, \circ\}) = 2, \quad W(v_2, \{\bullet, \circ\}) = 3,$$
$$W(v_2, \{\bullet, \circ\}) = 4, \quad W(v_3, \{\bullet, \circ\}) = 2, \quad W(v_3, \{\bullet, \circ\}) = 3, \quad W(v_4, \{\bullet, \bullet\}) = 2,$$
$$W(v_4, \{\bullet, \bullet\}) = 3, \quad W(v_5, \{\bullet, \bullet\}) = 3, \quad W(v_5, \{\bullet, \circ\}) = 3, \quad W(v_5, \{\bullet, \bullet\}) = 3.$$

Finalement, l'algorithme continue avec les ensembles de trois couleurs :

$$W(v_1, \{\bullet, \bullet, \circ\}) = 4, \quad W(v_2, \{\bullet, \bullet, \circ\}) = 5, \quad W(v_3, \{\bullet, \bullet, \circ\}) = 4,$$
$$W(v_4, \{\bullet, \bullet, \bullet\}) = 4, \quad W(v_5, \{\bullet, \bullet, \circ\}) = 4.$$

Lors de la dernière itération, l'algorithme trouve que le chemin colorful de poids minimum de taille quatre est de poids 4, ce qui est différent de l'algorithme naïf. Ceci est du au fait que trouver un chemin colorful de poids minimum n'est pas la même chose que de trouver un chemin de poids minimum. Il faut donc colorier plusieurs fois de manière aléatoire le graphe pour ne pas manquer des "bons" chemins.

Avant de discuter du problème des coloriages multiples aléatoires, considérons le temps d'exécution d'un essai. Celui-ci est de $O(2^k km)$. En effet, dans la table de programmation dynamique, il y a $2^k n$ cases : pour les n sommets, il y a deux choix (présence ou non) pour chacune des k couleurs. Pour chaque case, le calcul est au pire un minimum sur tous les arcs du graphe. Nous sommes bien en présence d'un algorithme de complexité paramétrée : la complexité est exponentielle en k mais linéaire en la taille de l'entrée. La Figure 4.3 illustre le gain de complexité entre la version naïve et la version color-coding.

L'assignement aléatoire des couleurs n'implique bien sûr aucune garantie sur le fait que le meilleur chemin recherché soit colorful. Plus précisément, il y a k^k manières d'assigner arbitrairement k couleurs aux sommets d'un chemin de taille k (on peut choisir n'importe quelle couleurs pour chaque noeud du chemin). Il y a $k!$ manières d'assigner des couleurs au chemin tel qu'aucune couleur n'apparaisse plus d'une fois (k choix de couleurs pour le premier noeud, puis $k-1$ pour le second, etc.). Ce qui donne la probabilité qu'un chemin soit colorful pour un essai donné égal à $\frac{k!}{k^k} > e^{-k}$. Il faut, par conséquent, appliquer l'algorithme un nombre suffisant de fois, avec un coloriage de graphe aléatoire, en sachant que la probabilité que l'algorithme ne trouve pas un chemin après $e^k \ln \frac{1}{\epsilon}$ essais est d'au plus ϵ, avec $\epsilon \in [0, 1]$. La combinaison de ces deux informations implique une complexité en $O(|ln\epsilon|.5.44^k km)$.

Nous avons donc vu que le color-coding améliore considérablement la complexité pour résoudre le problème de recherche de chemin de poids minimum. Les auteurs s'inquiètent assez peu du fait que l'algorithme soit aléatoire : le facteur d'erreur est logarithmique et peut-être choisi très petit sans perdre en efficacité. De plus, Alon et al. prouvent que l'algorithme peut être *dérandomisé*,

c'est à dire transformé en algorithme déterministe. Toutefois, leurs résultats restent inapplicables en pratique et ne présentent qu'un intérêt théorique.

Après avoir présenté le color-coding, Scott et al. proposent des extensions sous des contraintes biologiques. En particulier, ils montrent qu'en utilisant le même algorithme de programmation dynamique et en restant dans le même ordre de complexité, ils peuvent imposer un type de protéine apparaissant dans le chemin et forcer l'ordre des protéines dans le chemin selon leurs types. De plus, les auteurs proposent deux structures de recherche, plus complexes que les chemins : les arbres et les Graphe Série-Parrallèle, ceci dans le but de "capturer" d'autres chemins biologiques. Pour plus d'informations se référer à [46].

Scores Les poids sur les arrêtes du graphes sont estimés à partir de probabilités selon trois variables expérimentales. De plus, ils utilisent des données d'entraînement : les exemples positifs choisis depuis une base de données "référence" (MIPS) [39], les exemples négatifs provenant d'interactions choisies au hasard.

Ils donnent ensuite deux mesures de qualités à un chemin trouvé (de poids w). La première mesure est le pourcentage de chemins dans un réseau aléatoire qui ont un poids inférieur à w (un réseau aléatoire étant construit depuis le réseau original en mélangeant les flèches et les poids). La seconde mesure cherche à trouver l'enrichissement biologique d'un chemin, à trouver des fonctions communes. Cette méthode est introduite par NetworkBLAST [49]. Pour ce faire, ils associent les protéines à des processus biologiques connus, avec les annotations de Gene Ontology [4] (i.e. relations structurées telles "une autruche est un oiseau, un oiseau est un ovipare").

Résultats Ils appliquent leur algorithme sur le réseau de la levure. Les "meilleurs chemins" trouvés sont fonctionnellement enrichis et ont de meilleurs scores que sur les réseaux aléatoires. De plus, il retrouve des chemins connus de la littérature.

Synthèse et discussions Cet article concoit une des premières applications du color-coding, permettant de trouver des chemins de longueurs k en un temps raisonnable, avec k pouvant aller jusqu'à la dizaine. Les auteurs proposent de plus des extensions à cet algorithme sous des contraintes biologiques. Ils ajoutent des possibilités dans l'ordre des protéines et introduisent de nouvelles structures.

Les applications des auteurs n'utilisent pas ces nouvelles structures, nous n'avons donc pas d'expériences prouvant leur intérêt réel. Cet algorithme pourrait être utilisé sur d'autres réseaux que sur celui des interactions entre les protéines. Enfin, cet algorithme pourrait être utilisé dans le cadre d'alignement entre plusieurs espèces.

Hüffner et al. (2007)

Une analyse théorique montre que le color-coding est plus efficace qu'un algorithme naïf de programmation dynamique. Cependant, le facteur exponentiel de 5.44^k dans la complexité pire cas limite cet algorithme à des valeurs de k inférieures à dix. Hüffner et al. [27] présentent en 2007 quelques améliorations destinées à diminuer ce facteur pire cas.

Accélération du pire cas par ajout de couleurs Pour trouver un chemin de taille k avec les noeuds ayant des couleurs différentes, il faut clairement au moins k couleurs pour colorier aléatoirement le graphe. Les auteurs cherchent à savoir ce qui peut se passer si on augmente le nombre de couleurs. Dans un premier temps, utiliser plus de couleurs implique qu'un chemin dit intéressant à plus de chance d'être colorful. Ainsi, moins d'essais devraient être effectués. D'un autre côté, en utilisant plus de couleurs, l'algorithme par programmation dynamique prend plus de temps et occupe plus d'espace mémoire pour un essai donné. Les auteurs montrent qu'en utilisant $1.3k$

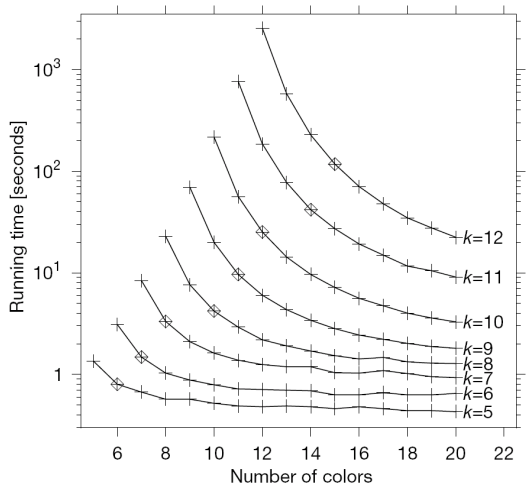


FIG. 4.4 – Ce graphique illustre le gain de temps de calcul en fonction du nombre de couleurs utilisées pour un k donné. Les valeurs sont le temps pour trouver les vingt chemins de poids minimum dans le réseau de la levure de Scott et al. [46]. On remarque que l'augmentation du nombre de couleurs accélère jusqu'à deux ordres de grandeur. Ce graphique est extrait de [27].

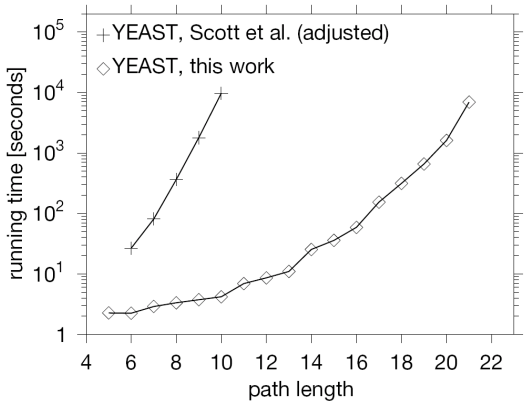


FIG. 4.5 – La figure représente les temps de calculs sur la levure reportés par l'article de Scott et al. [46] (les croix) ainsi que ceux mesurés avec l'implémentation de Hüffner et al. (les carrés). Ce graphique est extrait de [27].

couleurs, l'algorithme peut obtenir une complexité de $O(|ln\epsilon|.4.32^k m)$. La Figure 4.4 illustre le gain de temps en fonction du nombre de couleurs pour un k donné.

En plus de l'augmentation du nombre de couleurs, les auteurs proposent un ensemble de petites astuces pour accélérer le temps de calcul, mais de manière non-exacte, ce sont des heuristiques. On peut citer par exemple le fait de ne pas calculer les chemins dont le poids partiel dépasse déjà le poids du meilleur chemin connu à cet instant. Ils proposent également des stockages des ensembles de couleurs par *bits*, cumulés avec des arbres de Patricia[4].

Résultats Afin de pouvoir comparer leur algorithme à celui de Scott et al., les auteurs utilisent le même réseau de test, celui de la levure. Ils parviennent à accélérer significativement le temps de calcul à k égal. D'autre part, ils peuvent augmenter la taille des chemins jusqu'à 20, alors que Scott et al. étaient limités à 10. La Figure 4.5 illustre ces améliorations.

[4] `http://en.wikipedia.org/wiki/Radix_tree`

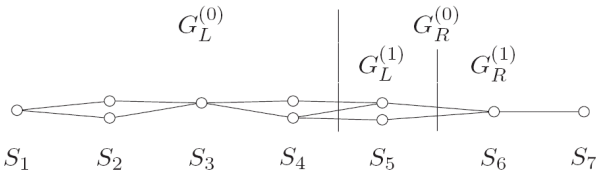


FIG. 4.6 – Illustration d'un *(k,t)-path*, avec $k = 10$ (nombre de protéines, nombre de noeuds), $t = 2$ (nombre de protéines maximum dans un ensemble), $l = 7$ (nombre d'ensembles). Seul les arcs entre les ensembles adjacents sont représentés, il n'y a aucune restriction concernant les autres arcs. Une itération de l'algorithme est illustrée, dans laquelle le graphe G est partitionné aléatoirement en deux parts $G_L^{(0)}$ et $G_R^{(0)}$, qui divisent également le (k,t)-path en deux structures plus petites d'à peu près la même taille. $G_R^{(0)}$ est partitionné par la suite en $G_L^{(1)}$ et $G_R^{(1)}$, contenant respectivement les ensembles $[S_5]$ et $[S_6, S_7]$. La Figure est extraite de [38].

Synthèse et discussions Les auteurs proposent un ensemble d'astuces pour accélérer le color-coding et augmenter la taille des chemins recherchés. Cependant, les tests sont effectués uniquement pour les recherches de chemins. Il serait intéressant de déterminer si ces améliorations sont aussi valides pour d'autres structures recherchées (arbres, graphes, etc.).

Lu et al. (2007)

En 2007, Lu et al. [38] ont proposé deux nouvelles améliorations pour la résolution du problème de recherche de meilleurs chemins dans un graphe. En poursuivant les idées de Scott et al., ils introduisent une nouvelle structure afin d'obtenir plus de flexibilité. De plus, pour la recherche de chemin, la complexité pire cas est meilleur qu'avec le color-coding, en utilisant un algorithme "diviser pour régner" randomisé. L'algorithme prend un graphe en **entrée** et propose leur structure avec le meilleur score en **sortie**.

Algorithme Les auteurs définissent une nouvelle structure non-linéaire, le *(k,t)-path*. Deux ensembles de sommets S_1 et S_2 du graphe G sont dit *connectés* si

- chaque sommet de S_1 est adjacent à au moins un sommet de S_2.
- chaque sommet de S_2 est adjacent à au moins un sommet de S_1.

Un *(k,t)-path* est un sous-graphe de G avec k sommets, partitionné en l sous-ensembles $S_1, S_2, ..., S_l$ tel que :

- le nombre de sommets dans chaque sous-ensemble est inférieur à t.
- S_i est connecté à S_{i+1} pour $1 \leq i < l$.

Les auteurs justifient l'intérêt biologique de la recherche de chemin prenant la forme de cette structure en avançant que les protéines dans le même ensemble pourraient avoir un rôle biologique similaire. Une illustration de cette structure est proposée dans la Figure 4.6.

Leur algorithme cherche **un** meilleur *(k,t)-path* en utilisant un algorithme diviser pour régner. Ainsi, ils vont "couper" en deux le graphe aléatoirement, un nombre $2,51 \times 2^k$ fois (aucune justification de ce nombre n'est donnée). Ils obtiennent alors deux sous-problèmes de recherche de *(k',t)-path*, avec k' environ égal à $k/2$. Ils continuent à partitionner le graphe récursivement, jusqu'à arriver au cas de base où $k \leq t$. Arrivé dans ce cas de base, les auteurs prouvent qu'il y a alors au plus $(2n)^k$ *(k,t)-path*, constructibles en $O(t(2n)^t)$. Ensuite, il y a fusion si les deux *(k',t)-path* sont "compatibles", c'est à dire si on peut les connecter, les mettre bout à bout pour former un *(k,t)-path*.

Au final, les auteurs prouvent que l'algorithme retrouve un *(k,t)-path*, s'il existe, avec une probabilité supérieure à 63%. Ainsi, pour obtenir une probabilité d'erreur inférieure à ϵ, il faut exécuter l'algorithme r fois tel que r satisfasse $1/e^r < \epsilon$. (e.g. pour $\epsilon = 0.001$, $r = 7$).

On remarque que pour $t = 1$, la structure est un chemin. Dans ce cas, leur algorithme à une complexité de $O(4^k n^{O(1)})$, ce qui est une amélioration par rapport au $O(5.44^k n^{O(1)})$ de Scott et al.

De plus, la complexité en espace n'est pas exponentielle ($O(nk.log(k)+m)$), ce qui n'est pas le cas du color-coding. Dans le cas général, la complexité en temps est de $O(4^k.n^{2t}.k^{t+log(t+1)+2.92}.t^2)$.

Scores Le graphe d'entrée n'est pas pondéré, la construction des *(k,t)-path* ne prend en compte que la connexité des sommets. Cependant, ils vont donner un score aux *(k,t)-path* trouvés. Pour cela, ils assignent un score sur les **noeuds** prenant en compte l'intérêt biologique, en fonction de leurs voisins et de l'ensemble auquel ils appartiennent. Cependant, ils envisagent d'ajouter un score sur les arcs pour éviter les faux positifs. Le score du *(k,t)-path* est par conséquent la somme des noeuds le composant.

Résultats Ils appliquent leur algorithme sur le réseau de la levure, obtenu depuis DIP. Avec $t = 2$, leur algorithme prend quelques minutes pour $k = 10$, quelques heures pour $k = 11$ et une journée pour $k = 12$. Grâce à leur structure, ils arrivent à détecter un chemin connu dans la littérature biologique qui n'était pas détecté par l'algorithme de Scott et al.

Synthèse et discussions Les auteurs proposent l'utilisation de leur algorithme au sein d'un graphe d'alignement, ce qui permettrait d'utiliser un algorithme de diviser pour régner dans le cadre d'un problème d'alignement de plusieurs réseaux. De plus, selon les auteurs, ce genre d'algorithme serait plus facile à dérandomiser que les algorithmes de color-coding. Il ne proposent cependant aucune solution qui pourrait être applicable en pratique.

Cependant, les auteurs concèdent que l'utilisation de structures complexes se justifie difficilement biologiquement. En effet, il existerait assez peu de chemins prenant cette allure. De plus, l'amélioration de la complexité ne concerne que la borne "pire-cas". Selon leur tests, il n'y aurait pas d'accélération significative du temps de calcul en moyenne.

Toutefois, cet article prouve qu'il est possible d'utiliser un algorithme de diviser pour régner afin de résoudre le problème de recherche de meilleur chemin dans un graphe. Il ouvre ainsi de nouvelles perspectives – peut-on allier les techniques de diviser pour régner avec le color-coding ?

4.2.2 Recherche selon motifs

Dans cette section, on présente des algorithmes cherchant des structures le plus conformes possible à des modèles, appelés requêtes. Celles-ci pourront respecter différentes topologies, allant du chemin au graphe.

QPath par Shlomi et al. (2006)

QPath est un algorithme écrit en 2006 par Shlomi et al. [50], dont le but est de rechercher des chemins respectant un motif dans un graphe. Il prend donc en **entrée** un réseau d'interactions protéiques ainsi qu'une requête sous la forme d'un chemin simple. Il donne en **sortie** le chemin du réseau respectant au mieux la requête. Écrit après PathBLAST, il en reprend les avantages et essaye d'en corriger les inconvénients.

L'algorithme QPath va "aligner" un chemin (requête) avec un réseau, en autorisant plusieurs insertions ou des délétions consécutives ; il existe tout de même une borne sur le nombre total d'in/del dans l'alignement final.

Afin d'améliorer la complexité par rapport à PathBLAST, QPath utilise un algorithme de programmation dynamique, amélioré par le color-coding. Le nombre de couleurs différentes utilisées n'est pas k, longueur du chemin, mais $k + N_{ins}$, avec N_{ins} nombre d'insertions autorisées. En effet, le chemin de sortie doit avoir les k couleurs correspondant à la taille de la requête, plus d'autres protéines *insérées*, qui doivent avoir une couleur différente.

La récurrence de l'algorithme de programmation dynamique consiste à effectuer la "meilleure" opération entre

- avancer dans le réseau et dans la requête
- avancer uniquement dans le réseau : une insertion
- avancer uniquement dans la requête (i.e. ne pas choisir de protéine homologue dans le réseau) : une délétion

tout ceci en gardant le résultat colorful.

La complexité en temps est de $O(2^{k+N_{ins}}.m.N_{del})$, k étant la taille de la requête, m le nombre d'arcs dans le graphe, N_{ins} et N_{del} respectivement le nombre d'insertions et de délétions autorisées. Ceci permet d'obtenir des chemins de longueur au plus 10, ce qui est une avancée par rapport à PathBLAST qui permettait d'obtenir des chemins deux fois plus petit.

Scores Les auteurs attribuent deux scores à un chemin : un score séquence, somme des homologies, mesurant la similarité entre le résultat et la requête, et un score interaction, qui est la somme des poids des arcs composant le chemin.

Résultats Pour évaluer l'utilité de leur algorithme, les auteurs l'appliquent sur les réseaux de la levure et de la mouche, téléchargés depuis DIP en 2004. Tout comme PathBLAST, leur premier test consiste à rechercher les chaines de protéines Kinase. L'algorithme en retrouve deux. Ensuite, ils tentent de caractériser le réseau de la mouche grâce aux résultats trouvés sur la levure. Ils extraient d'abord les meilleurs chemins dans la levure, qu'ils recherchent dans le réseau de la mouche. Dans 63% des cas, ces chemins existent également chez la mouche, avec au plus trois insertions/déletions. De plus, très peu de chemins sont trouvés en interdisant les insertions/deletions, ce qui suggère l'intérêt de leur autorisation. Une procédure équivalente est effectuée depuis des chemins connus de l'homme : l'algorithme en retrouve dans le réseau de la mouche.

Synthèse et discussions Les inconvénients principaux relevés sur PathBLAST semblent être résolus par QPath. En effet, la complexité de PathBLAST limitait la taille des chemins à cinq : QPath recule la limitation à dix. Les protéines apparaissent de manière unique dans les chemins résultats et plusieurs in/dels consécutifs sont autorisés.

Malheureusement, les auteurs ne donnent pas d'indications sur le taux d'insertions par rapport au taux de délétions. L' algorithme suivant va proposer des améliorations quant à la topologie des requêtes d'entrée.

QNet par Dost et al. (2007)

En 2007, Dost et al. [15] proposent une extension à QPath, qui accepte des requêtes sous formes de chemin. QNet lui ajoute la possibilité de donner en entrée des requêtes sous formes d'arbres ou de graphes. Il prend donc en **entrée** un réseau ainsi que la requête et produit en **sortie** la structure la plus proche de la requête.

Algorithme Dans un premier temps, les auteurs expliquent leur algorithme lorsque la requête à la forme d'un arbre. Il est similaire à celui qui était effectué sur les chemins, il utilise également de la programmation dynamique avec color-coding. Il effectue donc une série d'essais, en assignant aléatoirement les $k + N_{ins}$ couleurs sur le graphe. La formule de récurrence consistera donc à conserver le score maximum entre matcher le noeud de l'arbre avec le graphe, insérer un noeud du graphe dans le résultat, deleter un noeud de l'arbre. La complexité d'un essai donné est de $2^{O(k+N_{ins})}.m$, k étant le nombre de noeuds du graphe, ce qui est similaire à la complexité lorsque la requête est un chemin. La probabilité que l'alignement optimal soit colorful est d'au moins $e^{-k-N_{ins}}$, ce qui donne une complexité totale de $2^{O(k+N_{ins})}m\ln(1/\epsilon)$, pour avoir une probabilité de succès de $1-\epsilon$.

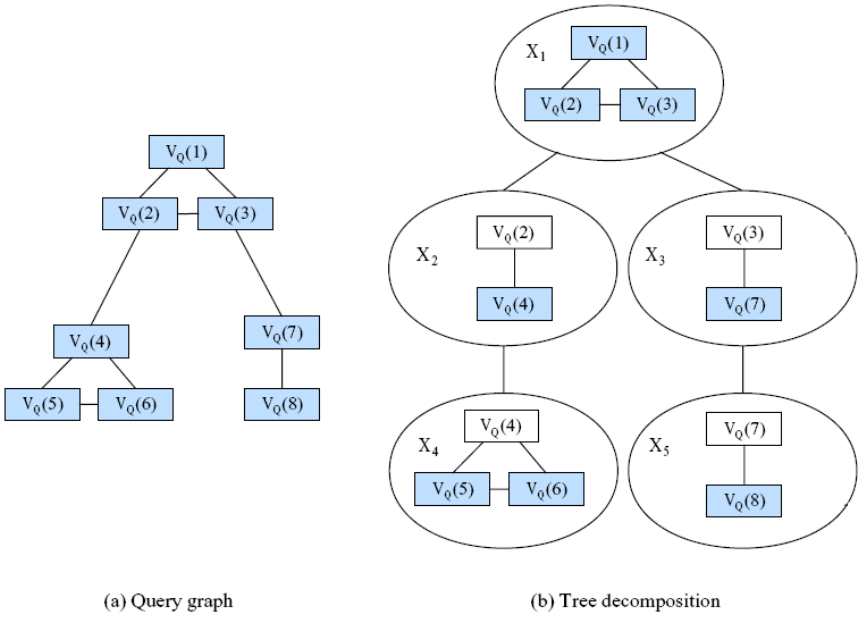


FIG. 4.7 – a) Un graphe de requête avec une treewidth de deux. b) Une décomposition arborescente du graphe de requête tel que chaque “super-noeud” n’a pas plus de trois noeuds de requête associés. Figure provenant de [15].

Dost et al. proposent de continuer la généralisation de la topologie des requêtes en autorisant les graphes. Cependant, ces derniers doivent être "proches" de la forme d’un arbre : ils utilisent la notion du *treewidth*. Intuitivement, la treewidth d’un graphe indique la distance à laquelle il est d’un arbre, le nombre d’"opérations" nécessaires pour le faire devenir un arbre. Par exemple, un arbre a une treewidth de 1. La treewidth maximale pour un graphe de n sommets est donc de $n-1$, valeur atteinte avec une clique de taille n.

Les auteurs font une *décomposition arborescente* (X, T) du graphe requête $G_Q = (V_Q, E_Q)$ telle que $T = (I, F)$ est un arbre binaire (i.e. deux fils) enraciné, et $X = \{X_i \subseteq V_Q : i \in I\}$ est une collection de sous-ensembles de V_Q, telle que $\bigcup_{i \in I} X_i = V_Q$ et que les deux conditions suivantes soient satisfaites :

1. pour chaque arc $(u, v) \in E_Q$, $\exists i \in I$ tel que $u, v \in X_i$.
2. si $i, j, k \in I$ et j est sur le chemin allant de i à k dans T, alors $X_i \cap X_k \subseteq X_j$.

La *treewidth* de la décomposition arborescente est $\max_{i \in I} |X_i| - 1$. Un exemple d’un graphe et de sa décomposition arborescente est donné en Figure 4.7.

Une fois la décomposition arborescente effectuée, l’algorithme peut réutiliser l’algorithme précédent utilisé lorsque la requête est un arbre. Il faudra alors maintenir une liste des sous-alignements optimaux. Pour l’implémentation, les auteurs doivent borner la treewidth de l’arbre par t : la complexité en temps est de $2^{O(k)} n^{t+1}$, k étant le nombre de noeuds dans le graphe et t la borne du treewidth.

L’article présente également une heuristique pour accélérer l’algorithme, en diminuant le nombre d’itérations. Avec cette heuristique, la probabilité d’obtenir un chemin colorful est augmentée. Ainsi, moins d’essais seront nécessaires.

Scores Le poids des arcs correspond à la probabilité de l’interaction. Le score d’une structure est la somme des poids des arêtes la composant.

Résultats Les auteurs n’ont implémenté que la partie acceptant les arbres en requête, la partie graphes reste donc uniquement théorique.

Dans un premier temps, les auteurs appliquent QNet sur la levure avec pour requête un ensemble d'arbres synthétiques (i.e. arbres choisis hasards dans le réseau de la levure, avec k, nombre de noeuds, allant de 5 à 9). Ils perturbent ces arbres avec jusqu'à deux insertions/délétions, et regardent si l'algorithme retrouvent ces arbres. Celui-ci ne prend que quelques secondes lorsque $k = 9$.

Dans un second temps, ils font des requêtes sur des chemins connus de la levure et de l'homme qu'ils recherchent dans le réseau de la mouche. Il s'agit de la Kinase, connue pour être présente dans plusieurs espèces. L'algorithme arrive a retrouver ces chemins.

Synthèse et discussions QNet est une nouvelle étape dans l'évolution des algorithmes de recherche de motifs. Il apporte deux nouvelles structures : les arbres et les graphes, toujours en se servant du color-coding. Cependant, les graphes de requêtes sont limités par la treewidth : la complexité est exponentielle en ce paramètre. Ainsi, on ne peut pas donner en requête n'importe quel graphe. Dans cet article, aucun test n'est effectué lorsque la requête est un graphe : on n'a donc pas d'indication sur la limite de treewidth en pratique, ni de confirmation sur l'intérêt biologique de ceux-ci.

Les auteurs considèrent qu'il faudrait développer des fonctions de scores plus appropriées pour une identification plus directe des chemins conservés. On pourrait trouver des algorithmes acceptant des structures encore plus générales pour la requête.

4.3 Remarques et perspectives

A la suite de cet état de l'art, quelques questions restent ouvertes :

- L'amélioration d'Hüffner et al. consistant à augmenter le nombre de couleurs pour accélérer la recherche de chemins est-elle toujours valide avec les structures différentes (comme les arbres) proposés par Scott et al. ? Si oui, dans quelle mesure ?
- Que vaut l'alignement de deux réseaux en utilisant la technique du color coding ? Si les résultats sont satisfaisant, peut-on réutiliser l'idée d'aligner successivement les réseaux les plus proches dans l'arbre phylogénétique pour aligner k réseaux ?
- Peut-on diminuer la limitation sur les graphes lors de la recherche de motif ? Plus largement, peut-on ajouter de nouvelles structures plus générales dans cette recherche de motifs ?
- Les techniques utilisant le diviser pour régner ont une complexité pire des cas meilleure que le color-coding. Qu'en est il en pratique ? Est-il possible d'allier les qualités de chacune des techniques ? (Kneis et al. parlent de divide-and-color [33])
- Existe t-il des méthodes assez efficaces pour dérandomiser les algorithmes de color-coding et les utiliser en pratique ? Quand est-il de la dérandomisation des algorithmes diviser pour régner ?
- Les recherches actuelles sont "locales", sur des petites structures, du à la modularisation des réseaux. Qu'en est il de l'organisation globale des réseaux ? Peut-on avoir un regard et des perspectives plus globales ?
- Les réseaux de protéines ne sont pas statiques dans le temps. Est-il possible de prendre en compte ces évolutions temporelles ?
- Peut-on appliquer les structures (triangulisation, structures d'ensemble, de multi-ensemble, etc.) et les problèmes (GRAPHMOTIF, MAXMOTIF, MINDELETION, etc.) connus de la théorie des graphes sur les réseaux d'interactions de protéines ?

	Entrée	Sortie	Algorithme	Scores	Complexité	Résultats
PathBlast (2003)	2 RIP	Un chemin (taille 5)	2 passes. Graphe d'alignement (gaps et mismatchs non consécutifs) puis recherche d'un chemin dans sous-graphes acycliques (ordonnancement aléatoire et supression d'arcs avant)	Pas de poids. Probabilités	$k!n^{O(1)}$ temps, $O(m+n)$ espace	5 chemins de la levure dans la bactérie. Duplication dans la levure
MaWish (2005)	2 RIP	Sous-réseaux conservés	Heuristique. Graphe d'alignement puis détection des réseaux très connectés : agrandissement glouton des sous-réseaux avec les protéines les mieux connectées	Selon l'évolution des réseaux (empirique, pas de modele proba)	$O(n.m)$	Sous-réseaux conservés entre homme et souris
NetworkBlast (2005)	k RIP (3 en pratique)	Chemins et clusters conservés	Graphe d'alignement (noeuds avec k protéines). Heuristiques avec graines (recherche + extension)	Scores sur sous-graphes	Non précisée	Interprét. bio supplémentaires avec k réseaux. Identification de structures communes à la levure, le ver et la mouche.
NetworkBlast-M (2008)	k réseaux	Collection de sous-réseaux	Nouvelle structure : graphe à k-couches + ensemble d'arcs "inter-couches". Recherche comme NetworkBlast.	Somme sur les layers	Non précisée	10 réseaux en minutes, plus rapide que NB, plus précis que Graemlin

FIG. 4.8 – Synthèse algorithmes d'alignements

	Problème	Entrée	Sortie	Algorithme	Scores	Complexité	Résultats
Scott et al. (2005)	Meilleur chemin	1 RIP	1 chemin (lg 10), 1 arbre, 1 2SPG	Color-coding + extensions (ordre & structures)	Poids selon probabilités	$O(2^k km)$ chaque essai $\rightarrow$ $5.44^k n^{O(1)}$ temps et $O(nk2^k + m)$ espace	Chemins de la littérature retrouvés
Hüffner et al. (2007)	Meilleur chemin	1 RIP	Un chemin	Acceleration du color-coding avec astuces (augmentation nombre de couleurs (1.3) -> moins d'essais)	Poids selon probabilités	$4.32^k.m$	Accélération des calculs, augmentation de la taille de k
Lu et al. (2007)	Meilleur chemin	1 RIP	Un (k,t)-path	Diviser pour régner. Coupe en deux le graphe et recherche un (k',t)-graph jusqu'a k<=t puis fusionne	Somme des poids sur les sommets	$O(4^k n^{2t}$ $k^{t+\log(t+1)+2.92}$ $t^2)$ temps, $O(n^t k \log k + m)$ espace. Chemins : $4^k n^{O(1)}$	Meilleur complexité pire cas, mais moins bon moyenne. Structure peu répandue mais trouve un nouveau chemin dans la levure
QPath (2006)	Recherche de motif	1 chemin, 1 RIP	1 chemin (lg 10)	Color-coding avec autorisarion de in/dels consécutifs	Poids selon probabilités	$2^{O(k+N_{ins})} m N_{del}$	Recherche dans la mouche et l'homme de chemins de la levure
QNet (2007)	Recherche de motif	1 arbre ou graphe et un RIP	Selon le motif	Color-coding. Décomposition du graphe en arbres + heuristique	Poids selon probabilités	Chaque essai : $2^{O(k+N_{ins})} m$ Graphe : $2^{O(k)} n^{t+1}$	Retrouve des chemins de la mouche et de la levure

FIG. 4.9 – Synthèse algorithmes de recherche de chemins

Chapitre 5

Notre algorithme

5.1 Idée générale et notations

Le but de notre stage est de fournir un algorithme capable de détecter un motif sous forme de graphe (la requête) dans un réseau d'interactions protéiques.

Comme indiqué dans le Chapitre 4, QPath [50] est un algorithme exact de programmation dynamique permettant de rechercher un chemin dans un réseau. QNet [15] permet d'utiliser des requêtes sous forme d'arbres. Les auteurs proposent également une manière théorique de rechercher des graphes en utilisant leur structure arborescente sous-jacente. En recherchant des graphes dans le réseau, notre algorithme propose une extension à QPath et une alternative à QNet.

Pour rechercher des graphes dans le réseau, QNet effectue une décomposition arborescente du motif ; ainsi, leur complexité de $2^{O(k)}n^{t+1}$, où k est la taille du motif et n la taille du réseau, est dépendante de t, la *largeur d'arborescence* de la requête. Une définition d'une décomposition arborecente est donnée dans la Section 4.2.2. On rappelle que cela consiste en la décomposition d'un graphe en séparateurs (i.e. sous-ensemble de sommets dont la supression rend le graphe non connexe) connectés dans un arbre. Ces décompositions ne sont pas uniques : la largeur d'arborescence d'un graphe est le minimum (parmis toutes les décompositions) de la taille moins un du plus grand sous-ensemble de sommets. Le calcul de cette largeur d'arboresence est NP-Difficile.

Un autre inconvénient de QNet est que lorsque le motif est un graphe, leur algorithme peut subir des pertes (i.e. on ne peut pas retrouver le graphe original depuis l'arbre résultat), car les auteurs suppriment des arcs, donc de l'information. Ils doivent tester l'algorithme avec plusieurs décompositions, elles ne sont pas uniques.

Enfin, les auteurs n'ont implémenté qu'une version de QNet supportant des motifs sous forme d'arbres. Ils écrivent cependant dans l'article qu'ils comptent écrire la version acceptant les graphes.

Si QNet propose un algorithme exact recherchant des arbres dans un réseau, une extension logique consiste à y rechercher des graphes. QNet propose donc une solution, seulement théorique, dépendant de la largeur d'arborescence d'un graphe et avec perte. C'est pourquoi nous avons réfléchi à un algorithme exact résolvant ce problème.

Afin de transformer le graphe en arbre et de pouvoir utiliser QNet, nous proposons une solution consistant à dupliquer les noeuds du graphe formant des cycles. Plusieurs noeuds dupliqués dans l'arbre représenterons en fait un seul et même noeud dans le graphe de départ. Sur la partie gauche de la Figure 5.1, on voit qu'en haut, le graphe comportait un cycle, qui a été "cassé" sur l'arbre en bas à gauche, en dupliquant le noeud q_1. Une fois le motif ne comprenant plus de cycles, on peut s'inspirer de l'algorithme de QNet afin de rechercher cet arbre dans le réseau. Cependant, ces noeuds dupliqués imposent de modifier l'algorithme afin de gérer ces noeuds : c'est ce que nous expliquons dans la Section 5.2. Nous verrons que le choix des noeuds à dupliquer est fondamental et modifie grandement la complexité générale de l'algorithme : nous exposons les différents choix et possibilités dans la Section 5.3

Le problème de recherche de motifs dans un réseau étant NP-Complet, la complexité d'un algorithme fournissant une solution exacte sera forcement exponentielle. Cependant, nous proposerons

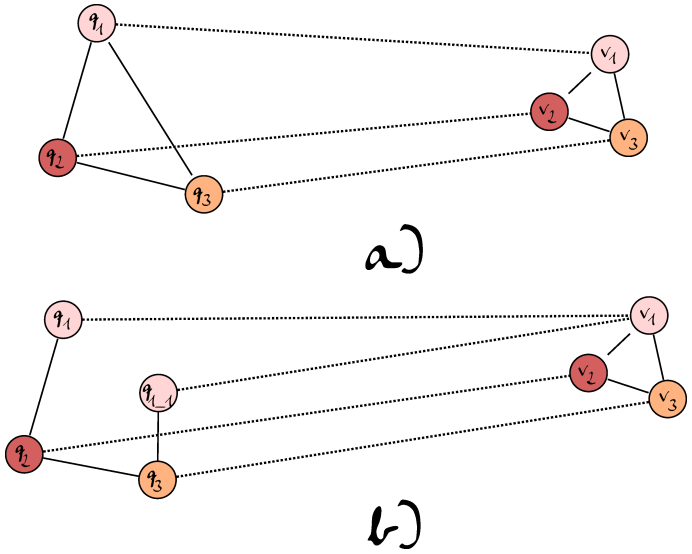


FIG. 5.1 – En a), le motif est un graphe. Il y a une bijection entre les noeuds du graphe et ceux du réseau. En b), le motif est un arbre, comportant deux noeuds dupliqués, symbolisant la même protéine. Il doivent être aligner avec la même protéine du réseau, c'est une sujection.

une complexité paramétrée par un "paramètre" différent de celui de QNet : le nombre minimal de sommets à supprimer pour rendre le graphe acyclique. Si l'on peut comparer théoriquement ces deux paramètres et donc les bornes pire cas, l'idéal serait de pouvoir comparer expérimentalement les deux algorithmes. Néanmoins, cela sera impossible étant donné que les auteurs de QNet n'ont pas implémenté celui-ci.

Indiquons maintenant les notations que nous utiliserons dans ce chapitre. Nous disposons d'un réseau d'interactions de protéines, noté $G_N = (V_N, E_N, w)$, représenté par un graphe non orienté, pondéré sur les arcs par la fonction $w : E_N \to \mathbb{R}$, représentant la probabilité de l'interaction entre deux noeuds du réseau.

On note $G_Q = (V_Q, E_Q)$ le motif de départ, également appelé requête. C'est un graphe orienté à k noeuds (i.e. $|V_Q| = k$), pouvant contenir des cycles.

On tranforme ce graphe en un arbre orienté $T_Q = (V'_Q, E'_Q)$, pouvant posséder des noeuds dupliqués. Les noeuds dupliqués d'une même famille représentent en fait une seule protéine de V_Q.

La fonction $h(q, v)$ représente un score de similarité entre un noeud de la requête $q \in V_Q$ et un noeud du réseau $v \in V_N$. Dans notre contexte, les noeuds représentent des protéines ; ainsi la similarité entre les protéines est calculée selon la similarité de leurs séquences d'acides aminés. On dira qu'un noeud q est *homologue* à un noeud v si leur score de similarité dépasse un certain seuil fixable.

Un *alignement* entre la requête T_Q et le réseau G_N est défini par : (i) un sous-graphe $G_A = (V_A, E_A)$ de G_N, (ii) une bijection entre les noeuds de G_Q et G_A, qui devient une surjection entre T_Q et G_A. En effet, dans ce second cas, l'ensemble des noeuds dupliqués d'une même famille sont une seule et même protéine et doivent donc tous être alignés avec la même protéine de G_N. La Figure 5.1 illustre ceci.

On dit qu'un noeud de la requête est *aligné* si son homologue dans le réseau apparait dans le résultat. Certaines protéines pourront être *insérées* dans G_A : ce sont des protéines qui n'étaient pas présentes dans G_Q. Des noeuds de la requête peuvent être *delétées*, auquel cas ils n'apparaissent pas dans le résultat. La délétion n'est autorisée que pour les noeuds ayant un degré deux, a l'instar de QNet. En pratique, le nombre de délétions (resp. d'insertions) sera limité par N_{del} (resp. N_{ins}), et son score de pénalitée sera symbolisé par δ_d (resp. δ_i).

Le problème de recherche de motifs dans un graphe peut alors être défini de cette façon : étant

donnés une requête G_Q, un réseau G_N, une fonction de similarité h, des scores de pénalités pour les délétions et les insertions, trouver un graphe d'alignement G_A dans G_N avec un score maximal. On rappelle que le score d'alignement est la somme : (i) des scores de similarité entre les noeuds alignés, (ii) des poids des arcs dans le sous-graphe d'alignement, (iii) d'un score de pénalité δ_d pour chaque délétion de noeud, (iv) d'un score de pénalité δ_i pour chaque insertion de noeud.

Ce problème est NP-Complet. Cependant, sa complexité est grandement influencée par la topologie de la requête. Pour rendre ce problème calculable, QNet utilise le *color-coding*, dont le principe pour la recherche de meilleurs chemins dans un réseau a été expliqué dans la Section 4.2.1. Cette technique permet de passer d'une complexité de $O(n^k)$ à $O(2^k)$, où n est la taille du réseau et k est la taille du chemin recherché.

QNet adapte cette technique à son algorithme de requête. Construire un alignement optimal consiste à étendre des sous-alignements optimaux, par programmation dynamique. Ajouter un noeud à l'alignement peut être effectué uniquement si ce noeud n'est pas déjà compris dans le sous-alignement. Ainsi, naïvement, chaque sous-alignement potentiel doit maintenir une liste d'au plus k noeuds déjà alignés. Ceci mène à $O(n^k)$ alignements potentiels. Avec le color-coding, on colorie a priori chaque noeud du réseau au hasard parmi k couleurs. On recherche alors un alignement colorful (i.e. comprenant une et une seule fois chacune des k couleurs). Ainsi, nous avons besoin de maintenir uniquement une liste des couleurs utilisées, de taille $O(2^k)$. Le calcul donne un résultat correct seulement si l'alignement est colorful, ce qui arrive avec une probabilité de $\frac{k!}{k^k} \simeq e^{-k}$. Alors, si le calcul est répété $\log(\frac{1}{\epsilon})e^k$ fois, on obtient un alignement optimal avec une probabilité d'au moins $1 - \epsilon$, pour tout ϵ.

5.2 Recherche de motifs

Dans cette section, nous expliquons la partie de notre algorithme consistant à rechercher un motif dans un réseau d'interactions protéiques. Nous considérons ici que le motif T_Q est un arbre, pouvant avoir plusieurs familles de noeuds dupliqués (en fait, une famille de noeuds dupliqués correspond à une seule protéine), nécessaires à la transformation préalable du graphe en arbre. L'ensemble des noeuds de cette famille devra être aligné de la même manière (i.e. tous deletés ou tous alignés avec le même noeud de G_N). L'algorithme devra retrouver T_Q dans le réseau, en autorisant des insertions et des délétions. Le résultat produit sera un graphe d'alignement G_A, sous-graphe de G_N.

5.2.1 Une solution par programmation dynamique

L'algorithme d'alignement d'arbre avec le réseau de QNet consiste à regarder pour chaque noeud de la requête, si le score de l'alignement est amélioré soit en alignant ce noeud et un noeud du réseau (un match), soit en insérant un noeud du réseau (pour atteindre un match plus loin), soit en supprimant ce noeud (aucun alignement n'est possible). Ceci est effectué pour une coloration du réseau donnée, et est répété pour l'ensemble des essais de colorations.

Notre algorithme reprend ce principe présenté par QNet. Cependant, dans notre cas, il existe des noeuds dupliqués. Nous devons donc adapter la récurrence.

En effet, chaque famille de noeuds dupliqués symbolise une seule et même protéine. Ainsi, tous les noeuds de cette famille doivent être alignés avec la même protéine dans le réseau (ou tous être deletés). Pour prendre en compte cela, nous décidons d'assigner au préalable (i.e. avant la récurrence), la protéine du réseau qui va être alignée par la famille. Pour tester toutes les possibilités, nous devons effectuer tous les assignements possibles, chaque famille avec chaque noeud du réseau. Une fois l'assignement effectué, nous exécutons la récurrence modifiée afin de prendre en compte l'assignement des familles. La fonction indiquant cet assignement sera noté AF (pour Assignement des Familles), tel que $AF : famille \rightarrow v \in V_N \cup delete$, où $famille$ représente une famille de noeuds dupliqués et $delete$ symbolise le fait de supprimer du résultat cette famille..

L'utilisation du color-coding nous impose donc d'utiliser un ensemble de couleurs, noté S_E, utilisées successivement lors d'un match ou d'une insertion. Pour avoir une réponse de taille suffisante, il faut donc mettre k couleurs dans l'ensemble, où k est le nombre de noeuds du motif. On doit également ajouter N_{ins} couleurs, correspondant aux éventuelles insertions de noeuds non prévues par rapport au motif.

Notre gestion des noeuds dupliqués nous oblige à utiliser un *multi-ensemble* S de couleurs (i.e. les éléments de l'ensemble peuvent apparaitre plusieurs fois) plutôt qu'un ensemble classique comme dans QNet. En effet, l'ensemble des noeuds appartenant à la même famille doivent utiliser la même couleur c : il faut donc que notre multi-ensemble de couleurs utilisables comprenne un nombre suffisant de cette couleur c.

On doit préalablement déterminer une racine arbitraire $root$, de degré 1. Pour chaque noeud $q \in V'_Q$, on notera ses fils par $q_1, ..., q_{n_q}$.

Le score maximum de l'alignement du motif avec le réseau est calculé de cette manière :

Algorithme 1 :

$T_Q = (V'_Q, E'_Q)$;
$G_N = (V_N, E_N)$;
$v \in V_N$;
$AlignementMax \leftarrow -\infty$;
$k \leftarrow |V'_Q|$;
$S_E \leftarrow$ ensemble de $k + N_{ins}$ couleurs;
pour chaque $\log(\frac{1}{\epsilon})e^k$ *colorations* **faire**
    pour chaque *Assignements* AF **faire**
        $S \leftarrow$ multi-ensemble de couleurs compatible avec AF;
        $Alignement \leftarrow \max_v W^M(root, v, S_{M-E}, 1, AF)$ + Score(AF));
        si $Alignement > AlignementMax$ **alors**
            $AlignementMax \leftarrow Alignement$;
            Sauvegarde $Coloration$, AF, v, $S_{MultiEnsemble}$;
        fin
    fin
fin
Chargement $Coloration, AF, v, S_{MultiEnsemble}$;
Backtrack($root, v, S_{MultiEnsemble}, 1, AF$);

On appelle $W^M(q, v, S, j, AF)$ le score maximal de l'alignement de l'arbre enraciné en q, qui est le noeud aligné avec v, et les sous-arbres enracinés par chacun des j premiers fils de q. L'alignement de cet arbre utilise les couleurs présentes dans S et l'assignement de familles indiqué dans AF.

La fonction Score(AF) renvoie le score d'alignement pour les familles de noeuds dupliqués (i.e. le score des matchs et des éventuelles délétions, pour chaque famille).

Score(AF) = $\sum_{(famille,v)\in AF} h(famille, v)$, avec $h(famille, v) = \delta_d$ si $AF(famille) = delete$, où $v \in V_N$ et où $famille$ représente une famille de noeuds dupliqués de la requête. Ce score est de $-\infty$ si v et la famille ne sont pas homologues.

Une coloration consiste à affecter une couleur aléatoire de S_E à chaque noeud du réseau. On disposera d'une fonction $c : v \in V_N \rightarrow couleur$, donnant la couleur d'un noeud du réseau.

Pour chacune des colorations, on construit l'ensemble des assignements de familles possibles. Ensuite, on tente d'aligner le motif (débutant à la racine) avec chacun des noeuds $v \in V_N$; on récupère le meilleur score de ces tests d'alignement. Dans le cas où la racine est un noeud dupliqué, on ne lance l'alignement que si le noeud v du réseau correspond bien à l'assignement dans AF de la racine.

Le multi-ensemble de couleurs S "compatibles" est composé de suffisamment de couleurs pour respecter l'assignement des familles de noeuds dupliqués avec un noeud du réseau. Par exemple, quand une famille de trois noeuds dupliqués est marquée comme assignée avec un noeud du

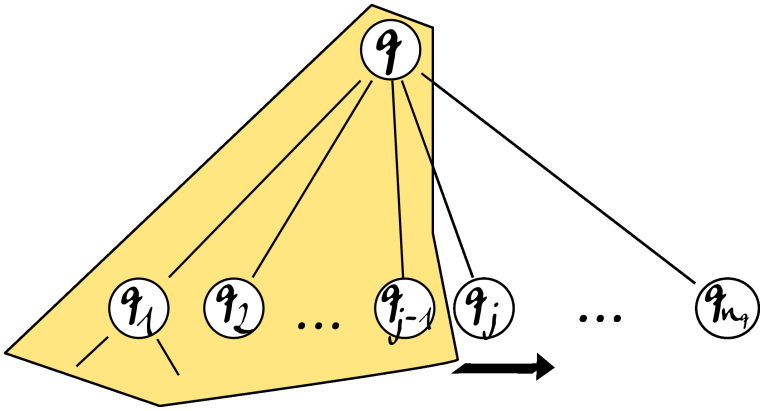


FIG. 5.2 – Illustration de la récurrence. À cet instant, on calcule $W^M(q, v, S, j, AF)$. On considère que l'alignement est déterminé pour les $j-1$ premiers sous-arbres enracinés par les fils de q et pour q lui-même (qui est aligné avec v) : on connait donc le score d'alignement de la zone pleine de gauche. Alors, on ajoute q_j à l'alignement, en testant les différentes possibilités (matcher q_j, insérer un voisin de v, supprimer q_j).

réseau ayant la couleur 5, il doit y avoir trois fois la couleur 5 dans le multi-ensemble.

La fonction W^M est calculée de la manière suivante :
Si $|S| \leq 1$, $W^M(q, v, S, j, AF) = -\infty$
Sinon,

$$W^M(q,v,S,j,AF) = \max_{\substack{u:(u,v)\in E_N \\ S'\subset S \\ c(v)\in S' \\ c(u)\in S-S'}} W^M(q,v,S',j-1,AF) + \begin{cases} \text{(* Match, fils } j \text{ *)} \\ W^M(q_j,u,S-S',n_{q_j},AF) + w(u,v), \\ \\ \text{(* Insertion, noeud } u \text{ *)} \\ W^I(q_j,u,S-S',AF) + w(u,v), \\ \\ \text{(* Deletion, fils } j \text{ *)} \\ W^D(q_j,v,S-S',AF) \end{cases}$$

Lors du calcul de cette valeur, on considère que q est aligné avec v, et on cherche à effectuer la meilleure opération pour le $j^{\text{ème}}$ fils de q. Le score rendu doit être le score d'alignement des $j-1$ premiers sous arbres enracinés par les fils de q, plus le score d'alignement du $j^{\text{ème}}$ fils. (Figure 5.2)

Le multi-ensemble S contient les couleurs disponibles pour l'alignement des j premiers sous-arbres des fils de q. On doit donc tester toutes les répartitions de couleurs entre l'ensemble des $j-1$ sous arbres (qui utilisent l'ensemble des couleurs de S') et le $j^{\text{ème}}$ sous-arbre (qui utilise l'ensemble des couleurs restantes, donc des couleurs se trouvant dans $S - S'$). On doit vérifier aussi que la couleur de v se trouve bien dans l'ensemble S', puisqu'elle n'a pas encore été utilisée. De la même manière, si on veut avoir u dans le résultat, sa couleur doit se trouver dans l'ensemble $S - S'$.

On considère donc que l'on trouve dans $W^M(q, v, S', j-1, AF)$ le meilleur score pour l'alignement des $j-1$ premiers sous-arbres, étant donné cette répartition de couleurs.

Rappelons que q est aligné avec v. Il faut alors aligner le noeud q_j, $j^{\text{ème}}$ fils de q, avec un voisin de v, pour continuer à construire le sous-graphe d'alignement G_A. On doit donc tester le score de l'alignement avec chaque u possible, u étant un voisin de v. Le score de l'alignement avec ce match est donné par $W^M(q_j, u, S - S', n_{q_j}, AF)$.

Mais, on doit également vérifier si un plus haut score est possible, en *insérant* u dans le résultat, c'est à dire de tel manière à ce qu'il ne soit aligné avec aucun noeud de la requête. Le noeud u est inséré entre q et q_j dans G_A. Le score de l'alignement une fois cette insertion effectuée est donné par $W^I(q_j, u, S - S', AF)$.

Enfin, on doit également vérifier le score de l'alignement si on supprime du résultat le noeud

q_j. On retrouve alors dans ce cas dans le résultat, le noeud q, suivi du fils de q_j. Le score de l'alignement étant donné cette suppression est donnée par $W^D(q_j, v, S - S', AF)$.

La meilleure action à effectuer pour le noeud q_j étant donné q et v alignés est donc le score maximum entre l'alignement de q_j avec un voisin de v, l'insertion d'un voisin de v et la suppression dans le résultat de q_j. Pour l'alignement et l'insertion, on compte dans le résultat la probabilité d'interaction entre u et v, donnée par $w(u, v)$, car ce sont les seuls cas où l'arc du réseau est conservé dans G_A.

Si l'ensemble de couleurs S de départ est vide, alors le score est de $-\infty$. En effet, à cette étape la récurrence considère que q et v sont alignés, il faut donc au moins une couleur à donner à q.

De la même manière, si S est un singleton (i.e. comporte qu'une seule couleur), alors le score est également de $-\infty$, sauf si q est une feuille, ce qui n'est pas le cas car $j > 0$. On doit à cette étape encore ajouter un noeud dans le résultat : la suppression de tous les noeuds restant est impossible car, pour respecter les choix de QNet, la suppression ne doit être effectuée que sur des noeuds ayant un degré 2, et il reste encore au moins une feuille dans la requête à cette étape, en plus de q.

Le score de l'alignement du noeud q en lui même est calculé de cette manière, une fois que l'alignement de l'ensemble de ses fils a été effectué, donc lorsque j est égal à 0.

$$W^M(q, v, S, 0, AF) = \begin{cases} \text{si } q \text{ n'est pas dupliqué} \\ \quad h(q, v) \\ \text{sinon si } AF(q)! = v \\ \quad -\infty \\ \text{sinon (* } q \text{ est dupliqué et } AF(q) == v \text{ *)} \\ \quad 0 \end{cases}$$

On compte le score d'alignement du noeud lui même, une fois que toutes ses feuilles ont été alignées. On considère donc le score de l'alignement de q avec v. Si q n'est pas un noeud dupliqué, alors on retourne son score d'homologie avec v donné par la fonction h.

Si q est un noeud dupliqué, alors le noeud du réseau avec lequel lui et les noeuds de sa famille doivent être aligné a déjà été établi par la fonction AF. Ainsi, si v n'est pas ce noeud en question, il faut interdire l'alignement, le score est de $-\infty$. Par contre, si v est bien le noeud désigné par la fonction AF, on doit compter un score de 0. En effet, pour éviter de compter ce score d'homologie pour chacun des dupliqués (on ne doit le compter qu'une seule fois, le score d'alignement du noeud représentant la famille avec le noeud du réseau), nous avons choisi d'effectuer ce comptage avant, dans Score(AF). La Figure 5.3 illustre ceci.

Lorsqu'on insère un noeud dans le résultat, le score de l'alignement est calculé de cette manière :

$$W^I(q, v, S, AF) = \begin{cases} \text{if } AF^{-1}(v) == 0 \text{ \&\& } |S| > 1 \\ \quad \max\limits_{\substack{u:(u,v)\in E_N \\ c(u)\in S-\{c(v)\}}} \begin{cases} W^M(q, u, S - \{c(v)\}, n_q, AF) + w(u, v) + \delta_i, \\ W^I(q, u, S - \{c(v)\}, AF) + w(u, v) + \delta_i, \end{cases} \\ \text{else} \\ \quad -\infty \end{cases}$$

On considère que le noeud v est inséré avant le noeud q. C'est de ce dernier que repart l'alignement ensuite. Il y a dans S l'ensemble des couleurs encore disponibles pour l'alignement.

L'idée est donc de considérer v comme étant dans le résultat, et de faire repartir l'alignement avec u, un voisin de v. On va donc tester avec tous les voisins de v, si le score est plus élevé en alignant u avec q (score donné par $W^M(q, u, S - \{c(v)\}, n_q, AF)$), ou en inserant u (score donné par $W^I(q, u, S - \{c(v)\}, AF)$).

Il faut supprimer de S la couleur de v, noeud inséré dans G_A.

On ajoute à chaque fois δ_i, le coût de cette insertion.

La Figure 5.4 illustre l'insertion d'un noeud dans le résultat.

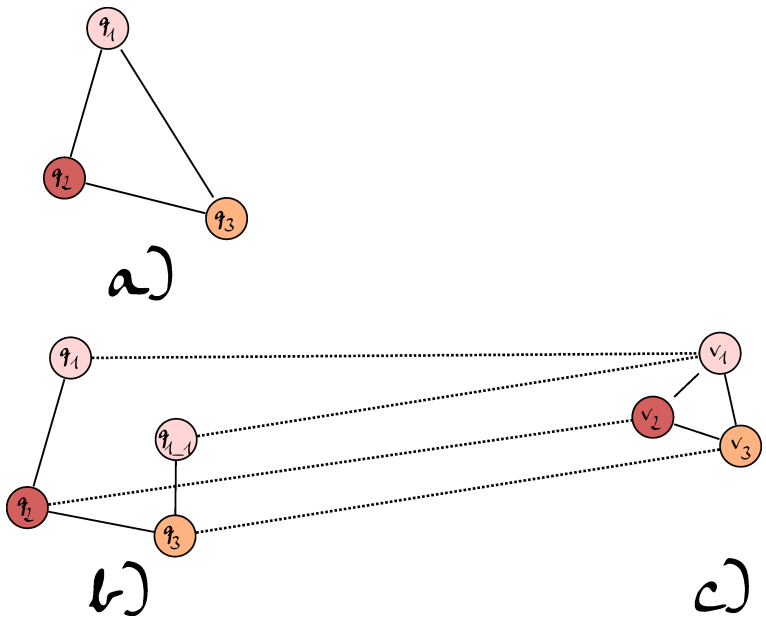


FIG. 5.3 – Illustration du problème du comptage du score d'homologie pour les noeuds dupliqués. En a), nous avons un graphe requête G_Q. La figure b) représente cette requête avec le cycle supprimé selon un algorithme présenté dans la Section 5.3. C'est donc un arbre T_Q, avec une famille de noeuds dupliqués (les noeuds q_1 et q_{1_1}). La figure c) représente le graphe d'alignement G_A, sous-graphe du réseau G_N. Les lignes pointillées entre b) et c) représente les homologies entre les protéines de la requête et du réseau. On observe donc bien que si on n'effectue pas de distinctions entre les noeuds dupliqués et non-dupliqués, on compte plusieurs fois le score d'homologie pour le noeud dupliqué, ici deux fois. Notre solution consiste à ne pas compter ce score d'homologie lors de la récurrence, mais a priori. En effet, étant donné que nous fixons l'affectation des noeuds dupliqués, on peut calculer au préalable le score d'alignement de l'ensemble des noeuds dupliqués, et ceci une seule fois par famille.

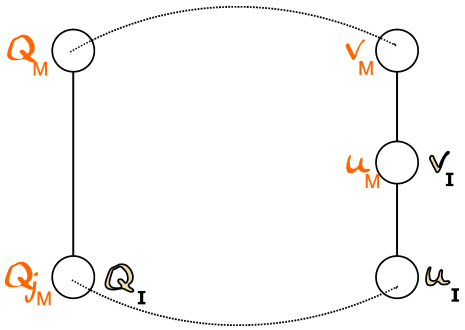


FIG. 5.4 – Insertion. La notation suffixée par M (resp. I) est celle adoptée dans W^M (resp. W^I). A gauche se trouve le motif, à droite le sous-graphe d'alignement. Dans W^M, q est aligné avec v et on cherche à insérer dans le résultat u, un voisin de v. Dans W^I, v est insérée, on cherche à relancer la récurrence en matchant q_j avec u, voisin de v.

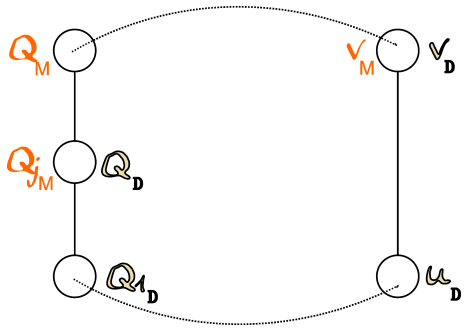


FIG. 5.5 – La délétion. La notation suffixée par M (resp. D) est celle adoptée dans W^M (resp. W^I). A gauche se trouve le motif, à droite le sous-graphe d'alignement. Dans W^M, q est aligné avec v. On va déléter q_j, fils de q. Ce noeud qui est dans le motif ne se trouvera pas dans le graphe d'alignement résultat. Dans W^D, q est délété, on cherche à relancer la récurrence en matchant le fils de q avec u, voisin de v.

Dans le cas de noeuds dupliqués, un test supplémentaire est nécessaire. En effet, si v, noeud du réseau, est destiné à être aligné avec un noeud dupliqué (représenté par $AF^{-1}(v)$), alors il est interdit de l'insérer (on renvoie $-\infty$). S'il est présent maintenant dans le résultat, alors on ne pourra plus effectuer l'alignement entre lui et le noeud dupliqué.

De plus, de la même façon que pour le score de match, s'il n'y a pas plus d'une couleur dans l'ensemble des couleurs disponibles, alors on doit interdire l'opération, car il reste encore au moins une feuille dans la requête qu'on ne pourra pas supprimer.

Lorsqu'un noeud est deleté, le score de l'alignement est calculé de cette manière (le degré d'un noeud est son nombre d'arcs entrant et sortant) :

$$W^D(q,v,S,AF) = \begin{cases} \text{si degré(q) != 2} \\ \quad -\infty \\ \text{sinon si } q \text{ n'est pas dupliqué} \\ \quad \max_{u:(u,v)\in E} \begin{cases} W^M(q_1,u,S,n_{q_1},AF)+w(u,v)+\delta_d, \\ W^I(q_1,u,S,AF)+w(u,v)+\delta_d, \\ W^D(q_1,v,S,AF)+\delta_d \end{cases} \\ \text{sinon} \\ \quad \text{si } AF(q) == deletion \\ \quad\quad \max_{u:(u,v)\in E} \begin{cases} W^M(q_1,u,S,n_{q_1},AF)+w(u,v), \\ W^I(q_1,u,S,AF)+w(u,v), \\ W^D(q_1,v,S,AF) \end{cases} \\ \quad \text{else} \\ \quad\quad -\infty \end{cases}$$

On considère que le noeud q, qui était dans la requête G_Q, ne sera pas dans le résultat G_A ; il est deleté. Le noeud v est aligné avec le père de q. Enfin, on trouve dans S l'ensemble des couleurs encore disponibles. Cet ensemble ne sera pas modifié, on n'ajoute aucune couleur dans le résultat final.

De la même manière que pour le match, on va, étant donnée la délétion de q, choisir l'action augmentant le plus le score de l'alignement, entre match, insertion ou délétion. L'alignement redémarre donc de q_1, premier et seul fils de q. La Figure 5.5 illustre cette délétion.

Si le degré du noeud à deléter n'est pas 2, alors on empêche cette délétion en indiquant un score de $-\infty$.

Pour prendre en compte les noeuds dupliqués, il suffit d'autoriser cette délétion si l'assignement du noeud était fixé comme tel dans l'ensemble AF, et de l'interdire sinon (en mettant un score de $-\infty$). Le coût de la délétion est compté une fois pour une famille, avant la récurrence dans Score(AF).

Un exemple d'exécution de la récurrence est donné en Figure 5.6.

Complexité

La complexité en temps pour un essai de l'algorithme de recherche d'arbre sans noeud dupliqué est selon les auteurs de QNet [15] de $2^{O(k+N_{ins})}m$, où k est le nombre de noeuds de la requête et m est le nombre d'arcs du réseau. Dans le cadre de notre algorithme, on exécute au pire $n^{|familles|}$ fois la programmation dynamique de QNet, où n est le nombre de noeuds du réseau et $|familles|$ le nombre de familles de noeuds dupliqués de la requête. Les modifications dans la programmation dynamique n'augmentent pas la complexité, ce ne sont que des tests supplémentaires effectués en temps constant. La complexité globale de notre algorithme dans le pire des cas est donc de $O(n^{|familles|}.2^{O(k+N_{ins})}m)$.

La complexité pire cas donné par les auteurs de QNet pour leur version théorique de recherche de graphes est de $2^{O(k)}n^{t+1}$, où k est la taille du motif, n le nombre de noeuds du réseau et t la treewidth de la requête. Ainsi, la différence pire cas entre nos deux algorithmes se situe au niveau de la puissance de n : pour QNet il s'agit de la treewidth plus un, pour notre algorithme il s'agit du nombre minimum de familles. Nous ne savons pas si ces deux paramètres sont théoriquement différents. Cependant, nous avons effectué des tests expérimentaux, afin de comparer ces deux paramètres sur des graphes aléatoires. La treewidth est calculée au moyen de l'algorithme exact proposé par la site `http://www.treewidth.com/`, le nombre de famille est calculé au moyen de notre algorithme exact que nous expliquerons dans la Section 5.3.2. La Figure 5.7 semble montrer que notre paramètre est plus petit pour la taille de graphe qui nous interesse. De plus, leur complexité dépend de la treewidth plus un : il semblerait que notre complexité pire cas soit meilleure.

En pratique, notre borne est largement sur-évaluée. En effet, chaque famille doit être assignée à un noeud différent du réseau, il n'y a donc pas n possibilités pour chaque famille. Le nombre d'exécutions de la programmation dynamique est de $\frac{n!}{(n-|familles|)!}$, nombre d'arrangements possibles.

De plus, on exécute la programmation dynamique uniquement si le score des assignations est différent de $-\infty$. Ce score s'obtient fréquemment : il suffit qu'une des familles soit assignée avec un noeud du réseau avec lequel il n'est pas homologue. Par conséquent, le score seuil pour BLAST indiquant si une protéine est homologue à une autre influence fortement le temps d'exécution de l'algorithme. Ainsi, le temps d'exécution moyen est très inférieur à la borne pire cas de la complexité.

On aimerait donc pouvoir comparer en pratique notre algorithme à QNet. Malheureusement, ceci est impossible tant que leur version prenant en compte les graphes n'est pas implémentée.

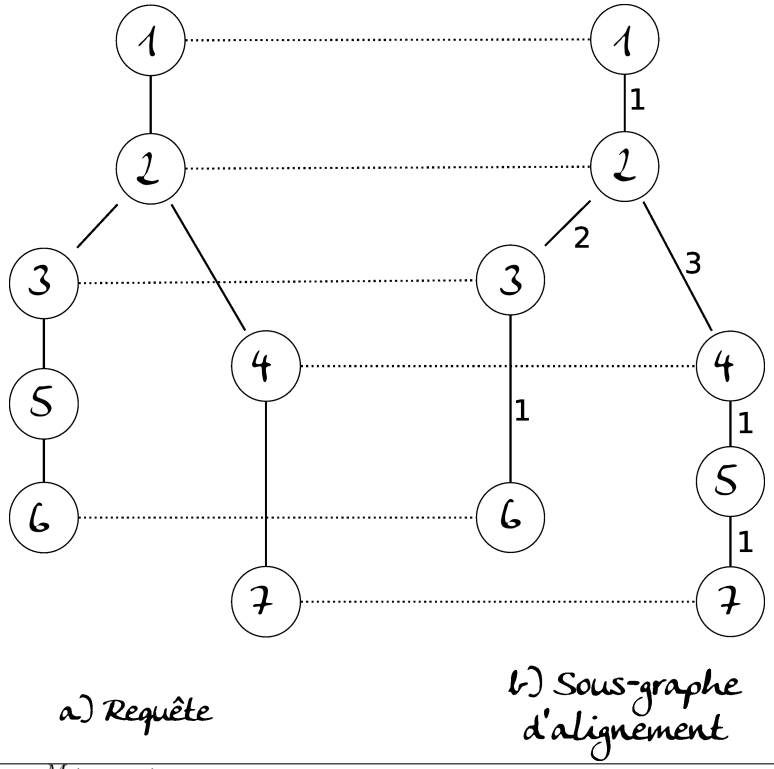
La complexité de notre algorithme est directement dépendante du nombre de familles dupliquées dans le motif, donc de la topologie de la requête. Il serait interessant d'avoir plus d'information biologiques sur la forme des motifs à rechercher, afin de mieux adapter notre algorithme et d'éventuellement prédire un temps moyen d'exécution.

5.2.2 Implémentation

Nous avons implémenté cette récurrence afin d'obtenir un programme prenant un réseau G_N, un motif arbre T_Q, une liste de noeuds dupliqués, donnant en sortie le sous-réseau correspondant au graphe d'alignement G_A. Ce programme est écrit avec le langage Python [1], agrémenté du package NetworkX [2].

Le calcul des homologies entre les protéines est effectué au préalable, à l'aide du programme BLAST [3]. On donne à ce programme les séquences d'acides aminés des protéines de la requête et du réseau et celui-ci nous donne un score *eValue* d'homologie. Le score d'homologie entre deux

[1] `http://www.python.org/`
[2] `https://networkx.lanl.gov/`



Etape1	$W^M(6,6,0) = 5$
Etape2	$W^M(7,7,0) = 5$
Etape3	$W^D(5,3) = W^M(6,6,0) + w(6,3) - \delta_d = 5 + 1 - 3 = 3$
Etape4	$W^M(3,3,0) = 5$
Etape5	$W^M(3,3,1) = 5 + 3 = 8$
Etape6	$W^I(7,5) = W^M(7,7,0) + w(7,5) - \delta_i = 5 + 1 - 3 = 3$
Etape7	$W^M(4,4,0) = 5$
Etape8	$W^M(4,4,1) = W^M(4,4,0) + W^I(7,5) + w(5,4) = 5 + 3 + 1 = 9$
Etape9	$W^M(2,2,0) = 5$
Etape10	$W^M(2,2,1) = W^M(2,2,0) + W^M(3,3,1) + w(3,2) = 5 + 8 + 2 = 15$
Etape11	$W^M(2,2,2) = W^M(2,2,1) + W^M(4,4,1) + w(4,2) = 5 + 9 + 3 = 17$
Etape12	$W^M(1,1,0) = 5$
Etape13	$W^M(1,1,1) = W^M(1,1,0) + W^M(2,2,1) + w(2,1) = 5 + 17 + 1 = 23$

FIG. 5.6 – Un exemple de requête et de son sous-graphe d'alignement correspondant. Les numéros dans les noeuds représentent des identifiants de protéines. Les lignes horizontales en pointillés représentent les homologies entre les protéines. Les chiffres sur les arcs du réseau représentent les poids des arcs. Dessous, on trouve un extrait des étapes de la programmation dynamique. Pour une question de simplicité, on a ignoré le coloriage des noeuds. Les scores sont fixés de la manière suivante : un match donne $+5$, une insertion ou une délétion coute 3. On remarque que le noeud 5 de la requête n'a pu être matché étant donné sa position dans le réseau. Il est donc supprimé de la requête, mais inséré dans le résultat, pour pouvoir matcher le noeud 7.

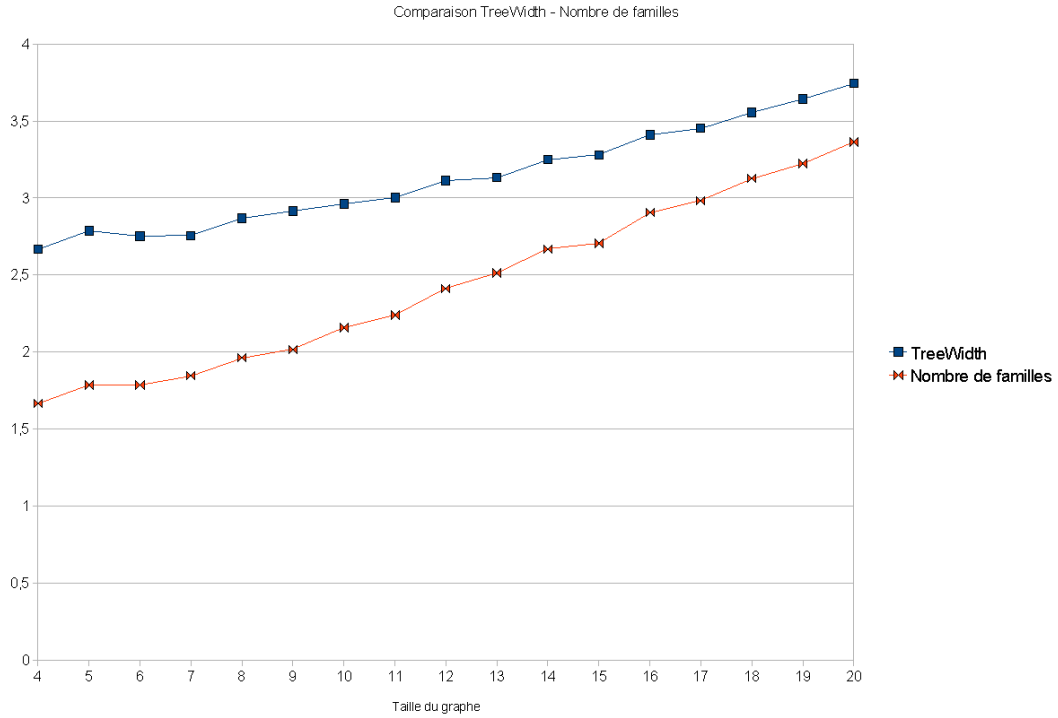


FIG. 5.7 – Comparaison de la treewidth et du nombre de familles. Pour chaque taille de graphe, on effectue la moyenne des taille sur 100 graphes connexes créés aléatoirement par la bibliothèque NetworkX. La treewidth est calculée au moyen d'un algorithme exact proposé par `http://www.treewidth.com/`, le nombre de famille par notre algorithme exact que nous verrons dans la Section 5.3.

protéines q et v est donné par $h(q, v)$ et est égal à $-\log(eValue)$, comme pour QNet. Plus l'$eValue$ est proche de 0, plus l'homologie entre les deux protéines est forte. On peut donner un seuil minimal de similarité, afin de ne garder qu'un certain nombre d'homologues pour chaque protéine.

L'implémentation de la récurrence devrait s'effectuer par une véritable programmation dynamique, calculant les valeurs pour les feuilles, puis remontant aux pères, dont la valeur est calculée avec ce qui à déjà était fait pour les fils.

Malheureusement, nous n'avons pas trouvé de moyen efficace pour ordonner ces calculs. Nous avons préféré effectuer une programmation par récurrence, respectant donc les formules de récurrences de l'algorithme. Cependant, on stocke toutes les valeurs que l'on calcule, pour ne pas avoir à recalculer plusieurs fois les même éléments. C'est donc une programmation dynamique "inverse" : on part de la racine, et on effectue par récurrence les calculs. Si la valeur a déjà été calculée, on la récupère. Sinon, on la calcule puis on la stocke.

La récurrence est effectuée de manière à tester l'alignement de la racine avec chacun des noeuds du réseau afin de "démarrer" la récurrence. Nous avons décidé de lancer les calculs uniquement si la racine était homologue avec le noeud du réseau. Ceci permet de limiter grandement le nombre de calculs.

De la même manière, nous ne testons pas le "match" si le noeud en lui-même n'est pas homologue avec le noeud du réseau. En effet, tel qu'est fait la récurrence, il faut d'abord calculer l'alignement pour chaque sous-arbre démarrant à ses fils, ce qui est inutile si le noeud n'est pas homologue.

Une fois les tables calculées, il faut déterminer le sous-arbre d'alignement résultat. Ceci est effectué par un *backtracking* "classique". Le principe est le suivant : étant donnée la table stockée, on doit déterminer quelle a été l'action effectuée pour obtenir ce résultat. En pratique, pour chaque noeud de la requête, un $\max$ sur les mêmes valeurs que la récurrence nous permet de savoir si

c'est un match, une insertion ou une délétion qui a été choisi auparavant. Récursivement, on recommence ce principe jusqu'à être descendu aux feuilles.

5.2.3 Tests

Comme précisé auparavent, la comparaison pratique avec QNet est impossible puisque leur algorithme n'est pas implémenté pour les graphes.

Nous avons pu cependant prendre une partie de leurs tests qu'ils avaient effectués pour rechercher des arbres. Les temps de calculs ne sont pas comparables : en effet, nous avons utilisé le langage python, qui est interprété, alors que QNet est codé en C++, langage compilé. Nous avons choisi python pour la rapidité de développement et ses bibliothèques intégrées (NetworkX). Cependant, nous avons pu "profiler" le programme et observer les parties consommatrices de temps : des améliorations sont encore possibles.

Les données d'interactions protéiques pour la levure et la mouche ont été obtenues depuis la base de données DIP [3] . La réseau de la levure contient 4 738 protéines et 15 147 interactions, celui de la mouche 7 481 protéines et 26 201 interactions.

QNet génère des motifs sous la forme d'arbres aléatoires de tailles 5 à 9, pris depuis le réseau de la levure et tente de les retrouver dans ce même réseau. Chaque motif est perturbé avec au plus 2 insertions/délétions. Nous avons pu également retrouver ces motifs. Nous avons pu étendre ces tests en recherchant des graphes (Figure 5.8).

Les chemins Mitogen-activated Protein Kinase (MAPK) sont une collection de chemins de transduction de signaux, jouant un rôle critique dans la réponse cellulaire à différents stimulis [14]. Ces chemins sont connus pour être conservés à travers différentes espèces et vont servir à tester notre algorithme.

Nous récupérons des chemins MAPK de l'homme à partir de la base de données KEGG [30], que nous recherchons dans le réseau de la mouche. Alors que QNet n'utilise pour ses tests uniquement des arbres, nous pouvons nous permettre de rechercher des graphes. Les résultats sont satisfaisant puisque nous retrouvons les chemins, avec très peu de modifications (Figure 5.9). Le seuil BLAST joue grandement dans le temps de calcul. Cependant, nous observons que celui-ci reste raisonnable pour une à deux familles de noeuds dupliqués.

5.3 Transformation de graphes en arbres

Dans la section précédente, nous avons vu que le motif T_Q possedait plusieurs noeuds dupliqués. Dans cette section, nous proposons un algorithme afin d'obtenir T_Q, en transformant le motif G_Q, qui peut comporter des cycles, en T_Q, qui est un arbre.

L'algorithme à mettre en place doit pouvoir transformer le graphe G_Q en arbre T_Q en supprimant les cycles, et ceci sans perte. Pour effectuer ceci, nous décidons de dupliquer les noeuds causant les cycles. Ainsi, à la fin de l'exécution de l'algorithme, T_Q ne comportera pas de cycles, mais plusieurs noeuds dupliqués.

L'algorithme est sans perte, il suffit de refusionner les noeuds dupliqués pour retrouver le graphe d'origine.

Le graphe en entrée est non-orienté. L'algorithme crée en sortie un graphe acyclique non-orienté. Il suffit alors de choisir un noeud et d'effectuer un parcours en profondeur pour obtenir un arbre orienté.

Nous verrons tout d'abord une approche naïve, efficace en temps, dupliquant les noeuds au hasard, puis une approche exacte, minimisant le nombre de noeuds dupliqués. Nous avons implémenté ces deux algorithmes avec le langage de programmation Python [4] et le package de gestion de graphes NetworkX [5] .

[3] http://dip.doe-mbi.ucla.edu/
[4] http://www.python.org/
[5] https://networkx.lanl.gov

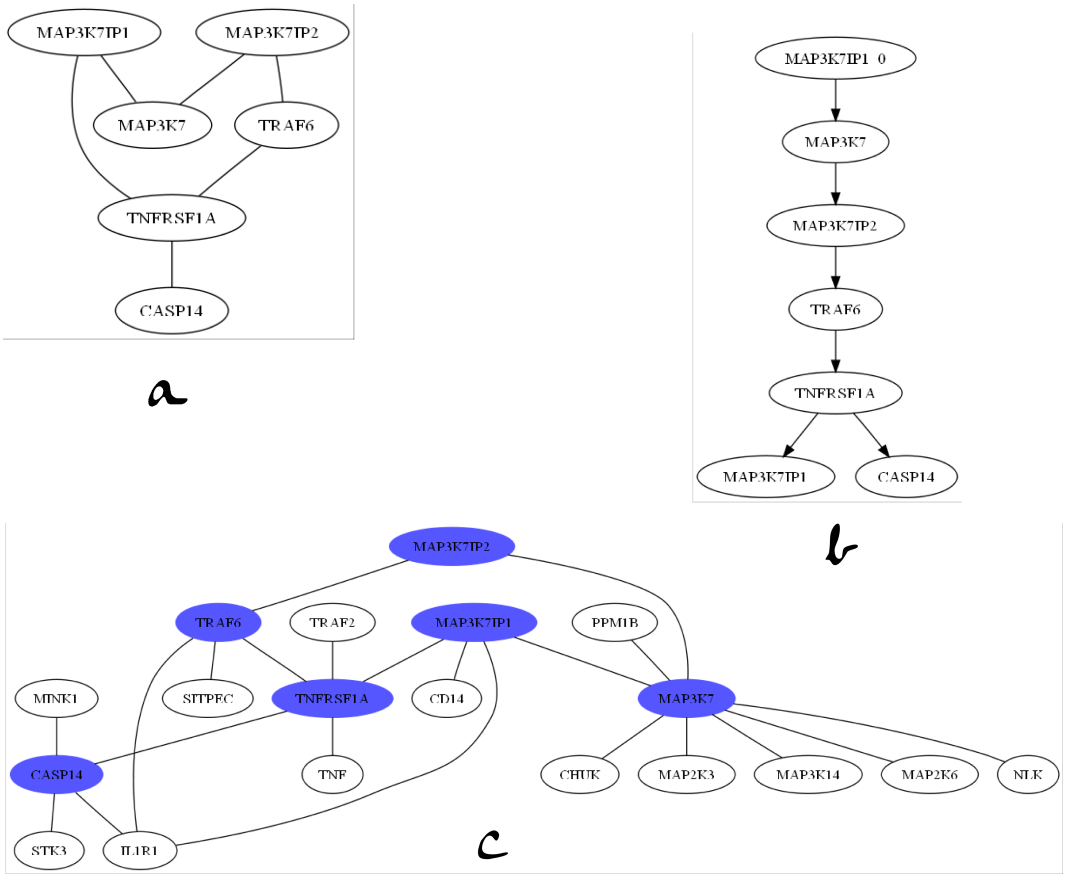


FIG. 5.8 – Exemple d'exécution de notre programme de recherche de graphe dans un réseau. En a), la requête initiale, sous forme de graphe. En b), la transformation de la requête en arbre, avec un noeud dupliqué. En c), un extrait du réseau, avec en noeuds pleins le sous-graphe d'alignement résultat. On a bien retrouvé un graphe dans le réseau. Le réseau est HSA KEGG, le motif a été choisi aléatoirement.

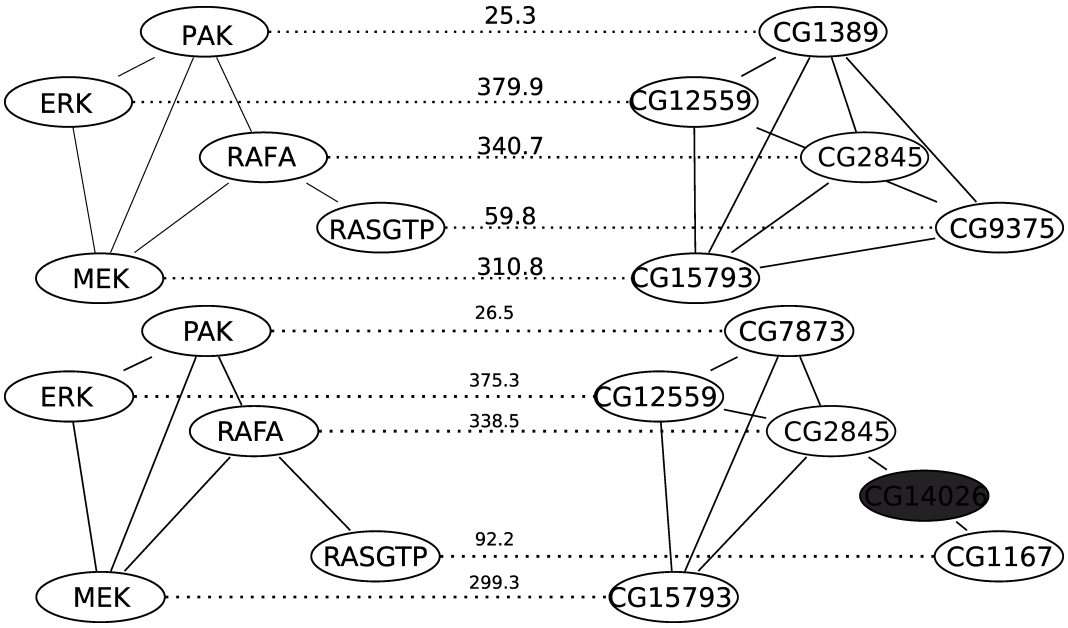


FIG. 5.9 – Deux types de résultats pour un même motif. A gauche, un motif MAPK de l'homme, récupéré depuis [30], avec trois cycles. A droite, le graphe d'alignement donné par notre programme dans le réseau de la mouche. Au centre se trouve les scores d'homologies entre ces deux protéines donnés par Blast. Notre programme a bien retrouvé un motif graphe dans un autre réseau, il y a conservation de ce motif. Sur la figure du dessous, la protéine CG14026 a été inséré dans le résultat afin de pouvoir aligner la protéine RASGTP avec la protéine CG1167.

5.3.1 Algorithme naïf

Soit $G_Q = (V_Q, E_Q)$, un motif. On définit une fonction $copys : v \in V_Q \rightarrow \mathbb{N}$, donnant le nombre de copies d'un noeud du motif. Egalement, $\forall v \in V_Q, v_{a\_i} \equiv$ la $i^{\text{ème}}$copie de v_a tel que $1 \leq i \leq copys(v_a)$.

L'algorithme est le suivant :

Algorithme 2 : Algorithme naïf

tant que *il existe un cycle c de taille m dans le graphe tel que* $c = v_1 \mapsto v_2 ... \mapsto v_m \mapsto v_1$ **faire**
    Dupliquer(v_m, v_1);
fin

Fonction Dupliquer(v_a, v_b)
soit $i =$ copys (v_b);
$V_Q = V_Q \cup \{v_{b\_i}\}$;
$E_Q = E_Q - \{(v_a, v_b)\}$;
$E_Q = E_Q \cup \{(v_a, v_{b\_i})\}$;

On effectue donc plusieurs parcours en profondeur successifs afin de retrouver les cycles dans le graphe (on colorie les noeuds en cours de visite et on détecte le cycle lorsque l'on atteint un noeud déjà colorié). Soit v_1 ce sommet déjà colorié et v_m le sommet précedent. Alors, on duplique v_1, on supprime l'arc allant de v_m à v_1, et on crée un nouvel arc, partant de v_m, mais arrivant sur le sommet nouvellement dupliqué. La Figure 5.10 illustre ces étapes.

La complexité de cet algorithme est de $O(|E_Q|^2)$. En effet, trouver un cycle par un parcours en profondeur est effectué en $O(|E_Q|)$ (au pire visite de tous les arcs), et afin d'obtenir un arbre, on

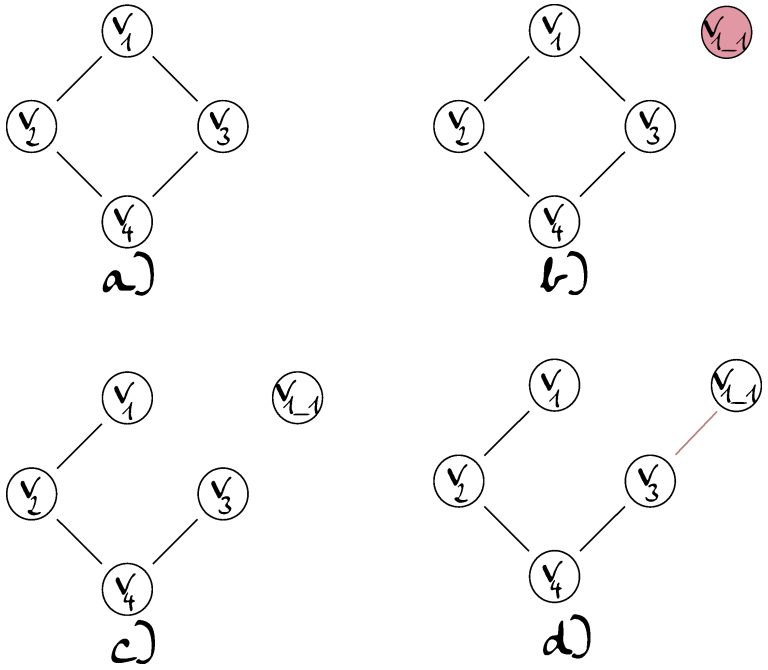


FIG. 5.10 – Etapes d'une duplication de noeud afin de casser un cycle. a) Graphe avec un cycle b) Le parcours en profondeur à détecté un cycle partant et revenant en v_1. On crée un noeud v_{1_1}, dupliqué du noeud v_1 c) On supprime l'arc (v_3,v_1) d) On crée l'arc (v_3,v_{1_1}). Finalement, le graphe n'a plus de cycles, et les noeuds v_1 et v_{1_1} représentent la même protéine.

doit dupliquer au plus $|E_Q| - |V_Q| + 1$ arcs.

On appelle *famille* de noeuds dupliqués l'ensemble des noeuds dérivés d'un même noeud. Par exemple, sur la Figure 5.10, le noeud v_1 et son dupliqué v_{1_1} font parti de la même famille.

Nous avons vu dans la section précédente que la complexité de l'algorithme de programmation dynamique retrouvant le motif dans le réseau est directement liée au nombre de familles de noeuds dupliqués. Il devient alors impératif de limiter au maximum la nombre de familles de noeuds dupliqués.

On remarque alors que l'approche naïve consistant à dupliquer les noeuds lorsqu'on découvre un cycle n'est pas optimale. En effet, comme montré sur la Figure 5.11, il est parfois possible de dupliquer plusieurs fois le même noeud, et ainsi ne pas augmenter le nombre de familles.

C'est pour cette raison que nous nous sommes intéressés au VERTEX FEEDBACK SET PROBLEM, donnant le nombre minimum de sommets à supprimer afin de rendre le graphe acyclique.

5.3.2 Vertex Feedback Set Problem

Le VERTEX FEEDBACK SET PROBLEM est un des premiers problèmes dont la NP-Complétude a été prouvée (il fait parti des 21 problèmes NP-Complet de Karp [31]). Il peut être formalisé de la manière suivante :
INSTANCE : Un graphe non-orienté $G = (V, E)$ et un entier positif k
QUESTION : Existe-t-il un sous-ensemble de sommets $X \subseteq V$ avec $|X| \leq k$ tel que G sans les sommets de X soit acyclique ?

Sa résolution donne le nombre minimal de sommets du graphe à supprimer pour rendre le graphe acyclique. Dans le cadre du stage, le résoudre nous permet de connaitre le nombre minimum de noeuds à dupliquer. En effectuant un algorithme exact, on obtiendra l'arbre correspondant au graphe, avec le nombre minimal de familles dupliquées. La complexité sera la plus petite possible pour l'algorithme de programmation dynamique retrouvant le motif dans le réseau.

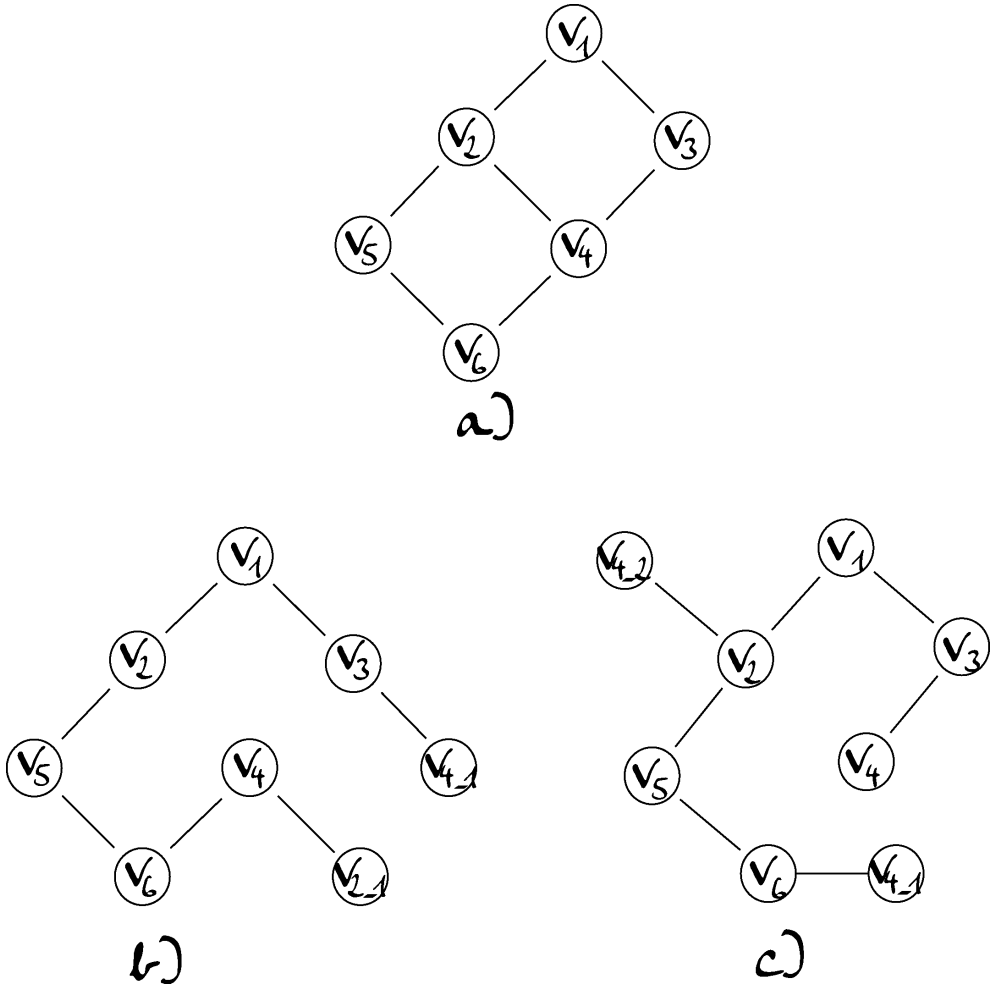


FIG. 5.11 – Cette figure montre que selon les choix de noeuds à dupliquer, on obtient un nombre différents de familles de noeuds dupliqués. a) Un graphe comportant plusieurs cycles. b) Selon un choix aléatoire de noeuds à dupliquer (par exemple des parcours en profondeurs successifs), l'algorithme duplique une fois le noeud v_2 et une fois le noeud v_4 afin de rendre le graphe acyclique. c) On voit qu'il était possible de rendre le graphe acyclique en dupliquant (plusieurs fois) un seul noeud. Le graphe est acyclique, avec une seule famille de noeud dupliqués (noeud v_4). Le nombre de noeuds dupliqués peut donc être variable pour un même graphe d'entrée.

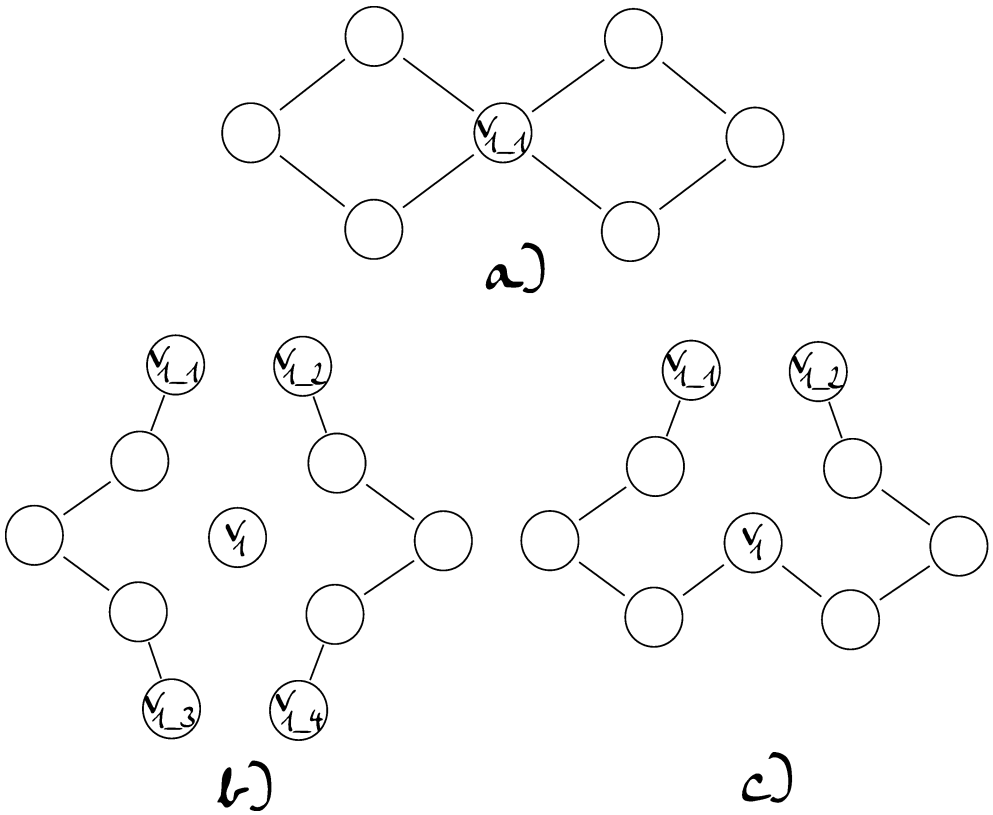


FIG. 5.12 – a) L'algorithme a déterminé que le noeud v_1 est à supprimer pour rendre le graphe acyclique b) Première idée : on duplique le noeud autant de fois qu'il a de voisin et on relie les voisins aux noeuds nouvellement dupliqués. Dans ce cas çi, on se retrouve avec un forêt (ensemble d'arbres). c) On duplique le noeud uniquement lorsque cela laisse le graphe connexe. Ici, la duplication et suppression des deux arcs restant sur v_1 entrainerait la perte de la connexité du graphe. On a bien un graphe acyclique en sortie.

Algorithme "brute force"

Dans le cadre de notre problème, nous avons choisi d'implémenter une version exacte "brute force" de cet algorithme, en testant si le graphe résultat est acyclique pour k allant de 1 à $|V_Q| - 1$, et en s'arrêtant une fois obtenue satisfaction (i.e. on a un ensemble X de k noeuds à supprimer pour rendre le graphe acyclique). La complexité exponentielle de l'algorithme découle du nombre de sous-graphes pour lesquels on teste l'acyclicité.

Nous devons alors adapter le problème : en effet, pour obtenir un arbre sans pertes il ne faut pas supprimer de noeuds. Pour chacun des noeuds de l'ensemble à supprimer, deux choix s'avèrent possibles.

La première idée consiste à dupliquer le noeud pour chaque voisin dans le graphe ; ainsi, tous les cycles sont supprimés. Seulement, cette méthode peut créer des forêts (voir Figure 5.12b). L'algorithme de programmation dynamique n'admet pas de forêt en entrée.

La seconde possibilité consiste à tester pour chaque arc à supprimer si sa suppression laisse le graphe connexe. Si oui, alors on peut effectuer la duplication. Cette méthode est illustrée par la Figure 5.12c.

Finalement, notre adaptation "brute force" de ce problème consiste à tester, pour tout sous-graphe de taille $|V_Q| - k$, si celui-ci est acyclique ou non. Ensuite, il faut dupliquer les noeuds de

l'ensemble à supprimer, en veillant à garder le graphe connexe. Voici l'algorithme :

Algorithme 3 : Algorithme "brute force"

```
1  pour k = 0 à |V_Q| faire
2      pour chaque G' : sous-graphe de G_Q de taille |V_Q| − k faire
3          si G' ne contient pas de cycle alors
4              pour tous les k noeuds u à supprimer faire
5                  pour tous les voisins v de u faire
6                      Dupliquer(v, u);
7                      si le graphe n'est plus connexe alors
8                          ignorer la duplication;
9                      fin
10                 fin
11             fin
12             retourner
13         fin
14     fin
15 fin
```

La complexité de cette algorithme "brute force" est exponentielle en la taille n du graphe. Il y a 2^n sous-graphes possibles, ainsi la complexité est de $O(2^n \times |E_Q|)$.

Autres algorithmes

La complexité de l'algorithme exact précédent est élevée. S'il est normal d'avoir un algorithme exponentiel pour résoudre un problème NP-Complet, il existe tout de même des alternatives :

- Guo et al. [24] proposent un algorithme exact de complexité paramétrée efficace en temps ($O(c^k.m)$, où c est une constante, k est la taille de l'ensemble à supprimer et m est le nombre d'arcs du graphe), en utilisant la technique de la compression itérative,
- Bafna, Berman et Fujito [8] exposent un algorithme polynomial de 2-approximation (i.e. la solution est au pire deux fois plus grande que la meilleure), Bar-Yehuda et al. [9] énoncent un algorithme linéaire de 4-approximation,
- Pardalos, Qian et Resende [40] soumettent une heuristique résolvant ce problème, qui fournirait en pratique de bonnes solutions dans un faible temps.

Ce sont différentes pistes utilisables afin d'accélérer cette transformation dans le cas où l'on voudrait pouvoir gérer des graphes de tailles plus importantes. Cependant, la complexité de l'algorithme de programmation dynamique recherchant le motif dans le réseau ne permet pas l'utilisation d'arbres dépassant une taille d'une dizaine de noeuds.

5.3.3 Tests

On dispose donc de deux algorithmes, un naïf, d'une complexité raisonnable, et un second, exact mais d'une complexité exponentielle, dupliquant le nombre minimal de familles. Nous avons réalisé plusieurs séries de tests afin de comparer la qualité des arbres obtenus et le temps nécessaire pour les construire.

On construit des graphes ayant un nombre de noeuds allant de quatre à vingt et un nombre d'arcs allant d'une fois à deux fois le nombre de noeuds. Et, lorsque ces deux paramètres sont fixés, on génère cinquante graphes aléatoires, à l'aide de la bibliothèque Network-X [6]. Sur chacun de ces graphes, on lance les algorithmes, et on relève le temps de calcul et le nombre de familles de noeuds dupliqués générées. La Figure 5.13 décrit ces données.

[6] `https://networkx.lanl.gov`

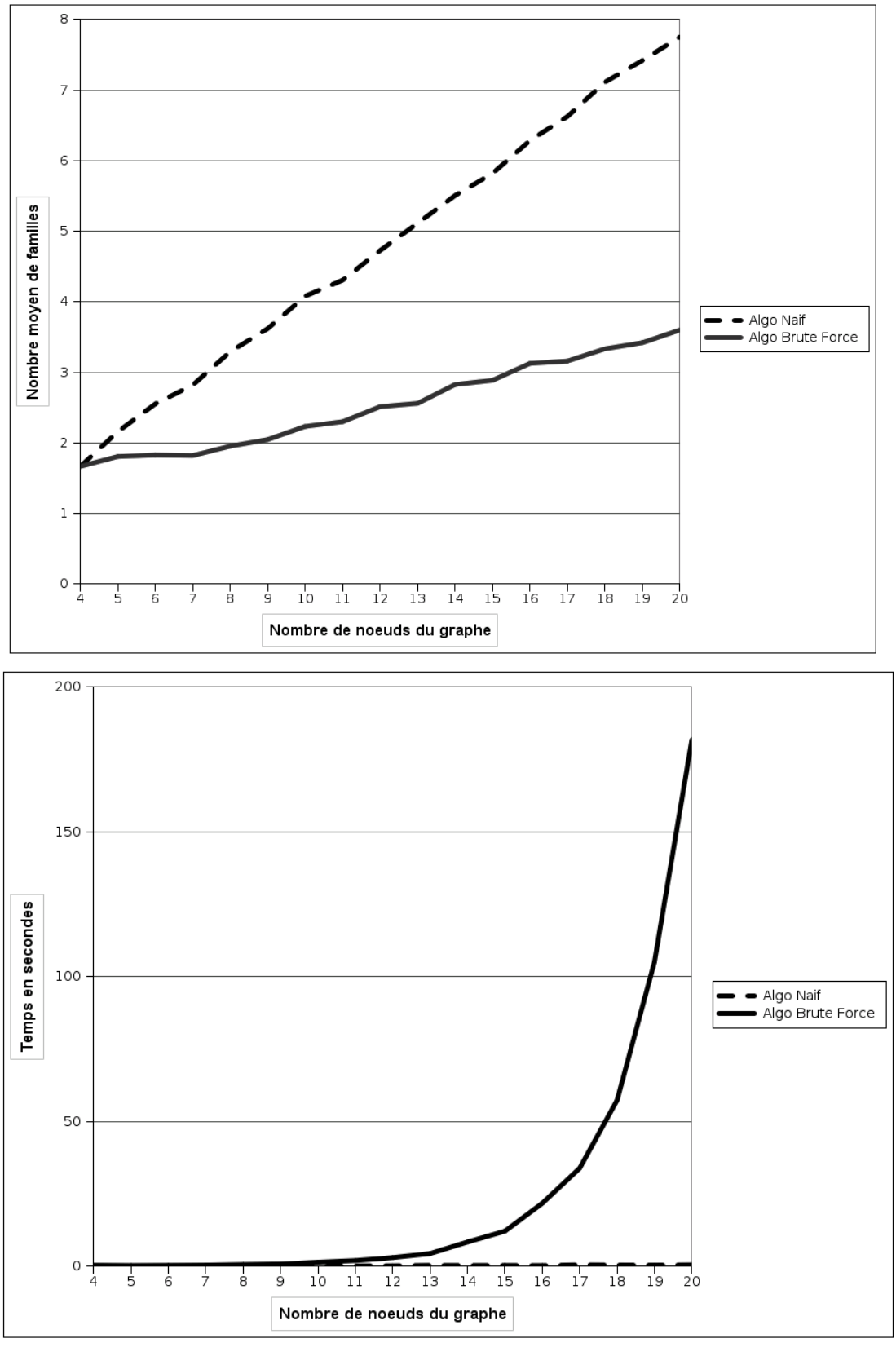


FIG. 5.13 – Graphiques indiquant le nombre de familles dupliqués et le temps de calcul pour chaque algorithme en fonction du nombre de noeuds dans le graphe. Les valeurs sont des moyennes effectuées sur plusieurs séries de cinquante tests. Tests réalisés sur une machine avec un processeur de 2Ghz et 512Mo de mémoire.

On remarque, comme prévu, que le nombre de familles dupliqués augmente très rapidement pour l'algorithme naïf, alors qu'il reste relativement faible pour l'algorithme exact. Également, le temps de calcul reste insignifiant pour l'algorithme naïf, alors qu'il augmente très rapidement dans le cas de l'algorithme exact. Cependant, il faut dépasser la vingtaine de noeuds pour que le temps de calcul soit réellement prohibitif.

Dans le cadre de notre étude, le nombre de noeuds du motif ne peut pas dépasser la dizaine, étant donnée la complexité de l'algorithme de programmation dynamique. Le temps de calcul de l'algorithme "brute force" reste très raisonnable pour cette taille de graphe. De plus, il est préférable d'utiliser cet algorithme car il minimise grandement le nombre de familles dupliquées par rapport à l'algorithme naïf. Le temps dépensé sera largement compensé par la baisse de la complexité dans l'algorithme de programmation dynamique.

Finalement, étant donné le type des données utilisées dans le cadre du problème de recherche de motif, il est conseillé d'utiliser un algorithme exact, permettant d'avoir le nombre minimal de familles dupliquées. De plus, un algorithme "brute force" suffit, les temps de calculs restant raisonnables pour les tailles typiques des motifs. Cependant, il existe des solutions, parfois exact comme l'algorithme de complexité paramétrée par compression itérative, permettant de diminuer les temps de calculs de cette transformation.

La Figure 5.14 illustre les résultats des deux algorithmes proposés par notre implémentation.

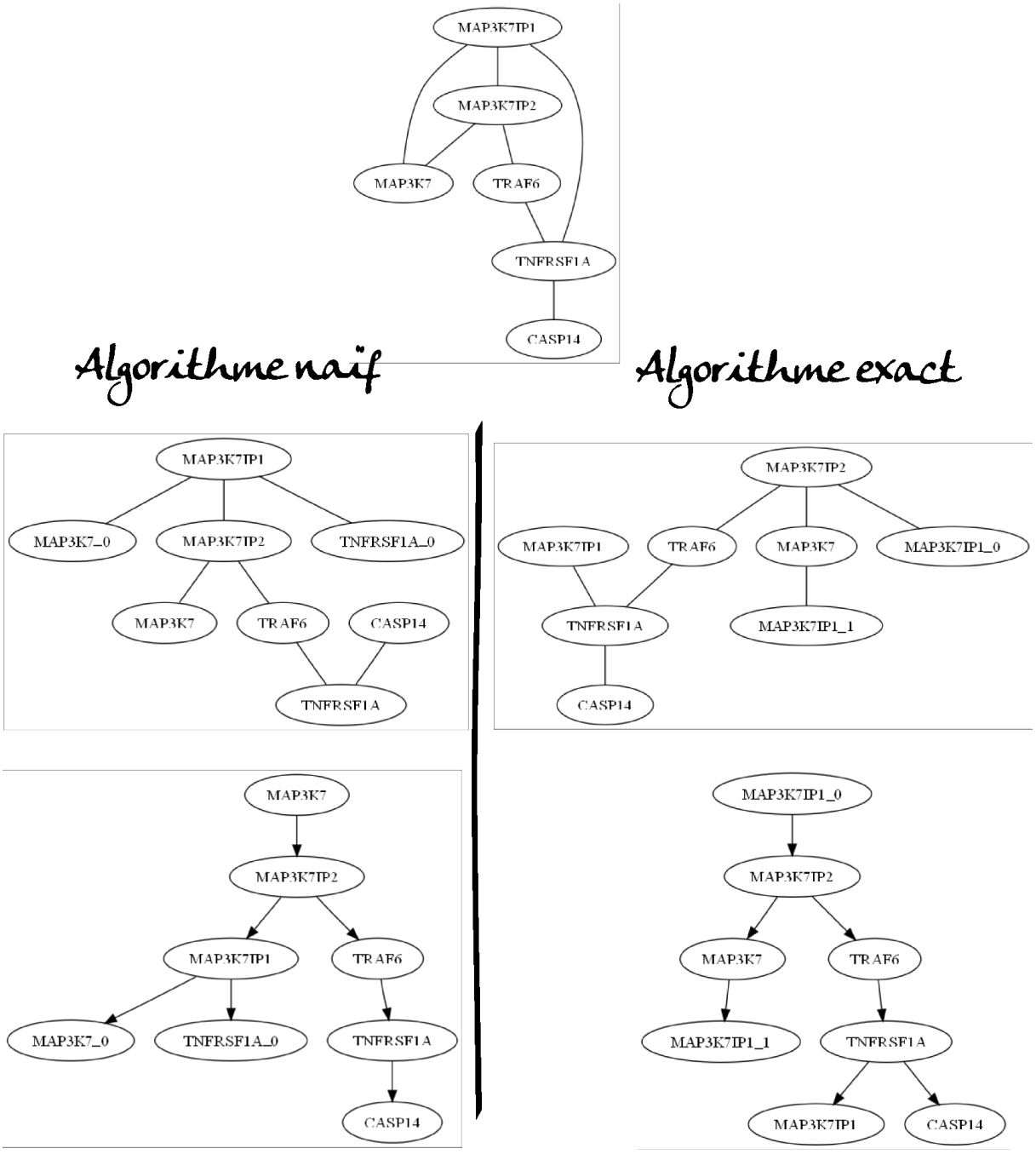


FIG. 5.14 – Schéma des étapes du déroulement des deux algorithmes de transformation de graphes en arbres. Sur le haut du schéma, se trouve un graphe de six protéines avec trois cycles. Sur la partie gauche, l'algorithme naïf a dupliqué deux protéines différentes (MAP3K7 et TNFRSF1A). En dessous, se trouve l'arbre orienté, avec le choix d'une racine (MAP3K7) suivi d'un parcours en profondeur. Sur la partie droite, l'algorithme exact a dupliqué qu'un seul type de protéines (MAP3K71P1), deux fois. En dessous, on obtient l'arbre orienté en choisissant MAP3K71P1_0 comme racine. Les graphes sont générés par graphviz [8].

Conclusion

Pendant la première partie du stage, nous avons défini deux problèmes principaux autour des réseaux d'interactions de protéines. Nous avons donc établi un état de l'art, afin de définir l'existant et de placer notre sujet par rapport aux recherches actuelles. Nous avons constaté qu'il existait un grand nombre de publications sur ce sujet récent.

Ensuite, durant la seconde partie du stage, nous nous sommes concentrés sur un problème précis, celui de la recherche de motif, et sur la réalisation d'un nouvel algorithme, présentant une extension à QPath [50] et une alternative à QNet [15]. Les auteurs de ce dernier proposent un algorithme pour la recherche de motif quand celui-ci est un graphe en effectuant une décomposition arborescente du graphe avec perte ; la complexité de l'algorithme dépend de la treewidth du graphe de requête.

Notre algorithme est quant à lui sans perte. Il transforme dans un premier temps le graphe en arbre, en effectuant une duplication des noeuds causant les cycles. Ensuite, nous avons modifié la recurrence proposée par QNet pour rechercher un arbre dans un réseau, afin d'effectuer la gestion de ces noeuds dupliqués, qui désignent une seule et même protéine dans le graphe de départ. Notre algorithme n'est plus dépendant de la treewidth du graphe, mais du nombre de noeuds à dupliquer pour y supprimer tous les cycles, sa complexité est de $n^{|familles|}2^{O(k)}m$, où n est le nombre de noeuds du réseau, $|familles|$ est le nombre de familles de noeuds dupliqués, k est la taille du motif et m est le nombre d'arcs du réseau. Par conséquent, on voit que le choix des noeuds à dupliquer est prépondérant. L'effectuer de manière aléatoire est rapide mais peu efficace. Nous nous sommes donc interessé au VERTEX FEEDBACK SET PROBLEM, problème NP-Complet, donnant le nombre minimum de noeuds à supprimer afin de rendre le graphe acyclique. Nous l'avons résolu de manière naïve par un algorithme de complexité exponentielle.

La complexité pire cas de notre algorithme est donc dépendante d'un paramètre différent que celui de QNet. Cette borne pire cas semble expérimentalement meilleure en notre faveur. De plus, cette borne est, pour notre algorithme, largement surévaluée. Il serait donc interessant de pouvoir comparer en pratique les temps de calculs de ces deux algorithmes. Malheuresement, ceci est impossible tant que les auteurs de QNet n'auront pas implémenté leur version sur les graphes.

Nous avons réussi à obtenir des résultats démontrant la validité de notre algorithme. En effet, nous avons recherché et retrouvé un motif graphe connu de l'homme dans le réseau d'interactions protéiques de la mouche. Ceci semble montrer une conservation de chemins à travers des espèces.

La thèse que je m'apprète à effectuer portera sur un sujet proche. Pour approndir ce stage, on pourrait reprendre les perspectives présentées dans la Section 4.3. On pourrait également essayer de trouver une complexité moyenne de l'algorithme, avec par exemple comme paramètre le nombre de protéines homologues moyen. On pourrait également tenter d'apporter une preuve théorique au fait que la treewidth est supérieure au nombre minimum de sommets à supprimer pour rendre le graphe acycliqe. Enfin, en ayant plus d'informations biologiques sur la topologie des requêtes, on pourrait en déduire éventuellement un nombre moyen de noeuds à dupliquer.

Durant ce stage, j'ai pu mieux intégrer la démarche de chercheur grâce à de fréquentes discussions avec mes encadrants. De plus, j'ai assisté à de nombreux séminaires, sur des sujets divers et variés (image, algorithmique, réseaux, linguistique). Mes encadrants m'ont également permis de m'intéresser à d'autres problèmes que ceux présentés dans ce rapport, dont un menant à un article en cours de soumission. J'espère que ma thèse se déroulera dans le même esprit que ce stage.

Bibliographie

[1] Protein data bank website - http ://www.rcsb.org/pdb/cgi/explore.cgi ?pid=10551087909401&pdbid=1mbo.

[2] N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the ACM (JACM)*, 42(4) :844–856, 1995.

[3] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *J. Mol. Biol*, 215(3) :403–410, 1990.

[4] M. Ashburner, CA Ball, JA Blake, D. Botstein, H. Butler, JM Cherry, AP Davis, K. Dolinski, SS Dwight, JT Eppig, et al. Gene ontology : tool for the unification of biology. The Gene Ontology Consortium. *Nat Genet*, 25(1) :25–9, 2000.

[5] Giorgio Ausiello, M. Protasi, A. Marchetti-Spaccamela, G. Gambosi, P. Crescenzi, and V. Kann. *Complexity and Approximation : Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[6] G.D. Bader. *Design and use of the biomolecular interaction network database (BIND) for storing and analyzing protein-protein interaction data*. PhD thesis, University of Toronto, 2003.

[7] J.S. Bader, A. Chaudhuri, J.M. Rothberg, and J. Chant. Gaining confidence in high-throughput protein interaction networks. *Nature Biotechnology*, 22(1) :78–85.

[8] V. Bafna, P. Berman, and T. Fujito. A 2-Approximation Algorithm for the Undirected Feedback Vertex Set Problem. *SIAM Journal on Discrete Mathematics*, 12 :289, 1999.

[9] R. Bar-Yehuda, D. Geiger, J. Naor, and RM Roth. Approximation algorithms for the feedback vertex set problem with applications to constraint satisfaction and bayesian inference. *SIAM journal on computing*, 27(4) :942–959, 1998.

[10] S. Batzoglou. The many faces of sequence alignment. *Briefings in Bioinformatics*, 6(1) :6–22, 2005.

[11] S.E. Calvano, W. Xiao, D.R. Richards, R.M. Felciano, H.V. Baker, R.J. Cho, R.O. Chen, B.H. Brownstein, J.P. Cobb, S.K. Tschoeke, et al. A network-based analysis of systemic inflammation in humans. *Nature*, 437(7061) :1032–1037, 2005.

[12] P. J. Cameron. *Combinatorics : Topics, Techniques, Algorithms*. Cambridge University Press, 1994.

[13] S.A. Cook. The complexity of theorem-proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[14] P. Dent, A. Yacoub, P.B. Fisher, M.P. Hagan, and S. Grant. MAPK pathways in radiation responses. *Oncogene*, 22 :5885–5896, 2003.

[15] B. Dost, T. Shlomi, N. Gupta, E. Ruppin, V. Bafna, and R. Sharan. QNet : A Tool for Querying Protein Interaction Networks. *RECOMB*, pages 1–15, 2007.

[16] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999.

[17] S. Fields and O. Song. A novel genetic system to detect protein–protein interactions. *Nature*, 340 :245–246, 1989.

[18] W.M. Fitch. Homology a personal view on some of the problems. *Trends in Genetics*, 16(5) :227–231, 2000.

[19] J. Flannick, A. Novak, B.S. Srinivasan, H.H. McAdams, and S. Batzoglou. Graemlin : General and robust alignment of multiple large interaction networks. *Genome Research*, 16(9) :1169, 2006.

[20] M.J. Fraunholz. Systems biology in malaria research. *Trends in Parasitology*, 21(9) :393–395, 2005.

[21] M.R. Garey and D.S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[22] A.C. Gavin, M. Boesche, R. Krause, P. Grandi, M. Marzioch, A. Bauer, J. Schultz, J.M. Rick, A.M. Michon, C.M. Cruciat, et al. Functional organization of the yeast proteome by systematic analysis of protein complexes. *Nature*, 415(6868) :141–147, 2002.

[23] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete mathematics : a foundation for computer science*. Reading, Mass. : Addison-Wesley, 1989.

[24] J. Guo, J. Gramm, F. Hüffner, R. Niedermeier, and S. Wernicke. Compression-based fixed-parameter algorithms for feedback vertex set and edge bipartization. *Journal of Computer and System Sciences*, 72(8) :1386–1396, 2006.

[25] Y. Ho, A. Gruhler, A. Heilbut, G.D. Bader, L. Moore, S.L. Adams, A. Millar, P. Taylor, K. Bennett, K. Boutilier, et al. Systematic identification of protein complexes in Saccharomyces cerevisiae by mass spectrometry. *Nature*, 415(6868) :180–183, 2002.

[26] L. Hood, JR Heath, ME Phelps, and B. Lin. Systems biology and new technologies enable predictive and preventative medicine. *Science*, 306(5696) :640–3, 2004.

[27] F. Hüffner, S. Wernicke, and T. Zichner. Algorithm engineering for color-coding to facilitate signaling pathway detection. *Proceedings of the 5th Asia-Pacific Bioinformatics Conference*, 2007.

[28] H. Jeong, S.P. Mason, A.L. Barabasi, Z.N. Oltvai, et al. Lethality and centrality in protein networks. *Nature*, 411(6833) :41–42, 2001.

[29] M. Kalaev, V. Bafna, and R. Sharan. Fast and Accurate Alignment of Multiple Protein Networks. *RECOMB*, 2008.

[30] M. KANEHISA, S. GOTO, S. KAWASHIMA, Y. OKUNO, and M. HATTORI. The KEGG resource for deciphering the genome. *Nucleic acids research*, 32 :277–280, 2004.

[31] R.M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, 43 :85–103, 1972.

[32] B.P. Kelley, R. Sharan, R.M. Karp, T. Sittler, D.E. Root, B.R. Stockwell, and T. Ideker. Conserved pathways within bacteria and yeast as revealed by global protein network alignment. *Proceedings of the National Academy of Sciences*, 100(20) :11394, 2003.

[33] Joachim Kneis, Daniel Mölle, Stefan Richter, and Peter Rossmanith. Divide-and-color. In *WG*, pages 58–67, 2006.

[34] B. Korte and J. Vygen. *Combinatorial Optimization : Theory and Algorithms*. Springer, 2002.

[35] M. Koyuturk, A. Grama, and W. Szpankowski. Pairwise local alignment of protein interaction networks guided by models of evolution. *RECOMB*, pages 48–65, 2005.

[36] D. L. Kreher and D. R. Stinson. *Combinatorial Algorithms : Generation, Enumeration, and Search*. CRC Press, 1999.

[37] L.A. Levin. Universal search problems. *Problemy Peredachi Informatsii*, 9(3) :265–266, 1973.

[38] S. Lu, F. Zhang, J. Chen, and S.H. Sze. Finding Pathway Structures in Protein Interaction Networks. *Algorithmica*, 48(4) :363–374, 2007.

[39] HW Mewes, C. Amid, R. Arnold, D. Frishman, U. Guldener, G. Mannhaupt, M. Munsterkotter, P. Pagel, N. Strack, V. Stumpflen, et al. MIPS : analysis and annotation of proteins from whole genomes. *Nucleic Acids Research*, 32(Database Issue) :D41, 2004.

[40] P.M. Pardalos, T. Qian, and M.G.C. Resende. A Greedy Randomized Adaptive Search Procedure for the Feedback Vertex Set Problem. *Journal of Combinatorial Optimization*, 2(4) :399–412, 1998.

[41] W.R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences of the United States of America*, 85(8) :2444–2448, 1988.

[42] J.B. Pereira-Leal, E.D. Levy, and S.A. Teichmann. The origins and evolution of functional modules : lessons from protein complexes. *Philosophical Transactions of the Royal Society B : Biological Sciences*, 361(1467) :507–517, 2006.

[43] S. Peri, J.D. Navarro, R. Amanchy, T.Z. Kristiansen, C.K. Jonnalagadda, V. Surendranath, V. Niranjan, B. Muthusamy, TKB Gandhi, M. Gronborg, et al. Development of Human Protein Reference Database as an Initial Platform for Approaching Systems Biology in Humans. *Genome Research*, 13 :2363–2371, 2003.

[44] Teresa Reguly, Ashton Breitkreutz, Lorrie Boucher, Bobby-Joe Breitkreutz, Gary Hon, Chad Myers, Ainslie Parsons, Helena Friesen, Rose Oughtred, Amy Tong, Chris Stark, Yuen Ho, David Botstein, Brenda Andrews, Charles Boone, Olga Troyanskya, Trey Ideker, Kara Dolinski, Nizar Batada, and Mike Tyers. Comprehensive curation and analysis of global interaction networks in saccharomyces cerevisiae. *Journal of Biology*, 5(4) :11, 2006.

[45] B. Ren et al. Genome-Wide Location and Function of DNA Binding Proteins. *Science's STKE*, 290(5500) :2306–2309, 2000.

[46] J. Scott, T. Ideker, R.M. Karp, and R. Sharan. Efficient Algorithms for Detecting Signaling Pathways in Protein Interaction Networks. *Journal of Computational Biology*, 13(2) :133–144, 2006.

[47] R. Sharan and T. Ideker. Modeling cellular machinery through biological network comparison. *Nature Biotechnology*, 24 :427–433, 2006.

[48] R. Sharan, T. Ideker, B. Kelley, R. Shamir, and R.M. Karp. Identification of Protein Complexes by Comparative Analysis of Yeast and Bacterial Protein Interaction Data. *Journal of Computational Biology - RECOMB*, 12(6) :835–846, 2004.

[49] R. Sharan, S. Suthram, R.M. Kelley, T. Kuhn, S. McCuine, P. Uetz, T. Sittler, R.M. Karp, and T. Ideker. Conserved patterns of protein interaction in multiple species. *Proceedings of the National Academy of Sciences*, 102(6) :1974, 2005.

[50] T. Shlomi, D. Segal, E. Ruppin, and R. Sharan. QPath : a method for querying pathways in a protein-protein interaction network. *feedback*, 2006.

[51] S. Tornow and HW Mewes. Functional modules by relating protein interaction networks and gene expression. *Nucleic Acids Research*, 31(21) :6283–6289, 2003.

[52] P. Uetz, Y.A. Dong, C. Zeretzke, C. Atzler, A. Baiker, B. Berger, S.V. Rajagopala, M. Roupelieva, D. Rose, E. Fossum, et al. Herpesviral Protein Networks and Their Interaction with the Human Proteome, 2006.

[53] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001.

[54] C. von Mering, R. Krause, B. Snel, M. Cornell, S.G. Oliver, S. Fields, and P. Bork. Comparative assessment of large-scale data sets of protein-protein interactions. *Nature*, 417(6887) :399–403, 2002.

Table des figures