Chapitre 5

Les méthodes indirectes de comparaisons

Tell all the Truth but tell it slant -
Success in Circuit lies ...

Poème #1129. Emily Dickinson.
The Complete Poems of Emily Dickinson.
Thomas H. Johnson (Ed.). Faber and Faber, 1975.

Ce chapitre est consacré aux méthodes indirectes de comparaison de chaînes. À trois exceptions près qui seront clairement indiquées, tous les algorithmes introduits ont été réalisés dans le cadre de ce travail.

Nous commençons par rappeler le problème, et en particulier le fait que les comparaisons entre chaînes d'un ensemble sont faites désormais en utilisant des objets externes à ces chaînes appelés des modèles, et nous le posons en termes formels. Puis nous décrivons deux des trois algorithmes élaborés à ce jour par d'autres que nous pour essayer de le résoudre.

Nous réalisons ensuite une visite générale de l'approche que nous proposons nous-mêmes. Cette visite approfondit des idées déjà suggérées par les méthodes combinatoires exactes directes présentées à la fin du chapitre 4. Nous reprenons ainsi le principe d'une construction par récurrence des objets recherchés. Ce principe a été introduit dans le chapitre précédent sous la forme d'une attribution d'étiquettes, nous le revoyons ici comme un parcours dans un arbre et comme une technique de partitionnement ou de raffinement d'ensembles. L'algorithme prototype et une première analyse intuitive de sa complexité sont alors discutés.

Après cette visite générale, nous passons à une visite détaillée des méthodes que nous avons élaborées. Les ressemblances entre mots que ces méthodes permettent de dégager correspondent à celles définies dans la dernière section du chapitre 3. Les algorithmes proposés se divisent en deux grands groupes, celui pour lequel les symboles de l'alphabet constituent l'unité de comparaison entre mots dans les chaînes, l'autre pour lequel cette unité est elle-même composée de mots. Le premier groupe se subdivise encore en trois sous-groupes, le premier reflétant une idée biologique de la ressemblance ayant pour base une distance (nombre de mutations séparant deux objets), la seconde essayant de se libérer de tout point de vue sur la nature de ce qui est semblable afin d'optimiser, dans une certaine limite, sur l'alphabet des modèles, et enfin le dernier qui tente de conjuguer les deux approches.

Nous n'abordons pas ici le problème du traitement à appliquer aux objets trouvés (les modèles et leurs occurrences). Ce chapitre ne comporte non plus ni les critiques que nous désirons porter nous-mêmes sur le travail accompli, ni une discussion de ce qu'il serait intéressant d'essayer de réaliser encore à partir des bases établies ici. Ces aspects sont importants mais ne seront discutés que dans les deux derniers chapitres.

Celui que nous présentons maintenant comporte des aspects techniques, il nous semble en effet utile de discuter non seulement les idées générales mais aussi les détails d'implémentation des divers algorithmes, ainsi que les variantes et contraintes qu'il est possible et intéressant d'y introduire. Un effort a cependant été fait dans l'organisation du chapitre pour qu'un lecteur non intéressé par ce niveau de précision puisse les éviter. L'essentiel des idées de ce chapitre se trouve ainsi regroupé dans la section consacrée à un aperçu général de nos méthodes et dans le tableau-résumé de la fin.

5.1 Le problème formellement posé

Commençons par rappeler le problème spécifique que nous essayons de résoudre. Il s'agit toujours bien sûr de comparer plusieurs chaînes simultanément entre elles afin de repérer des mots présents dans ces chaînes qui se 'ressemblent' suivant une certaine définition de similarité. Dans la seconde partie du chapitre précédent, nous avons présenté diverses méthodes permettant de réaliser ce repérage. D'une manière un peu grossière, nous pouvions y distinguer deux approches. La première avait pour caractéristique d'autoriser une recherche souple, elle était cependant soit exacte mais trop coûteuse en temps de calcul (programmation dynamique appliquée au cas local), soit rapide mais faisant appel à des heuristiques (par exemple MACAW — section 4.2.2.1.2.2). La seconde, composée des méthodes locales combinatoires (section 4.2.3), était quant à elle rapide et exacte mais relativement peu souple. Les mots repérés étaient soit des mots identiques ou équivalents, soit des mots similaires mais liés les uns aux autres par une relation simple.

Notre problème est donc d'introduire de plus en plus de flexibilité dans la notion de similarité entre mots tout en demeurant exhaustifs dans la recherche de ces mots et relativement efficaces.

Ce que nous voulons c'est trouver tous les groupes de mots similaires suivant une des quatre définitions de similarité données dans la section 3.5 du chapitre 3 (définitions 3.5.3, 3.5.8, 3.5.12 et 3.5.15). Pour réaliser cela, les mots ne sont plus comparés directement entre eux, mais le sont à des objets externes que nous avons appelé des modèles dont une description a été fournie dans la section 3.5. Les groupes de mots similaires le sont parce qu'il existe un tel modèle auquel tous 'ressemblent' d'une certaine façon.

Une définition formelle du problème général que nous souhaitons résoudre peut alors être la suivante :

Le problème de la construction des modèles *Étant donné un ensemble de N chaînes $s1, s2, ..., sN \in \Sigma^*$ et un entier q entre 2 et N (représentant une contrainte de quorum — définition 3.3.5), les problèmes que nous nous proposons de résoudre sont :*

Problème 1 *Trouver tous les modèles, ainsi que leurs ensembles d'occurrences, d'une longueur k qui sont présents (L, EL, ECL ou w-présents) dans au moins q des chaînes de l'ensemble;*

Problème 2 *Trouver la plus grande longueur k_{max} pour laquelle il existe au moins un modèle présent (à nouveau, L, EL, ECL ou w-présent) dans au moins q des chaînes de l'ensemble et résoudre le problème 1 pour $k = k_{max}$.*

La question est maintenant, comment apporter une solution à ce problème général?

Sous des formes simples, des objets très semblables à des modèles ont été utilisés par d'autres avant nous, et nous commençons par présenter leur approche. Nous appelons cette approche naïve car elle consiste à engendrer tous les modèles possibles et à les rechercher dans l'ensemble des chaînes à comparer.

5.2 Approche naïve : engendrer tous les modèles

5.2.1 Des modèles comme des mots : l'algorithme de Waterman

Dans une série d'articles publiés depuis 1984 [Galas *et al.*, 1985] [Waterman, 1984b] [Waterman *et al.*, 1984b] [Waterman, 1986] [Waterman, 1989] [Waterman, 1990], Waterman a introduit l'idée d'utiliser un objet externe contre lequel sont effectuées toutes les comparaisons des mots présents dans un ensemble de chaînes. À notre connaissance, il est le premier à avoir employé cette technique dans une approche purement algorithmique (des approches semblables ont été élaborées dans le domaine de la reconnaissance de formes ou de l'apprentissage mais avec une philosophie différente [Gascuel and Danchin, 1986]).

Pour Waterman, un modèle, qu'il appelle un motif consensuel ('consensus pattern'), est un mot défini sur le même alphabet que celui des chaînes à comparer. L'application la plus immédiate de sa méthode concerne les séquences d'acides nucléiques, mais elle est également valable pour les protéines. Par souci de clarté, nous la décrivons sur l'exemple des séquences nucléiques.

L'idée de l'algorithme est très simple. Elle consiste à engendrer tous les modèles possibles, c'est-à-dire tous les mots possibles de taille k sur $\Sigma = \{A, C, G, T \text{ ou } U\}$ pour un k donné, et à rechercher ces mots en autorisant un certain nombre d'erreurs (substitutions et éventuellement trous). Un mot u dans une des chaînes est ainsi une occurrence d'un modèle m si u présente au plus e erreurs avec m. Cette approche consiste donc à rechercher les modèles L-présents dans un ensemble de chaînes suivant la définition 3.5.2. Si aucune contrainte supplémentaire n'était imposée, les groupes de mots similaires trouvés correspondraient à ceux de la définition 3.5.3.

Comme Waterman doit engendrer tous les modèles possibles afin de localiser ceux qui l'intéressent (nous verrons bientôt quels sont ceux qu'il va considérer comme intéressants), il est limité dans la longueur k des modèles qu'il peut identifier. En pratique, cette longueur est en général fixée entre 4 et 6 pour les acides nucléiques. Pour accélérer les calculs et parce que rechercher des modèles (et leurs occurrences) relativement petits sur tout l'espace des chaînes peut produire beaucoup de 'bruit', Waterman limite de plus sa recherche à l'intérieur d'une fenêtre de taille W placée sur chacune des chaînes. Bien sûr, cela suppose que ces dernières on pu être pré-alignées sur la base d'un critère biologique quelconque (par exemple, le début de transcription de l'ADN en ARN [Galas *et al.*, 1985]). Le choix de la taille W est un compromis entre efficacité et sensibilité de la recherche. La méthode consiste à chercher pour chaque modèle possible de longueur k les mots de même longueur (si seules les substitutions sont autorisées) ou de longueur égale plus ou moins e (si les trous sont également permis) qui se situent à l'intérieur de la fenêtre et qui présentent au plus e erreurs avec le modèle. Chacun des modèles m reçoit alors un score donné par la formule suivante :

$$score(m) = \sum_{0 \leq d \leq e} \lambda_d q_{md}$$

où q_{md} est le nombre de chaînes pour lequel la meilleure occurrence de m se trouve à une distance d de m et λ_d pour $0 \leq d \leq e$ est un ensemble de poids fixés à l'avance. En général, nous avons :

$$\lambda_d = 1 - \frac{d}{k}.$$

Le modèle ayant le meilleur score (en cas d'égalité, un modèle est choisi arbitrairement) constitue le modèle 'intéressant' pour cet emplacement de la fenêtre. Celle-ci est alors glissée vers la droite et la recherche d'un autre modèle se poursuit. Le glissement de la fenêtre peut être effectué de diverses manières. Par exemple, celle-ci peut se déplacer d'une position seulement vers la droite sur chacune des chaînes [Galas *et al.*, 1985] [Waterman, 1989]. Les modèles trouvés à chaque glissement peuvent alors se superposer ou même être identiques du point de leurs 'noms' et/ou de leurs ensembles d'occurrences [Galas *et al.*, 1985] [Waterman, 1989] (voir la figure 5.1) bien que l'on puisse également exiger que cela ne soit pas le cas [Waterman, 1986]. Dans une autre version [Waterman, 1986], le modèle trouvé par rapport au placement initial de la fenêtre constitue un premier point d'ancrage des chaînes, qui sont alors alignées sur ce modèle (ses occurrences), puis la fenêtre est glissée vers la droite pour venir se placer à la première position qui suit ces occurrences sur les chaînes. La recherche d'un nouveau modèle dont les occurrences sont forcément différentes de celles du premier recommence alors (voir la figure 5.2). Outre le fait qu'il permet de repérer certains consensus, ce second processus produit ainsi naturellement un alignement (heuristique) des chaînes.

L'extension de cette méthode à la comparaison de séquences de protéines est immédiate. Les modèles sont toujours des mots (en général de longueur 4) définis cette fois sur l'alphabet des acides aminés, et le score d'un modèle correspond à la somme des scores de tous les alignements du modèle avec sa meilleure occurrence située à l'intérieur de la fenêtre sur chaque chaîne [Waterman, 1989]. Il correspond donc à un score Étoile Centrale (voir la section 3.4.1.5 du chapitre 3).

La complexité en temps de l'algorithme est $O(n.N.W^2.B)$, où n et N sont toujours la longueur moyenne et le nombre des chaînes respectivement et où B est une fonction de $\mid \Sigma \mid^k$. Suivant l'implémentation qui en est faite, l'algorithme peut avoir une complexité en espace très réduite, en $O(k)$.

5.2.2 Des modèles comme des mots à positions fixes et positions variables : l'algorithme de Smith

L'idée d'un modèle n'apparaît que de manière implicite dans l'article de Smith destiné à la comparaison de protéines [Smith *et al.*, 1990]. Il définit ainsi un modèle, qu'il appelle un motif, comme étant composé d'un arrangement de deux (ou plus de deux) acides aminés séparés entre eux par des distances fixes. Par exemple, $m = A...Q.H..I$ est un modèle (motif) composé de 4 acides aminés (A, Q, H, I) séparés les uns des autres par une distance égale à 3, 1 et 2 acides aminés respectivement. Un mot dans une chaîne est une occurrence du modèle m s'il peut lui-même être écrit sous la forme $A...Q.H..I$ où les points représentent n'importe quel acide aminé.

L'algorithme de Smith consiste, étant donné deux paramètres k et d (des entiers non nuls), à engendrer tous les modèles de la forme $a_1 d_{12} a_2 d_{23} a_3 ... d_{(k-s)k} a_k$ où $a_i \in \Sigma$ et $0 \leq d_{(d-i)i} \leq d$, puis à rechercher l'occurrence de ces modèles dans les chaînes. Les a_i correspondent aux positions fixes des modèles et les acides aminés entre deux d'entre eux aux positions variables. Pour réaliser cette recherche, Smith utilise un tableau de taille $\mid \Sigma \mid^k d^{k-1}$ où tous les modèles possibles sont stockés. Il n'a besoin alors d'effectuer qu'un seul passage sur chacune des chaînes afin de repérer les occurrences des modèles, avec un arrêt de durée d^{k-1} sur chaque position. La complexité en temps de l'algorithme est ainsi $O(max\{n.N.d^{k-1}, \mid \Sigma \mid^k .d^{k-1}$(initialisation du tableau)$\})$ et celle en espace $O(\mid \Sigma \mid^k .d^{k-1} + n.N.d^{k-1})$, le dernier facteur indiquant que,

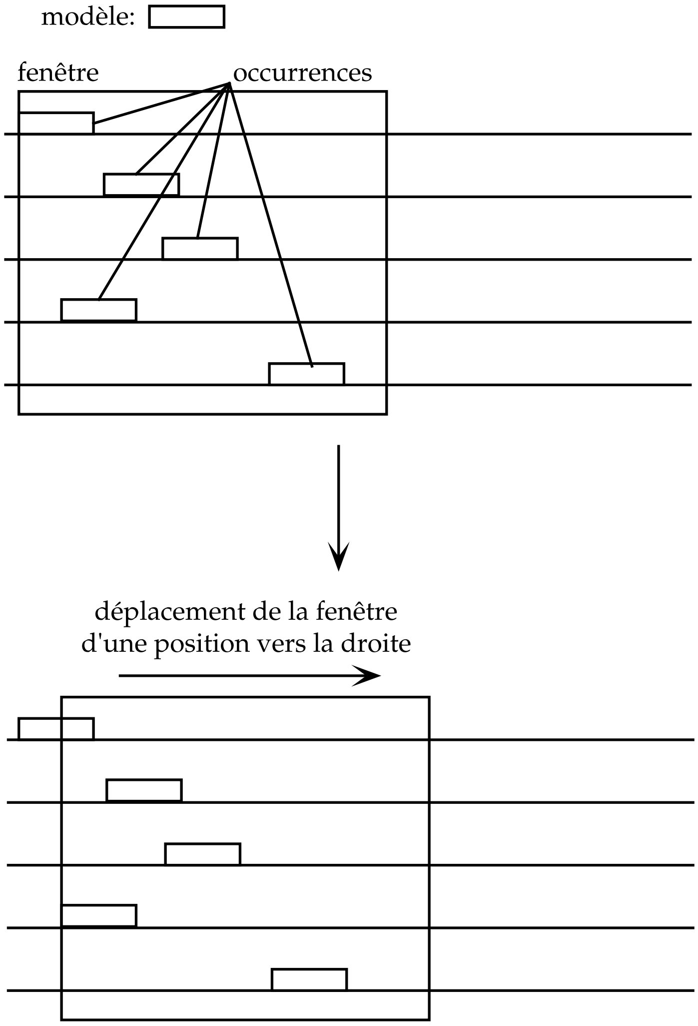

Figure 5.1: Construction des modèles dans l'algorithme de Waterman : premier type de glissement de la fenêtre.

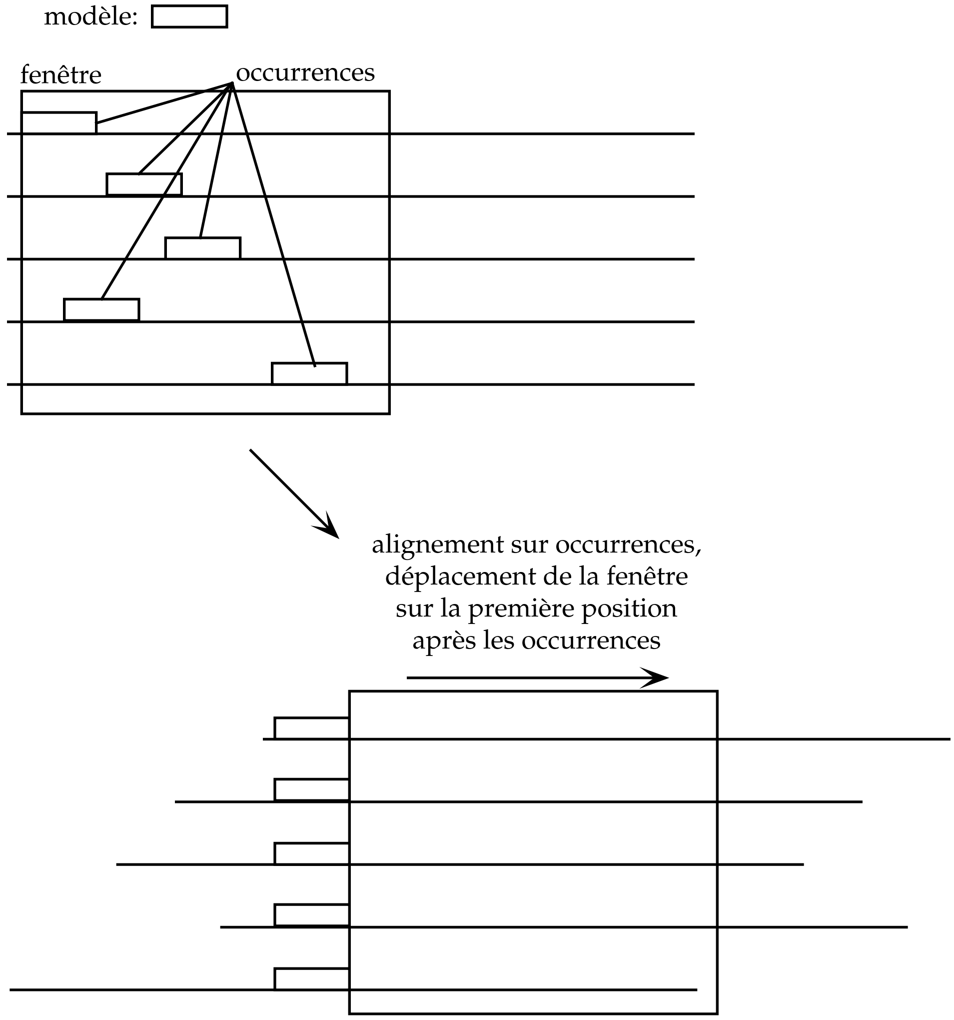

Figure 5.2: Construction des modèles dans l'algorithme de Waterman : second type de glissement de la fenêtre.

dans la pire des hypothèses (si les chaînes sont composées du même symbole), chaque position peut faire référence à d^{k-1} mots appartenant à des modèles différents. Il est clair qu'une telle approche n'est praticable que pour des petites valeurs de k (en général 3 ou 4) et de d (en général 10). De plus, afin de limiter le nombre de modèles trouvés, seuls ceux satisfaisant une contrainte de quorum sont retenus, et parmi ceux-là, seuls ceux ayant le meilleur score sont reportés par l'algorithme, où le score d'un modèle correspond à un score SP de ses occurrences. Par contre, la recherche des modèles se fait sur tout l'espace des chaînes simultanément et non plus uniquement à l'intérieur d'une fenêtre de taille fixe comme cela est le cas pour Waterman. Aucune de ces deux méthodes n'est cependant assez flexible, soit dans la définition de ressemblance entre mots, soit dans sa réalisation pratique (en particulier elles ne permettent de résoudre que le Problème 1), et le reste de ce chapitre est consacré à l'introduction de méthodes indirectes de comparaison plus générales et plus efficaces.

5.3 Aperçu général

5.3.1 La construction par récurrence des modèles

5.3.1.1 La construction comme un parcours dans un arbre

5.3.1.1.1 L'arbre de tous les modèles et celui des modèles valides

Nous allons examiner ce que résoudre le problème général signifie en considérant le cas le plus simple où les modèles sont des mots (ou des sous-ensembles unitaires de l'alphabet) et où aucune erreur (substitution ou trou) n'est autorisée entre un modèle et ses occurrences. Le problème est alors simplement de localiser tous les mots répétés exactement dans $N \geq 1$ chaînes, où le nom des modèles associés à chaque ensemble de mots est simplement le nom de l'un quelconque de ces mots. C'est donc le problème traité par KMR (section 4.2.3.4). Pour simplifier, nous supposons également que les $N \geq 1$ chaînes que nous voulons comparer sont concaténées en une seule (voir l'extension de KMR réalisée par Landraud, section 4.2.3.4) et nous oublions pour l'instant les détails d'implémentation permettant d'éliminer les mots répétés à cheval sur deux chaînes et de vérifier sur laquelle des chaînes est effectivement présent un modèle.

Dans ces conditions, considérons l'arbre de tous les modèles possibles de longueur au plus k (un exemple d'un tel arbre est donné dans la figure 5.3 pour $k = 3$ et un alphabet constitué de 2 symboles). Supposons que chaque nœud de l'arbre représente un récipient contenant les positions d'une chaîne s où le modèle qui donne son nom au nœud est présent de manière exacte. Si s était une chaîne aléatoire et infinie, chaque nœud comporterait au moins une telle position. Et trouver tous les modèles, ainsi que leurs occurrences, présents au moins une fois dans s correspondrait à donner la liste de toutes les feuilles de cet arbre (c'est-à-dire à donner leur noms et à énumérer le contenu de leurs récipients).

Bien sûr, cela est un cas extrème, puisque s est en pratique une chaîne de longueur finie. De plus, s représente en fait le plus souvent un ensemble de chaînes et les modèles qui nous intéressent en général sont ceux qui satisfont une contrainte de quorum q. Nous ne devons ainsi fournir que la liste des feuilles de l'arbre correspondant aux modèles qui sont effectivement présents au moins q fois dans la chaîne (nous appelons ces modèles des modèles valides). En prenant à nouveau comme exemple un alphabet à deux symboles, nous montrons dans la figure 5.4 à quoi ressemble cet arbre pour un ensemble de trois chaînes et pour $q = 3$. Observons qu'il s'agit d'un arbre déjà beaucoup plus élagué que celui de la figure 5.3. Il serait encore plus élagué pour des valeurs plus élevées de k.

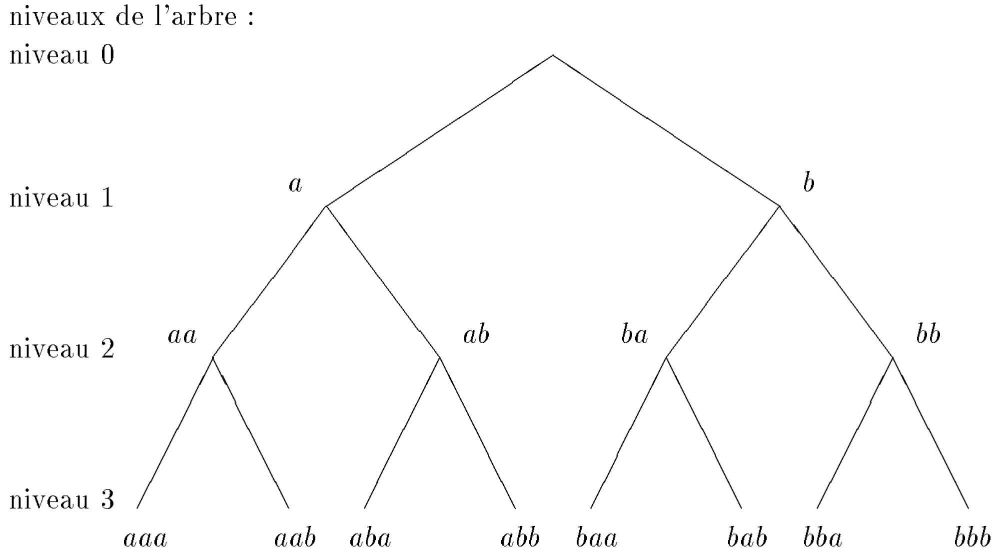

Figure 5.3: Arbre de tous les modèles de longueur au plus 3 sur un alphabet à 2 symboles.

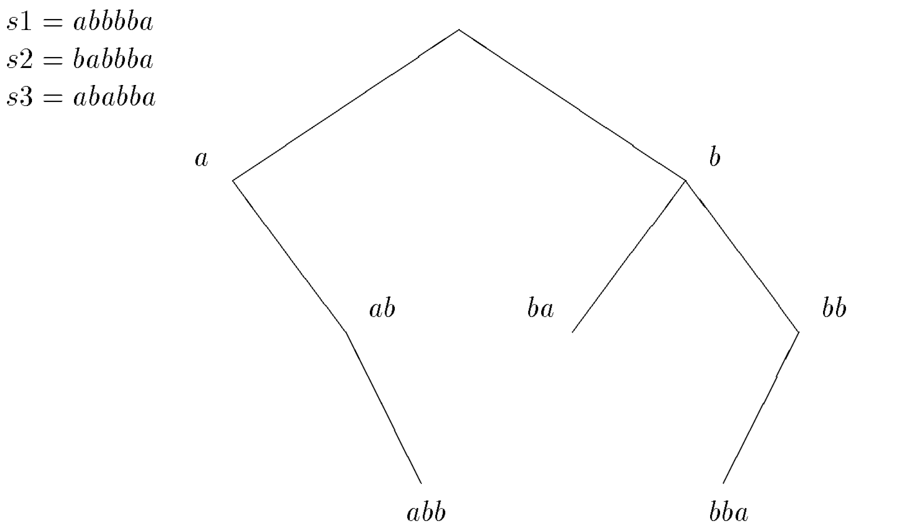

Figure 5.4: Arbre de tous les modèles valides de longueur au plus 3 sur un alphabet à 2 symboles pour un ensemble de 3 chaînes et avec $q = 3$.

Le degré d'élagage de l'arbre va en fait dépendre non seulement de k mais aussi des chaînes, de la valeur attribuée à q et de l'autorisation ou non de substitutions et d'erreurs. D'une façon générale cependant, l'arbre des modèles valides de longueur k va contenir beaucoup moins de nœuds que l'arbre de tous les modèles.

Ces considérations sont très importantes car il est évident que pour obtenir le résultat désiré, c'est-à-dire fournir une liste des feuilles de l'arbre de tous les modèles valides (par opposition à tous les modèles possibles), nous devons traverser cet arbre d'une manière ou d'une autre. En effet, la seule façon que nous aurions de connaître directement le contenu d'un récipient à un nœud quelconque (excepté ceux localisés au premier niveau) serait de construire une table indexée ou de hachage comme cela est fait par Leung, Martinez ou Santibanez (voir la section 4.2.3.2), ou encore Waterman ou Smith (section 5.2). Cela pourrait être très inefficace (puisque tous les nœuds ne sont pas présents) et très coûteux en espace pour des arbres avec plus de 3 ou 4 niveaux. Une façon plus naturelle et performante de calculer le contenu d'un récipient à n'importe quel niveau en-dessous du premier est donc d'utiliser comme information le contenu des récipients des nœuds situés à des niveaux moins élevés dans l'arbre (ceux qui se trouvent plus près de la racine de l'arbre) et qui auraient été déterminés dans une étape antérieure. Dans le cas d'une relation d'identité et lorsqu'aucune erreur n'est autorisée, un arbre ou un automate des suffixes pourraient également être employés mais ces structures sont relativement lourdes à établir alors que l'arbre des modèles est parcouru sans jamais être effectivement construit. Une partie de l'information que cet arbre représente est bien sûr maintenue en mémoire par les algorithmes comme KMR/KMRC ou ceux que nous allons proposer ici, nous verrons un peu plus loin quelle partie exactement, mais celle-ci ne compose qu'un sous-ensemble réduit de toute l'information qui doit être stockée dans une structure comme un arbre ou un automate des suffixes. Par ailleurs, la construction et l'utilisation de ces dernières structures ne permet pas facilement, du moins pour l'instant, de tenir compte de relations de similarité plus complexes (substitutions conservatives modélisées par une couverture, erreurs, couverture combinatoire pondérée, similarité basée sur des mots).

Observons que l'arbre des modèles valides n'est pas connu a priori. Cependant, il peut être facilement déterminé en réalisant un parcours de l'arbre de tous les modèles possibles qui s'arrête aussitôt qu'un nœud est rencontré dont le modèle correspondant ne vérifie pas la contrainte de quorum. En effet, si le modèle qui donne son nom à un nœud x ne vérifie pas ce quorum, aucun modèle de plus grande longueur l'ayant comme préfixe ne peut lui-même vérifier le quorum. Le sous-arbre de l'arbre de tous les modèles possibles ayant le nœud x comme racine peut donc être coupé. De cette manière, ce qui est réellement parcouru est l'arbre des modèles valides seulement, plus éventuellement $\mid \Sigma \mid$ nœuds (dans ce cas les modèles sont des mots) à chaque feuille de l'arbre comme des 'regards-en-avant' (en anglais, des 'look-aheads'). En fait, nous verrons qu'il est possible dans certains cas de lancer moins de regards en avant.

Les modèles et leurs ensembles d'occurrences sont ainsi trouvés récursivement.

5.3.1.1.2 Le parcours dans l'arbre : en largeur ou en profondeur

Considérons toujours le cas le plus simple où les modèles sont des mots et la relation de ressemblance entre ceux-ci et leurs occurrences est l'identité (aucune erreur n'est donc autorisée). Nous venons de voir que parcourir l'arbre des modèles valides (ceux qui vérifient un certain quorum q) afin de fournir la liste des feuilles de niveau k ou k_{max} (donner leurs noms et énumérer le contenu de leurs récipients) correspond exactement à ce que fait KMR et les classes d'équivalence que l'algorithme établit sont implicitement associées à des modèles.

La façon la plus efficace de traverser cet arbre pour atteindre ses feuilles utilise alors, dans ce cas, le lemme 4.2.1 ou le lemme 4.2.2 de KMR avec $a = b$ tant que cela est possible, puis le lemme 4.2.1 avec $a \neq b$. Utiliser le lemme 4.2.1 ou le lemme 4.2.2 avec $a = b$ correspond en fait à réaliser une exploration en largeur de l'arbre où seuls les niveaux qui sont des puissances de 2 sont visités (c'est-à-dire le contenu de leurs récipients calculés).

Lorsque les modèles sont maintenant explicitement introduits ainsi que des définitions plus flexibles de la ressemblance entre eux et les mots dans les chaînes, des lemmes souvent très semblables à ceux de KMR (ou KMRC) peuvent être établis permettant de calculer l'ensemble des occurrences d'un modèle quelconque à partir des ensembles d'occurrences de modèles strictement plus courts. Ces lemmes seront donnés plus loin mais il est relativement aisé de voir dès à présent que, lorsque des erreurs sont autorisées (définitions 3.5.3 et 3.5.8 de ressemblance) ou une pondération introduite sur les éléments d'une couverture (définition 3.5.12), seule une forme modifiée du lemme 4.2.1 peut être employée pour construire les modèles de longueur k à partir de ceux de longueur $k' < k$. Nous ne pouvons plus en effet réaliser une superposition de deux occurrences afin d'en obtenir une troisième qui garantisse que cette dernière continuera de vérifier les mêmes propriétés par rapport à son modèle que les deux premières par rapport aux leurs (nombre d'erreurs ou pondération des éléments de la couverture en-dessous de la valeur maximale permise — voir la figure 5.5). En revanche, lorsqu'un système de score des sous-mots d'une occurrence est utilisé (définition 3.5.15), c'est une forme modifiée du lemme 4.2.2 qui est seule applicable, à condition que b soit <u>strictement</u> plus petit que a (voir la figure 5.6).

Dans le premier cas, nous pouvons donc toujours doubler la longueur des modèles construits à chaque étape, et dans le second cas nous pourrions presque la doubler (i.e. nous pourrions l'augmenter de $k - w + 1$ où w est la longueur des sous-mots dont le score doit être au-dessus d'un certain seuil). Cependant lorsqu'une telle opération de dédoublement n'est plus possible et qu'il faille utiliser la superposition (lemme 4.2.2) pour continuer à étendre les modèles, alors la longueur de ces derniers ne peut plus être incrémentée que d'une unité à chaque fois jusqu'à ce que la longueur désirée soit atteinte, ou jusqu'à la plus grande longueur pour laquelle cette contrainte est encore vérifiée, sauf à garder en mémoire les ensembles d'occurrences de tous les modèles pour plusieurs longueurs de ces derniers.

De plus, si nous explorions l'arbre en largeur en avançant par puissance de 2 (ou presque), nous aurions à nouveau besoin d'obtenir et puis de stocker tous les ensembles d'occurrences de tous les modèles de longueur k qui se trouvent L/EL/ECL/w-présents au moins q fois dans les chaînes avant de pouvoir construire les modèles de longueur $2k$ (ou $2k - w + 1$). Lorsque la relation de ressemblance entre modèles et mots est très flexible (ce que nous souhaitons), les modèles de longueur petite (jusqu'à 4 ou 5) sont presque tous présents et chacun possède beaucoup d'occurrences. Un tel mode de parcours pourrait en conséquence exiger trop d'espace puisque tous les nœuds du niveau k sont nécessaires afin de construire les nœuds du niveau $2k$ (ou $2k - w + 1$).

Un moyen plus économique en mémoire pour construire ces modèles consiste donc à réaliser un parcours en profondeur de l'arbre des modèles. En effet, si au lieu de doubler la longueur des modèles à chaque étape, nous étendons chaque modèle séparément vers la droite d'une unité à la fois, alors pour obtenir les ensembles d'occurrences d'un modèle donné, nous n'avons besoin que de l'ensemble d'occurrences de son préfixe, plus les ensembles d'occurrences de tous les modèles de longueur 1 ou w. Cela correspond à un parcours dans l'arbre dont la complexité en temps présente un rapport $\frac{k}{\log k}$ avec le parcours réalisé 'à la façon KMR'. Comme k est en pratique petit (autour de 10-20), ce rapport est faible. Il est également relativement négligeable par rapport à $n.N$ (où n est la longueur moyenne des chaînes).

Quel que soit le mode de parcours adopté, celui-ci est bien sûr associé à une certaine opération qui doit être effectuée à chaque nœud x afin de pouvoir descendre vers ses nœuds-fils

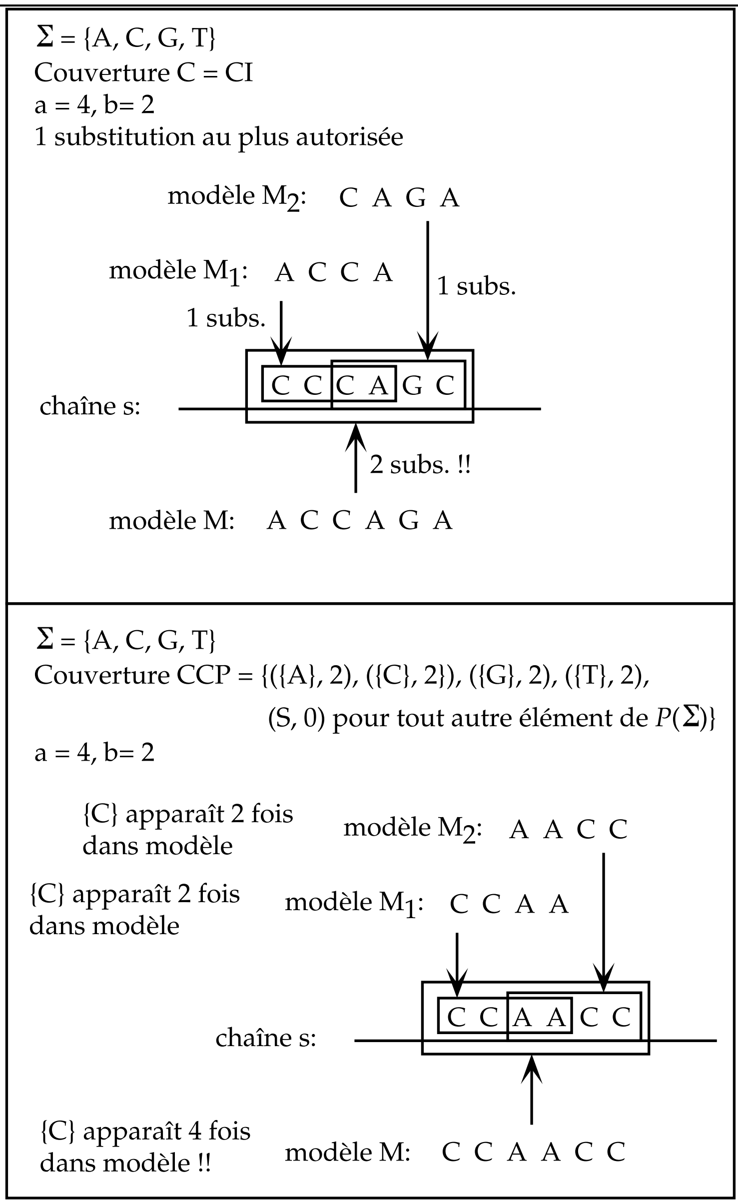

Figure 5.5: Problème si le second lemme de KMR est utilisé (avec $b < a$) lorsque des erreurs ou une pondération des ensembles de la couverture sont introduites.

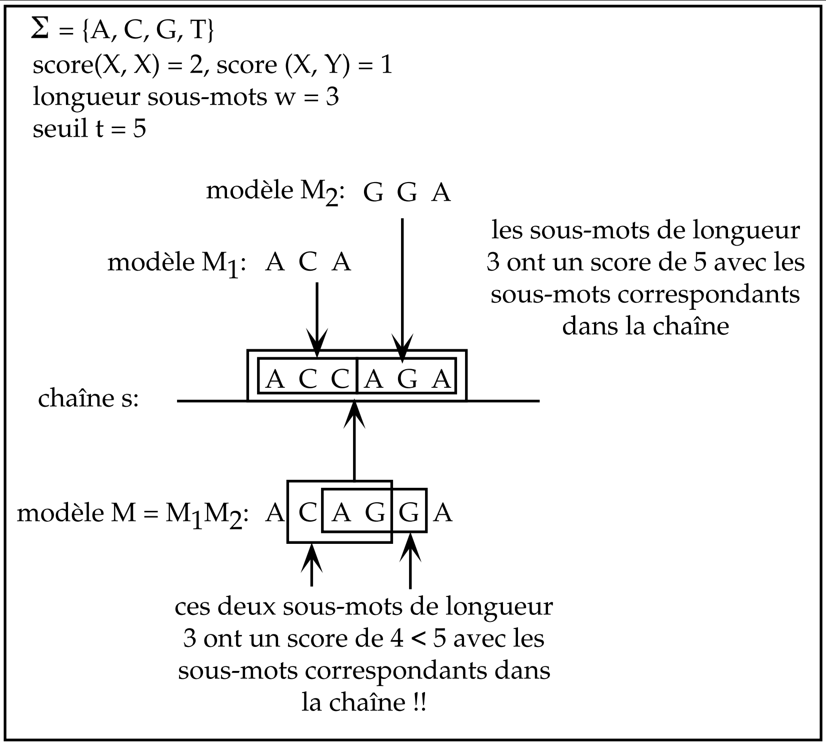

Figure 5.6: Problème si le premier lemme de KMR est utilisé lorsque la définition de ressemblance ayant pour unité de comparaison des mots est adoptée.

dans l'arbre des modèles. Cette opération peut être vue comme une attribution d'étiquettes, c'est ce que nous avons fait dans la section 4.2.3.4. Elle peut également être décrite comme un partitionnement d'ensembles (de positions dans les chaînes) lorsque la relation entre modèles et mots est l'identité ou une relation d'équivalence, ou comme un raffinement de ces mêmes ensembles dans tous les autres cas.

5.3.1.2 Le construction comme un partitionnement/raffinement d'ensembles

Considérons un parcours dans l'arbre des modèles valides. À nouveau ici, nous commençons par supposer que nous travaillons avec une relation de similarité entre modèles et mots qui est l'identité et que les chaînes sont concaténées en une seule, notée s. Supposons par ailleurs également que nous avons atteint un certain nœud x situé au niveau $k-1$. Cela signifie que nous venons de construire un modèle valide m (les modèles sont dans ce cas des mots) de longueur $k-1$, c'est-à-dire, nous venons de calculer son ensemble d'occurrences dans les chaînes. Construire le modèle de longueur k ayant m comme préfixe (i.e. établir quelles sont ses occurrences), et donc passer au niveau supérieur dans l'arbre, peut être fait en plaçant dans le récipient associé au nœud étiqueté par ma pour tout $a \in \Sigma$ les positions i du récipient associé à m pour lesquelles $s_{i+m} = a$. Chaque position d'une occurrence de m dans s va donc aller se placer dans un et un seul des récipients rattachés aux modèles ma de longueur k. Ce placement est simplement déterminé par le symbole a qui suit l'image u associée à chaque occurrence de m (voir la figure 5.7). Ce processus représente bien une opération de partitionnement, suivant Σ, réalisée sur l'ensemble des occurrences de m. Et traverser l'arbre des modèles correspond donc ainsi à effectuer une telle opération sur chaque nœud visité de l'arbre.

Lorsque la relation entre modèles et mots n'est plus l'identité, ni une relation d'équivalence, une position donnée d'un récipient situé à un certain niveau $k-1$ peut glisser vers plusieurs récipients différents du niveau k, ou même glisser plusieurs fois vers un même récipient de ce niveau. Des exemples de ces deux situations sont présentés dans les figures 5.8 et 5.9. Ce phénomène sera plus précisément décrit lorsque nous entrerons dans le détail de chacun des algorithmes. Nous y montrerons alors que dans certaines situations, ce problème de la 'multiplication' d'une position peut être en partie contourné. Parcourir l'arbre des modèles équivaut encore une fois à réaliser une certaine opération sur les contenus des récipients rattachés à chaque nœud de l'arbre, simplement l'opération n'est plus une partition mais fait référence à un raffinement des ensembles de positions associés aux modèles.

Observons que dans ce parcours dans l'arbre des modèles, il n'est pas toujours nécessaire de suivre un nombre de branches égal à $\mid \Sigma \mid$ (si les modèles sont des mots) ou p (si les modèles sont des produits cartésiens d'ensembles d'une couverture) à partir d'un nœud x vers ses nœuds fils. Cela a été suggéré déjà dans le cas d'une relation d'identité (il suffit de regarder quels symboles sont effectivement présents dans les chaînes), mais ce n'est pas la seule situation où une telle économie est possible. Cette remarque a un effet sur la complexité de l'algorithme et sera précisée plus loin.

Par contre, il paraît assez intuitif de considérer que le parcours dans l'arbre va nécessiter dans le pire des cas n étapes si les modèles ne sont rallongés que d'une unité à la fois, et qu'elle ne peut être réalisée en $\log n$ étapes que lorsqu'est adoptée l'idée de KMR de doubler la longueur des mots répétés/modèles à chaque fois (ce qui correspond à visiter uniquement les niveaux de l'arbre qui sont une puissance de 2).

Cette intuition est fausse. Dans le cas d'un partitionnement au moins (i.e., lorsque la relation est l'identité), il a été montré par Aho [Aho *et al.*, 1974] qu'une suite de n partionnements demande au plus $O(n. \log n)$ opérations. Aho observe en effet que le partitionnement d'un

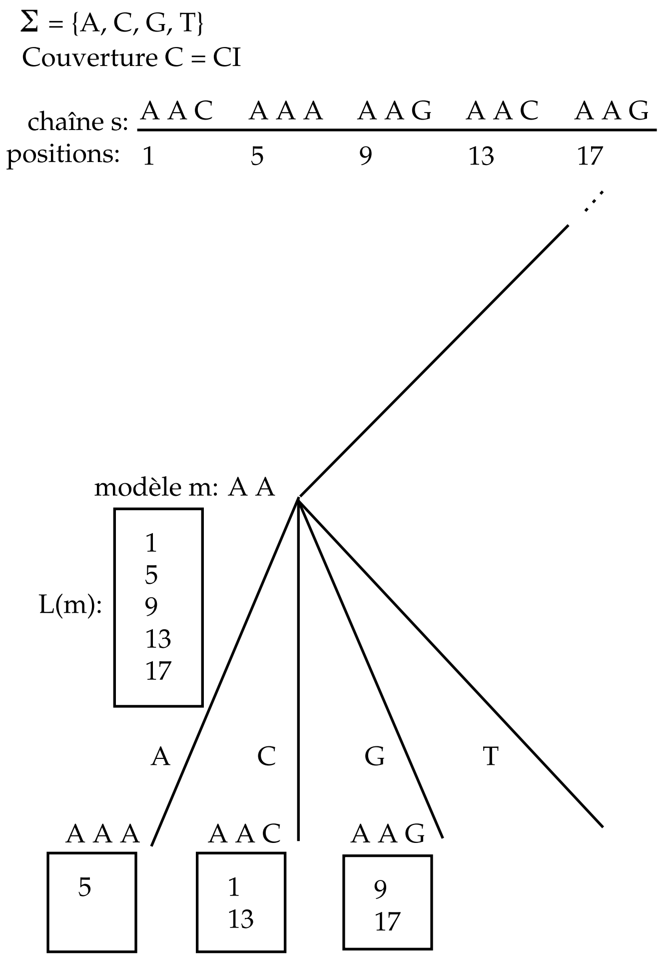

Figure 5.7: Partionnement des positions de début des occurrences d'un modèle m de longueur k afin d'obtenir les occurrences du modèle de longueur $k+1$ ayant m comme préfixe.

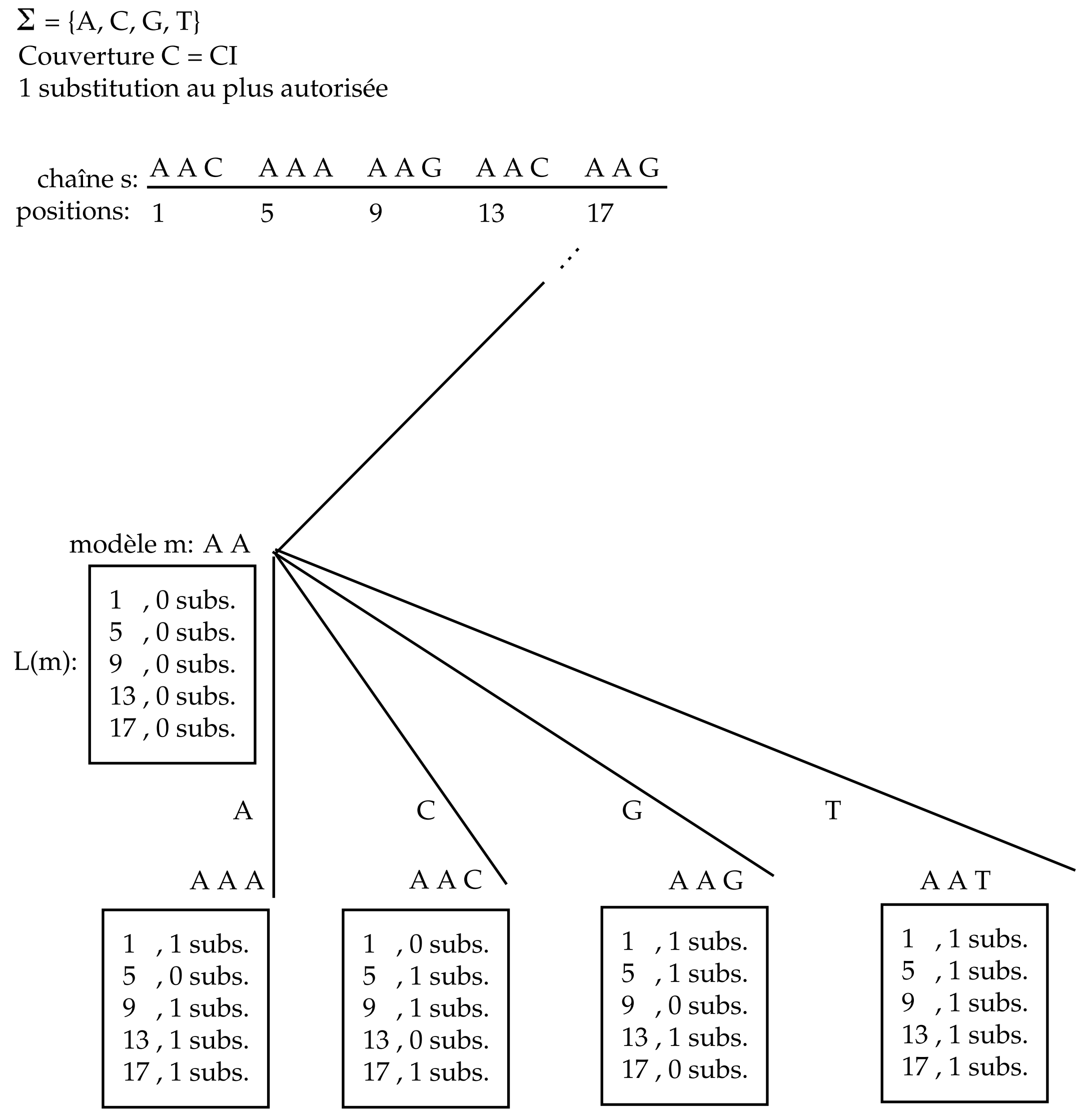

Figure 5.8: 'Multiplication' des positions : glissement vers plusieurs des récipients du niveau suivant de l'arbre.

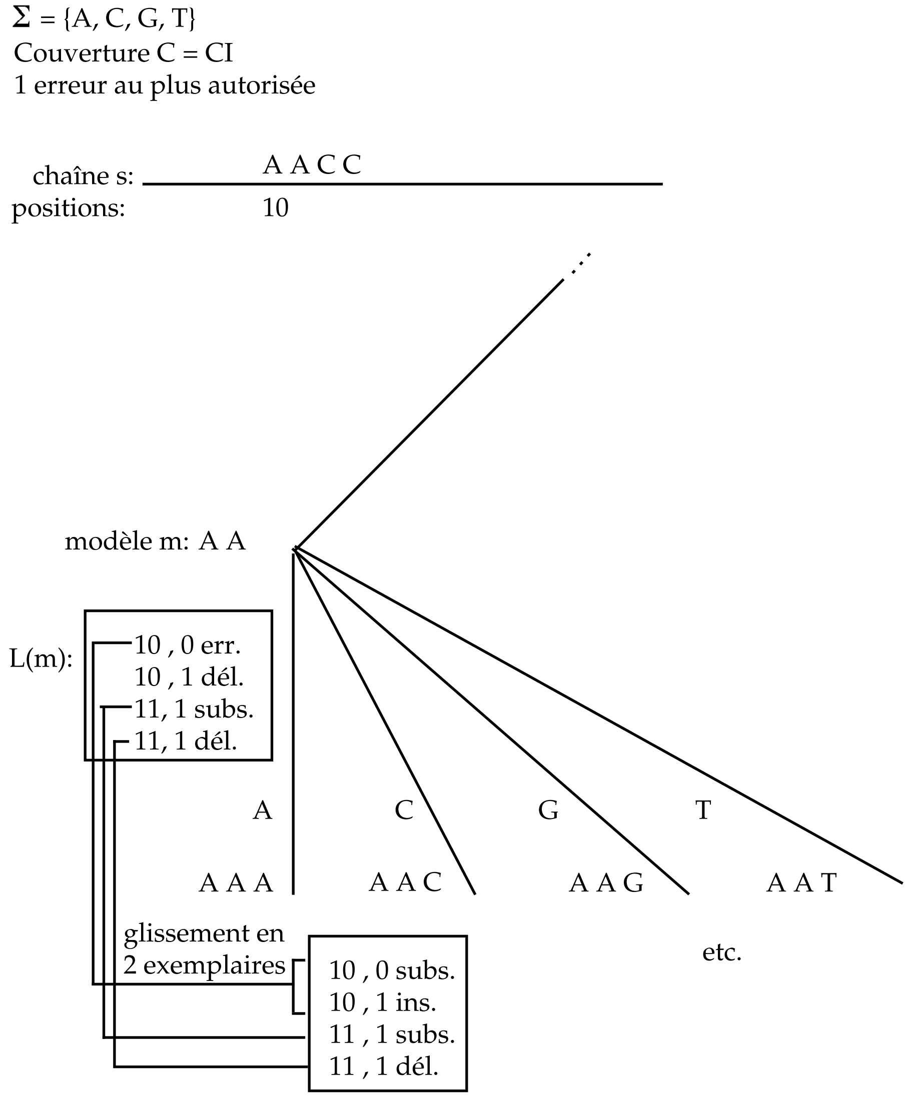

Figure 5.9: 'Multiplication' des positions : glissement en plusieurs 'exemplaires' vers un (même) récipient du niveau suivant de l'arbre.

/* Entrée */

q : contrainte de quorum

k : une longueur fixe ou la plus grande possible

/* Sortie */

Tous les modèles de longueur k qui ont des occurrences dans au moins q des chaînes

Algorithmes pour résoudre le problème général :

1. Déterminer les ensembles d'occurrences de tous les modèles de longueur 1 (distance basée sur les symboles) ou de longueur w (distance basée sur des mots) qui vérifient la contrainte de quorum q;

2. Appliquer des opérations de partionnement ou de raffinement successives afin de déterminer les ensembles d'occurrences des modèles de longueurs croissantes qui vérifient la contrainte de quorum q jusqu'à ce que cette longueur soit égale à k (Problème 1) ou soit la plus longue possible (Problème 2).

En termes de parcours d'un arbre, cela peut être reformulé de la façon suivante:

Algorithmes pour résoudre le problème général :

1. Déterminer les ensembles d'occurrences de tous les modèles de longueur 1 (distance basée sur les symboles) ou de longueur w (distance basée sur des mots) qui vérifient la contrainte de quorum q;

2. Appliquer des opérations de partionnement ou de raffinement successives afin de traverser l'arbre des modèles et énumérer le contenu des récipients au niveau k (Problème 1) ou ceux au niveau le plus élevé (Problème 2).

Figure 5.10: Algorithme prototype pour résoudre le problème général.

ensemble en deux sous-ensembles peut en fait être réalisé en un temps qui est proportionnel au nombre d'éléments du plus petit des deux sous-ensembles uniquement. Crochemore et Cardon [Cardon and Crochemore, 1982] [Crochemore, 1981] ont prouvé alors que cette idée demeure valable même lorsqu'un ensemble est partitionné en $c > 2$ sous-ensembles : dans ce cas, le partitionnement demande un temps proportionnel au nombre d'éléments des $c - 1$ plus petits sous-ensembles.

Nous disposons maintenant des idées de base pour pouvoir indiquer un prototype de tous les algorithmes que nous présenterons dans la suite de ce chapitre.

5.3.2 Un premier prototype d'algorithme

L'algorithme prototype pour résoudre le problème général (voir section 5.1) et qui forme le squelette de tous les algorithmes que nous avons établis et introduisons dans ce chapitre est donné dans la figure 5.10.

Un premier aperçu intuitif de sa complexité est présenté dans la section suivante.

5.3.3 Complexité : un premier aperçu

Nous allons d'abord considérer la complexité en temps de la méthode que nous proposons. La forme récursive de l'algorithme prototype, qui sera retrouvée dans tous les algorithmes particuliers montrés plus loin, conduit à une formulation de sa complexité en temps qui est également récursive.

Ainsi la question principale que nous devons nous poser afin d'établir cette complexité est : supposons que nous ayons construit tous les modèles d'une certaine longueur $k-1$ pour $k \geq 1$, combien d'opérations devons-nous alors réaliser afin d'obtenir les modèles de longueur k à partir de ceux que nous connaissons?

Avec tout ce que nous venons de voir, il apparaît assez clairement que trois facteurs vont être déterminants dans la complexité en temps de l'algorithme. Le premier est bien sûr le nombre total de modèles de longueur $k-1$ et de leurs occurrences — notons ce nombre $N_{ModOcc}(k-1)$. Celui-ci est en général lié au produit du nombre total de modèles par le nombre maximal d'occurrences de chaque modèle, ce dernier nombre étant lui-même en rapport avec la longueur totale $n.N$ des chaînes. Le second facteur déterminant de la complexité est le nombre maximal de branchements à suivre à chaque étape dans le parcours de l'arbre — notons-le $N_{MaxBranches}$ — qui correspond au nombre de regards-en-avant qu'il faut lancer à chaque nœud, ou de branches par lesquelles il faut glisser pour passer d'un niveau au suivant. Ce facteur est en théorie indépendant de k. Enfin le troisième facteur est fonction de la longueur des modèles à rechercher — notons-le $f(L_M)$. Si nous appelons $CompTemps$ cette complexité, nous aurons alors dans tous les cas quelque chose de la forme :

$$CompTemps = N_{ModOcc}(k-1) \times N_{MaxBranches} \times f(L_M).$$

Observons que la contrainte de quorum q n'est pas prise en compte dans cette formule, du moins de façon directe. Elle pourrait l'être dans le calcul de $N_{ModOcc}(k-1)$, mais en fait son influence sur le nombre des modèles va dépendre très fortement des données et n'est donc pas considérée dans la valeur théorique (pire des cas) de la complexité.

Toute la difficulté maintenant d'établir une 'bonne' borne supérieure pour $CompTemps$ va dépendre de la façon de calculer ces divers facteurs, en particulier $N_{MaxBranches}$ et $N_{ModOcc}(k-1)$, ainsi que le rapport de ce dernier avec le nombre de modèles possibles et la longueur des chaînes.

Une première manière de procéder afin de calculer une borne supérieure à $N_{ModOcc}(k-1)$ est de calculer tout simplement le nombre total de modèles pouvant être présents dans les chaînes. Cette approche est naïve car la seule chose que nous pouvons dire de cette borne dans l'absolu est qu'elle est égale à $\mid \Sigma \mid^{k-1}$ ou p^{k-1} suivant que les modèles de longueur $k-1$ sont définis comme étant des mots ou des produits d'ensembles d'une couverture comportant p éléments.

Une meilleure façon de faire consiste à prendre le problème à l'envers, c'est-à-dire à considérer les positions dans les chaînes et à calculer le nombre maximal de modèles auxquelles chacune de ces positions (indicatives du début d'une occurrence dans les chaînes) peut appartenir — notons ce nombre $N_{Modi}(k-1)$. Il dépend de la définition de ressemblance entre modèles et mots adoptée, mais est en général bien plus petit que $\mid \Sigma \mid^{k-1}$ ou p^{k-1} (dans le cas de KMR en particulier, $N_{Modi}(k-1)$ est égal à 1). Comme il y a au plus $n.N$ positions dans les $N \geq 1$ chaînes de l'ensemble à comparer, nous devrions alors obtenir :

$$N_{ModOcc}(k-1) \leq n.N \times N_{Modi}(k-1).$$

Cela n'est cependant pas encore tout à fait exact. Souvenons-nous (voir la section 5.3.1.2 ainsi que les figures 5.8 et 5.9) qu'une position peut se trouver présente plusieurs fois dans un même récipient de l'arbre des modèles (pour des occurrences/images différentes). La vraie borne supérieure pour $N_{ModOcc}(k-1)$ est donc :

$$N_{ModOcc}(k-1) \leq n.N \times F_{MultiPos} \times N_{Modi}(k-1)$$

où $F_{MultiPos}$ est un facteur multiplicatif qui indique le nombre de fois où une position peut appartenir à un même modèle (pour des images différentes) et dépend de la définition de ressemblance choisie. Nous verrons par la suite que ce facteur est en général petit par rapport à $N_{Modi}(k-1)$. Il n'est en fait différent de 1 que lorsque les trous sont permis dans une définition de ressemblance.

Considérons maintenant le facteur $f(L_M)$. Celui-ci est égal soit à la longueur des modèles recherchés, soit au log de cette longueur suivant que l'arbre est parcouru en longueur ou en largeur (et lorsque de plus seuls les nœuds situés aux niveaux qui sont des puissances de 2 sont visités).

Reste alors à calculer une borne supérieure pour le facteur $N_{MaxBranches}$. Pour cela, imaginons que nous avons atteint un certain nœud x situé au niveau $k-1$ dans l'arbre des modèles. Nous souhaitons voir si le modèle associé au nœud x peut être prolongé en un modèle de longueur k. Si la relation de similarité entre modèles et occurrences (mots dans les chaînes) est l'identité, alors une possibilité serait de descendre le long de $\mid \Sigma \mid$ branches à partir du nœud x pour vérifier si ma pour $a \in \Sigma$ demeure un modèle valide. Comme nous l'avons fait observé, il est possible dans ce cas particulier d'être plus efficace que cela puisqu'en pratique, nous n'avons besoin de développer que les branches étiquetées a issues du nœud x pour lesquelles a apparaît au moins une fois dans les chaînes à la suite d'une des occurrences de m. Ce fait est illustré dans la figure 5.11. Il demeure valable lorsque la ressemblance entre modèles et mots dans les chaînes est une relation d'équivalence, une relation unique non transitive comme dans Soldano [Soldano *et al.*, 1995] (voir sections 3.4.3.2 et 4.2.3.4), ou encore est basée sur une couverture (section 3.5.2.1.2) si les erreurs ne sont pas autorisées, ou sur une couverture combinatoire pondérée (section 3.5.2.1.3). Cela signifie qu'une position i appartenant à $N_{Modi}(k-1)$ modèles de longueur $k-1$ ne peut 'donner naissance' (i.e. ne peut aller appartenir) qu'à 1 ou g modèles de longueur k respectivement (où g mesure le degré de non-transitivité d'une couverture — voir 3.2.15). Ainsi le nombre total de branches liant le niveau $k-1$ au niveau k, et donc le nombre total d'opérations à réaliser pour construire les modèles de longueur k à partir de ceux de longueur $k-1$, est au plus égal à $1 \times N_{Modi}(k-1)$ ou $g \times N_{Modi}(k-1)$.

Lorsque des erreurs sont permises dans la comparaison d'un modèle avec les mots dans les chaînes, que ces erreurs soient comptées (sections 3.5.2.1.1 et 3.5.2.1.2) ou pondérées (section 3.5.2.2), une telle économie n'est plus réalisable. Examinons d'abord pourquoi lorsque les erreurs sont comptées (la relation entre modèles et mots mesure une distance). Il est assez aisé de voir qu'il suffit alors qu'il y ait une seule occurrence d'un modèle qui n'ait pas encore atteint son 'quota' d'erreurs permises (et cela sera le plus souvent le cas) pour que son prolongement d'une unité vers la droite puisse être une occurrence du modèle également rallongé d'une unité, quel que soit le symbole (de l'alphabet ou de la couverture) par lequel ce modèle est étendu. Cette constatation est illustrée dans la figure 5.12. Dans ce cas, toutes les $\mid \Sigma \mid$ ou p branches possibles doivent être développées pour cette occurrence-là. Éventuellement, une (ou plusieurs) de ces branches conduira à un modèle qui n'est plus valide et pourra être alors coupée. Mais dans le pire des cas, $\mid \Sigma \mid \times N_{Modi}(k-1)$ ou $p \times N_{Modi}(k-1)$ branches doivent être examinées entre les niveaux $k-1$ et k.

Lorsque la relation de ressemblance entre modèles et mots est celle de la définition 3.5.15 ayant pour unité de comparaison des mots, cette observation demeure valable dans la mesure où il suffit qu'une occurrence d'un modèle possède une bonne 'réserve' en termes du score de

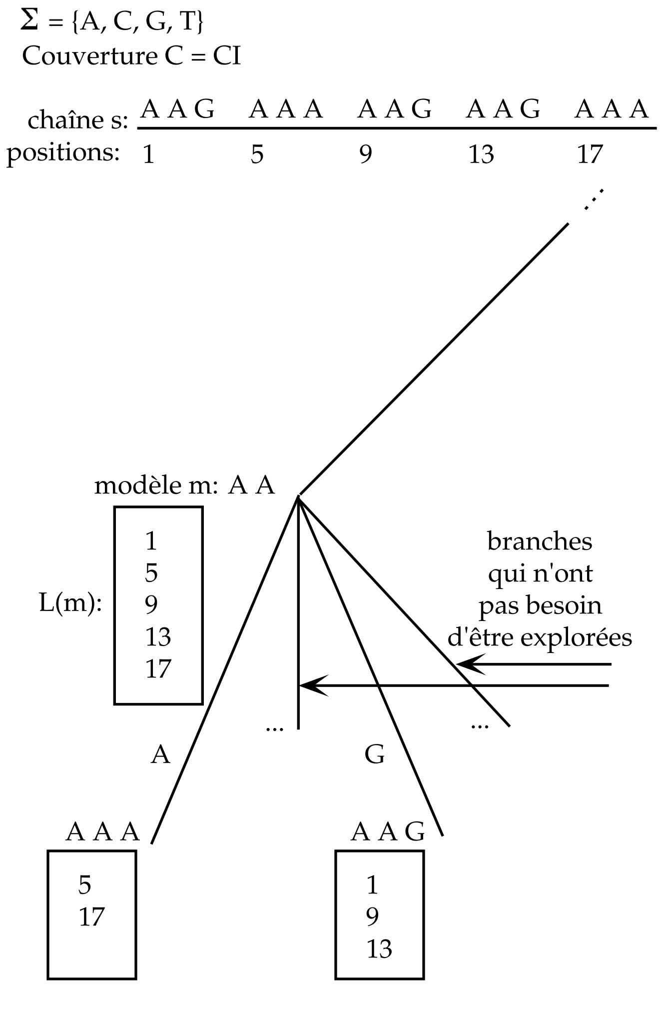

Figure 5.11: Branches à explorer à partir d'un nœud de l'arbre des modèles lorsque la notion de similarité ne permet pas d'erreurs (sauf les substitutions conservatives).

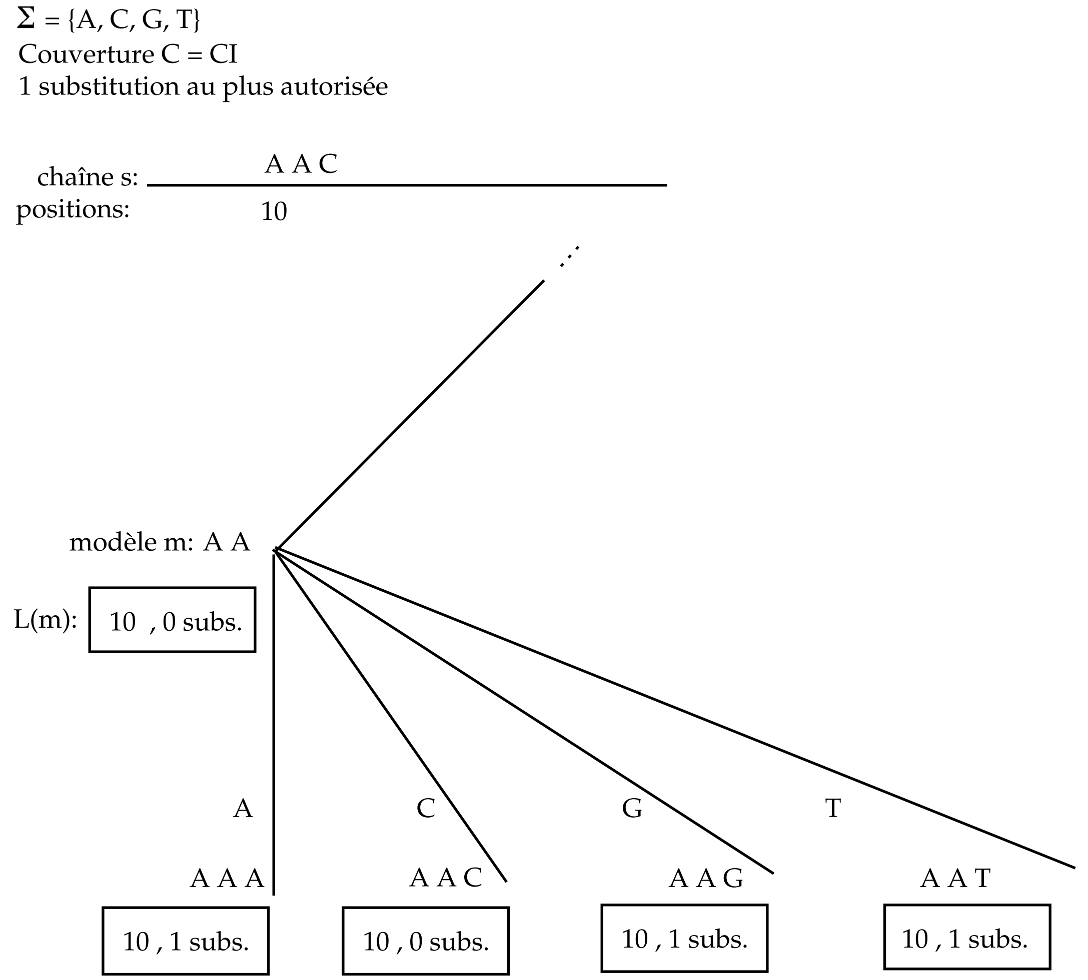

Figure 5.12: Branches à explorer à partir d'un nœud de l'arbre des modèles lorsque la notion de similarité permet des erreurs.

son dernier sous-mot avec le sous-mot correspondant du modèle de longueur $k-1$ pour qu'elle puisse 'donner naissance' à $\mid \Sigma \mid$ autres modèles de longueur k.

Ainsi, nous devrions avoir :

- $N_{MaxBranches} = 1$ ou g si les définitions de ressemblance adoptées sont les définitions 3.5.3 ou 3.5.8 avec $e = 0$, ou la définition 3.5.12;
- $N_{MaxBranches} = \mid \Sigma \mid$ ou p autrement.

À nouveau cette formule n'est pas tout à fait exacte, elle ne compte pas tout. Un facteur multiplicatif va intervenir encore une fois, que nous notons $F_{MultiBranche}$, et dont la raison est toujours liée aux trous. Considérons en effet le cas d'une définition de ressemblance basée sur une distance de Levenshtein. Il est assez facile de voir que, lorsque les trous sont autorisés, chaque occurrence d'un modèle de longueur $k-1$ va pouvoir engendrer potentiellement plusieurs occurrences du modèle de longueur k obtenu à partir du précédent (si le nombre d'erreurs de l'occurrence initiale par rapport au modèle était encore nul). Certaines de ces occurrences sont en fait 'fausses' dans la mesure où le nombre d'erreurs qui va se trouver associé à elles par cette opération ne représente pas leur vrai distance au modèle étendu mais cela ne peut être déterminé qu'a posteriori et l'opération doit donc être comptabilisée malgré tout. Ce phénomène de multiplication d'une occurrence lors de la descente d'un niveau dans l'arbre sera plus précisément décrite dans la section 5.4.2.1. Il a été succintement illustré sur un exemple dans la figure 5.9. Il permet de voir qu'en réalité $F_{MultiBranche} \times \mid \Sigma \mid \times N_{Modi}(k-1)$ ou $F_{MultiBranche} \times p \times N_{Modi}(k-1)$ opérations sont nécessaires pour passer du niveau $k-1$ au niveau k de l'arbre des modèles, où $F_{MultiBranche}$ indique le nombre d'occurrences de longueur k qu'une occurrence de longueur $k-1$ peut engendrer. Comme nous l'avions mentionné (section 5.3.1.2), une autre façon de voir les choses est de considérer ce facteur comme représentant le nombre de fois où une occurrence peut 'glisser' le long d'une même branche de l'arbre des modèles.

Pour les mêmes raisons, un facteur multiplicatif semblable à celui que nous venons de décrire doit être appliqué lorsque la définition 3.5.15 de ressemblance est adoptée si les trous sont permis.

Ainsi, la valeur de $N_{MaxBranches}$ corrigée par ce facteur indique en fait dans tous les cas le produit du nombre maximal de branches par le nombre maximal de 'glissades' qu'il faut effectuer par branche et nous avons :

- $N_{MaxBranches} \leq 1$ ou g si les définitions de ressemblance adoptées sont les définitions 3.5.3 ou 3.5.8 avec $e = 0$, ou la définition 3.5.12;
- $N_{MaxBranches} \leq F_{MultiBranche} \times (\mid \Sigma \mid$ ou $p)$ autrement.

La complexité en temps de l'algorithme prototype (et des algorithmes particuliers) peut donc être écrite sous la forme :

$$CompTemps \leq \underbrace{n.N \times F_{MultiPos} \times N_{Modi}(k-1)}_{\mathrm{N_{ModOcc}(k-1)}} \times \underbrace{\left\{ \begin{array}{l} 1 \; ou \; g \\ F_{MultiBranche} \times (\mid \Sigma \mid ou \; p) \end{array} \right.}_{\mathrm{N_{MaxBranches}}} \times \underbrace{\left\{ \begin{array}{l} k \\ \log k \end{array} \right.}_{\mathrm{f(L_M)}} .$$

Considérons maintenant la complexité en espace de l'algorithme. Celle-ci va également être composée d'un certain nombre de facteurs déterminants. Ceux-ci sont cependant différents suivant que l'arbre est parcouru en largeur ou en profondeur.

Dans le cas d'un parcours en largeur, nous pouvons profiter de tous les calculs que nous venons de faire afin de déterminer la complexité en temps de l'algorithme. En effet, ce qu'il faut compter ici est le nombre total de modèles et de leurs occurrences à un niveau k de l'arbre puisque c'est l'information qu'il nous faut stocker pour construire les modèles de longueur $k+1$ ou $2k$. Ce nombre correspond à $N_{ModOcc}(k)$. Si nous notons $CompEspace_{Largeur}$ la complexité en espace de l'algorithme par rapport à un parcours en largeur de l'arbre des modèles, nous avons alors :

$$CompEspace_{Largeur} \leq n.N \times F_{MultiPos} \times N_{Modi}(k).$$

Lorsque l'arbre est parcouru en profondeur, nous n'avons besoin de connaître que les modèles de longueur 1 plus tous ceux qui sont préfixes du modèle que nous sommes en train de construire. En fait, il est possible de ne conserver que ces derniers, les modèles de longueur 1 pouvant être facilement retrouvés (reconstruits) à chaque fois (en un temps $O(1)$). Si nous notons $CompEspace_{Longueur}$ la complexité en espace de l'algorithme par rapport à un parcours en profondeur, nous avons donc :

$$CompEspace_{Longueur} \leq n.N \times F_{MultiPos} \times (k-1)$$

puisqu'un modèle de longueur k a $k-1$ préfixes.

Dans ce qui suit, nous n'aurons plus qu'à déterminer les bornes supérieures pour les valeurs de $N_{Modi}(k-1)$, $F_{MultiPos}$ et $F_{MultiBranche}$ (qui vont dépendre de chaque l'algorithme).

5.4 Description détaillée

5.4.1 Introduction générale et conventions de notation

Nous allons présenter cette fois une visite détaillée de chacun des algorithmes composant notre approche.

Tous les algorithmes s'appuient sur un ou deux lemmes et sur une propriété permettant de construire les modèles (et donc de parcourir l'arbre) par longueurs croissantes. Ils sont en général donnés pour un parcours quelconque (en largeur ou en profondeur), mais ont été implémentés dans tous les cas pour un parcours en profondeur.

Dans tout ce qui suit, nous considèrerons que l'ensemble de $N \geq 1$ chaînes $s1, s2, ..., sN$ que nous souhaitons comparer est concaténé en une seule chaîne notée $s = s1s2...sN$. En fait, un symbole spécial sépare chacune des chaînes dans s, ceci afin de permettre d'éliminer facilement durant le déroulement de l'algorithme les mots qui se trouvent à cheval sur deux chaînes. De plus, un tableau auxiliaire de taille N conserve les positions de début (ou de fin) de chacune des chaînes $s1, s2, ..., sN$ dans la chaîne s. Comme les positions des mots dans s sont maintenues ordonnées dans les ensembles d'occurrences des modèles (cet ordre découle naturellement de l'implémentation), il est aisé grâce à ce tableau de s'assurer après sa construction qu'un modèle vérifie bien la contrainte de quorum.

Il est important de noter que chaque algorithme, ainsi que chaque version à l'intérieur d'un même ensemble d'algorithmes, est accompagnée d'une analyse de sa performance en pratique (dans l'absolu ou relative à celle des autres versions). En effet, le temps d'exécution observé en pratique va dépendre fortement de la valeur attribuée à la contrainte de quorum et, à travers cette valeur, des données. Lorsqu'il s'agira de résoudre le Problème 2 (trouver les modèles les plus longs), ces données vont également avoir en retour une influence sur la valeur de k_{max} et donc sur la complexité, théorique et pratique. C'est la raison pour laquelle donner seulement la formule pour cette complexité ne suffit pas à rendre compte du comportement effectif de l'algorithme. Diverses courbes sont donc à chaque fois présentées qui permettent de combler cette lacune. Pour les motifs mentionnés, il était important de fournir ces courbes par rapport à de 'vrais exemples' biologiques, mais notre propos n'est pas pour l'instant de discuter la signification des résultats obtenus en tant que tels. Cela ne sera fait en détail que dans le chapitre 6.

5.4.2 Les algorithmes utilisant des symboles comme unités de comparaison

5.4.2.1 Distance de Levenshtein classique ou ensembliste

5.4.2.1.1 Introduction

Le premier algorithme que nous allons présenter utilise des symboles comme unités de comparaison et travaille avec une distance de Levenshtein, classique ou ensembliste, entre modèles et mots dans les chaînes. Il a donc pour base la définition de ressemblance 3.5.3 ou celle 3.5.8 et les symboles en question sont ainsi soit des 'lettres' de l'alphabet Σ des monomères, soit des éléments d'une couverture C définie sur Σ. Nous avons montré en effet à la fin de la section 3.5.2.1.2 qu'il n'existe pas vraiment de différence entre des modèles qui sont des produits d'ensembles d'une couverture et des modèles qui sont des mots. Une 'lettre' a de l'alphabet Σ peut en effet être écrite sous la forme d'un singleton $\{a\}$, et un mot comme un produit cartésien de tels singletons. Nous ne présentons donc ici que la version la plus générale de l'algorithme, sachant qu'il suffit de travailler avec une couverture identité $C = CI = \{\{a\} \mid a \in \Sigma\}$ pour traiter également le cas des modèles définis comme des mots. Les deux approches (modèles comme des mots et modèles comme des produits cartésiens des éléments d'une couverture) ont été introduits dans [Sagot *et al.*, 1995a] et [Sagot *et al.*, 1995d].

L'algorithme est décrit progressivement. Nous donnons d'abord une version où les erreurs ne sont pas permises entre modèles et mots. Dans un deuxième temps seules les substitutions sont autorisées, et nous présentons alors une version de l'algorithme utilisant la distance de Hamming. Enfin, nous introduisons la version utilisant la distance de Levenshtein correspondant aux définitions de ressemblance 3.5.3 et 3.5.8 du chapitre 3.

Une fois cela fait, nous montrons des modifications de l'algorithme permettant de se débarrasser en partie du phénomène de la 'multiplication' d'une position dans les ensembles d'occurrences dont nous avons parlé dans la section 5.3.1.2. Elles impliquent une perte partielle d'information durant le déroulement de l'algorithme, information qui peut cependant être facilement récupérée à la fin.

Nous terminons cette section en introduisant deux variantes de l'algorithme principal où des contraintes supplémentaires sont imposées au niveau de la définition de ressemblance entre modèles et mots ou à celui caractérisant la validité d'un modèle. Ces variantes peuvent être utiles dans certaines applications, et ont pour effet d'accélérer les calculs en pratique.

5.4.2.1.2 Algorithme sans erreurs

5.4.2.1.2.1 Définition de récurrence

Avant de donner le lemme sur lequel l'algorithme s'appuie, nous introduisons une notation simplifiée pour les EL-occurrences d'un modèle lorsque $e = 0$.

Notation 5.4.1 *Soit M un modèle défini comme un produit d'ensembles d'une couverture C. Si (i, d_s, d_d, d_i) est une EL-occurrence de M dans s telle que $d_s = d_d = d_i = 0$, alors nous écrivons simplement : $i \in EL(M)$.*

Soit s une chaîne de longueur n. Soient M, M_1, M_2 trois modèles tels que $M = M_1M_2$ et $i \in \{1,...,n\text{-}|M|\text{+}1\}$ une position dans s. Nous avons le lemme suivant :

Lemme 5.4.1

$$i \in EL(M = M_1M_2)$$

$$\Longleftrightarrow$$

$$i \in EL(M_1), (i+ \mid M_1 \mid) \in EL(M_2).$$

De ce lemme nous pouvons facilement déduire une proposition permettant de construire $EL(M)$ à partir de $EL(M_1)$ et $EL(M_2)$. Avant de la présenter, nous introduisons la notation suivante :

Notation 5.4.2 *Soit $EL(M)$ un ensemble de EL-occurrences, nous notons :*

$$(EL(M))_{-b} = \{i \mid i + b \in EL(M)\}.$$

Il s'agit tout simplement de l'ensemble obtenu de $EL(M)$ en soustrayant b de chaque élément i dans $EL(M)$.

Proposition 5.4.1 $EL(M) = EL(M_1) \cap (EL(M_2))_{-|M_1|}$.

Les preuves du lemme et de la proposition sont immédiates et sont donc omises. Ces résultats nous permettent ainsi de construire les modèles EL-présents au moins q fois dans s par longueurs croissantes.

5.4.2.1.2.2 Algorithme

Une idée de l'algorithme pour résoudre le Problème 1 est donné dans la figure 5.13. Cet algorithme est appelé Poivre (ou Moivre dans sa version pour l'ADN/ARN utilisant une couverture unitaire). Il est implémenté de manière récursive. Comme l'arbre des modèles est parcouru en profondeur, la valeur de b dans la notation 5.4.2 est égale à 1 et les modèles M_2 de la proposition 5.4.1 sont les ensembles de la couverture C de Σ. Ce sont les modèles de longueur 1.

La résolution du Problème 2 est effectuée exactement de la même façon excepté que la condition d'arrêt de la descente le long d'une branche dépend maintenant exclusivement de la contrainte de quorum q. Dans l'implémentation, la seule chose qui change est que nous devons réaliser au tout début de la fonction Explore le test supplémentaire suivant :

```
si (l > k) {
        /* un plus long modèle vient d'être rencontré */
        rejeter les modèles stockés au préalable;
        k = l;
}.
```

Observons que cet algorithme produit des résultats très similaires à ceux obtenus par KMRC (section 4.2.3.4), ce qui n'est pas étonnant dans la mesure où les définitions de ressemblance utilisées par ces algorithmes (définitions 3.4.10 et 3.5.8) sont très proches (voir section 3.5.4). Les principales différences entre les deux portent sur le fait que KMRC fait un usage uniquement implicite des modèles et d'une couverture, que cette 'couverture implicite' doit être composée de cliques maximales (aucun sous-ensemble ne peut être contenu dans un autre), et finalement que KMRC doit effectuer un test de plus à la fin de chaque étape sur les cliques d'occurrences obtenues, ceci afin d'éliminer celles qui ne sont pas maximales (section 4.2.3.4). KMRC est par ailleurs implémenté 'à la façon KMR', la longueur des occurrences est donc doublée à chaque fois. L'algorithme doit de toutes façons parcourir l'arbre en largeur s'il veut pouvoir réaliser ces tests d'inclusions sur les cliques.

5.4.2.1.2.3 Complexité

Nous avons dans ce cas $N_{Modi}(k-1) = g^{k-1}$ et $F_{MultiPos} = 1$. Comme les erreurs, et en particulier les trous, ne sont pas permises, chaque occurrence ne peut glisser qu'une seule fois le long d'une branche, elle peut par contre engendrer g nouvelles occurrences, c'est-à-dire, $N_{MaxBranches} = g$. La complexité en temps de l'algorithme est donc $O(n.N.k.g^k)$.

La complexité en espace est $O(n.N.k)$ pour un parcours en profondeur (elle serait en $O(n.N.k.g^k)$ pour un parcours en largeur).

Les valeurs pour ces complexités correspondent bien sûr à un pire cas qui ne serait observé que s'il existait un symbole de l'alphabet appartenant à tous les sous-ensembles de la couverture et si, de plus, la chaîne s était composée uniquement de ce symbole. Une telle situation est bien sûr totalement irréaliste. D'une façon plus générale, nous pouvons en fait considérer que la

```
/* Entrée */
        s = s1s2...sN : une chaîne de longueur n = concaténation des chaînes s1, s2, ..., sN
        S_1, S_2, ..., S_p : une couverture C de Σ
        q : contrainte de quorum
        k : une longueur fixe
/* Sortie */
        Tous les modèles M de longueur k qui possèdent des instances dans au moins q
        des chaînes s1, s2, ..., sN
/* Principales structures de données */
        M = produit d'ensembles de C : implémenté comme un tableau d'entier entre 1
        et p
        EL(M) = ensemble d'occurrences d'un modèle M : implémenté comme une pile
        Modèles[i] = ensembles de C auxquels S_i appartient : implémenté comme une pile
        (cette structure n'est pas vraiment nécessaire, elle est donnée afin de faciliter la
        compréhension de l'algorithme)
        ExtensionPossible = ensemble des modèles qui peuvent être étendus : implémenté
        comme une pile
Main {
        pour (i ∈ [1..n])
                pour (j ∈ [1..p]) {
                        si (i ∈ S_j)
                                EL(S_j) = EL(S_j) ∪ {i};
                        Modèles[i] = Modèles[i] ∪ S_j;
                }
        pour (j ∈ [1..p])
                si (EL(S_j) vérifie quorum de q)
                        Explore(S_j, 1, EL(S_j));
}
Explore(M, l, EL(M)) {
        si (l = k)                                      /* fin de la récursion */
                stocke M et EL(M);
        sinon {                                         /* pas de la récursion */
                pour (i ∈ EL(M))
                        pour (M_1 ∈ Modèles[i + l]) {
                                EL(MM_1) = EL(MM_1) ∪ {i};
                                ExtensionPossible = ExtensionPossible ∪ {M_1};
                        }
                pour (M_1 ∈ ExtensionPossible)
                        si (EL(MM_1) vérifie quorum de q)
                                Explore(MM_1, l + 1, EL(MM_1));
        }
}
```

Figure 5.13: Algorithme Poivre/Moivre (couverture (éventuellement unitaire) et pas d'erreurs) pour résoudre le Problème 1.

valeur de g qui est réellement importante dans le calcul de la complexité est plutôt sa valeur moyenne $\overline{g}$ telle que celle-ci est donnée dans la définition 3.2.15, à savoir :

$$\overline{g} = \frac{\sum_a g_a}{\mid \Sigma \mid}$$

où g_a est le nombre d'ensembles de C auxquels le symbole a de Σ appartient, ou bien celle donnée par la formule :

$$\overline{g} = \frac{\sum_a p_a g_a}{\mid \Sigma \mid}$$

où p_a est la probabilité d'apparition du symbole a dans la chaîne s.

5.4.2.1.3 Distance de Hamming

5.4.2.1.3.1 Définition de récurrence

Avant de donner le lemme sur lequel repose l'algorithme permettant de traiter les substitutions entre modèles et mots dans les chaînes, nous introduisons une autre notation simplifiée pour les EL-occurrences d'un modèle lorsque $e > 0$ mais que les trous ne sont pas permis.

Notation 5.4.3 *Soit M un modèle défini comme un produit d'ensembles d'une couverture. Si (i, d_s, d_d, d_i) est une EL-occurrence de M dans s telle que $d_d = d_i = 0$, alors nous écrivons $(i, d_s) \in EL(M)$.*

Soit s une chaîne de longueur n. Soient M, M_1, M_2 trois modèles tels que $M = M_1M_2$ et $i \in \{1,...,n\text{-}|M|+1\}$ une position dans s. Alors le lemme suivant est vérifié :

Lemme 5.4.2

$$(i, d_s) \in EL(M = M_1M_2)$$
$$\Longleftrightarrow$$
$$(i, d_{s_1}) \in EL(M_1), (i+ \mid M_1 \mid, d_{s_2}) \in EL(M_2)$$
$$\textit{et } d_s = d_{s_1} + d_{s_2} \leq e$$

La proposition permettant de construire $EL(M)$ à partir de $EL(M_1)$ et $EL(M_2)$ est semblable à celle de la version sans erreurs, elle porte simplement sur des ensembles d'occurrences décrits différemment. Notons d'abord :

Notation 5.4.4 *Soit $EL(M)$ un ensemble de EL-occurrences, nous notons :*

$$(EL(M))_{-b} = \{(i, d_s) \mid (i + b, d_s) \in EL(M)\}.$$

Proposition 5.4.2 $EL(M) = EL(M_1) \cap (EL(M_2))_{-|M_1|}$.

5.4.2.1.3.2 Algorithme

Une idée de l'algorithme pour la résolution du Problème 1 est donné dans la figure 5.14. Cette version est appelée HPoivre (ou HMoivre dans sa version pour l'ADN/ARN).

La résolution du Problème 2 est effectuée de la même façon que pour l'algorithme sans erreur.

```
/* Entrée */
        e : le nombre maximal de substitutions autorisées
        Comme pour l'algorithme de la figure 5.13
/* Sortie */
        Comme pour l'algorithme de la figure 5.13
/* Principales structures de données */
        Modèles[i] = ensembles de la couverture C auxquels l'occurrence i appartient, avec
        le nombre de substitutions entre l'occurrence et le modèle : implémenté comme
        une pile (cette structure n'est pas vraiment nécessaire)
        Le reste comme pour l'algorithme de la figure 5.13
Main {
        pour (i ∈ [1..n])
                pour( j ∈ [1..p])
                        si (i ∈ S_j) {
                                EL(S_j) = EL(S_j) ∪ {(i, 0)};
                                Modèles[i] = Modèles[i] ∪ (S_j, 0);
                        }
                        sinon {
                                EL(S_j) = EL(S_j) ∪ {(i, 1)};
                                Modèles[i] = Modèles[i] ∪ (S_j, 1);
                        }
        pour (j ∈ [1..p])
                si (EL(j) vérifie le quorum q)
                        Explore(S_j, 1, EL(S_j));
}
Explore(M, l, EL(M)) {
        si (l = k)                                      /* base de la récursion */
                stocke M et EL(M);
        sinon {                                         /* pas de la récursion */
                pour ((i, d_s) ∈ EL(M))
                        pour ((M_1, d'_s) ∈ Modèles[i])
                                si (d'_s + d_s ≤ e) {
                                        EL(MM_1) = EL(MM_1) ∪ {(i, d_s + d'_s)};
                                        ExtensionPossible = ExtensionPossible ∪ {M_1};
                                }
                pour (M_1 ∈ ExtensionPossible)
                        si (EL(MM_1) vérifie le quorum q)
                                Explore(MM_1, l + | M_1 |, EL(MM_1));
        }
}
```

Figure 5.14: Algorithme HPoivre/HMoivre (couverture et distance de Hamming) pour résoudre le Problème 1.

5.4.2.1.3.3 Complexité

Soit i une position dans les chaînes. Calculons d'abord $N_{Modi}(k-1)$.

Soit u une EL-image correspondant à (i, d_s). En notant p le nombre d'ensembles dans la couverture C de Σ, nous avons $N_{Modi}(k-1) \leq C_1 \times C_2$ où:

- $C_1 = g^{k-1}$ est le nombre de produits d'ensembles de C auxquels un mot dans une chaîne peut appartenir (correspondant au nombre maximal de positions sans erreur de l'occurrence);
- $C_2 = \begin{pmatrix} k-1 \\ d_s \end{pmatrix} .(p-1)^{d_s}$ compte le nombre de substitutions.

Observons que le facteur C_1 surévalue le nombre de positions sans erreurs (il correspond en fait à $e = 0$). De même, le facteur C_2 peut être surévalué lorsque $g > 1$ car dans ce cas un symbole de u peut appartenir à plus d'un ensemble de la couverture.

En simplifiant et en majorant $N_{Modi}(k-1)$, nous obtenons :

$$N_{Modi}(k-1) \leq \frac{(k-1)!}{(k-1-d_s)!}.p^e.g^{k-1} \leq k^{d_s}.p^e.g^{k-1}$$

et ainsi :

$$N_{Modi}(k-1) \leq k^e.p^e.g^{k-1}.$$

$F_{MultiPos}$ est ici égal à 1 puisque les erreurs sont autorisées mais pas les trous.

Le facteur $N_{MaxBranches}$ est quant à lui majoré par $p \times F_{MultiBranche}$. Et ce dernier facteur est à nouveau égal à 1 (trous interdits). Ainsi $N_{MaxBranches} \leq p$.

La complexité en temps de l'algorithme est donc $O(n.N.k^{e+1}.p^{e+1}.g^k)$. Ce nombre correspond également au nombre maximal de nœuds effectivement présents au niveau k de l'arbre des modèles valides.

La complexité en espace est bornée supérieurement par $n.N.k$ comme dans le cas sans erreur et pour un parcours en profondeur. Elle serait cependant cette fois en $O(n.N.k^e.p^e.g^k)$ pour un parcours en largeur. Le gain ici est plus grand.

Une situation de pire cas pour cet algorithme est observée dans les mêmes circonstances que pour la version sans erreurs (chaîne s composée uniquement d'un symbole appartenant à tous les sous-ensembles de C).

5.4.2.1.4 Distance de Levenshtein

5.4.2.1.4.1 Principales différences avec la version sans erreurs ou distance de Hamming

Nous présentons maintenant le cas le plus général où au plus $e > 0$ erreurs (substitutions (non conservatives) et trous) sont autorisées entre un modèle et ses EL-occurrences. Les principales différences avec la version sans erreurs ou la version 'distance de Hamming' sont les suivantes :

1. Étant donné un modèle M, une position i d'une occurrence de M dans s peut être associée à plusieurs triplets (d_s, d_d, d_i). Chacune correspond à une image u (dont certaines peuvent être identiques) et indique une combinaison différente de substitutions, délétions et insertions permettant de passer de u à une instance de M en un nombre minimal

d'opérations (rappelons que, par convention, d_d et d_i sont le nombre de délétions et insertions réalisées sur u — voir la figure 3.25 du chapitre 3);

2. Étant donné un modèle M de longueur k et ses EL-occurrences, la longueur des EL-images correspondantes peut varier entre $\mid M \mid$ - e et $\mid M \mid + e$. Aussi, lorsque nous considérons l'occurrence (i, d_s, d_d, d_i) d'un modèle M de longueur k et que nous essayons d'étendre le modèle auquel il appartient, nous ne pouvons nous contenter de regarder la position $i + k$ pour voir quel est le prochain symbole dans s comme nous l'avions fait dans le cas sans erreurs ou avec seulement des substitutions. Nous devons maintenant considérer le fait que l'image u correspondant à l'occurrence (i, d_s, d_d, d_i) vérifie : $\mid u \mid = k$ - $déc$, où $déc = d_d - d_i$;

3. Lorsque nous juxtaposons deux EL-images u et u' des modèles M and M' respectivement, nous savons que $distance_{EL}(uu', MM') \leq distance_{EL}(u, M) + distance_{EL}(u', M')$ $\leq e$. Nous n'avons donc pas toujours une stricte égalité. En termes de programmation dynamique, cela signifie que la concaténation de deux chemins optimaux ne produit pas toujours un chemin optimal. Ce que nous pouvons garantir, c'est qu'au moins une des concaténations correspond à un chemin optimal (le lemme 5.4.2.1.4.2 présenté un peu plus loin nous garantit cela). Ainsi uu' est une EL-image de MM'. La $distance_{EL}$ de uu' à une plus proche instance de MM' est la plus petite parmi toutes celles qui sont obtenues et est celle de Levenshtein.

Ces considérations conduisent à un algorithme différent, plus complexe que celui donné précédemment.

5.4.2.1.4.2 Définitions de récurrence

Comme auparavant, soit s une chaîne de longueur n. Soient également M, M_1, M_2 trois modèles tels que $M = M_1M_2$ et $i \in \{1,...,n\text{-}|M|+1\}$ une position dans s. Le lemme 5.4.2 doit maintenant être écrit en deux parties qui ne sont plus symétriques :

Lemme 5.4.3

$$(i, d_s, d_d, d_i) \in EL(M)$$
$$\Longrightarrow$$
$$\exists\, d_{s_1}, d_{d_1}, d_{i_1}, d_{s_2}, d_{d_2}, d_{i_2}\ tel\ que$$
$$(i, d_{s_1}, d_{d_1}, d_{i_1}) \in EL(M_1), (i + \mid M_1 \mid - d_{d_1} + d_{i_1}, d_{s_2}, d_{d_2}, d_{i_2}) \in EL(M_2)$$
$$et\ d_s = d_{s_1} + d_{s_2},\ d_d = d_{d_1} + d_{d_2},\ d_i = d_{i_1} + d_{i_2}.$$

Lemme 5.4.4

$$(i, d_{s_1}, d_{d_1}, d_{i_1}) \in EL(M_1),\ (i + \mid M_1 \mid - d_{d_1} + d_{i_1}, d_{s_2}, d_{d_2}, d_{i_2}) \in EL(M_2)$$
$$et\ d_{s_1} + d_{d_1} + d_{i_1} + d_{s_2} + d_{d_2} + d_{i_2} \leq e$$
$$\Longrightarrow$$
$$(i, d_{s_1} + d_{s_2}, d_{d_1} + d_{d_2}, d_{i_1} + d_{i_2}) \in ES(M_1M_2 = M) \supseteq EL(M)$$
$$où\ ES(M) = \{(i, d_s, d_d, d_i) \mid d_s + d_d + d_i \leq e\}.$$

Avant de présenter la propriété permettant dans ce cas de construire $EL(M)$ à partir de $EL(M_1)$ et $EL(M_2)$, nous introduisons une notation, une opération de filtrage et une opération spéciale d'intersection sur les ensembles de EL-occurrences des deux modèles M_1 et M_2.

Notation 5.4.5 *Soit $EL(M)$ un ensemble de EL-occurrences. Nous notons :*

$$(EL(M))_{-b} = \{(i,d_s,d_d,d_i) \mid (i+b,d_s,d_d,d_i) \in EL(M)\}.$$

Définition 5.4.1 *Soit un ensemble $ES(M) = \{(i, d_s, d_d, d_i) \mid d_s + d_d + d_i \leq e\}$. Nous définissons l'opération $Filtre(ES(M))$ par :*

$$\begin{gathered} Filtre(ES(M)) \\ = \\ \{(i, d_s, d_d, d_i) \in ES(M) \mid d = d_s + d_d + d_i \textit{ est une distance}_{EL}\}. \end{gathered}$$

En d'autres termes :

$$\begin{gathered} Filtre(ES(M)) \\ = \\ \{(i, d_s, d_d, d_i) \in ES(M) \mid d_s + d_d + d_i \ \leq \ d'_s + d'_d + d'_i,\ \forall (i, d'_s, d'_d, d'_i) \ \in \ ES(M) \\ \textit{avec } d'_d - d'_i = d_d - d_i\}. \end{gathered}$$

La dernière condition exprime simplement le fait que deux occurrences doivent avoir même longueur, c'est-à-dire, $\mid M \mid$ - d_d + d_i = $\mid M \mid$ - d'_d + d'_i.

Définition 5.4.2 *Étant donné deux ensembles de EL-occurrences $EL(M_1)$ et $EL(M_2)$, nous définissons l'opération $\sqcap_{EL}$ par :*

$$\begin{aligned} EL(M_1) \sqcap_{EL} EL(M_2) = \{ & (i,d_s,d_d,d_i) \mid (i,d_{s_1},d_{d_1},d_{i_1}) \in EL(M_1), \\ & (i + d_{d_1} - d_{i_1},d_{s_2},d_{d_2},d_{i_2}) \in EL(M_2), \\ & d_s = d_{s_1} + d_{s_2},\ d_d = d_{d_1} + d_{d_2},\ d_i = d_{i_1} + d_{i_2} \\ & \textit{et } d_s + d_d + d_i \leq e\}. \end{aligned}$$

Des lemmes il vient alors :

Proposition 5.4.3 *$EL(M) = Filtre(EL(M_1) \sqcap_{EL} (EL(M_2))_{-|M_1|})$.*

5.4.2.1.4.3 Algorithme

Une idée générale de l'algorithme pour résoudre le Problème 1 est donnée dans la figure 5.15. Cet algorithme est appelé LePoivre (LeMoivre si $C = CI$). La modification qui doit être réalisée afin de traiter le Problème 2 est semblable à celle indiquée pour l'algorithme Poivre/Moivre.

5.4.2.1.4.4 Complexité

Soit i une position dans les chaînes. Étant donné trois valeurs d_s, d_d et d_i telles que $d_s + d_d + d_i \leq e$, nous calculons d'abord $N_{Modi}(k-1)$.

Soit u une EL-image correspondant à (i, d_s, d_d, d_i). En notant $déc = d_d - d_i$ et p le nombre d'ensembles dans la couverture C de Σ, nous avons $N_{Modi}(k-1) \leq C_1 \times C_2 \times C_3 \times C_4$ où:

- $C_1 = g^{k-1}$ est le nombre de produits d'ensembles de C auxquels un mot dans une chaîne peut appartenir (correspondant au nombre maximal de positions sans erreur de l'occurrence);

```
/* Entrée */
        e : le nombre maximal d'erreurs autorisées
        Le reste comme pour l'algorithme de la figure 5.13
/* Sortie */
        Comme pour l'algorithme de la figure 5.13
/* Principales structures de données */
        Modèles[i] = ensembles de la couverture C auxquels l'occurrence i appartient, avec
        le nombre de substitutions, délétions et insertions entre l'occurrence et l'instance
        la plus proche de l'ensemble : implémenté comme une pile
        (à nouveau ici, cette structure n'est pas vraiment nécessaire, les modèles de longueur
        1 pouvant être 'reconstruits' à chaque fois — elle est donnée pour faciliter)
        Le reste comme pour l'algorithme de la figure 5.13
Main {
        pour (i ∈ [1..n])
              pour (j ∈ [1..p])
                    si (i ∈ S_j) {
                          EL(S_j) = EL(S_j) ∪ {(i, 0, 0, 0)};
                          Modèles[i] = Modèles[i] ∪ (S_j, 0, 0, 0);
                    }
                    sinon {
                          EL(S_j) = EL(S_j) ∪ {(i, 1, 0, 0)} ∪ {(i, 0, 1, 0)}};
                          Modèles[i] = Modèles[i] ∪ (S_j, 1, 0, 0) ∪ (S_j, 0, 1, 0) ∪ (∅, 0, 0, 1);
                    }
        pour (j ∈ [1..p])
              si (EL(j) vérifie le quorum q)
                    Explore(S_j, 1, EL(S_j));
}
Explore(M, l, EL(M)) {
        si (l = k)                                       /* base de la récursion */
              stocke M et EL(M);
        sinon {                                          /* pas de la récursion */
              pour ((i, d_s, d_d, d_i) ∈ EL(M))
                    pour ((M_1, d'_s, d'_d, d'_i) ∈ Modèles[i + l − d_d + d_i])
                          si (d'_s + d'_d + d'_i + d_s + d_d + d_i ≤ e) {
                                ES(MM_1) = ES(MM_1) ∪ {(i, d_s + d'_s, d_d + d'_d, d_i + d'_i)};
                                ExtensionPossible = ExtensionPossible ∪ {M_1};
                          }
              pour (M_1 ∈ ExtensionPossible)
                    si (EL(MM_1) = Filtre(ES(MM_1)) vérifie le quorum q)
                          Explore(MM_1, 1 + | M_1 |, EL(MM_1));
        }
}
```

Figure 5.15: Algorithme LePoivre/LeMoivre (couverture et distance de Levenshtein) pour résoudre le Problème 1.

- $C_2 = \begin{pmatrix} k-1-d\acute{e}c \\ d_s \end{pmatrix}.(p-1)^{d_s}$ compte le nombre de substitutions;

- $C_3 = \begin{pmatrix} k-1-d\acute{e}c-d_s \\ d_i \end{pmatrix}$ compte le nombre d'insertions une fois que les substitutions ont été considérées;

- $C_4 = A^{d_d}_{k-1-d\acute{e}c}.p^{d_d}$ compte le nombre de délétions avec $A^{d_d}_{k-1-d\acute{e}c} = \frac{(k-1-d\acute{e}c)!}{(k-1-d\acute{e}c-d_d)!}$ (comme il s'agit juste de choisir des emplacements pour les délétions, ces évènements ne peuvent être distingués).

Comme précédemment, les facteurs C_1 et C_2 peuvent être surévalués.

En simplifiant et en majorant $N_{Modi}(k-1)$, nous obtenons que $N_{Modi}(k-1)$ est au plus égal à :

$$\frac{(k-1-d\acute{e}c)!(k-1-d\acute{e}c-d_s)!(k-1-d\acute{e}c)!}{(k-1-d\acute{e}c-d_s)!(k-1-d\acute{e}c-d_s-d_i)!(k-1-d\acute{e}c-d_d)!}.p^e.g^{k-1}.$$

Nous avons donc que :

$$N_{Modi}(k-1) \leq k^{d_s+d_i+d_d}.p^e.g^{k-1}$$

et ainsi :

$$N_{Modi}(k-1) \leq k^e.p^e.g^{k-1}.$$

Il nous faut maintenant calculer la valeur de $F_{MultiPos}$. Cela revient à calculer le nombre de triplets possibles (d_s, d_d, d_i) pour une position donnée i.

Soit u une EL-image d'un modèle M commençant à la position i. La longueur de u peut varier entre $\mid M \mid$ - e et $\mid M \mid + e$ uniquement, un total de $2e+1$ possibilités. Notons e' la distance Ensembliste de Levenshtein entre l'image et son modèle. Quel que soit le triplet (d_s, d_d, d_i) associé à u, nous devons avoir $d_s + d_d + d_i = e'$ et $d_d - d_i = \mid M \mid$ - $\mid u \mid = c$ où c est une constante. Aussi $d_d = e' + c$ - $\frac{d_s}{2}$ et il ne peut y avoir plus de $\lfloor\frac{e+1}{2}\rfloor$ tels triplets associés à u. Il ne peut y avoir donc plus de $\lfloor\frac{e+1}{2}\rfloor.(2e+1)$ éléments (d_s, d_d, d_i) pour lesquels $d_s + d_d + d_i$ représente une distance Ensembliste de Levenshtein entre u et un modèle M. Si toutefois nous désirons uniquement localiser les modèles et leurs occurrences et non pas aligner ces objets les uns par rapport aux autres exactement (ce qui est notre cas), alors deux occurrences situées à une même position i dans s, ayant même longueur et présentant le même nombre minimal total d'erreurs avec un modèle peuvent être considérées comme étant équivalentes bien qu'elles puissent différer individuellement dans le nombre de substitutions, délétions et insertions qu'elles présentent par rapport au modèle. Nous n'avons donc à considérer, pour chaque position i, que $(2e+1)$ occurrences possibles. Ainsi $F_{MultiPos} \leq 2e+1$.

Comme les trous sont cette fois-ci autorisés, le facteur $N_{MaxBranches}$ est majoré par $p \times F_{MultiBranche}$. Et ce dernier facteur est lui-même borné supérieurement par le nombre maximal d'insertions permises (égal au nombre maximal d'erreurs) plus deux (l'occurrence d'un modèle M de longueur $k-1$ n'est pas étendue (délétion) ou est prolongée sans trous par rapport à un modèle de longueur k ayant M comme préfixe). Ainsi $N_{MaxBranches} \leq (e+2).p$.

La complexité en temps de l'algorithme (opération de filtrage exclue) est donc $O(n.N.(e+2).(2e+1).k^{e+1}.p^{e+1}.g^k)$ (observons que, au contraire de ce qui se passait pour la version sans erreurs ou distance de Hamming, le nombre maximal de nœuds effectivement présents au niveau k de l'arbre des modèles valides peut être inférieur au nombre maximal d'opérations nécessaires pour y parvenir).

Finalement, si les éléments de $EL(M)$ sont maintenus en permanence ordonnés (cet ordre découle naturellement de l'implémentation de l'algorithme), l'opération de filtrage peut être réalisée en utilisant un tableau d'entiers de taille $(2e+1)$ où les meilleurs scores rencontrés pour chaque position i et longueur $\mid u \mid$ sont préservés. L'usage d'une pile additionnelle indiquant quelles cellules de ce tableau ont été modifiées pour chaque position i nous permet de ne l'initialiser qu'une seule fois au début de l'algorithme (avant l'entrée dans la fonction récursive Explore). Le filtrage peut ainsi être effectué en temps constant.

La complexité en temps de l'algorithme demeure donc en $O(n.N.(e+2).(2e+1).k^{e+1}.p^{e+1}.g^k)$.

Sa complexité en espace est bornée supérieurement par $n.N.k.F_{MultiPos}$ et est donc en $O(n.N.k.(2e+1))$ pour un parcours en profondeur (elle serait en $O(n.N.(2e+1).k^e.p^e.g^k)$ pour un parcours en largeur).

5.4.2.1.5 Perte d'information récupérable par la suite

5.4.2.1.5.1 Introduction

L'idée derrière les modifications de l'algorithme précédent qui sont présentées dans cette section consiste à accepter de perdre un peu d'information en cours de route de façon à la fois à essayer d'économiser en espace et à aller plus vite, sachant que cette information perdue n'est pas essentielle pour le déroulement de l'algorithme (<u>tous</u> les modèles valides sont quand même repérés) et qu'elle peut être facilement récupérée ensuite. En fait, la seule information qu'il est absolument nécessaire de préserver est celle concernant les modèles en tant que tels, c'est-à-dire leurs 'noms'. Les diverses versions données ici varient donc suivant la quantité d'information supplémentaire qu'elles maintiennent en mémoire.

5.4.2.1.5.2 Algorithmes

Essentiellement trois modifications sont possibles :

1. la première consiste précisément à ne garder que les 'noms' des modèles valides, aucune information concernant les occurrences de ces modèles n'étant conservée;

2. une seconde version conserve non seulement l'indication des modèles valides mais également, pour chacun d'eux, celle de la position de la première occurrence de ces modèles rencontrée sur chacune des chaînes;

3. la dernière enfin garde tous les noms des modèles qui sont valides, ainsi que toutes les positions où ces modèles sont présents dans les chaînes, mais ne distingue plus les occurrences en termes de quadruplets (i, d_s, d_d, d_i).

Concernant la première modification, il convient d'observer qu'elle n'est pas équivalente à l'approche naïve qui consiste à engendrer tous les modèles d'une certaine longueur car elle tient compte du quorum. Les modèles sont ainsi toujours construits par longueurs croissantes et un modèle M est invalidé, de même que tous les modèles ayant M pour préfixe, si M ne vérifie plus la contrainte de quorum. L'algorithme est alors très simple puisqu'il se réduit à une recherche avec erreur d'un motif dans un texte. Comme la longueur des modèles trouvés sur des exemples biologiques ne dépasse pas en général la taille d'un mot machine (32 bits), nous avons choisi d'utiliser pour cette recherche l'algorithme Agrep de Wu et Manber [Wu and Manber, 1992a] [Wu and Manber, 1992b] qui lui-même est une version étendue de l'algorithme 'Shift-Add' de Baeza-Yates et Gonnet [Baeza-Yates and Gonnet, 1992]. La recherche d'un motif de longueur k dans un texte de longueur n avec au plus e erreurs se fait en $O(n.e)$. Elle est ainsi indépendante de la longueur du motif (si celui-ci est plus petit qu'un mot machine).

La seconde modification utilise le même algorithme que précédemment, simplement cette fois, pour chaque modèle M, sont conservées les positions de début de la première occurrence de M sur chacune des chaînes (au maximum N positions sont ainsi préservées). Lorsqu'un modèle M est allongé en un modèle MS avec $S \in C$, la recherche d'une première occurrence de MS sur chacune des chaînes peut se faire alors en toute sécurité non plus à partir du début des chaînes, mais à partir de la position précédemment obtenue sur chacune d'elles.

La troisième version est plus complexe et ne fait pas appel à des algorithmes de recherche d'un motif dans un texte. Sa validité repose sur l'observation suivante. Supposons qu'un modèle M soit présent de manière multiple à la position i d'une chaîne s, c'est-à-dire, plusieurs occurrences du modèle ayant des longueurs différentes se trouvent à la position i avec un nombre d'erreurs pouvant varier entre 0 et e. Si M est allongé en un modèle MS, il n'est pas possible à ce stade de savoir lesquelles parmi les occurrences de M localisées à la position i vont pouvoir être prolongées. Il faut donc toutes les conserver. Mais considérons maintenant le problème d'une autre façon. Supposons que cette fois ce qui est préservé est la position de fin des occurrences de M et non celle de début. Chaque position de s peut ainsi à nouveau être associée à plusieurs occurrences de M. Pour savoir si la position $i+1$ est la fin d'une occurrence du modèle MS obtenu par allongement de M, il suffit de vérifier que la meilleure occurrence de M (celle présentant le plus petit nombre d'erreurs) dont la fin se situe à la position i peut être prolongée. En effet, si cela n'était pas le cas, aucune des autres occurrences se terminant en i ne pourrait l'être non plus. Pour chaque modèle M, il suffit donc de conserver l'ensemble des positions i indiquant la fin d'au moins une occurrence de M, ainsi que la distance de la meilleure occurrence se terminant en i. Une position i ne peut ainsi appartenir qu'une seule fois à l'ensemble $EL(M)$ des occurrences de M et non plus $2e+1$ fois comme auparavant. Au niveau de l'implémentation, une précaution est toutefois nécessaire dans la mesure où les positions de fin des occurrences d'un modèle allongé MS peuvent ne pas être produites de manière ordonnée (voir la figure 5.16). Afin de maintenir un tel ordre dans les listes d'occurrences des modèles et ne pas y introduire deux fois une même position, il faut donc, à chaque fois que nous voulons ranger une nouvelle position dans ces listes, examiner si elle ne s'y trouve pas déjà. Deux cas de figures sont alors possibles :

- la position — notons-la i — actuellement considérée ne se trouve pas dans $EL(MS)$. Nous sommes sûrs alors que la dernière position de la liste correspondant à $EL(MS)$ est plus petite que i et qu'il suffit donc d'ajouter i à la fin de cette liste (en pratique, en haut de la pile puisque c'est ainsi que ces listes sont implémentées). La preuve fait l'hypothèse que les ensembles d'occurrences sont maintenus en permanence ordonnés (par positions croissantes) et est par induction sur la longueur des modèles. Supposons par l'absurde qu'il existe une position $i+p$ avec $p > 0$ telle que $i+p$ indique la fin d'une occurrence de MS qui aurait été placée dans $EL(MS)$ auparavant. Comme $EL(M)$ est ordonnée (hypothèse d'induction), cela signifie qu'une occurrence de MS se terminant en $i+p$ présentait au moins $p+1 \leq e$ insertions par rapport à MS. Mais alors, l'occurrence se terminant en i présenterait p insertions de moins et aurait également été placée dans $EL(MS)$ au moment où $i+p$ l'a été (par rapport à la même occurrence de M), ce qui contredirait le fait que i ne se trouve pas dans $EL(MS)$;
- la position i se trouve déjà dans $EL(MS)$. Dans ce cas, cette position n'y est pas placée une seconde fois et seul le nombre d'erreurs associé (correspondant à la distance minimale) est éventuellement modifié.

Dans les deux situations, il faudra examiner au plus e des derniers éléments de $EL(MS)$ à chaque fois que l'on voudra y placer une position nouvelle. Dans la meilleure des hypothèses

Σ = {A, C, G, T}
Couverture C = CI
1 erreur au plus autorisée

chaîne s: A A A A
positions: 10

occurrences du modèle A A (par positions de fin):

10	A	1 délétion
11	A	1 délétion
11	A A	0 erreur
11	A A	1 insertion
12	A	1 délétion
12	A A	0 erreur
12	A A A	1 insertion

ordre dans lequel les occurrences du modèle AAA
sont produites (par positions de fin):

11	A A	1 délétion
12	A A	1 délétion
12	A A A	0 erreur
11	A A	1 délétion
13	A A A	1 insertion
12	A A A	1 insertion
13	A A	1 délétion
13	A A A	0 erreur
12	A A	1 délétion
14	A A A A	1 insertion

Figure 5.16: Exemple de l'ordre suivant lequel les occurrences de longueur $k+1$ sont produites à partir de celles de longueur k dans la troisième version modifiée de LePoivre.

(qui sera fréquemment observée pour des modèles plus longs), une seule vérification suffira (la fin de l'occurrence précédemment placées dans la liste se trouvant déjà à une distance supérieure à e de celle que l'on veut y introduire).

5.4.2.1.5.3 Complexités

Pour les deux premières versions de l'algorithme modifié, les facteurs de multiplicité $F_{MultiPos}$ et $F_{MultiBranches}$ disparaissent. Les autres se trouvent multiplés par le nombre d'opérations pour rechercher un motif dans un texte, ici N chaînes de longueur moyenne n.

La première version effectue cette recherche à partir du début des chaînes à chaque fois, la complexité est donc en $O(n.N.e.n.N.k^{e+1}.p^{e+1}.g^k)$ où k est une longueur fixe ou la plus grande possible, auquel s'ajoute le temps nécessaire pour récupérer toutes les occurrences à la fin (rappelons que seul le nom des modèles valides est initialement identifié). Ce second temps dépend bien sûr du nombre de ces modèles. Notons ce nombre $N_{ModèlesValides}$. Dans le pire des cas, il est égal à $n.N.(2e+1).N_{modi}(k)$, mais en pratique il peut être bien plus petit que cela. La complexité en temps totale est donc en $O(n^2.N^2.e.k^{e+1}.p^{e+1}.g^k + k.e.N_{ModèlesValides})$. Celle en espace est en $O(k)$: il suffit de conserver le nom du modèle en construction.

La complexité en temps de la seconde version est similaire à celle de la première excepté que, comme cette fois la position de la première occurrence de chaque modèle sur chacune des chaînes est conservée, celles-ci ne sont parcourues qu'une seule fois par rapport à chaque modèle. Par contre, il faut refaire la recherche à partir du début de l'occurrence du modèle préfixe de celui en train d'être construit, ce qui signifie qu'à chaque allongement d'un modèle, $N.k.e$ comparaisons 'en trop' sont effectuées. La complexité totale en temps est donc en $O(n.N^2.k.e.k^{e+1}.p^{e+1}.g^k + k.e.N_{ModèlesValides})$. Celle en espace est en $O(N.k)$: il faut conserver le nom du modèle en construction et les positions de ses premières occurrences sur chaque chaîne.

Enfin, dans la complexité en temps de la troisième et dernière version, le terme multiplicatif des positions $F_{MultiPos}$ disparaît et est substitué par le nombre maximal de 'regards dans la liste ordonnée' qu'il faut réaliser : $e+1$. À cela s'ajoute le temps pour récupérer toutes les occurrences à la fin. Ici, toutes les positions sont connues, il suffit donc de retrouver les mots correspondants (leurs longueurs). Cela peut se faire avec un simple algorithme de programmation dynamique sur des objets de longueur k (modèle) et $k+e$ (occurrence) respectivement. La complexité en temps finale est donc en $O(n.N.(e+1).(e+2).k^{e+1}.p^{e+1}.g^k) + k.(k+e).N_{ModèlesValides})$. La complexité en espace est en $O(n.N.k)$. Cette dernière version devrait donc être à la fois un peu plus rapide que la version originale (LePoivre), tout en étant souvent bien plus économe en mémoire. Cela sera expérimentalement vérifié dans la section performance.

5.4.2.1.6 Variantes : introduction de contraintes supplémentaires

5.4.2.1.6.1 Distribution uniforme des erreurs

L'algorithme présenté ci-dessus nous permet de résoudre le problème général énoncé dans la section 5.1 par rapport aux définitions de ressemblance 3.5.3 et 3.5.8. D'un point de vue pratique cependant, ces définitions peuvent être trop larges. Ainsi, toutes les erreurs d'une occurrence par rapport à un modèle peuvent apparaître groupées au début ou à la fin des deux objets, or cela n'est pas toujours très satisfaisant du point de vue de la biologie. Nous pouvons donc être amenés à introduire des contraintes sur la façon dont sont distribuées les erreurs entre un modèle et ses occurrences. En particulier, il est aisé de modifier l'algorithme de telle sorte que cette distribution soit uniforme.

Étant donné un modèle M et une quelconque de ses images u, nous demandons dans ce cas que chaque sous-modèle de longueur $l \leq \mid M \mid$ de M possède au plus un nombre $e_l < e$ d'erreurs avec le sous-mot correspondant de u, c'est-à-dire au plus un total de e_l substitutions, délétions ou insertions avec ce sous-mot. Lorsque M est étendu en un modèle $M' = MM_1$ avec $M_1 \in C$, le mot étendu correspondant $u' = ua$ avec $a \in \Sigma$ peut donc ne plus être une image de M' si $a \notin M_1$ et si u présente e_l erreurs avec les dernières $l - 1$ positions de M. En termes d'implémentation, ce qui change par rapport à l'algorithme de la section 5.4.2.1.4 ou 5.4.2.1.5 est que nous devons maintenant conserver, pour chaque image u, les positions (relativement au modèle M) des dernières e_l erreurs dans u.

Par exemple, pour $e_l = 1$, une valeur entière additionnelle doit être stockée avec chaque occurrence. Cette valeur — notons-la p_l — est incrémentée à chaque extension d'une image contre un modèle tant qu'aucune erreur n'est introduite. Lorsque une image u ne peut être étendue 'exactement' par rapport à un modèle $M' = MM_1$, nous devons vérifier que $p_l > e_l$. Si cela est le cas, et si le nombre total d'erreurs de u par rapport à M est moins que e, alors l'extension de u est une occurrence de M' et p_l est remis à 0. Cette manière de procéder restreint, surtout au début de la construction, le nombre de modèles valides.

5.4.2.1.6.2 Présence exacte des modèles au moins une fois

Il est tout à fait possible qu'un modèle soit présent au moins q fois dans les chaînes de l'ensemble sans être jamais instancié dans aucune de celles-ci (c'est-à-dire, sans jamais être présent avec 0 erreur). Dans le cadre spécifique d'une recherche sur banque, il peut être intéressant de demander qu'un modèle soit instancié au moins dans la séquence requête. Cette exigence produit une seconde variante de l'algorithme où nous demandons que chaque modèle M possède au moins une occurrence i dont l'image correspondante u appartienne à M, c'est-à-dire soit une instance de M dans s. Afin de vérifier cela, nous avons besoin uniquement de placer une variable booléenne dans le corps de la récursion de la fonction Explore qui indique si $EL(MM_1)$ contient au moins une occurrence ne présentant aucune erreur avec son modèle. Si cela est le cas, MM_1 est un modèle valide.

Cette contrainte peut considérablement réduire l'espace de recherche des modèles et, en conséquence, accélérer l'algorithme. Cette accélération sera d'autant plus grande que le nombre d'erreurs autorisé entre un modèle et ses occurrences est lui aussi grand.

Bien sûr, les deux variantes présentées ici peuvent être combinées.

5.4.2.1.7 Performance en pratique

Nous présentons ici diverses figures qui indiquent les performances pratiques de H/LeMoivre, H/LePoivre et des modifications des deux algorithmes décrites dans la section 5.4.2.1.5. Toutes ces mesures ont été effectuées sur une Sparc Station 20.

Les trois premières figures, 5.17, 5.18 et 5.19, concernent H/LeMoivre. L'ensemble de chaînes avec lequel nous avons travaillé est constitué de 950 séquences d'ADN de longueur 40 nucléotides, dont chacune comporte deux signaux, l'un très court représentant le codon de début de la traduction (en général ATG) et un autre un peu plus long (constitué de 9 à 10 bases) correspondant au site de fixation d'un complexe macromoléculaire (protéine et ARN ribosomique) appelée ribosome qui participe au processus de la traduction. La couverture utilisée dans ce cas est, bien sûr, la couverture identité CI (voir notation 3.2.2) et les modèles sont donc ici des mots sur le même alphabet que celui des chaînes. Les figures 5.17 et 5.18 donnent les courbes du temps d'exécution (en secondes) de HMoivre et LeMoivre respectivement, par rapport à un

Variation erreurs	HMoivre		LeMoivre	
	Valeur théor.	Valeur moy.	Valeur théor.	Valeur moy.
$0 \to 1$	12	9,2	54	15,6
$1 \to 2$	12	3	26,7	5,6
$2 \to 3$	12	1,8	21	2,4

Tableau 5.1: Rapport (théorique et pratique) des temps d'exécution de H/LeMoivre lorsque le nombre d'erreurs varie de e à $e+1$ pour $q = 80\%$, $e = 0, 1, 2$ et k fixe.

nombre de chaînes variant entre 50 et 950 avec un pas de 50, un quorum fixé à 80%, et pour $k = 3$ (courbes 1) ou $k = k_{max}$ (courbes 2). Ce qu'il est important d'observer dans ces séries de courbes est que :

- elles démontrent bien la linéarité de H/LeMoivre par rapport au nombre N de chaînes, et donc par rapport à la longueur totale $n.N$ de celles-ci (puisque leurs longueurs est toujours la même);
- ces courbes indiquent que le temps d'exécution de ces algorithmes croît moins lentement avec le nombre d'erreurs que la valeur théorique de la complexité ne le laisserait supposer ainsi que l'indique le tableau 5.1.

Notons que le temps d'exécution demeure linéaire avec la longueur totale $n.N$ des chaînes même lorsque des modèles de longueur maximale sont recherchés. Cette dernière varie cependant peu dans cet exemple, pour des N croissants et par rapport au même nombre d'erreurs, sans doute à cause de la présence du site de fixation du ribosome qui est relativement bien conservé partout. Cette longueur maximale est de 3, (6 ou 5), (8 ou 7) et (10 ou 9) pour HMoivre pour e égal à 0 jusqu'à 3 respectivement; elle est de 3, (6 ou 5), 8 et 10 pour LeMoivre.

Les courbes 1 de la figure 5.19 indiquent quant à elles le temps d'exécution (en secondes) de HMoivre (courbe 1a) et de LeMoivre (courbe 1b) pour un quorum variant entre 5% et 100% avec un pas de 5% lorsque les plus longs modèles sont recherchés sur 100 chaînes avec 2 substitutions ou 2 erreurs permises respectivement. Dans ce cas, les valeurs de k_{max} ont varié entre 15 et 6 pour HMoivre et entre 16 et 6 pour LeMoivre. Les courbes 2 de la même figure indiquent enfin le temps d'exécution des deux algorithmes pour k variant de 2 à 11 avec un pas de 1, $N = 10$ chaînes, $q = 50\%$ et $e = 2$.

Les trois figures suivantes, 5.20, 5.21 et 5.22 concernent H/LePoivre cette fois. Dans ce cas, le jeu d'essai a consisté en un ensemble de 38 protéines appartenant à deux sous-familles de nitrogénases, à savoir les chaînes alpha et beta de la composante 1 [Pau, 1989]. Nous travaillerons à nouveau avec ces protéines, qui forment un complexe oligomérique responsable de la fixation de l'azote, dans le chapitre des illustrations. Cette famille contient au moins deux motifs consensus (PROSITE [Bairoch, 1992] numéros PS00699 et PS0090), tous deux correspondant à des sites actifs de la protéine. La longueur de ces chaînes varie entre 441 et 533 acides aminés, avec une moyenne de 500. La couverture C utilisée est :

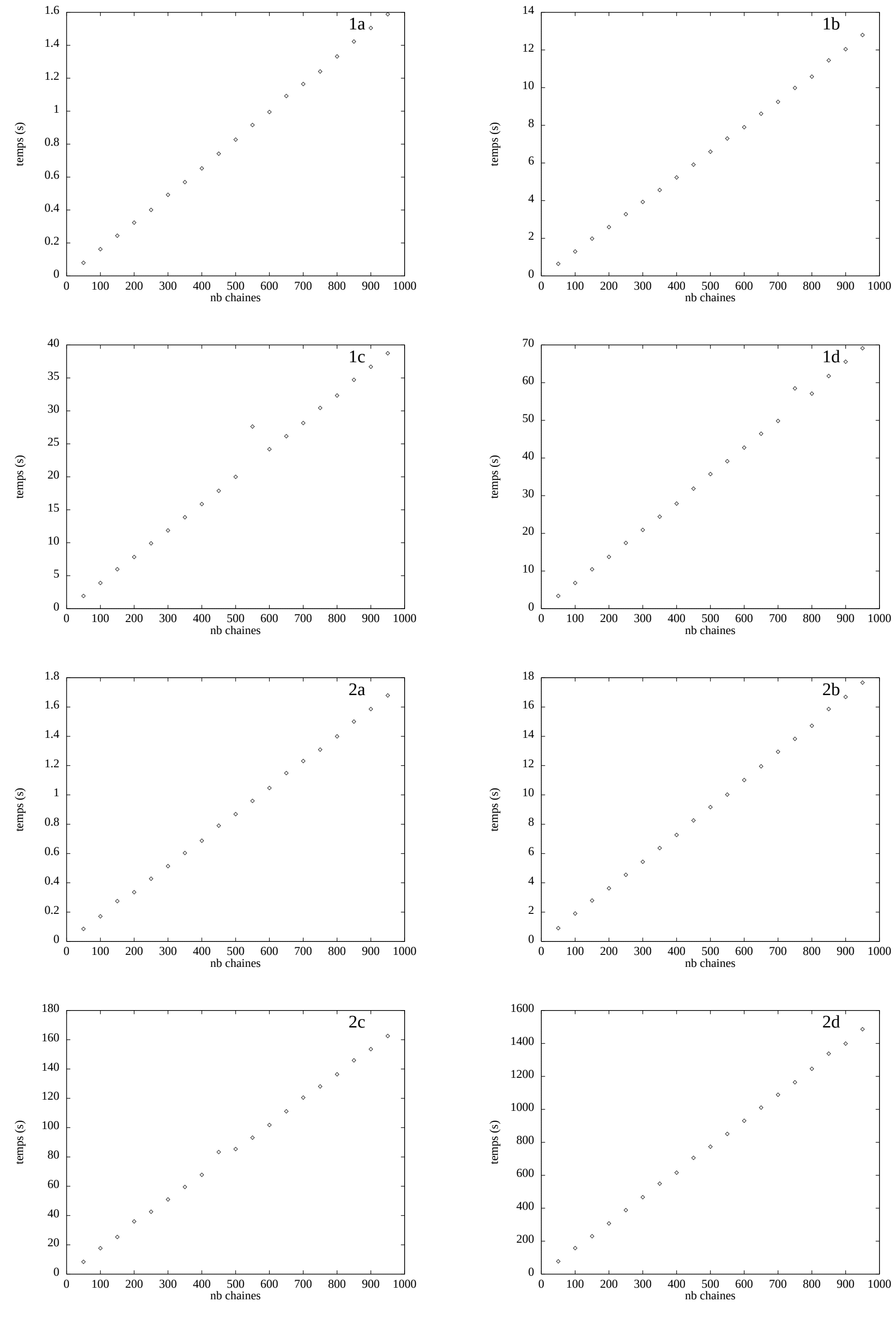

Figure 5.17: Performance de HMoivre : temps (en secondes) $\times$ nombre de chaînes pour $k = 3$ (courbes 1) et $k = k_{max}$ (courbes 2), $q = 80\%$ et pour 0, 1, 2 et 3 substitutions (courbes a, b, c et d respectivement).

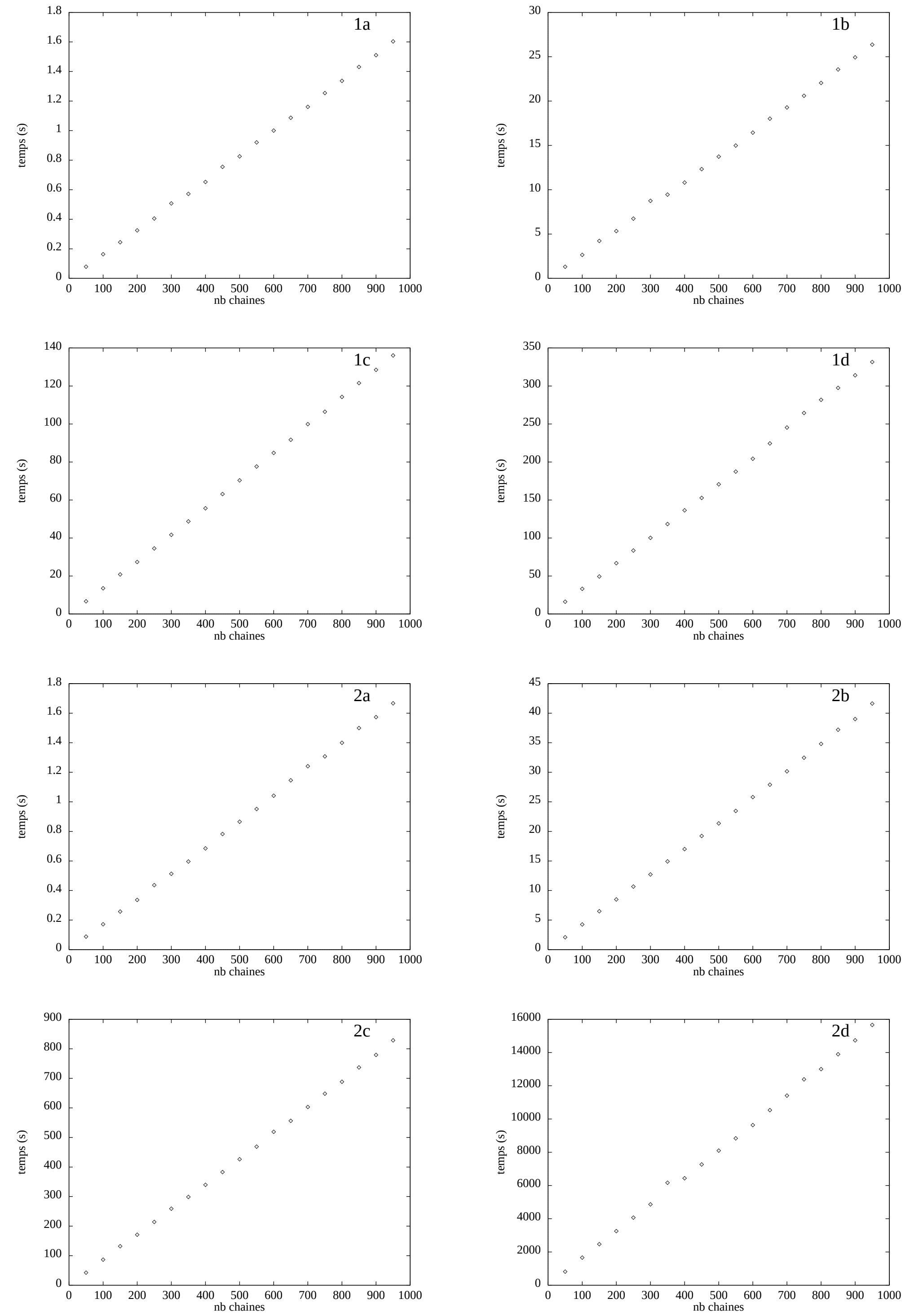

Figure 5.18: Performance de LeMoivre : temps (en secondes) $\times$ nombre de chaînes pour k = 3 (courbes 1) et $k = k_{max}$ (courbes 2), $q = 80\%$ et pour 0, 1, 2 et 3 erreurs (courbes a, b, c et d respectivement).

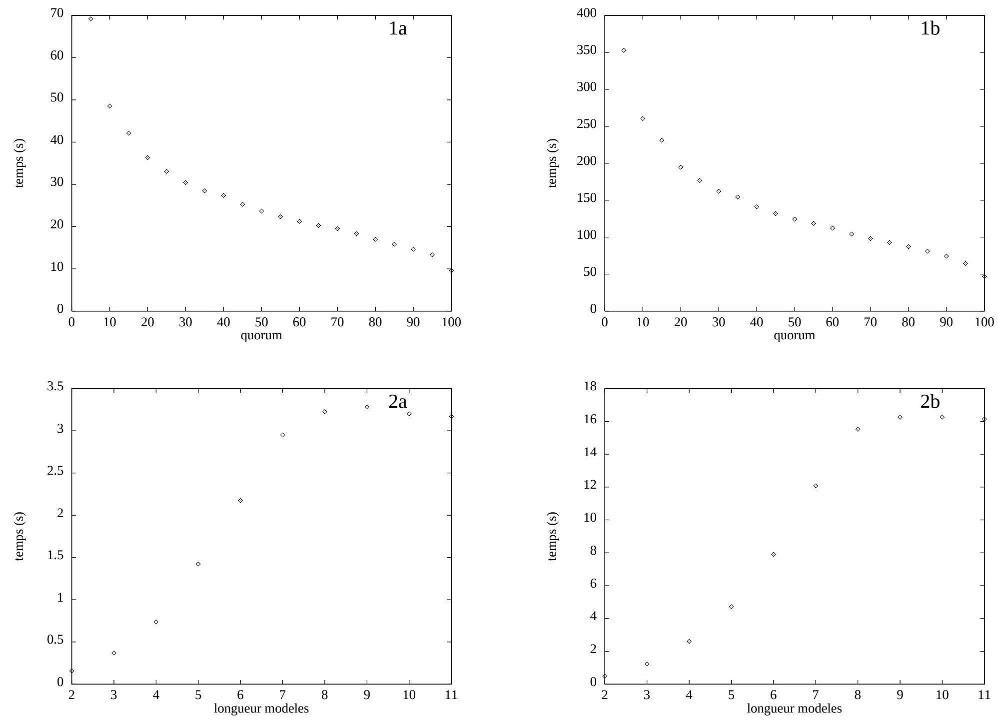

Figure 5.19: Performance de H/LeMoivre (courbes a/b) : temps (en secondes) $\times$ quorum (courbes 1 : N = 100 et $k = k_{max}$) et temps (en secondes) $\times$ k (courbes 2 : N = 10 et q = 100%).

Variation erreurs	HPoivre		LePoivre	
	Valeur théor.	Valeur moy.	Valeur théor.	Valeur moy.
$0 \to 1$	40	36	180	81
$1 \to 2$	40	10	89	17

Tableau 5.2: Rapport (théorique et pratique) des temps d'exécution de H/LePoivre lorsque le nombre d'erreurs varie de e à $e+1$ pour $e = 0$ ou 1, $q = 100\%$ et pour k fixe.

$S_1 = \{A, C, G, S, T\}$ — très petit
$S_2 = \{A, C, D, G, N, S, T, V\}$ — petit
$S_3 = \{F, H, W, Y\}$ — aromatique
$S_4 = \{V, L, I, M\}$ — hydrophobe
$S_5 = \{K, R, H\}$ — basique
$S_6 = \{D, E\}$ — acide
$S_7 = \{Q, N\}$ — amide
$S_8 = \{P\}$ — proline

Elle est inspirée du diagramme de Taylor (section 3.2.3.3.1). Comme auparavant, les figures 5.20 et 5.21 indiquent la variation du temps avec le nombre de chaînes (et donc avec la longueur totale puisque les longueurs des chaînes sont approximativement les mêmes) pour HPoivre et LePoivre. Cette fois cependant, nous ne présentons les courbes que pour e allant de 0 à 2 lorsque k est fixé et pour e égal à 0 ou 1 lorsque $k = k_{max}$. Le quorum dans ce cas est égal à 100% et le nombre de chaînes varie entre 2 et 38 avec un pas de 2.

Les observations faites à propos des courbes pour H/LeMoivre demeurent valables pour H/LePoivre, et en particulier le tableau 5.2 montre que le temps d'exécution croît moins lentement avec le nombre d'erreurs que ne le laisserait supposer la complexité théorique. Les longueurs maximales observées varient cette fois de 16 à 6 et de 30 à 9 pour HPoivre et un nombre de substitutions égal à 0 ou 1 respectivement et de 16 à 6 et de 30 à 10 pour LePoivre pour les mêmes nombres d'erreurs (substitution ou trou). La différence ici est plus importante que celle observée pour H/LeMoivre sur le jeu d'essai précédent. En particulier, k_{max} décroît abruptement au départ, c'est pourquoi nous avons retiré les deux premiers points de la courbe 2b ($N = 2, 4$) pour lesquels k_{max} est égal à 30 et le temps d'exécution approximativement 3000 secondes. À partir de huit chaînes, la valeur de k_{max} demeure pratiquement constante.

Les courbes 1 de la figure 5.22 indiquent le temps d'exécution (en secondes) de HPoivre (courbe 1a) et LePoivre (courbe 1b) lorsque le quorum varie entre 50% et 100% avec un pas de 5% et que les plus longs modèles sont recherchés sur 20 chaînes avec 1 erreur permise (substitution uniquement ou substitution et trou respectivement). Les valeurs obtenues pour k_{max} se sont situées entre 16 ($q = 50\%$) et 10 ($q = 100\%$) pour les deux algorithmes. Les courbes 2 de la même figure indiquent le temps d'exécution lorsque k varie de 3 à 18 avec un pas de 1 pour 6 chaînes, $q = 100\%$ et $e = 1$.

Enfin, les trois dernières figures de cette section permettent de comparer les temps d'exécution des trois versions modifiées de LePoivre présentées dans la section 5.4.2.1.5 avec celui de LePoivre lui-même pour $q = 100\%$ et 0 (figure 5.23), 1 (figure 5.24) et 2 erreurs (figure 5.25). Le

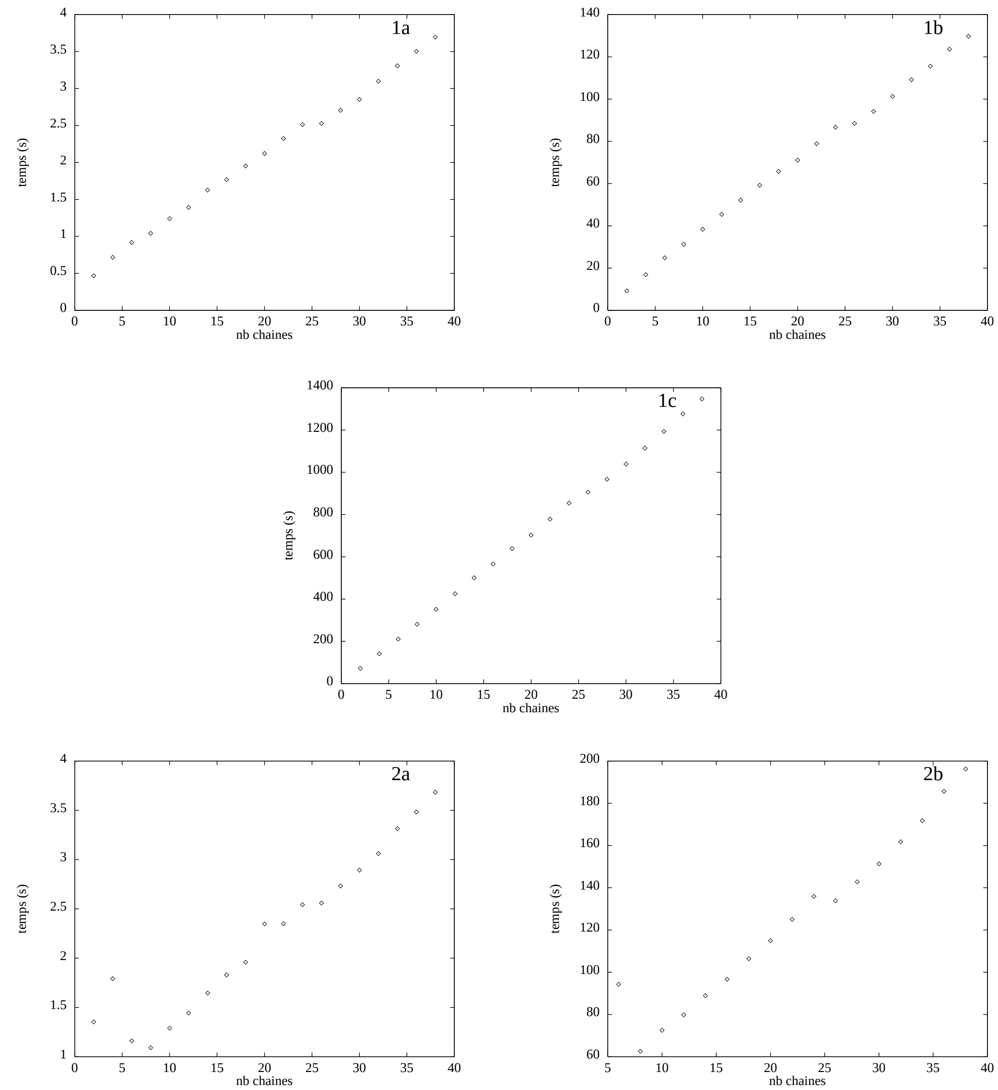

Figure 5.20: Performance de HPoivre : temps (en secondes) $\times$ nombre de chaînes pour $k = 5$ (courbes 1) et $k = k_{max}$ (courbes 2), $q = 100\%$ et pour 0, 1 et 2 substitutions (courbes a, b et c respectivement).

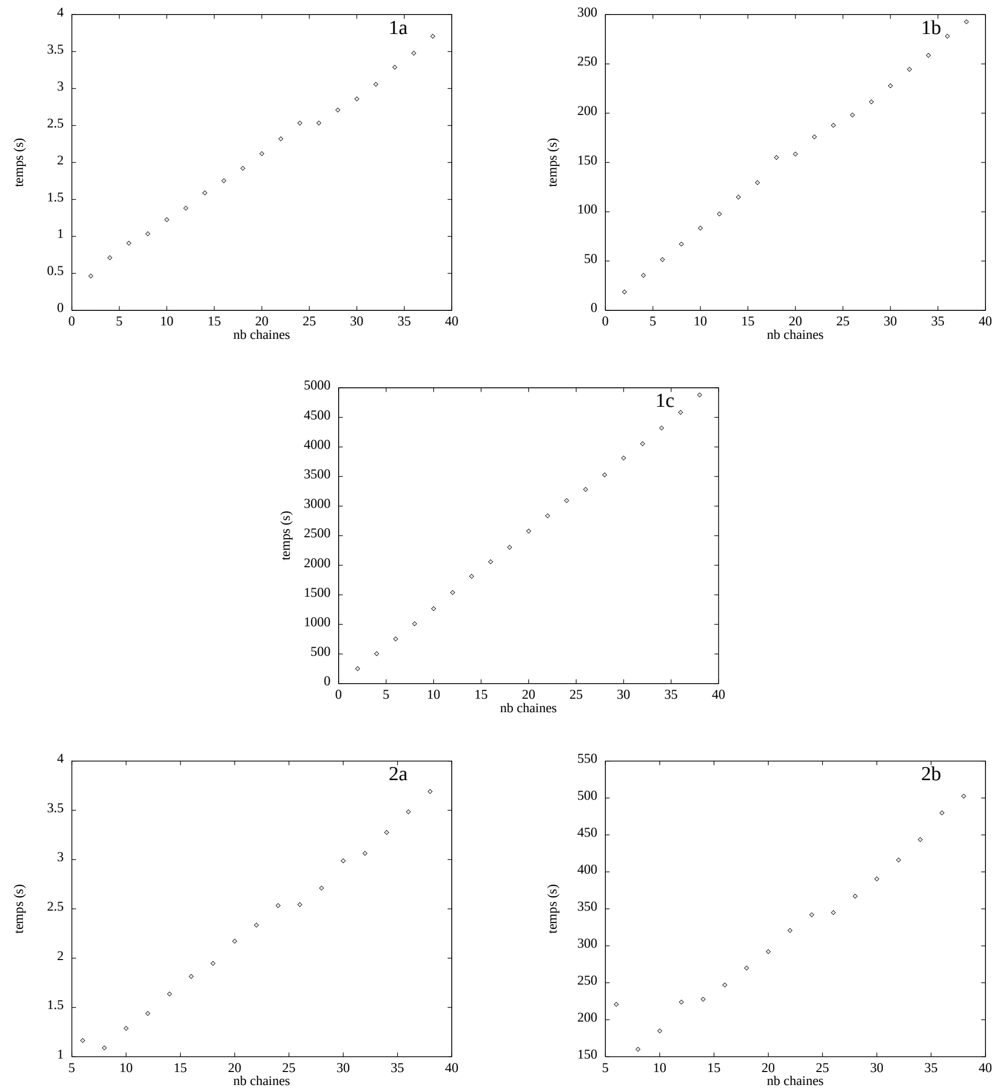

Figure 5.21: Performance de LePoivre : temps (en secondes) $\times$ nombre de chaînes pour $k = 5$ (courbes 1) et $k = k_{max}$ (courbes 2), $q = 100\%$ et pour 0, 1 et 2 erreurs (courbes a, b et c respectivement).

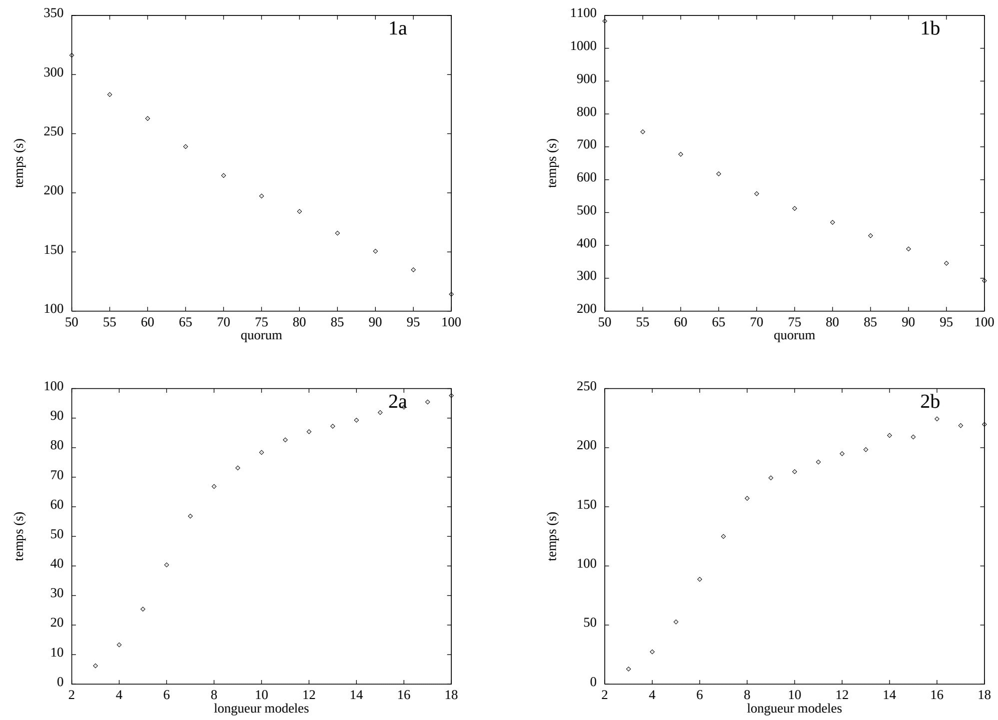

Figure 5.22: Performance de H/LePoivre (courbes a/b) : temps (en secondes) $\times$ quorum (courbes 1 : $N = 20$ et $k = k_{max}$) et temps (en secondes) $\times$ k (courbes 2 : $N = 6$ et $q = 100\%$).

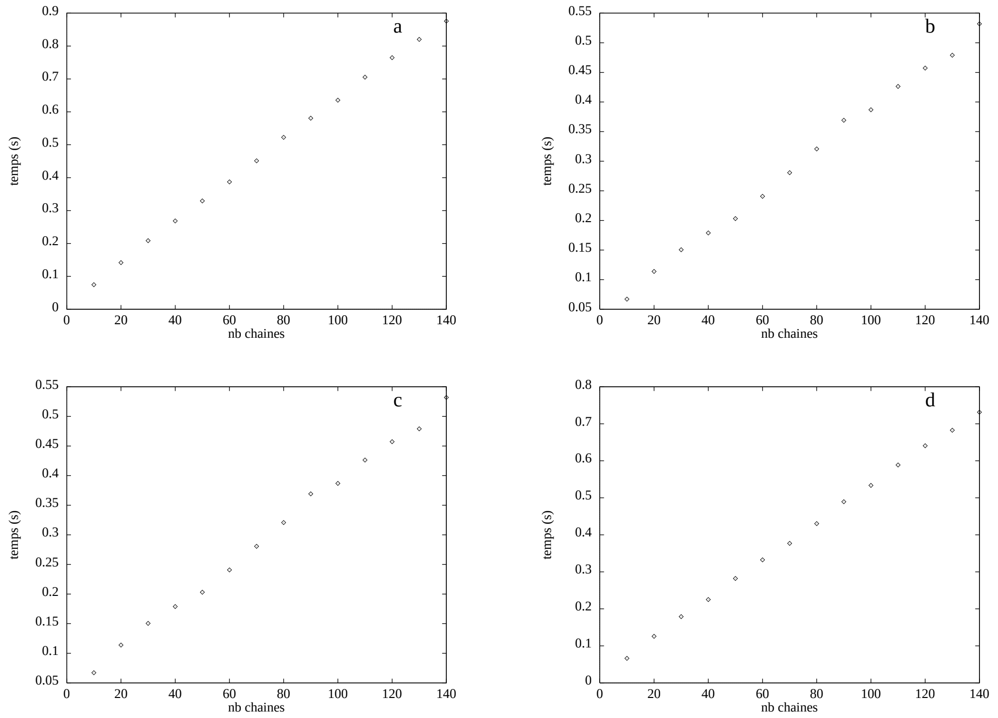

Figure 5.23: Comparaison des temps d'exécution (en secondes) de LePoivre (courbe a) et de ses trois versions plus économiques en mémoire pour $q = 100\%$ et $e = 0$ (courbes b, c et d pour les versions 1, 2 et 3 respectivement).

jeu d'essai a consisté cette fois en 140 chaînes d'ADN de longueur moyenne 100 nucléotides. Ces chaînes correspondent à des promoteurs de *Bacillus subtilis*. Elles sont utilisées à nouveau dans les illustrations et le lecteur est renvoyé au chapitre 6 pour plus de détails les concernant. Il suffit pour l'instant de dire qu'elles présentent toutes 2 à 3 sites impliqués dans le processus de transcription de l'ADN en ARN chez *subtilis*. La couverture utilisée dans le cas de cet exemple a été l'identité. Dans les trois figures, la courbe a fait référence à LePoivre et les courbes b, c et d aux trois versions modifiées de LePoivre.

Ces courbes montrent que le temps d'exécution des versions modifiées 1 et 2 devient d'autant plus grand par rapport à celui de l'algorithme original que le nombre d'erreurs est grand, alors que celui de la version 3 devient d'autant plus petit que ce nombre croît. Le tableau 5.3 donne une indication de cette tendance jusqu'à 7 erreurs pour $q = 100\%$, $N = 10$ et $k = 7$. Observons également qu'à partir de 4 erreurs, le rapport du temps d'exécution de la version modifiée 3 et de LePoivre devient plus grand que la valeur théorique maximale (égale à 2). Une explication à cela est sans doute que les regards qui doivent être en permanence portés sur les ensembles d'occurrences des modèles dans la version 3 sont en pratique moins nombreux que la valeur donnée dans la formule de la complexité (égale à $e + 1$ pour chaque occurrence traitée). En d'autres termes, le calcul de la complexité du pire des cas pour la version modifiée 3 est plus proche de la réalité que celui de l'algorithme LePoivre. Enfin, le tableau 5.4 indique les besoins en mémoire de LePoivre et de cette troisième version modifiée pour $e = 2$, $q = 50\%$, $k = 10$ et

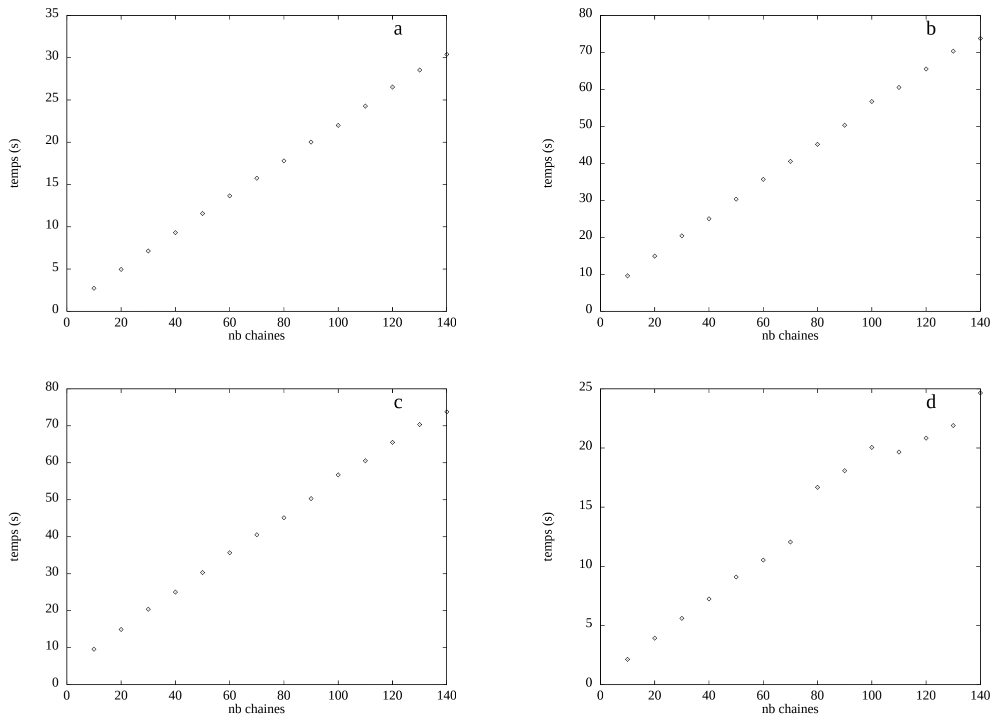

Figure 5.24: Comparaison des temps d'exécution (en secondes) de LePoivre (courbe a) et de ses trois versions plus économiques en mémoire pour $q = 100\%$ et $e = 1$ (courbes b, c et d pour les versions 1, 2 et 3 respectivement).

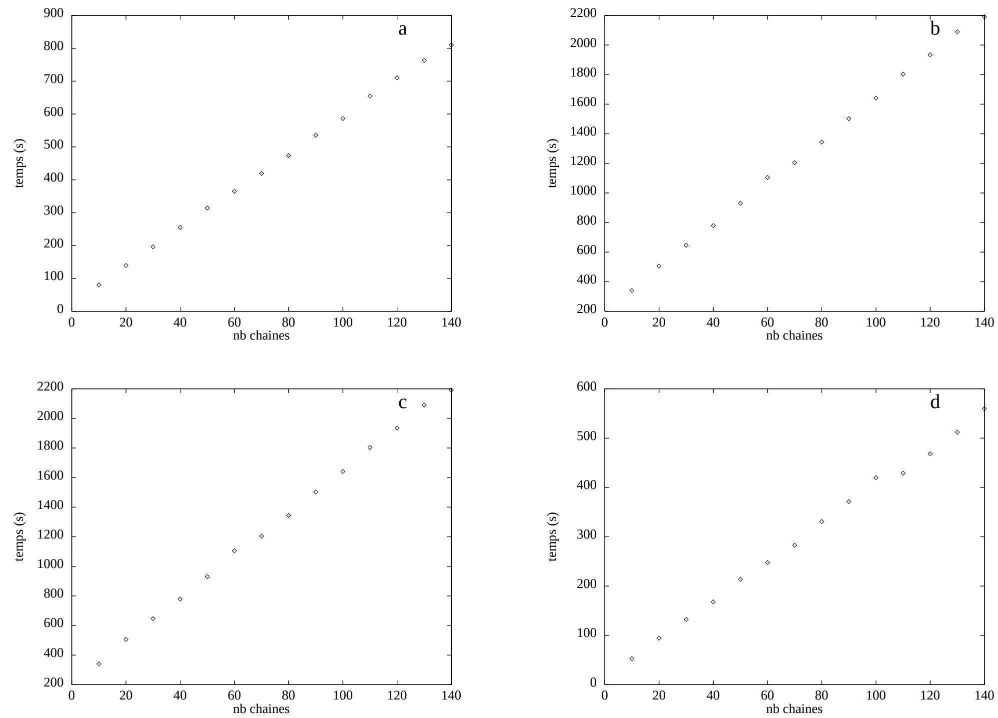

Figure 5.25: Comparaison des temps d'exécution (en secondes) de LePoivre (courbe a) et de ses trois versions plus économiques en mémoire pour $q = 100\%$ et $e = 2$ (courbes b, c et d pour les versions 1, 2 et 3 respectivement).

Nombre d'erreurs	Temps	
	LePoivre	Modification Version 3
1	1.67	1.20
2	33.06	21.38
3	254.54	145.76
4	1035.63	456.84
5	2631.05	847.09
6	4508.46	1237.58
7	6102.20	1502.38

Tableau 5.3: Comparaison du temps d'exécution (en secondes) de LePoivre et de sa version modifiée 3 pour un nombre d'erreurs croissant, $q = 100\%$, $N = 10$, $k = 7$.

un nombre de chaînes variant entre 20 et 140 avec un pas de 20 (la quantité mesurée correspond au nombre d'occurrences à stocker). L'occupation mémoire des versions 1 et 2 est constante lorsque k est fixé et égale à k et $N.k$ respectivement.

5.4.2.2 Couverture combinatoire pondérée

5.4.2.2.1 Introduction

L'algorithme que nous introduisons maintenant et qui a été présenté dans [Sagot and Viari, 1996] travaille avec une relation de ressemblance différente de celles utilisées par les algorithmes de la section précédente. Elle correspond à la définition 3.5.12 où les modèles représentent des produits d'ensembles d'une couverture combinatoire pondérée. Une telle couverture contient souvent plus d'éléments que les couvertures habituellement employées par les algorithmes précédents et, surtout, son degré de non-transitivité est en moyenne bien plus élevé. Ainsi, dans le cas d'une couverture combinatoire CCP définie sur l'alphabet Σ des nucléotides, CCP contient 15 éléments (tous les sous-ensembles de Σ sauf le sous-ensemble vide) et le degré g de non-transitivité de CCP est égal à 8. Dans ce cas, nous avons également $\overline{g} = 8$.

Les difficultés d'une telle approche doublement combinatoire du problème de la recherche de groupes de mots similaires peuvent donc être considérables. Il est clair en particulier que dans le cas des protéines, où $\mid \Sigma \mid = 20$, la couverture utilisée ne pourra être totalement combinatoire. Nous reviendrons là-dessus plus loin. Pour l'instant cependant, nous allons voir comment traiter une couverture combinatoire complète (tous les sous-ensembles de Σ peuvent être autorisés au moins une fois dans les modèles) dans le cas où Σ est l'alphabet des nucléotides.

L'algorithme est encore une fois présenté de manière progressive. Nous commençons par donner la formule de récurrence permettant de construire les modèles (i.e. leurs ensembles d'occurrences) par longueurs croissantes. Puis nous décrivons un premier algorithme direct pour résoudre le problème général par rapport à la définition de ressemblance adoptée ici. Nous présentons ensuite deux améliorations de cette version initiale qui nous permettent de résoudre le même problème de façon plus efficace et rapide en pratique et/ou en théorie. La première introduit une minimalité gauche-droite des ensembles composant un modèle et la seconde réalise

Nombre de chaînes	Nombre d'occurrences	
	LePoivre	Modification Version 3
20	628798	453830
40	609136	429261
60	639261	448940
80	551173	386660
100	625088	438628
120	731256	510860
140	708499	495232

Tableau 5.4: Mémoire occupée (indiquée ici en nombre d'occurrences devant être stockées) par LePoivre et sa version modifiée 3 pour $e = 2$, $q = 50\%$, $k = 10$ et un nombre de chaînes variant entre 20 et 140.

une esquisse de l'espace solution avant de l'explorer en détail. Dans ce second cas, l'amélioration obtenue dans le temps d'exécution peut parfois être importante.

5.4.2.2.2 Définition de récurrence

Soient $s \in \Sigma^*$, $X \in \Sigma$ et i une position dans s. Soient $CCP = \{(S, w_S)\}$ une couverture combinatoire pondérée de Σ et $M = M'S \in CCP^k$. Nous avons alors le lemme suivant :

Lemme 5.4.5

$$(i, u = u'X) \in Occ(M = M'S)$$

$$\Longleftrightarrow$$

$$(i, u') \in Occ(M'),\ X \in S,\ (p(S, M') + 1) \leq w_S$$

où $p(S, M')$ est le nombre de fois où l'ensemble S apparaît dans le modèle M'.

Ce lemme fournit une façon simple de construire l'ensemble $Occ(M)$ à partir de l'ensemble $Occ(M')$ où $\mid M' \mid = \mid M \mid$ - 1. L'ensemble $Occ(M)$ pour $\mid M \mid = 1$ et $w_M \geq 1$ est obtenu en glissant M le long de la chaîne s et en plaçant dans $Occ(M)$ tous les éléments (i, s_i) pour lesquels $s_i \in M$. Bien sûr, si $w_M = 0$, c'est-à-dire, si l'ensemble M n'est pas autorisé dans les modèles, alors $Occ(M) = \emptyset$.

5.4.2.2.3 Algorithme direct

5.4.2.2.3.1 Algorithme

Une idée de l'algorithme direct pour résoudre le Problème 1 est donnée dans la figure 5.26. La résolution du Problème 2 ne nécessite qu'une modification mineure au début de la fonction Explore comme pour les algorithmes précédents.

```
/* Entrée */
        s = s1s2...sN : une chaîne de longueur n = concaténation des chaînes s1, s2, ..., sN
        CCP = {{S}, w_S} : une couverture combinatoire de Σ
        q : contrainte de quorum
        k : une longueur fixe
/* Sortie */
        Tous les modèles M de longueur k ayant des instances dans au moins q des chaînes
/* Principales structures de données */
        M = produit d'ensembles de la couverture CCP
        Occ(M) = ensemble des occurrences d'un modèle M : implémenté comme une pile
        p_M = tableau de taille 2^|Σ| defini par p_M[S] = nombre de fois où l'ensemble
        S apparaît dans l'ensemble M
        Modèles[i] = ensembles de CCP auxquels s_i appartient : implémenté comme une
        pile (cette structure n'est pas vraiment nécessaire)
        ExtensionPossible : comme pour les algorithmes précédents
Main {
        pour (i ∈ [1..n])
                pour (j ∈ [1..2^|Σ|])
                        si ((w_{S_j} ≥ 1) et (i ∈ S_j)) {
                                p_{S_j}[S_j] = 1;
                                Occ(S_j) = Occ(S_j) ∪ {i};
                                Modèles[i] = Modèle[i] ∪ S_j;
                        }
        pour (j ∈ [1..2^|Σ|])
                si (Occ(j) vérifie quorum de q)
                        Explore(S_j, 1, Occ(S_j), p_M);
}
Explore(M, l, Occ(M), p_M) {
        si (l = k)                                      /* fin de la récursion */
                stocke M et Occ(M);
        sinon {                                         /* pas de la récursion */
                pour (i ∈ Occ(M))
                        pour (M_1 ∈ Modèles[i + l])
                                si ((p_M[M_1] + 1) ≤ w_{M_1}) {
                                        p_{MM_1}[M_1] = p_M[M_1] + 1;
                                        Occ(MM_1) = Occ(MM_1) ∪ {i};
                                        ExtensionPossible = ExtensionPossible ∪ {M_1};
                                }
                pour (M_1 ∈ ExtensionPossible)
                        si (Occ(MM_1) vérifie quorum de q)
                                Explore(MM_1, l + 1, Occ(MM_1), p_{MM_1});
        }
}
```

Figure 5.26: Algorithme direct (couverture combinatoire pondérée) pour résoudre le Problème 1.

5.4.2.2.3.2 Complexité

La complexité qui est obtenue ici est la même que celle observée pour l'algorithme Poivre (section 5.4.2.1.2), simplement la couverture est définie différemment. Cette complexité est donc $O(n.k)$ en espace et $O(n.N.k.g^k)$ en temps, avec cependant cette fois, g (ainsi que $\overline{g}$) égal à 8. Rappelons que, dans le cas de LePoivre, nous travaillions avec des valeurs de $\overline{g}$ proches de l'unité ($\overline{g} \sim 1.2$)

Bien sûr, une fois encore le quorum q n'est pas pris en considération (c'est-à-dire, nous faisons comme si $q = 1$). De plus, nous considérons également dans ce calcul que $w_S = \infty$ pour tout $S \in \mathcal{P}^+(\Sigma)$.

Malgré cela, une complexité proportionnelle à g^k lorsque $g = 8$ est excessive, et nous présentons donc dans la prochaine section une première amélioration de cet algorithme direct.

5.4.2.2.4 Minimalité gauche-droite

5.4.2.2.4.1 Idée principale

Considérons l'exemple suivant :

Exemple 5.4.1 *Soient* $q = 3$,

$s1 = ACAA$
$s2 = CAGA$
$s3 = AGAA$
$s4 = CTGA$

et :

$$CCP = \left\{ \begin{array}{l} (\{X\}, \infty)\ \forall X \in \Sigma = \{A, C, G, T\}, \\ (\{X, Y\}, 1)\ \forall X, Y \in \Sigma, \\ (\{X, Y, Z\}, 0)\ \forall X, Y, Z \in \Sigma, \\ (\{A, C, G, T\}, 1) \end{array} \right\}.$$

Alors $M_1 = \{A, C\}\{A, C, G, T\}\{A, G\}\{A, T\}$ *et* $M_2 = \{A, C\}\{A, C, G, T\}\{A, G\}\{A\}$ *sont tous deux des modèles valides de longueur 4. Cependant le modèle* M_1 *n'apporte aucune information supplémentaire par rapport au modèle* M_2 *puisque* $Occ(M_1) = Occ(M_2)$ *et* $M_2 \subseteq M_1$ *(où* $(M_2 = \prod_{i=1}^{k} S_2^i) \subseteq (M_1 = \prod_{i=1}^{k} S_1^i)$ *si* $S_2^i \subseteq S_1^i$ *pour* $1 \leq i \leq k$*).*

Ceci demeure vrai pour tous les modèles M_1' *ayant le modèle* M_1 *comme préfixe propre : l'information qu'ils apportent est superflue par rapport à celle contenue dans les modèles* M_2' *de même longueur ayant* M_2 *comme préfixe propre. Aucune information n'est donc perdue si le sous-arbre de l'arbre des modèles ayant* M_1 *pour racine n'est jamais traversé.*

Observons que l'exemple précédent est très différent de celui indiqué ci-dessous :

Exemple 5.4.2 *Soient* CCP *et* q *comme avant et maintenant :*

$s1 = ACAA$
$s2 = CAGA$
$s3 = AGAA$
$s4 = ATGA.$

Dans ce cas, $M_1 = \{A, C\}\{A, C, G, T\}\{A, G\}\{A\}$, $M_2 = \{A\}\{A, C, G, T\}\{A, G\}\{A\}$ *et* $M_3 = \{A\}\{A, C, G, T\}\{A, G\}\{A, T\}$ *sont tous des modèles valides de longueur* 4 *mais si* M_3 *est superflu par rapport à* M_2 *puisque* $Occ(M_3) = Occ(M_2)$ *et* $M_2 \subseteq M_3$*, cela n'est plus vrai pour le* *modèle M_1 par rapport à M_2. La raison en est que, si comme produit d'ensembles, nous avons bien $M_2 \subseteq M_1$, nous avons aussi par ailleurs que $Occ(M_1) \neq Occ(M_2)$.*

Les modèles M_1 et M_2 peuvent donc conduire à des solutions finales différentes qui sont toutes deux valides et l'arbre des modèles ne peut être élagué au nœud étiqueté par aucun de ces modèles.

Ces exemples montrent que, étant donné deux modèles valides M_1 et M_2, si les deux conditions suivantes sont vérifiées simultanément :

c1 $Occ(M_1) = Occ(M_2)$

c2 $M_2 \subseteq M_1$

alors l'information apportée par le modèle M_1 est superflue en termes aussi bien d'un produit d'ensembles de la couverture que de ses occurrences. Il peut donc être éliminé, et l'arbre des modèles élagué au nœud étiqueté par ce modèle. En d'autres termes, lorsqu'à une étape donnée de l'algorithme, les ensembles d'occurrences de deux modèles sont identiques, nous n'avons besoin de conserver que celui qui correspond à un produit d'ensembles minimaux par rapport à l'autre. Cela peut représenter une économie supplémentaire en temps (et en espace).

En pratique cependant, les deux conditions ci-dessus ne sont pas celles que nous vérifions effectivement. Au lieu de cela, ce qui est vérifié est si, étant donné deux modèles $M_1 = MS_1$ et $M_2 = MS_2$, la condition suivante est vrai :

c3 $Occ(M) \cap Occ(S_1) = Occ(M) \cap Occ(S_2)$ (il s'agit de la même condition que c1 écrite d'une autre façon)

c4 $S_2 \subseteq S_1$

Ceci signifie que la minimalité des ensembles dans les modèles est toujours relative aux derniers ensembles $S \in \mathcal{P}^+(\Sigma)$ qui sont successivement concaténés à un même modèle M, cette minimalité n'est jamais vérifiée par rapport à tous les modèles. Clairement, c4 est une condition plus faible que c2. En voici un exemple :

Exemple 5.4.3 *Soient CCP comme dans l'exemple 5.4.1, $q = 3$ et :*

$s1 = ACAA$
$s2 = CAGT$
$s3 = AGAA$
$s4 = ATGA$

Alors $M_1 = \{A,C\}\{A,C,G,T\}\{A,G\}$ et $M_2 = \{A\}\{A,C,G,T\}\{A,G\}$ sont tous deux des modèles valides et non superflus de longueur 3 et les deux doivent être préservés. Les modèles $M_3 = \{A\}\{A,C,G,T\}\{A,G\}\{A\}$ et $M_4 = \{A,C\}\{A,C,G,T\}\{A,G\}\{A\}$ de longueur 4 sont aussi valides mais M_4 est superflu car $Occ(M_3) = Occ(M_4)$ et $M_3 \subseteq M_4$. Dans notre implémentation cependant, tous deux sont conservés puisque $M_3 = M_2\{A\}$, $M_4 = M_1\{A\}$ et $M_2 \neq M_1$. Aussi la minimalité de M_3 par rapport à M_4 ne peut être vérifiée.

La vérification des conditions c3 et c4 plutôt que celle de c1 et c2 est nécessaire car les modèles sont construits en profondeur. Le désir d'éviter un parcours en largeur de l'arbre n'est cependant pas la seule raison pour laquelle nous ne cherchons pas pour l'instant à vérifier les conditions c1 et c2 totalement. Une seconde motivation est que cette vérification serait impossible à réaliser sans que cela ajoute à la complexité de l'algorithme car elle impliquerait des comparaisons entre tous les modèles d'une certaine longueur. Enfin une troisième et dernière motivation également importante est qu'en fait même un parcours en largeur ne permettrait pas de vérifier c1 et c2. En effet, certains modèles minimaux ne vont jamais être engendrés. Illustrons cela sur un exemple :

Exemple 5.4.4 *Soient CCP comme dans l'exemple 5.4.1, $q = 4$ et :*

$s1 = CAAA$
$s2 = ATAA$
$s3 = AAGA$
$s4 = AAAC$

Alors $M_1 = \{A,C\}\{A,T\}\{A,G\}\{A,C,G,T\}$ et $M_2 = \{A,C,G,T\}\{A,T\}\{A,G\}\{A,C\}$ sont tous deux des modèles valides et non superflus de longueur 4 puisque, si $Occ(M_1) = Occ(M_2)$, aucune des deux inclusions $M_1 \subseteq M_2$ ou $M_2 \subseteq M_1$ n'est vérifiée. Tous deux sont donc, dans ce sens, minimaux. Ces modèles devraient alors être conservés, s'ils étaient engendrés. Mais quel que soit le type de parcours effectué dans l'arbre des modèles, le modèle M_2 ne sera jamais créé (en tant qu'étiquette seulement puisqu'en tant qu'ensemble d'occurrences il apparaît sous le 'nom' de $\{A,C\}\{A,T\}\{A,G\}\{A,C,G,T\}$) parce que la construction des modèles se fait, toujours, de la gauche vers la droite!

Il est important de comprendre que ces modèles qui ne sont pas engendrés en tant qu'étiquettes n'invalident pas l'affirmation que l'algorithme est capable de trouver tous les ensembles d'occurrences correspondant à des modèles valides — la recherche des groupes de mots similaires suivant la définition 3.5.12 de ressemblance demeure donc exhaustive par rapport à ces ensembles et non par rapport aux modèles. En d'autres termes, s'il arrive qu'un modèle soit 'oublié', alors il existe nécessairement un autre modèle qui présente le même ensemble d'occurrences.

Il y a ainsi deux facteurs qui font que la minimalité des modèles que l'algorithme construit, à travers la vérification des conditions c3 et c4, n'est pas absolue. Le premier est lié au fait que nous parcourons l'arbre en profondeur. Des modèles non minimaux (superflus) peuvent être conservés. Le second facteur découle du fait que le processus de construction possède une direction dans le temps. Celle-ci est elle-même conséquence de la pondération finie attribuée à certains éléments de la couverture. Elle est donc intrinsèque au principe de développement par récurrence de l'algorithme.

La propriété qui est vérifiée dans tous les cas par les conditions c3 et c4 est alors ce que nous avons appelé une minimalité gauche-droite. Nous pouvons affirmer aussi que si un modèle M donné est conservé, tous les modèles de plus courtes longueurs en ordre lexicographique de cardinalité des ensembles les composant qui vérifient le quorum sont également préservés. Ceci est le cas de M_4 et M_3 dans l'exemple 5.4.3.

La question maintenant est, comment vérifier ces conditions de manière efficace?

Soit M' le modèle que nous sommes en train d'allonger d'une unité vers la droite. Soient $Occ(M')$ l'ensemble de ses occurrences et $S = \{s_{i+|M'|} \in \Sigma \mid i \in Occ(M')\}$. En d'autres termes, S est l'ensemble des symboles de Σ qui suivent chacune des occurrences de M' dans s.

Alors les modèles $M = M'S'$ pour $S' \in CCP$ qui doivent être préservés sont ceux pour lesquels :

1. $(p(S', M') + 1) \leq w_{S'}$
   et
2. (a) $(S' \subseteq S)$
   ou

Soit M' le modèle que nous essayons d'étendre d'une unité vers la droite;
Soit $S = \{s_{i+|M'|} \in \Sigma \mid i \in Occ(M')\}$;
Soit $\text{Sup}(S')$ l'union de tous les ensembles S'' tels que $S' \subset S''$ et $\mid S'' \mid = \mid S' \mid + 1$;
EnsembleValable $= \emptyset$;

pour chaque ensemble $S' \in CCP$ pris par cardinalité <u>croissante</u>
    si $(S' \subseteq S)$
        si $((p(S', M') + 1) \leq w_{S'})$
            $M = M'S'$ est un modèle valide;
        sinon
            EnsembleValable = EnsembleValable $\cup$ $\text{Sup}(S')$;
    sinon si ($S' \in$ EnsembleValable)
        si $((p(S', M') + 1) \leq w_{S'})$
            $M = M'S'$ est un modèle valide;
        sinon
            EnsembleValable = EnsembleValable $\cup$ $\text{Sup}(S')$;

Figure 5.27: Vérification des conditions c3 et c4 (couverture combinatoire pondérée — première amélioration de l'algorithme direct).

(b) S' est un <u>plus petit</u> ensemble tel que $S \subset S'$ et tel qu'il existe $S'' \subseteq S$ avec $(p(S'', M') + 1) > w_{S''}$.

Bien sûr, $M'S'$ doit en outre vérifier la contrainte de quorum. Observons aussi qu'il peut y avoir plus d'un ensemble S' verifiant 2(b).

5.4.2.2.4.2 Algorithme

Une idée de l'algorithme correspondant à l'implémentation des deux points c3 et c4 ci-dessus est donné dans la figure 5.27.

5.4.2.2.4.3 Complexité

Si les structures de données $\text{Sup}(S)$ pour tout S dans la couverture et EnsembleValable sont implémentées comme des tableaux de booléens et si les opérations d'union et d'appartenance sont également booléennes, alors la complexité en temps que demande la vérification des conditions c3 et c4 est $O(p)$. Elle est donc linéaire avec le nombre des éléments de la couverture. La complexité en temps totale de l'algorithme est ainsi majorée par $O(n.N.k.g^k.p)$. Celle en espace demeure la même que pour l'approche directe.

Bien que la complexité en temps soit plus mauvaise en théorie, en pratique cela n'est pas le cas, la raison étant que moins de modèles sont produits et que l'espace de recherche est plus réduit. Des tests ont montré (voir la section performance) que le facteur en temps gagné dans l'exécution de l'algorithme est en général égal à 4 pour les acides nucléiques (cette version s'exécute 4 fois plus rapidement). Nous introduisons dans la section suivante une seconde idée qui peut améliorer encore plus la performance de l'algorithme, parfois de manière importante.

5.4.2.2.5 Esquisse de l'espace solution

5.4.2.2.5.1 Idée principale

La seconde idée pour améliorer la performance de l'algorithme (à la fois théorique et pratique) est issue de l'expérience suivante. Nous avions lancé l'algorithme à la recherche des plus longs modèles présents à 100% dans 4 chaînes aléatoires de longueur 100 définies sur l'alphabet $\{A, C, G, T\}$ avec la couverture :

$$CCP = \left\{ \begin{array}{l} (\{X\}, \infty)\ \forall X \in \Sigma, \\ (S, 1)\ \forall S \in \mathcal{P}^+(\Sigma),\ \mid S \mid \neq 1 \end{array} \right\}.$$

L'algorithme a nécessité 875 secondes sur une station de travail Silicon Graphics (R4000) pour produire une réponse, même après que la première modification (section 5.4.2.2.4) ait été implémentée.

Nous avions alors relancé l'algorithme dans les mêmes conditions que précédemment, avec cette fois :

$$CCP = \left\{ \begin{array}{l} (\{X\}, \infty)\ \forall X \in \Sigma, \\ (\{A, C, G, T\}, 2^{|\Sigma|-1} - \mid \Sigma \mid) \\ (S, 0)\ \forall S \in \mathcal{P}^+(\Sigma),\ \mid S \mid \neq 1\ et\ S \neq \Sigma \end{array} \right\}$$

c'est-à-dire, les seuls ensembles permis dans les modèles étaient les ensembles unitaires et le joker (en anglais, 'wild card' ou 'don't care symbol'), ce dernier ensemble étant autorisé au plus 11 (= 15 - 4) fois dans les modèles. De plus, nous avions recherché tous les modèles de longueur $k = 15$, qui correspondait à la longueur des plus longs modèles trouvés dans le premier cas.

En termes d'ensembles d'occurrences, il est clair que nous avons : espace solution(expérience 1) $\subseteq$ espace solution(expérience 2).

Le temps mis par l'algorithme pour produire une réponse à l'expérience 2 avait été de 2 secondes sur la même machine.

En termes d'une recherche dans l'espace de tous les modèles, l'expérience 2 peut être vue comme effectuant une exploration à 'gros grain' de cette espace, dans laquelle plusieurs branches de l'arbre sont examinées ensembles (en une seule fois). Cette expérience simple montre qu'une telle approche peut parfois être plus efficace que celle qui consiste à essayer de rechercher les réponses 'fines' dès le départ. Si cette première exploration conserve strictement le contour général de l'espace solution tout en étant capable d'éliminer une grande partie de l'espace de recherche initial, alors dans une seconde phase de l'algorithme, nous n'avons plus besoin d'explorer que le sous-espace délimité par la première phase. Cette manière de faire peut être avantageuse si l'exploration à gros grain est réalisable de façon plus rapide que l'exploration à grain fin et si l'espace solution qui est esquissé est plus réduit que l'espace de départ, ou si la première exploration plus 'grossière' est capable non seulement de limiter cet espace mais également de l'organiser d'une certaine manière.

5.4.2.2.5.2 Algorithme

Étant donné la couverture CCP définie par :

$$CCP = \left\{ \begin{array}{l} (\{X\}, a)\ \forall X \in \Sigma = \{A, C, G, T\}, \\ (\{X, Y\}, b)\ \forall X, Y \in \Sigma, \\ (\{X, Y, Z\}, c)\ \forall X, Y, Z \in \Sigma, \\ (\{A, C, G, T\}, d) \end{array} \right\}$$

1. Résolution du Problème 1 — recherche de tous les modèles de longueur k satisfaisant le quorum q :

1.1. réaliser une première recherche avec la couverture CCP' définie par :

$$CCP' = \left\{ \begin{array}{l} (\{X\}, a)\ \forall X \in \Sigma = \{A, C, G, T\}, \\ (\{X, Y\}, 0)\ \forall X, Y \in \Sigma, \\ (\{X, Y, Z\}, 0)\ \forall X, Y, Z \in \Sigma, \\ (\{A, C, G, T\}, b + c + d) \end{array} \right\}$$

1.2. pour chaque modèle M trouvé lors de la première recherche
concaténer ses occurrences et lancer une version légèrement modifiée de l'algorithme (voir la section 5.4.2.2.5.2), cette fois avec la couverture CCP et en recherchant tous les modèles valides de longueur k

2. Résolution du Problème 2 — recherche de tous les modèles de plus grande longueur satisfaisant le quorum q :

2.1. réaliser une première recherche avec la couverture CCP' donnée ci-dessus
soit k'_{max} la longueur des plus longs modèles trouvés

2.2. pour chaque modèle M trouvé lors de la première recherche
concaténer ses occurrences et lancer une version légèrement modifiée de l'algorithme (voir la section 5.4.2.2.5.2) avec cette fois la couverture CCP et en recherchant tous les modèles valides de longueur k'_{max}

2.3. si aucun modèle valide n'est trouvé
faire {
résoudre le Problème 1 pour $k = k'_{max}$ - 1, puis $k = k'_{max}$ - 2 etc
}
jusqu'à ce qu'au moins un modèle valide soit trouvé
(observons que cette dernière étape peut aussi être réalisée de manière dichotomique)

Figure 5.28: Une idée de la version en deux phases de recherche de l'algorithme Combi (couverture combinatoire pondérée — seconde amélioration de l'algorithme direct).

où a, b, c, d sont des entiers non-négatifs, et dans le cas de b, c, et d, finis, le schéma à suivre afin de résoudre le Problème 1 ou le 2 est celui indiqué dans la figure 5.28. L'algorithme final est appelé Combi.

La modification à apporter à l'algorithme dans la seconde phase (points 1.2 et 2.2) concerne uniquement la première étape, c'est-à-dire la construction des ensembles $Occ(M)$ pour $\mid M \mid = 1$ et $w_M \geq 1$. Ces ensembles sont maintenant obtenus en glissant le long de la chaîne s, mais cette fois seules sont placées dans $Occ(M)$ les positions i pour lesquelles $s_i \in M$ et i est un multiple de k (puisque nous travaillons dans cette seconde étape avec une chaîne formée par la concaténation des occurrences).

En reprennant nos expériences précédentes, si nous exécutons cette version en deux phases de l'algorithme sur le même ensemble de 4 chaînes aléatoires de longueur 100 chacune et recherchons les plus longs modèles présents dans 100% des chaînes, nous obtenons la réponse correcte en 38 secondes, ce qui représente un gain d'un facteur 25 par rapport à la version antérieure.

La version finale est donc 100 fois plus rapide que l'approche directe sur cet exemple particulier (rappelons que la première amélioration allait déjà 4 fois plus vite). Des tests supplémentaires (voir section performance) indiquent le gain observé sur d'autres exemples. Ils montrent que celui-ci dépend beaucoup des données, alors que le gain obtenu par l'exigence d'une minimalité gauche-droite est plus constant.

5.4.2.2.5.3 Complexité

Il est aisé de voir que la complexité en temps de la première phase de l'algorithme est $O(n.N.k.2^k)$ (g ainsi que $\overline{g}$ sont ici tous deux égaux à 2) et que celui de la seconde phase est $O(m.k.p)$ où m est le nombre total d'occurrences trouvées dans la première phase. En général, m est bien plus petit que $n.N.g^k$ (voir la section performance). Comme les deux phases de l'algorithme doivent être répétées au plus k'_{max} fois où k'_{max} est la plus grande longueur des modèles trouvés avec la couverture CCP', la complexité totale en temps de l'algorithme est bornée supérieurement par $O(n.N.(k'_{max})^2.2^{k'_{max}} + m.(k'_{max})^2.p)$.

En réalité, la valeur de k_{max} par rapport à la couverture CCP peut être déterminée de manière dichotomique à partir de la valeur k'_{max} par rapport à la couverture CCP'. La borne supérieure pour la complexité est donc $O(n.N.k'_{max}. \log k'_{max}.2^{k'_{max}} + m.k'_{max}. \log k'_{max}.p)$. Une telle technique n'est en fait pas utile en général (voir performance) car k_{max} est le plus souvent proche de k'_{max}. Lorsque des modèles de longueur fixe k sont recherchés, cette borne devient $O(n.N.k.2^k + m.k.p)$.

Comme pour l'approche directe, l'introduction d'une contrainte de quorum $q > 1$ et le fait que $w_S \neq \infty$ pour la plupart des ensembles S de la couverture, signifient qu'en pratique le comportement moyen de l'algorithme est bien meilleur que cela (voir ci-dessous).

Observons enfin que la complexité en espace de cette version finale de l'algorithme est bornée supérieurement par $O(n.N.k'_{max}.2^{k'_{max}})$ puisque tous les modèles de longueur k'_{max} de la première phase ainsi que leurs occurrences doivent être conservés pour la seconde phase.

5.4.2.2.5.4 Observation importante concernant les modèles

Il est important de noter que, jusqu'à maintenant, nous n'avons pas tenu compte de quels sous-ensembles pouvaient être présents en première et dernière positions d'un modèle considéré valide. En particulier, un tel modèle peut très bien être de la forme $M = \{\Sigma\}M'\{\Sigma\}$, c'est-à-dire débuter et se terminer par le joker. Il est clair que de tels modèles ne sont souvent pas très intéressants, et il est relativement facile de modifier l'algorithme de telle sorte que les modèles repérés (d'une longueur fixe ou de la plus grande longueur possible) ne contiennent le joker ni en première ni en dernière position.

Exiger l'absence du joker en première position implique modifier l'appel de la fonction Explore à la fin de la fonction principale de l'algorithme 5.26. Cet appel ne se fait désormais que pour les $S_i \in CCP$ tels que $S_i \neq \{\Sigma\}$. Dans le cas où des modèles d'une longueur fixe sont recherchés, ne sont stockés que les modèles $M = M'S$ pour lesquels $S \neq \{\Sigma\}$. Enfin, lorsque ce sont les plus longs modèles qui nous intéressent, les modèles de longueur plus petite conservés au préalable ne sont rejetés que si le modèle valide nouveau ne possède pas de joker en dernière position.

Cette contrainte imposée au modèle ne change rien à la complexité théorique de l'algorithme mais elle peut avoir une influence relativement importante sur le temps d'exécution en pratique ainsi que sur le volume du résultat final obtenu. Cela est montré dans la section performance.

5.4.2.2.6 Variantes : introduction de contraintes supplémentaires

5.4.2.2.6.1 Pondération sur les ensembles limitée par la pondération sur le joker

Travailler avec un algorithme en deux phases nous fournit une manière facile d'imposer des contraintes supplémentaires sur le nombre d'ensembles qui peuvent apparaître dans les modèles valides outre celle concernant l'absence de joker en première et dernière position (voir la section 5.4.2.2.5.4). Ceci est obtenu en fixant le poids attaché à $\{\Sigma\}$ dans CCP' non pas égal à $b+c+d$ mais égal plutôt à une valeur notée f strictement plus petite que $b+c+d$. Dans ce cas, les modèles ne peuvent contenir plus de f ensembles 'spéciaux', où par un ensemble 'spécial' nous entendons n'importe quel ensemble non unitaire spécifié dans la seconde phase. Cette contrainte additionnelle a prouvé son utilité dans un certain nombre d'applications (voir le chapitre 6). Un cas particulier très employé en pratique consiste à établir $b = c = d = f$. Ceci équivaut à permettre la présence de f sous-ensembles non unitaires dans les modèles, quels qu'ils soient.

5.4.2.2.6.2 Pondération sur les ensembles groupés par cardinalité

Dans ce cas, les pondérations sont affectées aux ensembles de la couverture groupés par cardinalité. Un exemple d'une telle pondération est donnée ci-dessous.

Exemple 5.4.5 *Soient* $\Sigma = \{A, C, G, T\}$ *et :*

$$CCP_{card} = \left\{ \begin{array}{l} (S, \infty)\ si\ \mid S \mid = 1, \\ (S, 3)\ si\ \mid S \mid = 2, \\ (S, 2)\ si\ \mid S \mid = 3, \\ (\{A, C, G, T\}, 1) \end{array} \right\}.$$

En d'autres termes, les sous-ensembles de $\{\Sigma\}$ ayant 2 symboles sont permis 3 fois dans les modèles, ceux ayant 3 symboles le sont 2 fois et le joker 1 fois. Le modèle $\{A, C\}\{A, C\}\{A, C\}$ $\{A, C, G, T\}\{A, C, G\}\{A, C, G\}$ est ainsi un modèle possible, c'est-à-dire est un élément de CCP_{card}. La modification à apporter à l'algorithme 5.26 est immédiate à réaliser : le tableau p_M compte désormais la présence des sous-ensembles par leur cardinalité et doit donc avoir une taille égale à $\mid \Sigma \mid$.

5.4.2.2.6.3 Nombre de jokers variable décrémenté à chaque retour à la première phase

Enfin, dans une dernière variante de l'algorithme, le nombre de jokers autorisés lors de la première phase peut être décrémenté à chaque étape. Ainsi initialement il est fixé égal à $b+c+d$, cependant si la seconde phase ne réussit à identifier aucun modèle valide et que l'algorithme doit revenir à la première, ce nombre est fixé à $(b+c+d)-1$ et ainsi de suite. Le nombre de jokers varie donc avec k'_{max}. Bien évidemment , cette variante est une heuristique. Elle trouve sa justification dans le fait que si aucun modèle de longueur k_{max} n'a pu être trouvé dans la seconde phase à partir de ceux repérés dans la première, alors il y a relativement peu de chances pour qu'un modèle de longueur $k'_{max}-1$ de la première phase et ayant le même nombre $b+c+d$ de jokers qu'au départ résulte en un modèle valide final. Nous verrons dans la section performance que cette heuristique fonctionne correctement la plupart du temps (i.e. elle permet de trouver les mêmes modèles que la variante exacte). Elle a tendance à être prise en défaut pour des faibles quorums.

5.4.2.2.6.4 Pondération sur les ensembles nulle presque partout (cas des protéines)

Ainsi que nous l'avons mentionné en introduction à cette section, il n'est pas envisageable, du moins pour l'instant, de travailler avec une couverture totalement combinatoire lorsque nous voulons comparer des protéines. Dans ce cas en effet, $\mid \Sigma \mid = 20$ et, par conséquent, $\mid \mathcal{P}^+(\Sigma) \mid = (2^{20} - 1)$.

Plusieurs approches peuvent être considérées afin de réussir néanmoins à traiter le cas de ces macromolécules. Toutes reposent sur le même principe, à savoir que la pondération affectée aux sous-ensembles de l'alphabet des acides aminés sera nulle presque partout <u>sauf</u> pour un sous-ensemble limité d'éléments de $\mathcal{P}^+(\Sigma)$. C'est dans le choix de ce sous-ensemble que les méthodes varient.

Celle adoptée par Neuwald [Neuwald and Green, 1994] consiste à établir que la pondération d'un sous-ensemble S de $\mathcal{P}^+(\Sigma)$ ne sera strictement positive que si :

- $S = \{\Sigma\}$;
  <u>ou</u>
- $S = \{a\}$ avec $a \in \Sigma$;
  <u>ou encore</u>
- $S = \{a, b\}$ avec $a, b \in \Sigma$ et $\mathcal{M}(a, b) \geq t$ où $\mathcal{M}$ est une matrice de similarité (par exemple PAM) et t est une valeur seuil fixée à l'avance.

Neuwald n'utilise en fait pas le concept d'une couverture combinatoire pondérée mais une contrainte très similaire est employée. L'idée d'un modèle composé de produits de certains sous-ensembles de l'alphabet des acides aminés apparaît également clairement sous le nom de motif. Ce dernier doit vérifier les conditions suivantes :

1. un motif valide doit débuter et se terminer par un sous-ensemble unitaire;
2. un motif valide doit comporter un nombre fixe de sous-ensembles de cardinalité 1 ou 2;
3. la longueur maximale des motifs est limitée ce qui, en même temps que la condition 2, limite effectivement le nombre de jokers permis.

La méthode de construction des motifs est techniquement assez proche de celle que nous suivons (version minimalité), mais la condition d'élagage de l'arbre des motifs est différente. Celle-ci porte en effet non seulement sur une contrainte de quorum q mais aussi sur la signification statistique d'un motif. Ainsi dans leur cas, des critères statistiques influencent la recherche des motifs <u>dès le départ</u>, ce que nous avons voulu éviter de faire pour des raisons qui seront discutées plus loin. Il est très aisé d'appliquer le concept d'une pondération à la couverture avec laquelle Neuwald *et al.* travaillent, ou n'importe quel autre couverture. Pour éviter d'introduire un point de vue comme cela est fait dans le cas de Neuwald à travers l'usage d'une matrice de substitution, il est possible de fixer des critères plus combinatoires afin de décider de la pondération à affecter à un élément de $\mathcal{P}^+(\Sigma)$. Par exemple, nous pouvons établir que ne seront permis dans un modèle que le joker (un nombre fini de fois) ainsi que tous les ensembles de cardinalité inférieure à une certaine valeur c. Il est clair qu'en pratique, c ne peut être très grand.

Nous présentons plus loin une autre façon encore d'aborder la question, qui diffère de celle employée par Neuwald ou suggérée plus haut dans le choix du sous-ensemble et surtout dans le fait que nous allons, contrairement à Neuwald, également pouvoir travailler avec des erreurs.

5.4.2.2.7 Performance en pratique

Nous présentons maintenant les performances de Combi et de ses variantes sur un jeu d'essai composé des mêmes 140 chaînes de promoteurs de *Bacillus subtilis* utilisé dans la section 5.4.2.1.7.

Ces performances sont indiquées sous forme de tableaux et non de courbes cette fois, parce qu'il est important dans ce cas d'indiquer et de discuter le rôle qu'exercent plusieurs autres facteurs sur le comportement de l'algorithme. Ces facteurs sont essentiellement :

- le nombre de fois où l'algorithme doit exécuter les deux phases avant de réussir à obtenir des modèles valides lorsque la longueur des modèles recherchés est la longueur maximale k_{max};
- la longueur maximale k'_{max} des modèles obtenus lors du premier passage de l'algorithme par la phase 1 et celle k_{max} obtenue à la fin;
- le nombre de modèles et d'occurrences observés dans la première et dernière phases de l'algorithme pour $k = k_{max}$.

Les valeurs de ces divers facteurs et les temps d'exécution correspondants (observés sur une Sparc Station 20) sont donnés pour plusieurs couvertures et variantes de Combi. Les couvertures utilisées ont été :

- pour le tableau 5.5 :

$$CCP1 = \left\{ \begin{array}{l} (\{X\}, \infty)\ \forall X \in \Sigma = \{A, C, G, T\}, \\ (\{X, Y\}, 1)\ \forall X, Y \in \Sigma, \\ (\{X, Y, Z\}, 1)\ \forall X, Y, Z \in \Sigma, \\ (\{A, C, G, T\}, 0) \end{array} \right\}$$

  avec, en première phase :

$$CCP1' = \left\{ \begin{array}{l} (\{X\}, \infty)\ \forall X \in \Sigma = \{A, C, G, T\}, \\ (\{X, Y\}, 0)\ \forall X, Y \in \Sigma, \\ (\{X, Y, Z\}, 0)\ \forall X, Y, Z \in \Sigma, \\ (\{A, C, G, T\}, 4) \end{array} \right\};$$

- pour le tableau 5.6 :

$$CCP2 = \left\{ \begin{array}{l} (\{X\}, \infty)\ \forall X \in \Sigma = \{A, C, G, T\}, \\ (\{X, Y\}, 1)\ \forall X, Y \in \Sigma, \\ (\{X, Y, Z\}, 1)\ \forall X, Y, Z \in \Sigma, \\ (\{A, C, G, T\}, 1) \end{array} \right\}$$

  avec, en première phase :

$$CCP2' = \left\{ \begin{array}{l} (\{X\}, \infty)\ \forall X \in \Sigma = \{A, C, G, T\}, \\ (\{X, Y\}, 0)\ \forall X, Y \in \Sigma, \\ (\{X, Y, Z\}, 0)\ \forall X, Y, Z \in \Sigma, \\ (\{A, C, G, T\}, 4) \end{array} \right\};$$

- pour le tableau 5.7 :

$$CCP3 = \left\{ \begin{array}{l} (\{X\}, \infty)\ \forall X \in \Sigma = \{A, C, G, T\}, \\ (\{A, G\}, 2), (\{C, T\}, 2), (\{X, Y\}, 0)\ \textit{pour toute autre paire}\ \{X, Y\} \in \Sigma^2, \\ (\{X, Y, Z\}, 0)\ \forall X, Y, Z \in \Sigma, \\ (\{A, C, G, T\}, 0) \end{array} \right\}$$

avec, en première phase :

$$CCP3' = \left\{ \begin{array}{l} (\{X\}, \infty)\ \forall X \in \Sigma = \{A, C, G, T\}, \\ (\{X, Y\}, 0)\ \forall X, Y \in \Sigma, \\ (\{X, Y, Z\}, 0)\ \forall X, Y, Z \in \Sigma, \\ (\{A, C, G, T\}, 4) \end{array} \right\}$$

(dans ces trois cas, il s'agit donc de la variante de Combi pour laquelle la pondération sur les ensembles est limitée par la pondération sur le joker en première phase décrite dans la section 5.4.2.2.6.1);

- pour le tableau 5.8 :

$$CCP1_{card} = \left\{ \begin{array}{l} (S, \infty)\ si\ \mid S \mid = 1, \\ (S, 2)\ si\ \mid S \mid = 2, \\ (S, 1)\ si\ \mid S \mid = 3, \\ (\{A, C, G, T\}, 1) \end{array} \right\}$$

avec en première phase :

$$CCP1'_{card} = \left\{ \begin{array}{l} (S, \infty)\ si\ \mid S \mid = 1, \\ (S, 0)\ si\ \mid S \mid = 2, \\ (S, 0)\ si\ \mid S \mid = 3, \\ (\{A, C, G, T\}, 4) \end{array} \right\};$$

- pour le tableau 5.9 :

$$CCP2_{card} = \left\{ \begin{array}{l} (S, \infty)\ si\ \mid S \mid = 1, \\ (S, 2)\ si\ \mid S \mid = 2, \\ (S, 0)\ si\ \mid S \mid = 3, \\ (\{A, C, G, T\}, 1) \end{array} \right\}$$

avec en première phase :

$$CCP2'_{card} = \left\{ \begin{array}{l} (S, \infty)\ si\ \mid S \mid = 1, \\ (S, 0)\ si\ \mid S \mid = 2, \\ (S, 0)\ si\ \mid S \mid = 3, \\ (\{A, C, G, T\}, 2) \end{array} \right\};$$

- pour le tableau 5.10 :

$$CCP3_{card} = \left\{ \begin{array}{l} (S, \infty)\ si\ \mid S \mid = 1, \\ (S, 2)\ si\ \mid S \mid = 2, \\ (S, 2)\ si\ \mid S \mid = 3, \\ (\{A, C, G, T\}, 0) \end{array} \right\}$$

avec en première phase :

$$CCP3'_{card} = \left\{ \begin{array}{l} (S, \infty)\ si\ \mid S \mid = 1, \\ (S, 0)\ si\ \mid S \mid = 2, \\ (S, 0)\ si\ \mid S \mid = 3, \\ (\{A, C, G, T\}, 4) \end{array} \right\}$$

(ces trois séries de tableaux analysent donc le comportement de la variante de Combi décrite dans la section 5.4.2.2.6.2 pour laquelle la pondération porte sur les ensembles groupés par cardinalité; le tableau 5.9 combine cette variante à celle de la section 5.4.2.2.6.1);

- enfin, pour les tableaux 5.11 et 5.12 :

$$CCP = \left\{ \begin{array}{l} (\{X\}, \infty)\ \forall X \in \Sigma = \{A, C, G, T\}, \\ (\{X, Y\}, 1)\ \forall X, Y \in \Sigma, \\ (\{X, Y, Z\}, 1)\ \forall X, Y, Z \in \Sigma, \\ (\{A, C, G, T\}, 1) \end{array} \right\}$$

avec, en première phase :

$$CCP' = \left\{ \begin{array}{l} (\{X\}, \infty)\ \forall X \in \Sigma = \{A, C, G, T\}, \\ (\{X, Y\}, 0)\ \forall X, Y \in \Sigma, \\ (\{X, Y, Z\}, 0)\ \forall X, Y, Z \in \Sigma, \\ (\{A, C, G, T\}, 11) \end{array} \right\}$$

Ces tableaux appelent plusieurs commentaires. En ce qui concerne les tableaux 5.5 à 5.10, nous constatons que le nombre de modèles obtenus peut varier considérablement avec le nombre de chaînes pour une même valeur de quorum, et cela d'une manière qui est, malheureusement, tout à fait imprévisible. La même variation serait observée si l'on maintenait le nombre de chaînes fixe et que l'on faisait varier cette fois le quorum. Décrémenter l'un ou l'autre augmente en général le nombre de modèles jusqu'au moment où des modèles plus longs (et moins nombreux) sont identifiés. Le degré de non-transitivité de la couverture possède également une grande influence sur le nombre des modèles. En effet, lorsque ce degré est élevé, on observe beaucoup plus de modèles qui sont des variantes proches les unes des autres. La raison en est que plusieurs éléments de la couverture peuvent être utilisés aux positions les plus variables des ensembles d'occurrences. De plus, lorsque le quorum n'est pas égal à 100% (comme c'est le cas ici), chacun de ces éléments peut conduire à une version légèrement modifiée d'un modèle. En ce qui concerne les applications que nous visons, ce type de comportement peut ne pas être négatif. Un exemple de ceci sera donné dans le chapitre des illustrations. Cet exemple suggère cependant qu'il serait intéressant dans certains cas d'ajouter une étape à l'algorithme, dans laquelle les modèles obtenus par l'algorithme tel qu'il est conçu aujourd'hui pourraient être regroupés en des objets plus généraux (des méta-modèles), c'est-à-dire, des modèles de modèles. Une autre possibilité serait de réaliser une évaluation statistique de ces modèles et de les classer ensuite selon leur 'pertinence'.

Toujours en ce qui concerne ces six tableaux, observons que, ainsi que nous l'avions mentionné à la fin de la section 5.4.2.2.6.3, la contrainte qui consiste à décrémenter le nombre de jokers à chaque retour à la première phase de l'algorithme peut avoir une influence importante sur le temps d'exécution. Nous pouvons également vérifier dans ces tableaux que le nombre total d'occurrences trouvées dans la première phase est, d'une part bien inférieur à $n.N.g^k$ et, d'autre part, généralement inférieur au nombre total d'occurrences trouvées en première phase.

Couverture $CCP1$ - pas de décrémentation du nombre de jokers entre les 2 phases								
N	nombre de cycles	k'_{max}	k_{max}	#mod. phase 1	#mod. phase 2	#occ. phase 1	#occ. phase 2	temps (s)
10	2	10	9	225	284	2182	2141	5.19
50	2	9	8	516	36	44121	2003	26.95
100	3	9	7	774	40701	174573	5652372	273.29
140	2	8	7	760	35590	238184	7015157	334.43
Couverture $CCP1$ - nombre de jokers décrémenté de 1 entre les 2 phases								
N	nombre de cycles	k'_{max}	k_{max}	#mod. phase 1	#mod. phase 2	#occ. phase 1	#occ. phase 2	temps (s)
10	6	10	5	2	14	22	154	3.07
50	5	9	5	2	14	138	966	7.75
100	3	9	7	251	170	42463	20316	25.37
140	2	8	7	247	161	57802	27761	62.84

Tableau 5.5: Comportement de Combi pour la couverture $CCP1$ et la variante 5.4.2.2.6.1 toute seule ou combinée à la variante 5.4.2.2.6.3 et pour $q = 70\%$.

Couverture $CCP2$ - pas de décrémentation du nombre de jokers entre les 2 phases								
N	nombre de cycles	k'_{max}	k_{max}	#mod. phase 1	#mod. phase 2	#occ. phase 1	#occ. phase 2	temps (s)
10	1	10	10	4	3	30	21	1.69
50	2	9	8	516	4325	44121	253418	53.87
100	2	9	8	433	666	72454	82070	63.11
140	1	8	8	421	613	97335	110270	68.15
Couverture $CCP2$ - nombre de jokers décrémenté de 1 entre les 2 phases								
N	nombre de cycles	k'_{max}	k_{max}	#mod. phase 1	#mod. phase 2	#occ. phase 1	#occ. phase 2	temps (s)
10	1	10	10	4	3	30	21	1.69
50	3	9	7	1	1	50	48	7.04
100	3	9	7	251	4278	42463	571817	41.28
140	1	8	8	421	613	97335	110270	69.28

Tableau 5.6: Comportement de Combi pour la couverture $CCP2$ et la variante 5.4.2.2.6.1 toute seule ou combinée à la variante 5.4.2.2.6.3 et pour $q = 70\%$.

Couverture $CCP3$ - pas de décrémentation du nombre de jokers entre les 2 phases								
N	nombre de cycles	k'_{max}	k_{max}	#mod. phase 1	#mod. phase 2	#occ. phase 1	#occ. phase 2	temps (s)
10	4	10	7	1239	5	23787	37	14.78
50	4	9	6	586	43	88047	2466	46.04
100	4	9	6	559	11	167338	1260	81.09
140	3	8	6	550	5	229444	901	102.11
Couverture $CCP3$ - nombre de jokers décrémenté de 1 entre les 2 phases								
N	nombre de cycles	k'_{max}	k_{max}	#mod. phase 1	#mod. phase 2	#occ. phase 1	#occ. phase 2	temps (s)
10	6	10	5	2	4	22	44	3.10
50	5	9	5	2	4	138	276	8.48
100	5	9	5	53	7	9086	918	22.72
140	4	8	5	50	5	12062	1000	45.78

Tableau 5.7: Comportement de Combi pour la couverture $CCP3$ et la variante 5.4.2.2.6.1 toute seule combinée à la variante 5.4.2.2.6.3 et pour $q = 70\%$.

Couverture $CCP1_{card}$ - pas de décrém. du nombre de jokers entre les 2 phases								
N	nombre de cycles	k'_{max}	k_{max}	#mod. phase 1	#mod. phase 2	#occ. phase 1	#occ. phase 2	temps (s)
10	1	10	10	4	1	30	7	1.94
50	2	9	8	516	38	44121	2076	32.27
100	3	9	7	774	27393	174573	4025253	267.33
140	2	8	7	760	24337	238184	5043267	327.35

Couverture $CCP1_{card}$ - nombre de jokers décrémenté de 1 entre les 2 phases								
N	nombre de cycles	k'_{max}	k_{max}	#mod. phase 1	#mod. phase 2	#occ. phase 1	#occ. phase 2	temps (s)
10	1	10	10	4	1	30	7	1.93
50	3	9	7	1	1	50	48	7.49
100	3	9	7	251	657	42463	84459	30.88
140	2	8	7	247	558	57802	102640	80.00

Tableau 5.8: Comportement de Combi (variante 5.4.2.2.6.2) pour la couverture $CCP1_{card}$ et la variante 5.4.2.2.6.1 toute seule ou combinée à la variante 5.4.2.2.6.3 et pour $q = 70\%$.

Couverture $CCP2_{card}$ - pas de décrém. du nombre de jokers entre les 2 phases								
N	nombre de cycles	k'_{max}	k_{max}	#mod. phase 1	#mod. phase 2	#occ. phase 1	#occ. phase 2	temps (s)
10	1	8	8	2	4	16	28	0.36
50	2	7	6	159	640	13638	44709	5.47
100	1	6	6	126	363	21570	51206	6.63
140	1	6	6	126	299	29776	58895	8.96
Couverture $CCP2_{card}$ - nombre de jokers décrémenté de 1 entre les 2 phases								
N	nombre de cycles	k'_{max}	k_{max}	#mod. phase 1	#mod. phase 2	#occ. phase 1	#occ. phase 2	temps (s)
10	1	8	8	2	4	16	28	0.36
50	2	7	6	2	4	121	284	1.48
100	1	6	6	126	363	21570	51206	6.66
140	1	6	6	126	299	29776	58895	8.96

Tableau 5.9: Comportement de Combi (variante 5.4.2.2.6.2) pour la couverture $CCP2_{card}$ et la variante 5.4.2.2.6.1 toute seule ou combinée à la variante 5.4.2.2.6.3 et pour $q = 70\%$.

Couverture $CCP3_{card}$ - pas de décrém. du nombre de jokers entre les 2 phases								
N	nombre de cycles	k'_{max}	k_{max}	#mod. phase 1	#mod. phase 2	#occ. phase 1	#occ. phase 2	temps (s)
10	2	10	9	225	58	2182	431	5.28
50	2	9	8	516	1	44121	50	30.72
100	3	9	8	774	15867	174573	2237249	271.85
140	3	8	7	760	13324	238184	2662072	306.97
Couverture $CCP3_{card}$ - nombre de jokers décrémenté de 1 entre les 2 phases								
N	nombre de cycles	k'_{max}	k_{max}	#mod. phase 1	#mod. phase 2	#occ. phase 1	#occ. phase 2	temps (s)
10	5	10	5	2	14	22	154	3.19
50	5	9	5	2	14	138	966	8.28
100	2	9	7	251	69	42463	8341	26.88
140	2	8	7	247	51	57802	8852	66.97

Tableau 5.10: Comportement de Combi (variante 5.4.2.2.6.2) pour la couverture $CCP3_{card}$ et la variante 5.4.2.2.6.1 toute seule ou combinée à la variante 5.4.2.2.6.3 et pour $q = 70\%$.

N	#tours	k'_{max}	k_{max}	#mod. phase 1	#mod. phase 2	#occ. phase 1	#occ. phase 2	temps (s)
4	1	17	17	9	32	18	64	1.72
6	3	14	13	264	894	47	141	6.26
8	3	14	12	218	1311	8	32	21.57
10	4	14	11	246	2133	125	625	113.33
12	3	13	11	200	2170	14	84	74.47
14	4	13	10	177	2375	137	1007	110.18
16	4	13	10	179	2867	111	942	119.21
18	4	13	10	162	2952	16	153	111.98
20	4	13	10	151	3136	4	41	103.67
22	4	13	10	128	3012	3	34	87.51
24	5	13	9	144	3718	332	4320	152.74
26	5	13	9	132	3795	180	2569	149.12
28	4	12	9	136	4203	134	2051	155.47
30	4	12	9	151	4946	340	5654	194.48

Tableau 5.11: Comportement de l'algorithme Combi original pour $q = 70\%$, de 2 à 30 chaînes de promoteurs et en utilisant la couverture CCP.

Enfin, les tableaux 5.11 et 5.12 présentent une comparaison des trois approches, directe, minimalité et esquisse de l'espace solution, de Combi. Le premier des deux tableaux indique simplement les valeurs des divers paramètres intervenant dans le comportement de la version finale de Combi (en particulier le nombre de fois où l'algorithme doit exécuter les deux phases). Le second tableau compare les temps d'exécution des trois approches. Nous pouvons observer que :

- la version finale est d'autant plus performante par rapport aux approches directe ou minimalité que la valeur de k'_{max} est proche de celle de k_{max} (i.e. lorsque l'algorithme doit effectuer peu de cycles), elle peut par contre être légèrement moins rapide que la version minimalité si ces deux valeurs sont éloignées (par exemple ici pour N égal à 24, 26, 28 ou 30);
- le rapport des temps d'exécution des versions directe et minimalité est pratiquement constant et approximativement égal à 4 (la version directe est 4 fois plus lente que la version minimalité).

5.4.2.3 Couverture pondérée et distance de Levenshtein

5.4.2.3.1 Introduction

Nous avons mentionné dans la section 5.4.2.2.6.4 qu'il n'est pas possible, à l'heure actuelle,

Nombre de séquences	Temps		
	Direct	Minimalité	Esquisse
4	7648.70	620.14	1.72
6	991.60	306.28	6.26
8	526.19	183.89	21.57
10	539.67	179.57	113.33
12	477.28	146.90	74.47
14	367.25	110.76	110.18
16	404.23	115.45	119.21
18	404.68	118.35	111.98
20	397.48	107.63	103.67
22	377.62	108.46	87.51
24	388.93	99.43	152.74
26	394.55	98.61	149.12
28	418.61	103.58	155.47
30	498.08	122.58	194.48

Tableau 5.12: Comportement de l'algorithme Combi original, approches 'directe' (section 5.4.2.2.3), 'minimalité' (section 5.4.2.2.4) et 'esquisse de l'espace solution' (section 5.4.2.2.5) pour $q = 70\%$, de 2 à 30 chaînes de promoteurs et en utilisant la couverture CCP.

de travailler avec une couverture totalement combinatoire lorsque nous voulons comparer des protéines. Nous avons indiqué alors une façon de procéder lorsque nous voulons essayer malgré tout d'analyser ces macromolécules sans point de vue a priori.

En fait, le sous-ensemble le plus intéressant de Σ correspond à $\{\Sigma\}$, qu'il s'agisse d'analyser des protéines ou des acides nucléiques. La section 3.5.2.1.2 a montré en effet qu'introduire des jokers dans les modèles permet de repérer des régions d'une macromolécule dont toutes les positions ne sont pas impliquées dans l'activité biologique. Par ailleurs, il peut être important en plus d'autoriser des erreurs entre les modèles et leurs instances ainsi que nous l'avons fait dans l'algorithme LePoivre. Nous allons donc considérer que nous disposons maintenant d'une couverture comportant un certain nombre de sous-ensembles de l'alphabet Σ dont le joker, et que ce dernier est le seul élément de la couverture qui ne reçoit pas une pondération infinie. Notons z la pondération de $\{\Sigma\}$. Observons que la couverture peut être très dégénérée et comporter des éléments dont certains sont strictement inclus dans d'autres. Elle peut également être établie sur la base de critères biologiques (structuraux par exemple) ou combinatoires du genre de celui suggéré dans la section 5.4.2.2.6.4.

Nous allons alors établir qu'un mot dans une chaîne est une occurrence d'un modèle défini sur une telle couverture (ce modèle ne possède pas plus de z jokers) si ce mot se trouve à une distance Ensembliste de Levenshtein au plus égale à e du modèle. Nous introduisons d'abord formellement le lemme permettant de construire de tels modèles, puis nous indiquons comment obtenir l'algorithme à partir de ceux pour une distance de Levenshtein et pour une couverture combinatoire pondérée. Pour simplifier, nous nous contentons en fait de 'compter les erreurs' pendant le déroulement de l'algorithme, ce qui veut dire que, par rapport à la distance Ensembliste de Levenshtein et au lemme 5.4.4 de LePoivre, nous travaillons ici avec l'ensemble $ES(M)$ plutôt qu'avec l'ensemble $EL(M)$ pour M un modèle. Les 'vraies' occurrences en termes d'une distance de Levenshtein peuvent être calculées à la fin de l'algorithme comme cela était déjà fait dans les versions de LePoivre présentées dans la section 5.4.2.1.5.

5.4.2.3.2 Définitions de récurrence

Nous commençons par définir une couverture combinatoire pondérée restreinte $CCPR$ de l'alphabet Σ des monomères.

Définition 5.4.3 *Une couverture combinatoire pondérée restreinte $CCPR$ de Σ est définie de la façon suivante :*

$$CCPR = \left\{ \begin{array}{l} (S, \infty)\ \forall S \in C \ où\ C\ est\ un\ ensemble\ de\ sous-ensembles\ propres\ de\ \Sigma, \\ (S, 0)\ pour\ S \in \mathcal{P}^+(\Sigma) \setminus C, \\ (\{\Sigma\}, z)\ où\ z\ est\ un\ entier\ fini\ non-négatif \end{array} \right\}.$$

Dans le cas des acides nucléiques, nous avons en général $C = CI$, et dans celui des protéines C sera une couverture comme celles établie à partir du diagramme de Taylor [Taylor, 1986a] et donnée dans la figure 3.11 du chapitre 3.

Soient $s \in \Sigma^*$, i une position dans s, $CCPR$ une couverture combinatoire pondérée restreinte de Σ et $M_1 \in CCPR^k$. Nous avons alors le lemme suivant (rappelons que $Occ(M)$ désigne un ensemble un peu plus grand que celui des occurrences strictes d'un modèle M puisqu'il s'agit de l'ensemble de tous les mots ayant au plus $d \leq e$ erreurs avec M où d peut ne pas représenter une distance) :

Lemme 5.4.6

$$(i, d_s, d_d, d_i) \in Occ(M = M_1 S) \; avec \; S \in CCPR$$

$$\Longleftrightarrow$$

$$\exists \, d_{s_1}, d_{d_1}, d_{i_1} \; tel \; que \; (i, d_{s_1}, d_{d_1}, d_{i_1}) \in Occ(M_1) \; et \; :$$

$$[Condition \, 1 \; et \; (Condition \, 2 \; ou \; Condition \, 3)]$$

$$ou$$

$$[Condition \, 4]$$

avec :

Condition 1 $S \neq \{\Sigma\}$;

Condition 2 $s_i + \mid M_1 \mid - d_{d_1} + d_{i_1} \in S$;

Condition 3 $s_i + \mid M_1 \mid - d_{d_1} + d_{i_1} \notin S$ *et* $d_s + d_d + d_i = d_{s_1} + d_{d_1} + d_{i_1} + 1 \leq e$;

Condition 4 $S = \{\Sigma\}$ *et* $(p(S, M_1) + 1) \leq z$ *où* $p(S, M_1)$ *est le nombre de fois où l'ensemble* S *apparaît dans le modèle* M_1.

5.4.2.3.3 Algorithme

L'algorithme établi à partir du lemme et permettant de construire l'ensemble des occurrences d'un modèle M dans le sens d'une couverture combinatoire restreinte $CCPR$ est esquissé dans la figure 5.29. Pour les raisons indiquées dans la section 5.4.2.2.5.4, le joker est interdit en première ainsi qu'en dernière position d'un modèle.

```
/* Entrée */
        CCPR : une couverture combinatoire pondérée restreinte établie à partir
        d'une couverture C
        p : le nombre d'éléments dans C
        e, z : le nombre d'erreurs et de jokers autorisés respectivement
        Le reste comme pour les algorithmes précédents
/* Sortie */
        Comme pour les algorithmes précédents
/* Principales structures de données */
        Comme pour les algorithmes des figures 5.15 et 5.28, sauf le tableau Modèles[i] qui
        n'est pas utilisé
Main {
        pour (i ∈ [1..n])
                pour (S_j ∈ C)
                        si (i ∈ S_j)
                                Occ(S_j) = Occ(S_j) ∪ {(i, 0, 0, 0)};
                        sinon
                                Occ(S_j) = Occ(S_j) ∪ {(i, 1, 0, 0)} ∪ {(i, 0, 1, 0)}};
        pour (S_j ∈ C)
                si (Occ(S_j) vérifie le quorum q)
                        Explore(S_j, 1, Occ(S_j));
}
```

Figure 5.29 : Algorithme LeCombi (couverture combinatoire pondérée restreinte et distance de Levenshtein — ici simple comptage des erreurs) pour résoudre le Problème 1 — continue à la page suivante.

5.4.2.3.3.1 Complexité

Il est aisé de déduire de la complexité de LePoivre que la complexité en temps de LeCombi est égale à (opération de filtrage exclue) $O(n.N.(e{+}2).(2e{+}1).k^{e+1}.(p{+}1)^{e+1}.2^k)$ où p est le nombre d'éléments dans C. Dans le cas de l'ADN et de l'ARN en particulier, il est courant d'avoir $p = \mid \Sigma \mid$. La complexité en espace de l'algorithme est bornée supérieurement par $O(n.N.k.(2e+1))$ pour un parcours en profondeur.

5.4.2.3.4 Amélioration

Il est possible d'appliquer ici les mêmes modifications que pour l'algorithme LePoivre, en particulier celle de la section 5.4.2.1.5.

La complexité de l'algorithme est alors $O(n.N.(e+1).(e+2).k^{e+1}.(p+1)^{e+1}.2^k + k.(k+e).N_{ModèlesValides})$ en temps et $O(n.N.k)$ en espace.

C'est cette version améliorée dont nous analysons la performance en pratique dans la prochaine section.

5.4.2.3.5 Performance en pratique

Nous présentons dans cette section la performance de LeCombi lorsque nous faisons varier le

```
Explore(M, l, Occ(M)) {
    si (l = k)                                   /* base de la récursion */
        si (le joker n'est pas le dernier élément de M)
            stocke M et Occ(M);
    sinon {                                      /* pas de la récursion */
        pour ((i, d_s, d_d, d_i) ∈ Occ(M)) {
            pour (S = ∅)                         /* insertion */
                si (d_s + d_d + d_i + 1 ≤ e) {
                    Occ(MS) = Occ(MS) ∪ {(i, d_s, d_d, d_i + 1)};
                    ExtensionPossible = ExtensionPossible ∪ {S};
                }
            pour (S = {Σ})
                si ((p_M[{Σ}] + 1) ≤ z) {
                    Occ(MS) = Occ(MS) ∪ {(i, d_s, d_d, d_i)};
                    ExtensionPossible = ExtensionPossible ∪ {S};
                }
            pour (S ∈ C)
                si (s_i + l − d_d + d_i ∈ S) {
                                                 /* identité */
                    Occ(MS) = Occ(MS) ∪ {(i, d_s, d_d, d_i)};
                    ExtensionPossible = ExtensionPossible ∪ {S};
                }
                sinon si (d_s + d_d + d_i + 1 ≤ e) {
                                                 /* substitution ou délétion */
                    Occ(MS) = Occ(MS) ∪ {(i, d_s + 1, d_d, d_i)};
                    Occ(MS) = Occ(MS) ∪ {(i, d_s, d_d + 1, d_i)};
                    ExtensionPossible = ExtensionPossible ∪ {S};
                }
        }
        pour (S ∈ ExtensionPossible)
            si (Occ(MS) vérifie le quorum q)
                Explore(MS, l + | S |, Occ(MS));
    }
}
```

Figure 5.29: Algorithme LeCombi (couverture combinatoire pondérée restreinte et distance de Levenshtein — ici simple comptage des erreurs) pour résoudre le Problème 1.

Nombre de jokers	0 erreur		1 substitution		1 erreur	
	k_{max}	temps (s)	k_{max}	temps (s)	k_{max}	temps (s)
0	4	0.076	6	1.03	7	2.10
1	5	0.19	7	4.47	8	9.95
2	6	0.37	9	12.51	9	29.94
3	7	0.68	10	28.34	10	70.94
4	8	0.94	11	56.78	11	145.98
5	9	1.36	12	101.69	12	274.72
6	10	1.86	13	171.31	13	491.43
7	11	2.49	14	275.70	15	1323.78
8	12	3.15	15	420.75	15	2065.43
9	13	3.96	16	617.61	16	3115.67
10	14	4.89	17	963.95	18	2769.30
11	15	5.95	18	1233.85	19	3939.74
12	16	7.17	19	1689.71	20	5524.03

Tableau 5.13: Comportement de LeCombi (longueur maximale des modèles trouvés et temps d'exécution en secondes) pour la couverture CCP donnée dans le texte, $q = 100\%$, $N = 20$, $k = k_{max}$, un nombre de jokers variant entre 0 et 12, et 0 erreur, 1 substitution et 1 erreur (substitution ou trou).

nombre de jokers autorisés. Les modèles recherchés étaient les plus longs présents dans toutes les chaînes d'un ensemble réduit (20 chaînes) du jeu d'essai composé des promoteurs de *Bacillus subtilis* utilisé dans la section 5.4.2.1.7. Enfin, nous avons autorisé 0 erreur, 1 substitution ou 1 erreur au plus. La couverture employée est dans tous les cas la suivante :

$$CCP = \left\{ \begin{array}{l} (\{X\}, \infty) \; \forall X \in \Sigma = \{A, C, G, T\}, \\ (\{A, C, G, T\}, J), \\ (S, 0) \; pour\ tout\ autre\ ensemble \end{array} \right\}$$

avec J (nombre de jokers) variant entre 0 et 12.

Le tableau 5.13 indique les temps d'exécution obtenus sur une Sparc Station 20, ainsi que les valeurs des longueurs maximales des modèles. La figure 5.30 présente les courbes correspondantes, montrant l'évolution du temps d'exécution en fonction du nombre de jokers. Ces courbes ne sont pas linéaires mais il faut observer qu'ici la longueur maximale varie avec le nombre de jokers autorisés. Nous n'avons donc pas le même comportement avec LeCombi qu'avec H/LeMoivre ou H/LePoivre (voir section 5.4.2.1.6.2) pour lesquels la longueur maximale des modèles décroissait assez rapidement au début mais devenait vite ensuite pratiquement constante.

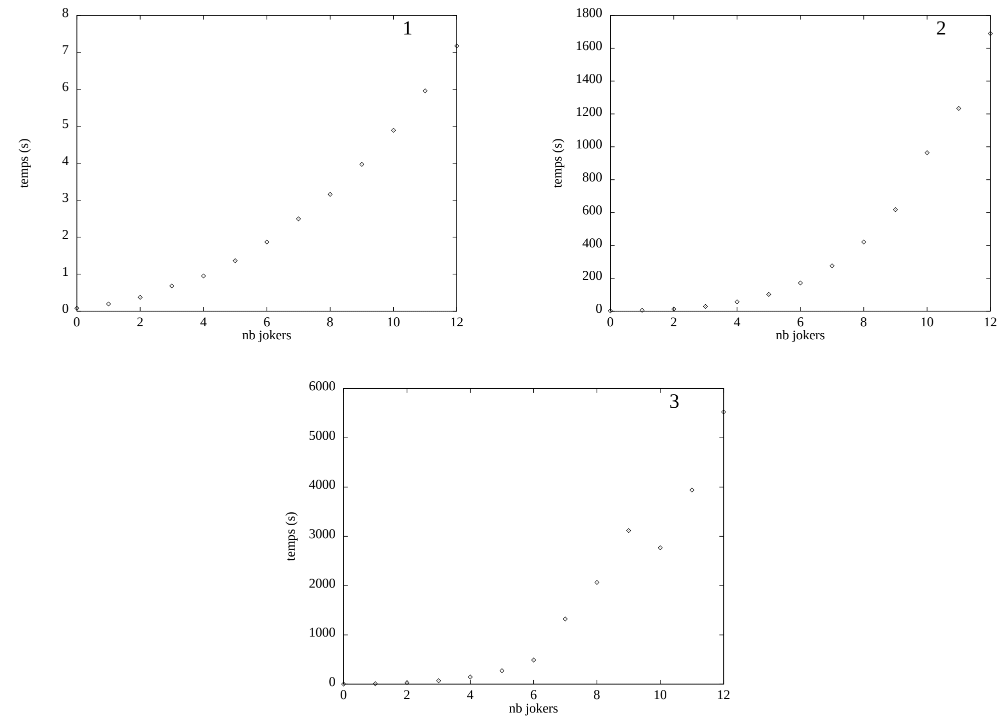

Figure 5.30: Performance de LeCombi : temps (en secondes) $\times$ nombre de jokers pour la couverture CCP donnée dans le texte, $q = 100\%$, $N = 20$, $k = k_{max}$ et 0 erreur, 1 substitution et 1 erreur (substitution ou trou) (courbes 1, 2 et 3 respectivement).

5.4.3 L'algorithme utilisant des mots comme unités de comparaison

5.4.3.1 Introduction

Le dernier algorithme que nous présentons travaille avec la définition de ressemblance 3.5.15. Il a été introduit dans [Sagot *et al.*, 1995c] et emploie une notion de similarité entre mots dans les chaînes et modèles (qui, dans ce cas, sont eux-mêmes des mots) qui repose sur les sous-mots de ces deux objets. Nous avons vu dans la section 3.2.4.1.1 du chapitre 3 que cette mesure peut souvent être plus flexible que celle basée sur la distance de Levenshtein utilisée par LePoivre (section 5.4.2.1.4).

L'algorithme est présenté progressivement. Nous commençons par la version sans trou, puis nous introduisons une version simplifiée de l'algorithme avec trous. Dans la définition 3.5.16, cette version consiste à ne permettre qu'au plus un seul trou par sous-mot de longueur w des modèles. Nous indiquons finalement comment peut être traité le cas où plus d'un trou est autorisé par sous-mot.

Nous allons voir que, dans tous les cas, l'algorithme construit d'abord les modèles de longueur w (ses unités de comparaison) ainsi que leurs ensembles d'occurrences. Dans un premier temps, nous n'allons pas nous soucier de comment ni où ces objets sont stockés, et allons donc simplement supposer qu'ils sont maintenus dans une table indexée. Nous montrerons ensuite quelles autres structures il est possible d'utiliser.

5.4.3.2 Version sans trou

5.4.3.2.1 Définition de récurrence

Soient s une chaîne de longueur n, $\mathcal{M}$ une matrice (que nous supposerons de similarité), $m = m_1...m_k$ un modèle, $i \in \{1,...,n- \mid m \mid +1\}$ une position dans s et $u = u_1...u_k$ le mot de longueur k commençant à la position i dans s. Alors le lemme suivant est vérifié :

Lemme 5.4.7 *Pour tout* $(k-1) \geq w$*:*

$$(i, u = u_1...u_k) \in K(m = m_1...m_k)$$

$$\Longleftrightarrow$$

$$(i, u_1...u_{k-1}) \in K(m_1...m_{k-1}) \; et$$

$$l_w(u, m) \geq t.$$

Rappelons que $l_w(u, m)$ indique le score des derniers sous-mots de longueur w de u et m. Ce lemme nous fournit une propriété constructive permettant d'obtenir $K(m_1...m_{k-1}m_k)$ à partir de $K(m_1...m_{k-1})$ pour tout $(k-1) \geq w$. Nous allons faire l'hypothèse pour l'instant que tous les modèles de longueur w qui sont w-présents dans s ont déjà été trouvés et que nous connaissons leurs ensembles d'occurrences. Nous discutons plus loin comment déterminer ces objets (section 5.4.3.2.2). Avant d'indiquer la propriété, observons que la valeur de $l_w(u, m)$ peut être calculée en $O(1)$ opérations puisque :

$$l_w(u, m) = l_w(u_1...u_{k-1}, m_1...m_{k-1}) - \mathcal{M}(u_{k-w}, m_{k-w}) + \mathcal{M}(u_k, m_k)$$

sachant que :

$$l_w(u_1...u_{k-1}, m_1...m_{k-1}) = score_{\mathcal{M}}(u_1...u_{k-1}, m_1...m_{k-1}) \; si \; (k-1) = w.$$

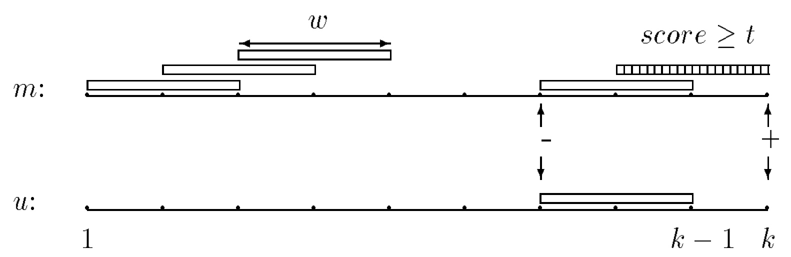

Figure 5.31: Extension d'un modèle.

Proposition 5.4.4 *Étant donné un ensemble d'occurrences* $K(m = m_1...m_{k-1})$ *avec* $(k-1) \geq w$*, nous avons :*

$$K(m = m_1...m_k) \\ = \\ \{(i, u = u_1...u_k) \mid (i, u_1...u_{k-1}) \in K(m_1...m_{k-1}) \text{ et } l_w(u, m) \geq t\}.$$

Si pour chaque élément (i, u) dans $K(m)$ nous conservons le score du dernier mot de longueur w de u et de m, alors nous pouvons construire les occurrences de n'importe quel modèle m connaissant uniquement les occurrences de ses préfixes de longueur $\mid m \mid$ - 1 (voir la figure 5.31).

Nous montrerons plus loin que si les valeurs de $\mathcal{M}(a, b)$ sont ordonnées pour tout $a \in \Sigma$, nous pouvons économiser un certain nombre d'opérations, à la fois au moment de la construction des modèles de longueur w et à celui de leur allongement.

5.4.3.2.2 Algorithme

L'idée de l'algorithme est donné dans la figure 5.33. Celui-ci s'appelle Klast (de Kmr + bLAST) et fonctionne en deux étapes distinctes : une étape d'initialisation et une étape d'extension. L'étape d'initialisation consiste à rechercher de manière efficace tous les modèles de longueur w qui sont w-présents dans au moins q des chaînes $s1, s2, ..., sN$. La construction de ces modèles n'a pas besoin d'être réalisée d'une manière complètement naïve, c'est-à-dire, nous n'avons pas besoin de vérifier pour tous les modèles possibles de longueur w s'ils ont des occurrences dans $s = s1s2...sN$. Une simple observation nous permet de limiter l'espace de recherche à cette étape. Celle-ci concerne le fait qu'un mot u de s ne peut être une w-occurrence d'un modèle m s'il existe $p \leq w$ tel que :

$$score_{\mathcal{M}}(u_1...u_p, m_1...m_p) + (w - p) * max(\mathcal{M}) < t$$

où $max(\mathcal{M})$ est la valeur maximale de la matrice de similarité $\mathcal{M}$.

Dans l'étape d'extension, nous considérons chaque modèle de longueur w, appelons m ce modèle, et nous essayons de l'allonger le plus loin possible, c'est-à-dire, tant que m a des occurrences dans au moins q des chaînes de l'ensemble. Cette seconde étape repose sur le lemme 5.4.7.

Soit $\Sigma = \{a, b, c, d, e\}$ et soit $\mathcal{M}$ la matrice suivante :

	a	b	c	d	e
a	5	3	0	2	1
b	3	4	1	2	4
c	0	1	5	4	2
d	2	2	4	4	1
e	1	4	2	1	4

Alors la matrice d'ordre $\mathcal{O}$ de $\mathcal{M}$ est la matrice donnée par :

	1	2	3	4	5
a	a	b	d	e	c
b	b	e	a	d	c
c	c	d	e	b	a
d	d	c	a	b	e
e	e	b	c	a	d

Figure 5.32: Exemple d'une matrice d'ordre.

Pendant les deux phases détaillées dans la figure 5.33, l'algorithme fait usage d'une matrice, appelée 'matrice d'ordre', qui est liée à $\mathcal{M}$ de la façon suivante :

Notation 5.4.6 *Soit une matrice $\mathcal{M}$, nous notons $\mathcal{O}$ la matrice d'ordre de $\mathcal{M}$ définie sur* $\mid \Sigma \mid \times \mid \Sigma \mid$ *par :*

$$\forall\ a \in \Sigma,\ \forall\ j \in [1..(\mid \Sigma \mid - 1)],\ \mathcal{M}(a, \mathcal{O}(a, j)) \leq \mathcal{M}(a, \mathcal{O}(a, j + 1)).$$

Chaque rang de la matrice $\mathcal{O}$ correspond ainsi à un symbole de Σ, et présente les symboles de Σ par scores, avec ce symbole, décroissants (voir la figure 5.32).

5.4.3.2.3 Complexité

Si les modèles de longueur w ainsi que les positions de leurs occurrences sont conservés dans une table indexée (temps d'accès en $O(1)$), la complexité en temps de l'algorithme est au pire en $O(n.N. \mid \Sigma \mid^{w})$ pour la première étape. Celle de la seconde est $O(n.N.N_{MaxBranches}.k.N_{Modi}(k-1))$ où $N_{MaxBranches} = \mid \Sigma \mid$. Dans le cas de Klast, $N_{Modi}(k-1)$ va en pratique dépendre fortement de plusieurs facteurs : la chaîne s, la matrice utilisée, le choix du seuil t et du quorum q. Le calcul de sa valeur théorique est une question qui demeure en ouvert. La complexité globale en temps est donc $O(n.N. \mid \Sigma \mid^{w}) + O(n.N. \mid \Sigma \mid .k.N_{Modi}(k-1))$. Une observation importante cependant est que $N_{Modi}(k-1)$ ne dépend pas directement de n, bien que la contribution de $N_{Modi}(k-1)$ à la complexité en temps de l'algorithme puisse être grande si t est fixé à une petite valeur et $N_{Modi}(k-1)$ peut avoir alors une relation avec $n.N$.

La complexité en espace de l'algorithme est $O(max\{n.N. \mid \Sigma \mid^{w}, n.N.N_{Modi}(k-1)\})$ si les modèles de longueur w ainsi que leurs occurrences sont stockées dans une table indexée.

```
/* Entrée */
        s = s1s2...sN : une chaîne de longueur n = concaténation de N chaînes
        M : une matrice de similarité
        O : matrice d'ordre de M, comme dans la notation 5.4.6
        t, w > 0, q : seuil, longueur des sous-mots et contrainte de quorum respectivement
        k : une longueur fixée
/* Sortie */
        Tous les modèles m de longueur k ayant des occurrences dans au moins q chaînes
/* Principales structures de données */
        K(m) = ensemble des occurrences d'un modèle m - contient pour chaque occurrence
        non seulement sa position i et l'image correspondante u mais aussi la valeur
        l_w(u, m) notée plus simplement l_w : implémenté comme une pile
        ExtensionPossible = ensemble des modèles qui vérifient le quorum q et peuvent
        donc être étendus : implémenté comme une pile
        T(m) = ensemble des occurrences du modèle m de longueur w : implémenté
        comme une pile
        E = ensemble des modèles de longueur w qui ont au moins une occurrence dans s
Main {
        pour (i ∈ [1..(n − w + 1)])
                Énumère (i, i − 1, λ, 0, O, T, E);
        pour (m ∈ E)
                si (T(m) ≠ ∅)
                        si (T(m) vérifie le quorum q)
                                Explore (m, w, T(m));
}

/* Étape d'initialisation */
/* d et f sont les positions de début et fin respectivement du sous-mot de longueur */
/* l de s, et m est le modèle en construction                                      */
Énumère (d, f, m, l, O, T, E) {
        si (l = w)                                  /* base de la récursion */
                si (score_M (s_d...s_f, m) ≥ t) {
                        T(m) = T(m) ∪ {(d, s_d...s_f);
                        E = E ∪ {m};
                }
        sinon {                                     /* pas de la récursion */
                si ((l > 0) et (score_M (s_d...s_f, m) + (max(M) * (w − l)) < t))
                        retourner;                  /* élagage */
                pour (j ∈ [1.. | Σ |]) {
                        a = O(s_{f+1}, j);
                        Énumère (d, f + 1, ma, l + 1, O, T, E);
                }
}
```

Figure 5.33 : Algorithme Klast (distance ayant pour base des mots, pas de trou) pour résoudre le Problème 1 — continue à la page suivante.

```
/* Étape d'extension */
Explore(m = m_1...m_l, l, K(m)) {
        si (l = k)                                  /* base de la recursion */
                stocke m et K(m);
        sinon {                                     /* pas de la recursion */
                pour ((i, u = s_i...s_{i+l-1}, l_w) ∈ K(m))
                        pour (j ∈ 1.. | Σ |) {
                                a = O(s_{i+l}, j);
                                l'_w = l_w - M(s_i, m_1) + M(s_{i+l}, a);
                                si (l'_w ≥ t) {
                                        K(ma) = K(ma) ∪ {(i,us_{i+l}, l'_w)};
                                        ExtensionPossible = ExtensionPossible ∪ {ma};
                                }
                                sinon
                                        sort de la boucle 'pour j';
                        }
                pour (ma ∈ ExtensionPossible)
                        si (K(ma) vérifie le quorum q)
                                Explore(ma, l + 1, K(ma));
        }
}
```

Figure 5.33: Algorithme Klast (distance ayant pour base des mots, pas de trous) pour résoudre le Problème 1.

5.4.3.3 Version avec trou

5.4.3.3.1 Esquisse de l'algorithme

Nous esquissons maintenant une extension de Klast qui autorise des trous (insertions et délétions) entre les modèles et leurs occurrences. Les notations sont celles de la définition 3.5.16. Dans l'implémentation actuelle, la distance entre deux trous ne peut être plus petite que w (c'est-à-dire, $e_s = 1$) et le score pour aligner un trou avec un symbole est une constante fixée, pour simplifier, à zéro. De plus, les trous ne sont pas autorisés dans le premier sous-mot. Un certain nombre de variables supplémentaires sont maintenant nécessaires. Pour chaque w-occurrence $(i, u_1...u_{k-1})$ d'un modèle $m_1...m_{k-1}$, une variable p_s indique ainsi la distance du dernier trou présent dans l'alignement du modèle et de l'occurrence et une variable p_t indique le nombre total de trous. Lorsqu'un modèle est étendu d'un symbole, p_s est incrémenté et un trou est autorisé entre le modèle m étendu et l'occurrence correspondante u également étendue seulement si $l_w(u, m) \geq t$, $p_s > w$ et $p_t < e_t$. L'idée de l'algorithme est donnée dans la figure 5.34.

Dans la section 5.4.3.2.1, nous avions indiqué que la valeur de $l(u = u_1...u_k, m = m_1...m_k)$ pouvait être calculée simplement à partir de celle de $l(u_1...u_{k-1}, m_1...m_{k-1})$ pour $(k-1) \geq w$ en $O(1)$ opérations. Ce calcul demeure constant ici, cependant la formule permettant de passer d'une valeur à l'autre fait maintenant intervenir le type de trou introduit en dernier lieu dans l'alignement de $u_1...u_{k-1}$ et $m_1...m_{k-1}$. Nous avons ainsi :

- si $p_s \geq w$:
  $l_w(u, m) = l_w(u_1...u_{k-1}, m_1...m_{k-1}) - \mathcal{M}(u_{k-w}, m_{k-w}) + \mathcal{M}(u_k, m_k)$;
- si $p_s < w$:
  - si le dernier trou introduit était une insertion :
    $l_w(u, m) = l_w(u_1...u_{k-1}, m_1...m_{k-1}) - \mathcal{M}(u_{k-w-1}, m_{k-w}) + \mathcal{M}(u_k, m_k)$;
  - si le dernier trou introduit était une délétion :
    $l_w(u, m) = l_w(u_1...u_{k-1}, m_1...m_{k-1}) - \mathcal{M}(u_{k-w+1}, m_{k-w}) + \mathcal{M}(u_k, m_k)$.

Au niveau de l'algorithme, il faut donc conserver une information de plus pour chaque occurrence, qui concerne justement le type du dernier trou introduit.

L'extension à plus d'un trou par sous-mot va exiger non plus deux valeurs indiquant respectivement la position du dernier trou (p_s) et son type, mais deux tableaux de bits (c'est-à-dire en fait toujours deux entiers) dont les indices i indiquent si une erreur (délétion ou insertion) a été introduite dans u face à la position m_{k-i+1} de m et son type (par exemple bit à 0 pour une délétion et à 1 pour une insertion).

5.4.3.3.2 Complexité

La complexité en temps de l'algorithme pour la première étape est au pire en $O(n.N. \mid \Sigma \mid^w)$ comme précédemment. Celle de la seconde est $O(n.N.N_{MaxBranches}.k.F_{MultiPos}.N_{Modi}(k-1))$ où $N_{MaxBranches}$ est égal à $(e_s+2) \mid \Sigma \mid$ et $F_{MultiPos}$ est égal à $(2e_t+1)$. La complexité globale est donc $O(n.N. \mid \Sigma \mid^w) + O(n.N.(e_s+2).(2e_t+1). \mid \Sigma \mid .k.N_{Modi}(k-1))$.

La complexité en espace de l'algorithme est $O(max\{n.N. \mid \Sigma \mid^w, n.N.(2e_t+1).N_{Modi}(k-1)\})$.

```
/* Entrée */
        e_t, e_s : nombre total de trous permis et distance minimale entre deux trous
        Le reste comme pour Klast
/* Sortie */
        Comme pour Klast
/* Principales structures de données */
        K(m) = ensemble des occurrences d'un modèle m - contient les mêmes informations
        que pour Klast avec de plus l'indication du nombre p_t de trous déjà introduits
        dans le mot, la distance p_s du dernier trou (initialement p_s = +∞), et une variable
        booléenne type indiquant le type de trou introduit (éventuellement) dans le dernier
        sous-mot
        Le reste comme pour Klast
/* Fonction principale : comme pour Klast */
/* Étape d'extension */
Explore(m = m_1...m_l, l, K(m)) {
        si (l = k)                                    /* base de la recursion */
              stocke m et K(m);
        sinon {                                       /* pas de la recursion */
              pour ((i, u = s_i...s_{i+l-1}, l_w, p_t, p_s) ∈ K(m))
                    pour (j ∈ 1.. | Σ |) {
                          p_s = p_s + 1;
                          a = O(s_{i+l}, j);
                          si (p_s > w)
                                l'_w = l_w - M(s_i, m_1) + M(s_{i+l}, a);
                          sinon si (type = insertion)
                                l'_w = l_w - M(s_{i-1}, m_1) + M(s_{i+l}, a);
                                sinon si (type = délétion)
                                      l'_w = l_w - M(s_{i+1}, m_1) + M(s_{i+l}, a);
                          si (l'_w ≥ t) {
                                K(ma) = K(ma) ∪ {(i,us_{i+l}, l'_w, p_t, 0)};
                                ExtensionPossible = ExtensionPossible ∪ {ma};
                          }
                          sinon si (p_t < e_t et p_s > w) {
                                K(ma) = K(ma) ∪ {(i,us_{i+l}, l'_w, p_t + 1, 0)};
                                ExtensionPossible = ExtensionPossible ∪ {ma};
                          }
                    }
              pour (ma ∈ ExtensionPossible)
                    si (K(ma) vérifie le quorum q)
                          Explore(ma, l + 1, K(ma));
        }
}
```

Figure 5.34: Algorithme GKlast (distance ayant pour base des mots, trous autorisés) pour résoudre le Problème 1.

5.4.3.4 Conservation des modèles initiaux

Dans ce qui précède, nous avons considéré que les modèles initiaux (ceux de longueur w) sont conservés dans une table indexée. D'autres structures de données sont bien sûr possibles. Nous avons ainsi réalisé une implémentation où ces modèles sont rangés dans un arbre AVL. Chaque nœud de l'arbre représente un entier codant pour le modèle et pointe sur une liste chaînée des positions où ce modèle est présent dans s. Comme pour la table indexée, la construction de l'arbre se fait en parcourant s et en plaçant la position i de chaque mot u de longueur w dans la liste de tous les modèles m de même longueur dont u est une occurrence. La recherche de m dans l'arbre prend à chaque fois dans le pire des cas un temps proportionnel au logarithme du nombre de modèles qui y ont été placés auparavant. L'algorithme est dans ce cas plus lent que la version utilisant une table, mais la structure en arbre a l'avantage de ne pas comporter plus de cellules qu'il n'y a de modèles de longueur w. Nous aurions pu également employer une table de hachage qui aurait représenté un compromis entre table indexée et arbre en termes d'espace en mémoire et rapidité de calculs. En pratique cependant, surtout lorsque nous souhaitons demeurer souples dans la définition de la similarité entre objets, presque tous les modèles de longueur w possèdent au moins une occurrence dans s, et ainsi pratiquement toutes les cellules de la table indexée se retrouvent remplies. C'est donc la structure que nous avons adopté en définitive.

5.4.3.5 Variantes : introduction de contraintes supplémentaires

5.4.3.5.1 Présence exacte (par sous-mots) des modèles au moins une fois

Comme nous l'avons mentionné, la complexité en temps de l'algorithme, quelle que soit la version, dépend du nombre maximal de modèles d'une certaine longueur qui se trouvent w-présents dans s. Ce nombre peut être réduit de manière importante si, en plus d'exiger qu'un modèle m soit w-présent au moins q fois dans s, nous imposons la contrainte supplémentaire que chaque sous-mot de longueur w de m soit <u>exactement</u> présent au moins une fois dans les chaînes (ce ne sera pas toujours nécessairement la même). Cela signifie que les modèles sont en fait construits à partir de sous-mots dans les chaînes.

Soit $EK(m)$ l'ensemble de w-occurrences de m dans le sens de la définition 3.5.15 de ressemblance. Alors :

$$EK(m) = \begin{cases} K(m) & si\ \forall j \in \{1, ..., \mid m \mid - w + 1\},\ \exists u \in K(m)\ tel\ que \\ & score_M(u_j...u_{j+|m|-w+1}, m_j...m_{j+|m|-w+1}) \geq t \\ & et\ u_j...u_{j+|m|-w+1} = m_j...m_{j+|m|-w+1}; \\ \emptyset & autrement. \end{cases}$$

Il est important d'observer qu'un modèle m peut ne jamais être lui-même exactement présent dans s, puisque la constrainte ne concerne que ses sous-mots de longueur w. La modification à apporter à l'algorithme afin de respecter cette contrainte de présence se fait de manière très similaire à celle indiquée pour la seconde variante de H/LePoivre (section 5.4.2.1.6.2).

5.4.3.5.2 Seuil sur la moyenne de tous les sous-mots

Il est possible d'assouplir enocre la contrainte imposée au score des sous-mots des modèles avec les sous-mots des mots dans les chaînes. Au lieu d'exiger que chacun de ces scores soit supérieur ou égal à une certaine valeur seuil t, nous pouvons définir qu'un mot u dans une chaîne est une occurrence d'un modèle m si et seulement si la moyenne des scores de chacun de ses sous-mots avec le sous-mot correspondant de m est supérieur ou égal à t. Plus formellement :

Lemme 5.4.8 *Pour tout* $(k-1) \geq w$*:*

$$(i, u_1...u_k) \in K(m_1...m_k)$$

$$\Longleftrightarrow$$

$$(i, u = u_1...u_{k-1}) \in K(m = m_1...m_{k-1}) \; et$$

$$\frac{\sum_{i=1}^{k} l_w(u_1...u_i + w - 1, m_1...m_{i+w-1})}{k - w + 1} = \frac{\sum_{i=1}^{k} score(u_i...u_i + w - 1, m_i...m_{i+w-1})}{k - w + 1} \geq t.$$

La modification à apporter à l'algorithme est simple : il suffit pour chaque modèle m de conserver une variable supplémentaire indiquant la moyenne obtenue jusqu'alors.

Observons que cette variante de l'algorithme initial est celle qui s'est révélée la plus utile en pratique ainsi que nous le verrons dans le chapitre consacré aux illustrations.

5.4.3.6 Performance en pratique

Nous présentons dans cette section les performances de Klast (original et sa variante travaillant avec un seuil établi sur la moyenne des scores des sous-mots — voir section 5.4.3.5.2) et de GKlast.

Comme nous l'avons mentionné, le temps d'exécution de l'algorithme dépend des données, de la matrice de similarité utilisée, de la valeur du seuil t et de celle du quorum q d'une façon que nous ne sommes pas capable d'établir de manière précise pour l'instant. Il est donc encore plus important dans ce cas d'évaluer le comportement de l'algorithme en pratique sur un 'vrai' ensemble test que pour les algorithmes présentés précédemment. Dans ce but, nous avons choisi comme jeu d'essai 80 protéines appartenant cette fois à la famille des facteurs d'élongation. Nous savons que les membres de cette famille possèdent plusieurs régions bien conservées (par exemple des sites de fixation de GTP). Par ailleurs, la longueur de ces protéines est assez homogène (environ 540 acides aminés).

Les figures 5.35 et 5.36 indiquent les courbes du temps d'exécution de Klast (courbes 1), Klast dans sa variante 'seuil sur la moyenne des sous-mots' (courbes 2) et GKlast (courbes 3) pour les valeurs suivantes de paramètres :

- matrice :

  Courbes a PAM250 normalisée (voir figure 3.7)

  Courbes b BLOSUM62 [Henikoff and Henikoff, 1992] (voir section 3.2.2.2)

- longueur des sous-mots : $w = 3$
- seuil : $t = 210$
- longueur des modèles recherchés : $k = 5$
- quorum : $q = 70\%$
- nombre de trous autorisés (pour GKlast) : $e_s = 1$ (1 trou au plus par sous-mot de longueur w)

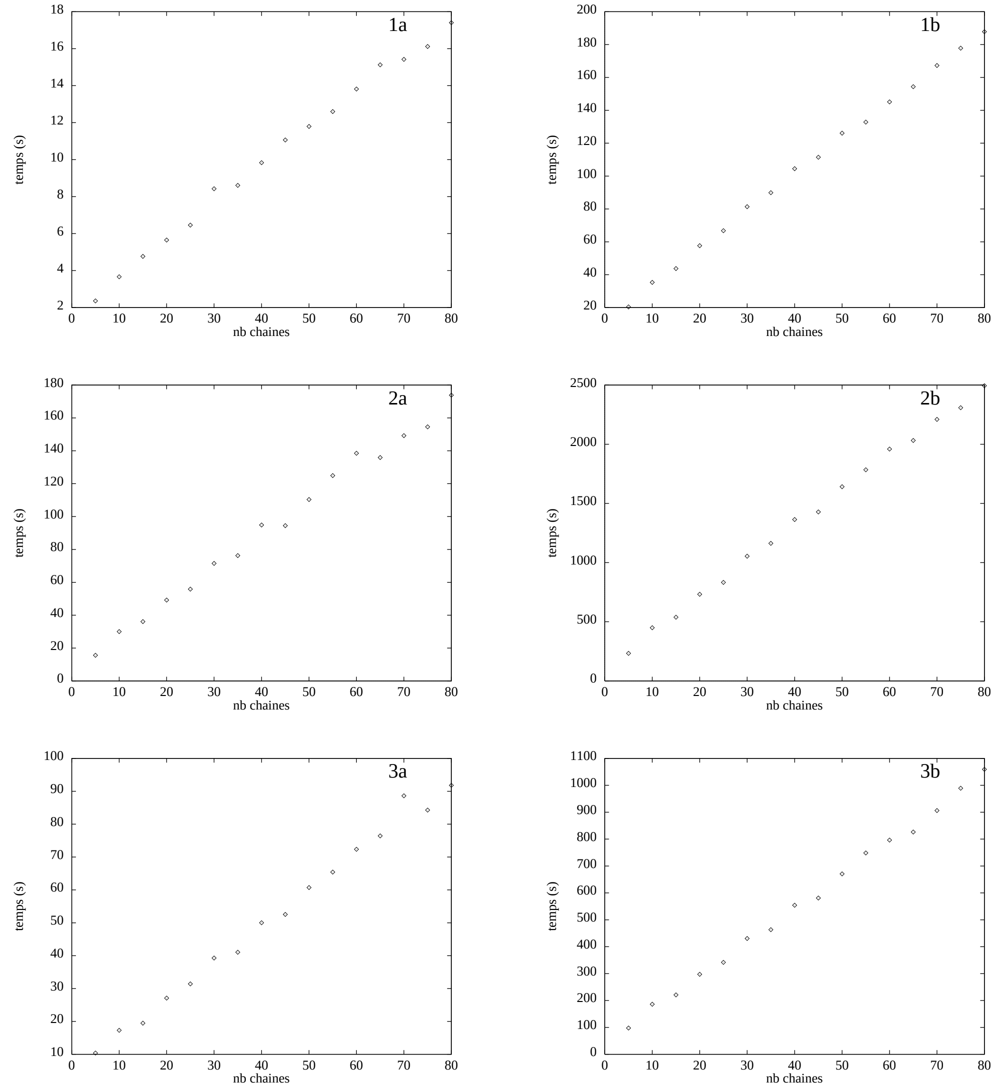

Figure 5.35: Performance de Klast (original et variante 5.4.3.5.2) et GKlast (courbes 1, 2 et 3 respectivement) : temps (en secondes) $\times$ nombre de chaînes pour $w = 3$, $t = 210$, $k = 5$, $q = 70\%$, $e_s = 1$ (GKlast) et la matrice PAM250 et BLOSUM62 (courbes a et b respectivement).

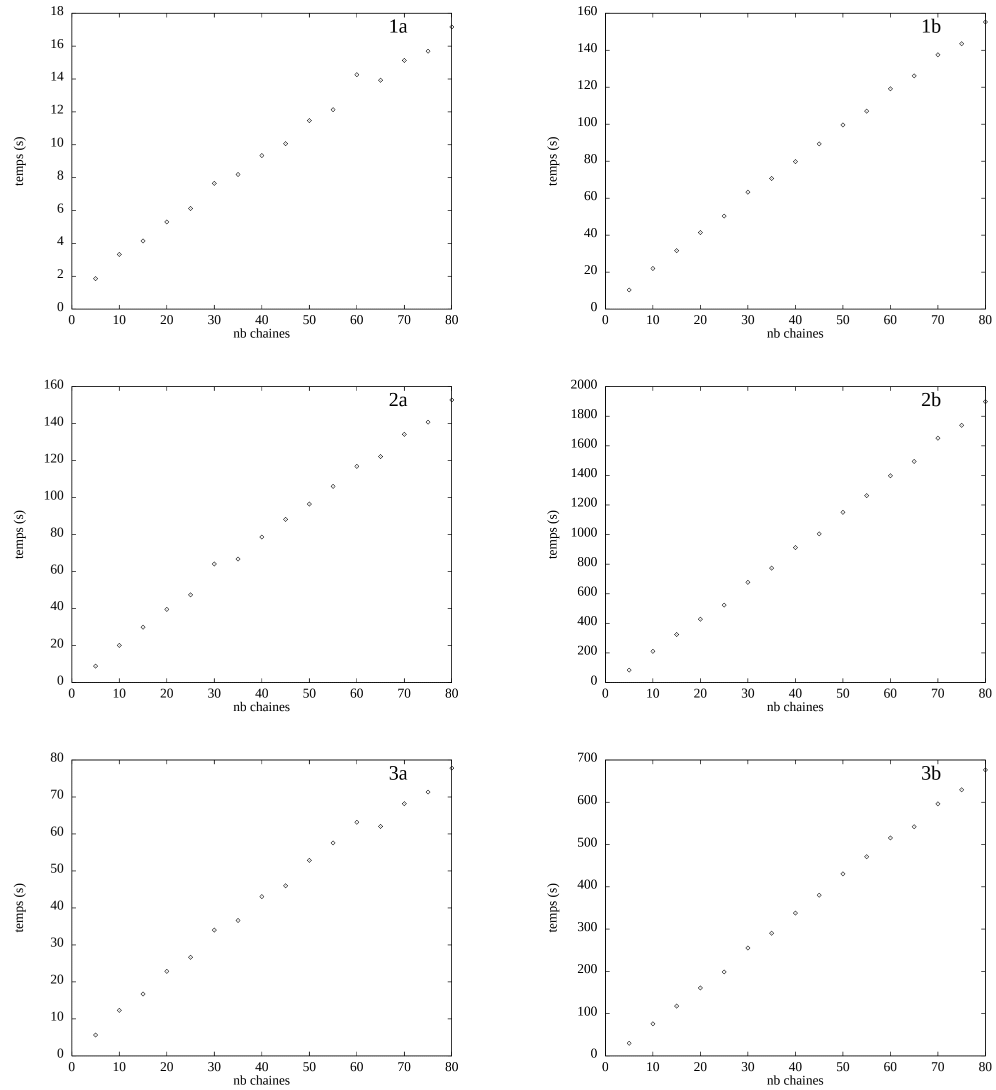

Figure 5.36: Performance de Klast (original et variante 5.4.3.5.2) et GKlast (courbes 1, 2 et 3 respectivement) : temps (en secondes) $\times$ nombre de chaînes pour $w = 3$, $t = 210$, $k = 5$, $q = 70\%$, $e_s = 1$ (GKlast) et la matrice PAM250 et BLOSUM62 (courbes a et b respectivement) lorsque la présence exacte des modèles (par sous-mots) est exigée au moins une fois.

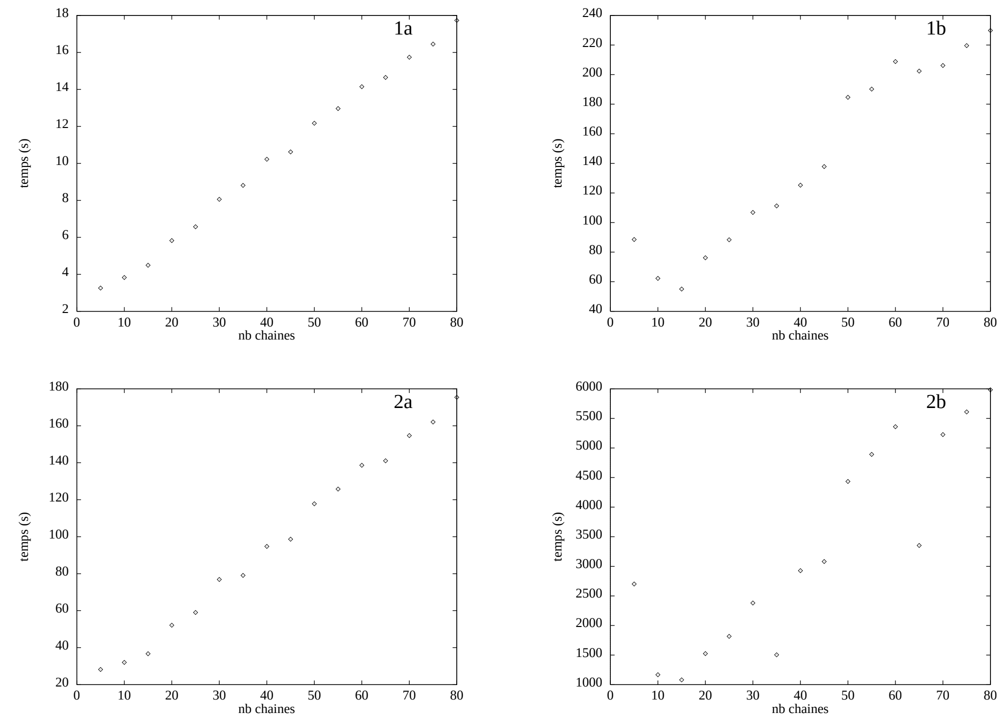

Figure 5.37: Performance de Klast original et GKlast (courbes 1, 2 et 3 respectivement) : temps (en secondes) $\times$ nombre de chaînes pour $w = 3$, $t = 210$, $k = k_{max}$, $q = 70\%$, $e_s = 1$ (GKlast) et la matrice PAM250 et BLOSUM62 (courbes a et b respectivement).

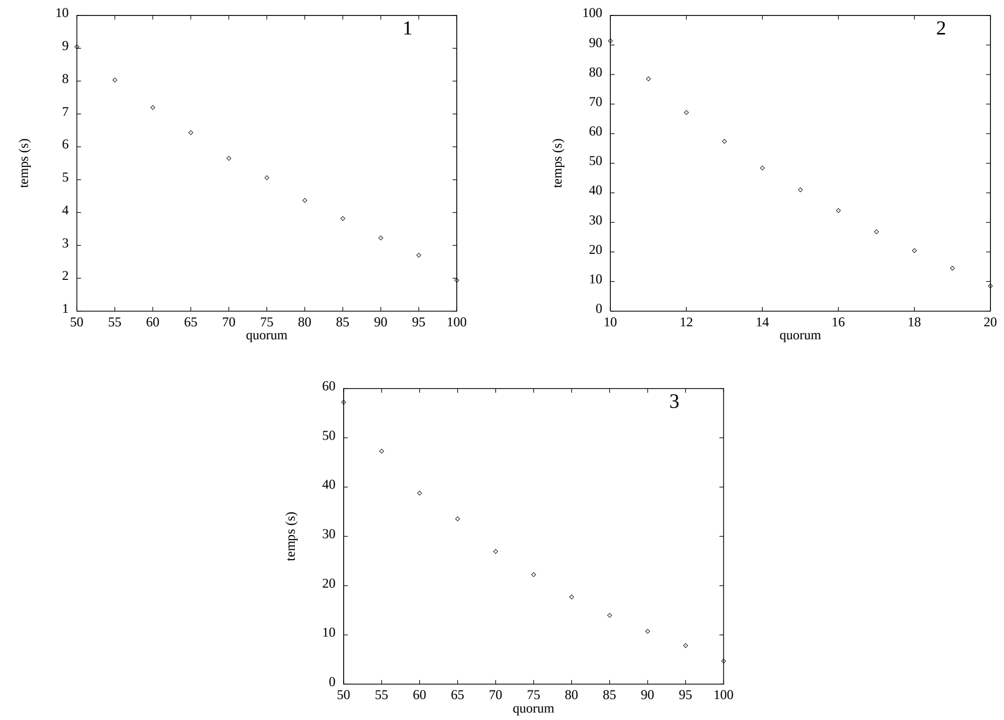

Figure 5.38: Performance de Klast (original et variante 5.4.3.5.2) et GKlast (courbes 1, 2 et 3 respectivement) : temps (en secondes) $\times$ quorum pour $w = 3$, $t = 210$, $k = 5$, $N = 20$, $e_s = 1$ (GKlast) et la matrice PAM250.

et lorsque le nombre de chaînes varie entre 5 et 80 avec un pas de 5. Dans la figure 5.36, nous exigeons de plus la présence exacte du modèle (par sous-mots) au moins une fois dans les chaînes de l'ensemble (variante de la section 5.4.3.5.1).

Nous pouvons déjà observer sur ces premières courbes que :

- quelle que soit la version de l'algorithme employé, sa complexité est bien en pratique linéaire en N, et donc en la longueur totale des chaînes puisque celles-ci possèdent approximativement le même nombre de résidus;
- le temps d'exécution est d'autant plus long que la matrice de similarité est plus dégénérée pour une même valeur de seuil (dans ce cas, la matrice la plus 'molle' est BLOSUM62);
- exiger la présence exacte du modèle (par sous-mots) au moins une fois dans les chaînes permet d'accélérer le temps de construction des modèles, mais le gain obtenu est relativement faible (environ 20 à 25%);
- comme nous pouvions nous y attendre GKlast est plus lent que Klast;
- la moins rapide des trois versions est celle qui travaille avec un seuil sur la moyenne des sous-mots (variante 5.4.3.5.2).

L'explication de ce dernier point est qu'un plus grand nombre de modèles est produit lorsque le seuil est établi sur la moyenne des scores de tous les sous-mots plutôt que sur celui du dernier (jusqu'à 11 fois plus de modèles peuvent être obtenus dans nos exemples). L'algorithme doit donc manipuler une masse plus importante d'objets durant toute son exécution.

Lorsque les modèles de longueur maximale sont recherchés, ceux-ci sont également plus longs, aussi bien pour cette variante de Klast que pour GKlast. Dans le cas de la variante, le rapport des longueurs par rapport à la version originale est en moyenne de 2,5 pour PAM250. Le temps d'exécution est quant à lui environ 44 fois plus grand. Ces rapports sont de 2 et 34 pour GKlast relativement à Klast. La figure 5.37 donne les courbes du temps d'exécution de Klast (courbes 1) et GKlast (courbes 2) en fonction des mêmes valeurs de paramètres.

Enfin, la figure 5.38 indique les courbes du temps d'exécution de Klast (courbe 1), Klast dans sa variante 'seuil sur la moyenne des sous-mots' (courbe 2) et GKlast (courbe 3) en fonction, cette fois, du quorum et pour les valeurs suivantes de paramètres :

- matrice : PAM250 normalisée (voir figure 3.7)
- longueur des sous-mots : $w = 3$
- seuil : $t = 210$
- longueur des modèles recherchés : $k = 5$
- nombre de chaînes : $N = 20$
- nombre de trous autorisés (pour GKlast) : $e_s = 1$ (1 trou au plus par sous-mot de longueur w)

Le quorum quant à lui varie entre 50 et 100 avec un pas de 5. La présence exacte du modèle au moins une fois dans les chaînes n'est pas exigée.

5.5 Résumé des différentes versions

Un résumé de toutes les définitions de similarité entre des mots et des algorithmes permettant de résoudre le problème général énoncé dans la section 5.1 est donné dans le tableau 5.14.

Modèle	Relation entre modèle et occurrences	Algorithme	Lemme	Complexité en temps
Σ^k	I_k	KMR Moivre	 # 5.4.1	$O(n.N.\log k)$
Σ^k	$distance_{EH} \leq e$	HMoivre	# 5.4.2	$O(n.N.k^{e+1}.\mid \Sigma \mid^{e+1})$
Σ^k	$distance_{EL} \leq e$	LeMoivre	# 5.4.3 et 5.4.4	$O(n.N.(e+2).(2e+1).k^{e+1}.\mid \Sigma \mid^{e+1})$
Σ^k	$score_{\mathcal{M}} \geq t$	Klast	# 5.4.7	$O(n.N.\mid \Sigma \mid^{w} +$ $n.N.\mid \Sigma \mid .N_{Modi}(k))$
Σ^k	$score_{\mathcal{M}} \geq t$ $p_s \leq e_s$ $p_t \leq e_t$	GKlast		$O(n.N.\mid \Sigma \mid^{w} +$ $n.N.(e_s+2).(2e_t+1).\mid \Sigma \mid .N_{Modi}(k))$
C^k	R_k $\in$	KMRC et KMRS Poivre	 # 5.4.3 et 5.4.4	$O(n.N.\log k.g^{2k}.k)$ (KMRC) $O(n.N.k.g^k)$ (Poivre)
C^k	$distance_{EH} \leq e$	HPoivre	# 5.4.3 et 5.4.4	$O(n.N.k^{e+1}.p^{e+1}.g^k)$
C^k	$distance_{EL} \leq e$	LePoivre Modification 3	# 5.4.3 et 5.4.4	$O(n.N.(e+2).(2e+1).k^{e+1}.p^{e+1}.g^k)$ $O(n.N.(e+1).(e+2).k^{e+1}.p^{e+1}.g^k)$
CCP^k	$\in$	Combi	# 5.4.5	$O(n.N.k'_{max}.\log k'_{max}.2^{k'_{max}}+$ $m.k'_{max}.\log k'_{max}.p)$ ou $O(n.N.k.2^k + m.k.p)$
CCP^k	$distance_{EL} \leq e$	LeCombi	# 5.4.6	$O(n.N.(e+1).(e+2).k^{e+1}.p^{e+1}.2^k)$ $O(n.N.(e+1).(e+2).k^{e+1}.p^{e+1}.g^k)$

Tableau 5.14: Résumé de toutes les définitions de similarité et des algorithmes présentés ou rappelés dans ce chapitre (KMR/KMRC et KMRS).