UNIVERSITE BRETAGNE LOIRE

université angers

Thèse de Doctorat

Vincent VIGNERON

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'Université d'Angers
Label européen
sous le sceau de l'Université Bretagne Loire

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Informatique et applications, section CNU 27
Unité de recherche : Laboratoire d'étude et d'informatique d'Angers (LERIA)

Soutenue le 8 décembdre 2017

Programmation par contraintes et découverte de motifs sur données séquentielles

JURY

Rapporteurs :	**M. Lakhdar SAÏS**, Professeur des universités, Université d'Artois
	M. Patrice BOIZUMAULT, Professeur des universités, Université de Caen Normandie
Examinateurs :	**M^me^ Béatrice DUVAL**, Professeur des universités, Université d'Angers
	M^me^ Christelle VRAIN, Professeur des universités, Université d'Orléans
Directeur de thèse :	**M. David LESAINT**, Professeur des universités, Université d'Angers

Remerciements

Je tiens en premier lieu à remercier mon directeur de thèse, David Lesaint, pour son encadrement qu'il s'agisse de conseils pour les articles à lire, des propositions pour les directions prises au cours de ma thèse, des relectures approfondies de ma thèse afin d'en affiner la terminologie, ou de l'opportunité que j'ai eu d'échanger avec le laboratoire Insight Institute of Data Analysis de Cork dans le cadre du projet Ulysse.

Je tiens également à remercier mes examinateurs pour avoir accepté de relire ma thèse.

Je tiens également à remercier le personnel du LERIA (Laboratoire d'Étude et de Recherche en Informatique d'Angers) et du département informatique pour m'avoir accueilli. Notamment, les secrétaires pour leur sympathie et leur aide dans les tâches administratives. Je tiens également à remercier les collègues qui m'ont aidé pour mes enseignements et avec qui j'ai pu avoir des échanges conviviaux le midi.

Je tiens également à remercier ma famille pour avoir tenté de comprendre mon sujet de thèse en vain - il faut l'avouer je ne leur ai pas vraiment facilité la tâche. Je tiens également à remercier Maheva relectures pointilleuses et très poussées de ma thèse. Je la remercie aussi pour son soutien moral pour les moments difficiles de ma thèse, en particulier pour la période de fin de rédaction.

Table des matières

1 Introduction **7**

1.1 Introduction 7

1.2 Problème de satisfaction de contraintes et programmation par contraintes . . . 8

1.2.1 Problème de Satisfaction de Contraintes 8

1.2.2 Méthodes de résolution 10

1.3 Fouille de données et apprentissage artificiel 24

1.3.1 Recherche d'itemsets 24

1.3.2 Recherche de motifs séquentiels 27

1.3.3 Comparaison des méthodes 35

1.3.4 Algorithmes sur les textes 35

1.4 Unification des problèmes de fouilles 37

2 Maximal Matrix Problem **39**

2.1 Notations 40

2.2 Calcul de matrices maximales 40

2.2.1 Extension incrémentale en ligne 43

2.2.2 Prédicat de base de données 45

2.2.3 Type matriciel et prédicats pour fouille d'itemsets 46

2.2.4 Type matriciel et prédicat pour recherche de motifs sans répétitions . . 47

2.2.5 Type matriciel et prédicat pour recherche de motifs avec répétitions . . 48

2.2.6 Type matriciel pour recherche de motifs à base de patrons 49

2.2.7 Caractérisation et prédicat de couverture minimale 49

2.2.8 Illustration 50

2.2.9 Typologie des prédicats matriciels 50

2.3 Le langage MMP 51

2.4 Modélisation QCSP pour structure matricielle 56

2.5 Complexité 58

2.6 Matrices maximales n-aires : définitions et prédicats 62

2.6.1 Prédicats d'ordre binaire 63
2.6.2 Structure matricielle pour la recherche d'ordre partiel fréquent 64

3 Résolution de MMP **67**
3.1 Introduction 67
3.2 Ordre Total 68
3.2.1 Sous-séquences fréquentes sans répétition de caractères 68
3.2.2 Sous-séquence fréquente avec répétition 73
3.2.3 Sous-séquence fréquente avec répétition et semi-fermée 77
3.2.4 Sous-séquence fréquente et discriminante 77
3.2.5 Contrainte globale dédiée à l'exclusion 78
3.3 Ordre Partiel 81
3.3.1 Contrainte globale dédiée à l'exclusion 86
3.3.2 Expérimentations 93

4 Méthode approchée pour le calcul de motifs partiellement ordonnés **101**
4.1 Introduction 101
4.2 Algorithmes génétiques et mémétiques 102
4.3 Calcul de motifs partiellement ordonnés maximaux pour l'exclusion 103
4.4 Approche mémétique 104
4.4.1 Représentation des individus 104
4.4.2 Algorithme général 105
4.4.3 Expérimentations 108

5 Deux algorithmes dédiés à l'extraction de motifs séquentiels **113**
5.1 Introduction 113
5.2 Calcul de patrons maximaux 114
5.3 Factorisation de blocs maximaux par parcours de treillis 119
5.3.1 Algorithme APriori 120
5.3.2 Adaptation d'APriori pour le calcul d'itemsets maximaux 122
5.3.3 Calcul des blocs maximaux pour l'enracinement 126
5.4 Approche dynamique 136
5.4.1 Algorithme 137
5.4.2 Expérimentations 144

Conclusion générale **151**

1

Introduction

1.1 Introduction

La fouille de données et l'apprentissage artificiel sont des champs de l'informatique qui visent à extraire de l'information à partir d'un ensemble de données. À ce titre, la fouille de données sert davantage à identifier de l'information - comme des motifs récurrents - dans un ensemble de données. D'une part, l'exemple canon de la fouille de données est l'analyse du panier d'une ménagère dans le cadre de l'étude des tickets de caisses d'une enseigne, afin d'identifier des ensembles de produits achetés simultanément. D'autre part, l'apprentissage artificiel vise à apprendre des règles qui seront appliquées dans la classification de nouvelles données, ce qui recouvre la classification supervisée et non-supervisée. La première cherche à discriminer, à l'aide de motifs ou par d'autres moyens à l'instar des arbres de décision, plusieurs ensembles de données disjoints. Nous citerons notamment le cas de la différenciation d'espèces animales à partir de photos. La classification non-supervisée, quant à elle, s'emploie à découvrir des groupes clairement identifiables au sein d'un jeu de données non-classées, permettant, par exemple, de regrouper des villes en fonction de leur situation géographique ou en fonction de leurs secteurs d'activités (tourisme, industrie, commerce,...).

Ce chapitre introduit des problèmes classiques de fouille de données et des problèmes connexes de classification. Nous présentons, pour chaque problème, des algorithmes dédiés ainsi que des méthodes de résolution basées sur une modélisation à haut niveau d'abstraction

(programmation par contraintes ou SAT). Les approches PPC et SAT sont récentes et proposent une alternative intéressante aux approches ad-hocs. En effet, elles permettent d'adapter facilement le modèle du problème à résoudre aux besoins utilisateurs sans changer les algorithmes sous-jacents. Par ailleurs, elles intègrent aisément des critères d'optimisation. Enfin, elles bénéficient de mécanismes de résolution efficaces dépendant des paradigmes de modélisation. Cependant, ces modélisations de haut niveau s'avèrent moins efficaces que les algorithmes dédiés. Cette thèse suit donc cette approche générique pour calculer des motifs discriminants. Nous établirons un cadre formel basé sur une modélisation avec des contraintes, appelé (MMP), pour représenter certains problèmes de fouille de données.

Ainsi, nous présenterons un état de l'art des deux domaines associés au sujet de cette thèse : la programmation par contraintes et la fouille de données. Nous décrirons la première d'un point de vue généraliste, tandis que la présentation de la seconde se concentrera sur les problèmes de recherche d'itemsets et de sous-séquences fréquents. De fait, les sujets traités couvrent principalement la recherche de motifs dans des données séquentielles, par exemple la recherche de sous-séquences ou sous-chaînes communes. Étant donné notre objectif qui consiste à modéliser des problèmes de fouille de données avec des méthodes de (PPC) nous fournirons une liste non-exhaustive de travaux déjà réalisés à propos de ceux-ci. La programmation par contraintes emploie des mécanismes appelés contraintes globales, dont certains seront décrits plus avant afin d'aider à la compréhension.

Cet état de l'art n'abordera pas l'emploi de la fouille de données pour aider le processus de résolution de la programmation par contraintes même si des travaux ont étudié le sujet.

1.2 Problème de satisfaction de contraintes et programmation par contraintes

La programmation par contraintes est un paradigme de programmation qui repose sur la modélisation de problèmes de satisfaction de contraintes (CSP) ([37] [45]) et leur résolution à l'aide de méthodes dédiées (recherche arborescente, propagation, etc.). Nous présentons d'abord le cadre formel des problèmes de satisfaction de contraintes, puis les principales méthodes de résolution utilisées en PPC.

1.2.1 Problème de Satisfaction de Contraintes

Un CSP est un triplet composé d'un ensemble de variables, d'un ensemble de domaines de valeurs pour les variables et d'un ensemble de contraintes limitant chacune les affectations de valeur possibles pour un sous-ensemble de variables.

Définition 1 (Problème de satisfaction de contraintes : CSP)
Un CSP est un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ tel que :

- $\mathcal{X} = \{x_1, \ldots, x_n\}$ est un ensemble de variables ;
- $\mathcal{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ est un ensemble de domaines finis tel que $x_k \in \mathcal{D}_k$ ($k \in [\![n]\!]$ [1]) ;
- $\mathcal{C} = \{C_1, \ldots, C_m\}$ est un ensemble de contraintes. Pour $i \in [\![m]\!]$, C_i est une paire $(\mathcal{X}_i, \mathcal{R}_i)$ telle que $\mathcal{X}_i = (x_{i_1}, \ldots, x_{i_k}) \in \mathcal{X}^k$ et $\mathcal{R}_i \subseteq \mathcal{D}_{i_1} \times \cdots \times \mathcal{D}_{i_k}$. k est appelée *portée* de C_i et $\mathcal{R}_i$ *relation* de C_i.

Le cadre CSP n'impose aucune restriction sur la nature des domaines (booléens, entiers, etc.) et des langages de contraintes associés (relations propositionnelles, arithmétiques, etc.). Selon le domaine, les contraintes peuvent s'exprimer soit en intension (formule propositionnelle, etc.), soit en extension (ensemble des n-uplets de valeurs valides).

Une solution d'un CSP est une affectation des variables qui est consistante avec les contraintes. Une affectation est un choix de valeurs pour un sous-ensemble des variables du CSP.

Définition 2 (Affectation)
Une affectation $\mathcal{A}$ pour un CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ est une paire $(\mathcal{X}_\mathcal{A}, \mathcal{V}_\mathcal{A})$ telle que $\{x_{\mathcal{A}_1}, \cdots, x_{\mathcal{A}_1}\} \subseteq \mathcal{X}$ et $\mathcal{V}_\mathcal{A} \in \mathcal{D}_{\mathcal{A}_1} \times \cdots \times \mathcal{D}_{\mathcal{A}_k}$. $\mathcal{V}_\mathcal{A}$ est le k-uplet de valeurs affectées aux variables de $\mathcal{X}_\mathcal{A}$.

Une affectation est dite totale si elle porte sur l'intégralité des variables du CSP, sinon elle est dite partielle.

Définition 3 (Affectation totale et partielle)
Une affectation $\mathcal{A}$ pour un CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ est totale si $\mathcal{X}_\mathcal{A} = \mathcal{X}$.
Une affectation $\mathcal{A}$ pour un CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ est partielle si $\mathcal{X}_\mathcal{A} \subset \mathcal{X}$.

Une affectation est consistante avec une contrainte si sa projection sur les variables de la contrainte appartient à la projection de la relation associée à la contrainte sur les variables de l'affectation.

Définition 4 (Affectation consistante)
Une affection $\mathcal{A} = (\mathcal{X}_\mathcal{A}, \mathcal{V}_\mathcal{A})$ d'un CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ est consistante avec une contrainte $C_i = (\mathcal{X}_i, \mathcal{R}_i) \in \mathcal{C}$ telle que $\mathcal{X}_i \subseteq \mathcal{X}_\mathcal{A}$ si et seulement si $\mathcal{X}_i = \mathcal{X}_\mathcal{A}$ et $\mathcal{V}_{\mathcal{A}_{|\mathcal{X}_i}} \in \mathcal{R}_i$.

Nous dirons, également, qu'une affectation satisfait une contrainte si elle est consistante avec cette contrainte.

1. $[\![n]\!]$ dénote l'ensemble $\{1, 2, \cdots, n\}$.

Définition 5 (Solution d'un CSP)
Soient $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ un CSP et $\mathcal{A}$ une affectation totale pour P. $\mathcal{A}$ est solution de P si et seulement si $\mathcal{A}$ satisfait C_i pour tout $C_i \in \mathcal{C}$.

Exemple 1. Soit le CSP composé des variables $\mathcal{X} = \{x_1, x_2, x_3\}$, des domaines $\mathcal{D}_1 = \{1, 2\}$, $\mathcal{D}_2 = \{2, 3, 4\}$ et $\mathcal{D}_1 = \{4, 7\}$ et des contraintes $C_1 = ((x_1, x_2, x_3), \{(1, 3, 4), (2, 2, 4)\})$ et $C_2 = ((x_1, x_2, x_3), \{(1, 2, 4), (1, 2, 7), (1, 3, 4), (1, 3, 7), (1, 4, 7), (2, 3, 4), (2, 3, 7), (2, 4, 7)\})$. La contrainte C_2 correspond à une contrainte de différence deux à deux entre ses variables tandis que C_3 impose l'égalité entre x_3 et la somme de ses deux autres variables. La figure 1.1 présente une formulation de ce CSP en intension sous la forme d'un programme par contraintes écrit dans le langage MiniZinc [40] (une description de la syntaxe est fournie en Annexe A). $A_p = ((x_1), (1))$ est une affectation partielle du CSP. $A_t = ((x_1, x_2, x_3), (1, 3, 4))$ est une affectation totale solution, du CSP car le triplet $(1, 3, 4)$ satisfait les deux contraintes C_1 et C_2. A_t est la seule solution du CSP, par exemple $((x_1, x_2, x_3), (2, 2, 4))$ n'est pas solution car le triplet $(2, 2, 4)$ ne satisfait pas C_2.

```
1 set of int: D1 = {1,2};
2 set of int: D2 = {2,3,4};
3 set of int: D3 = {4,7};
4
5 var D1: x1;
6 var D2: x2;
7 var D3: x3;
8
9 constraint x1 + x2 = x3;
10 constraint x1 != x2;
```

FIGURE 1.1 – Modèle PPC associé à l'exemple 1.

△

1.2.2 Méthodes de résolution

Les méthodes de résolution de CSP sont nombreuse et mettent en œuvre différentes approches : recherche arborescente, heuristiques, propagation de contraintes, élimination de symétries, décomposition, etc ([57]). Nous présentons ici les principes de la recherche arborescente, les choix d'heuristiques et le filtrage de domaine par propagation de contraintes, en nous attardant en particulier sur les contraintes globales `all_different` et `regular`.

1.2.2.1 Recherche arborescente

L'ensemble des affectations totales d'un CSP détermine l'espace de recherche qu'ont à explorer les méthodes de résolutions procédant par calcul des solutions. Cet espace de recherche peut se structure sous la forme d'un arbre qui associe à chaque variable une profondeur donnée et qui fait correspondre à chaque nœud une affectation possible pour la variable du niveau considéré en associant à chaque nœud interne autant de fils qu'il existe d'affectations possibles pour la variable suivante. Toute branche menant de la racine à un nœud caractérise donc une affectation partielle du CSP, et dans le cas d'une feuille, une affectation totale. Du fait de l'ordonnancement des variables, de l'exhaustivité et de la non-redondance des choix de valeur pour chaque variable, nous garantissons alors la bijection entre feuilles de l'arbre et affectations totales du CSP. Dans ce cadre, la recherche arborescente par retour-arrière (backtracking [18]) est une méthode de parcours en profondeur d'abord qui consiste à reconsidérer l'affectation de variable la plus récente en cas d'inconsistance. Nous distinguons deux méthodes - *générer-et-tester* et *tester-et-générer* - selon que le test de consistance s'effectue uniquement sur les feuilles, ou bien sur chaque nœud interne.

L'algorithme 1 décrit la méthode *générer-et-tester*. et la figure 1.2 montre l'arbre de recherche développé pour résoudre le CSP de l'exemple 1.

La figure 1.3 présente les arbres de recherche développés par l'approche *tester-et-générer*. Un nœud vert dénote une solution d'un CSP, un nœud rouge dénote une affectation inconsis-

tante.

Algorithme 1 – `Générer et Tester`

Entrée : $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ un CSP
Sortie : $\mathcal{A}$ une affectation totale
Données : S pile de paires variable/valeur, i indice de variable, v valeur, n nombre de variables du CSP

```
1  begin
2      A ← () ;
3      foreach v ∈ D_1 do ajouter(S, (1,v));
4      while not vide(S) do
5          (i, v) ← retirer(S) ;
6          V_{A_i} ← v ;
7          if i = n ∧ consistant(A, P) then return V_A ;
8          else
9              i ← i + 1 ;
10             foreach v ∈ D_i do ajouter(S, (i,v));
11         end
12     end
13     return ∅ ;        ▷ L'ensemble vide indique l'absence d'affectation consistante
14 end
```

Nous constatons que la méthode *tester-et-générer* crée moins de nœuds que *générer-et-tester*, car elle permet de détecter les affectations partielles inconsistantes évitant ainsi le parcours inutile de sous-arbres. L'ajout d'un processus de filtrage permet de limiter encore plus efficacement le nombre de nœuds explorés en retirant des domaines des variables restant à instancier des valeurs qui sont incompatibles avec l'affectation courante, ce qui limite le nombre de nœuds générés.

1.2.2.2 Filtrage des domaines

Les méthodes de filtrage sont fondamentalement des méthodes de point fixe qui réduisent itérativement les domaines de variable jusqu'à obtenir une propriété voulue - communément appelée degré de consistance - ou la preuve de l'inconsistance du CSP.

Les degrés de consistance les plus répandus sont la consistance d'arcs [37, 38, 39] (Arc-Consistency ou (AC)) et la consistance d'arcs généralisée (Generalized-Arc-Consistency ou (GAC)). L'arc-consistance est définie sur des CSP à contraintes binaires tandis que l'arc-consistance-généralisée s'applique au cas général des CSP. Pour une contrainte donnée, la consistance généralisée d'arcs élimine des domaines variables de la contrainte les valeurs qui n'appartiennent à aucune affectation satisfaisant de la contrainte.

L'algorithme 2 présente la procédure de filtrage GAC3 pour la consistance généralisée

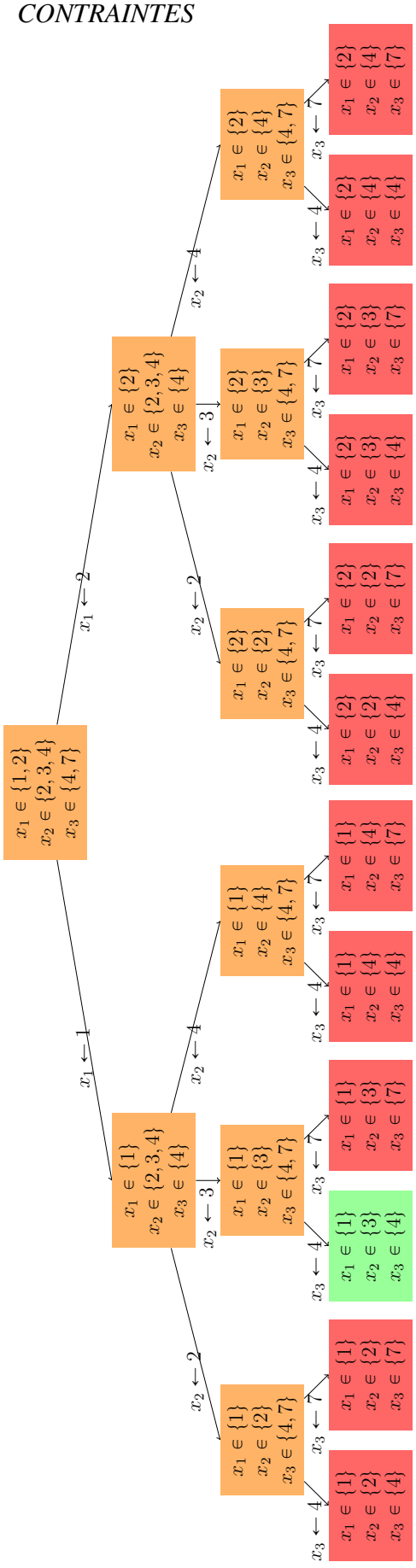

FIGURE 1.2 – Arbre développé par la méthode *générer-et-tester* pour le CSP de l'exemple 1

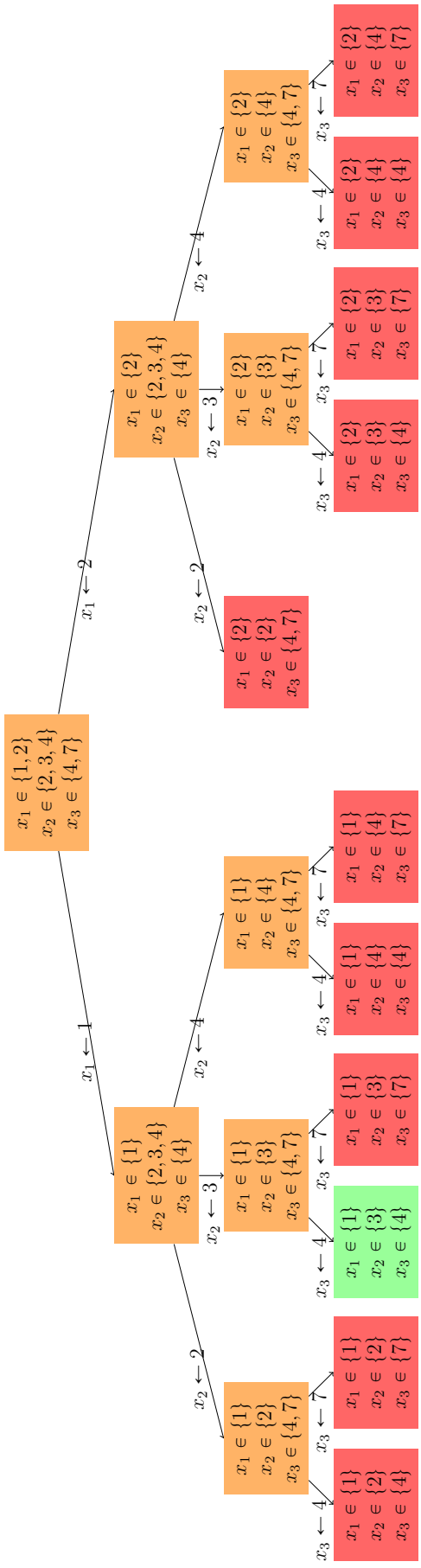

FIGURE 1.3 – Arbre développé par la méthode *tester-et-générer* pour le CSP de l'exemple 1

d'arcs [37]. Par construction, toute élimination de valeur préserve l'équivalence entre le CSP résultant et le CSP de départ. De plus, toute modification de domaine pour une variable entraîne une révision des contraintes dans lesquelles la variable est présente. Le processus est ainsi répété jusqu'à atteindre un point fixe. Le pendant de GAC3 pour l'arc-consistance est AC3 auxquelles ont été apportées de nombreuses améliorations (variantes AC4 [43, 44], AC6 [4] et AC2001 [61]).

Il existe d'autres degrés de consistance [] dont la consistance de bornes (Bounds Consistency ou (BC)) qui n'opère que sur les valeurs extrêmes des domaines dans le cas de domaines ordonnés. Dans le cas général, la consistance de bornes réduit de façon moindre le domaine des variables par rapport à la consistance d'arcs, même s'il existe des classes de contraintes pour lesquelles les deux consistances sont équivalentes (c'est notamment le cas pour les contraintes d'inégalité tel que $x < y$). La consistance de bornes est toutefois moins coûteuse en temps de calcul.

Exemple 2. Soit le CSP composé des variables $x_1 \in \{2\}$ et $x_2 \in \{2, 3\}$ et de la contrainte $x_1 \neq x_2$. La réduction de domaine par arc-consistance laisse inchangé le domaine de x_1, mais réduit le domaine de x_2 à $\{3\}$, car la contrainte ne sera jamais satisfaite si x_2 prend la valeur 2.

△

Les méthodes de filtrage peuvent être utilisées soit avant de mener une recherche de solutions soit en cours de recherche (nous parlerons de maintien de consistance). Dans le second cas, le principe consiste à déclencher la méthode de filtrage à chaque affectation de variable en intégrant l'élimination des valeurs non retenues pour la variable et en en propageant les effets sur les domaines des variables restantes. Lorsque le filtrage détecte l'inconsistance du CSP courant, il est alors inutile d'explorer le sous-arbre associé au nœud courant. Le nœud est dit en échec et la recherche se poursuit par retour-arrière.

Exemple 3. Nous illustrons ici le calcul de la fermeture arc-consistante du CSP présenté en exemple 1 par l'algorithme GAC3. Supposons que l'algorithme dépile d'abord la paire (x_3, $\mathcal{C}_2$). Le domaine de x_3 contient deux valeurs 4 et 7. Pour 4 nous trouvons une affectation qui vérifie la contrainte de somme avec $x_1 \leftarrow 2$ et $x_2 \leftarrow 2$. Par contre, aucune affectation consistante n'existe pour 7. Donc cette valeur est supprimée du domaine de x_3. Supposons que l'algorithme poursuive avec la paire (x_2, $\mathcal{C}_2$). Nous trouvons une affectation consistante pour les valeurs 2 et 3 en choisissant respectivement les valeurs 2 et 1 pour la variable x_1 (x_3 ne peut prendre que la valeur 4). Pour la valeur 4 aucune affectation n'est consistante, cette valeur est donc supprimée du domaine de x_2. En poursuivant l'exécution, l'algorithme ne procède à aucune réduction de domaine supplémentaire.

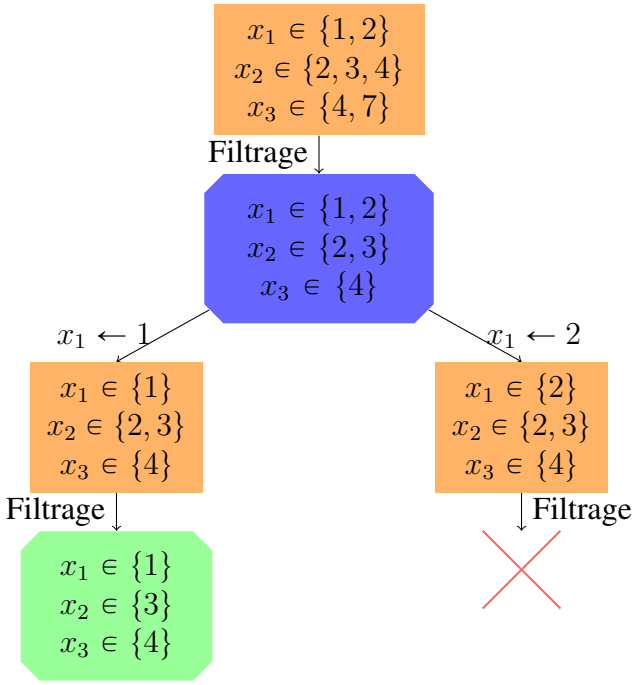

FIGURE 1.4 – Arbre de recherche développé par maintien d'arc-consistance sur le CSP de l'exemple 1.

La figure 1.4 montre l'arbre de recherche obtenu par maintien d'arc-consistance sur ce CSP. Le nœud bleu de la figure 1.4 correspond à la réduction de domaine effectuée au premier appel de GAC3 et décrite ci-dessus..

△

1.2.2.3 Heuristiques de branchement

La structure d'un arbre de recherche est conditionnée par l'ordre d'affectation choisi entre les variables et, pour chacune, l'ordre d'affectation choisi sur les valeurs de son domaine. Par défaut, ces choix sont invariants en cours de recherche et pré-établis à l'aide d'heuristiques (critères) statiques. Cependant, ces choix peuvent être réévalués en cours de recherche en fonction d'heuristique dynamique.

Parmi les heuristiques pour le choix de variables nous pourrons citer de manière non-exhaustive : l'heuristique de choix aléatoire, les heuristiques dépendant de la taille des domaines à un instant de la résolution, celles dépendant du nombre de contraintes associées à une variable, ou celles dépendant du nombre d'échecs dans lesquels la variable est impliquée, etc.

Parmi les heuristiques de choix de valeurs, nous citerons de la même manière des heuristiques de choix aléatoire, de choix de valeur minimale, maximale ou médiane.

Exemple 4. Reprenons le CSP présenté en exemple 1. L'exemple 3 propose une réduction

Algorithme 2 – GAC 3

Entrée : $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ un CSP
Sortie : La fermeture GAC de P si elle existe ou $false$ sinon

begin
  $Q \leftarrow \{(x_j, \mathcal{C}_i) | \mathcal{C}_i = (\mathcal{X}_i, \mathcal{R}_i) \in \mathcal{C} \wedge x_j \in \mathcal{X}_i\}$;
  while $Q \neq \varnothing$ **do**
    $(x_j, \mathcal{C}_i) \leftarrow$ `retirer(`Q`)` ;
    foreach $v \in \mathcal{D}_j$ **do**
      foreach $u \in [\![|\mathcal{D}_i|]\!] \ : \ i_u \neq j$ **do** $\mathcal{D}'_{i_u} \leftarrow \mathcal{D}_{i_u}$;
      $\mathcal{D}'_j = \{v\}$;
      $support \leftarrow false$;
      $\mathcal{V} \leftarrow \mathcal{D}'_{i_1} \times \cdots \times \mathcal{D}'_j \times \cdots \times \mathcal{D}'_{i_k}$;
      while $\neg support \wedge \mathcal{V} \neq \varnothing$ **do**
        $v_i \leftarrow$ `retirer(`v`)`;
        if $v_i \in \mathcal{R}_i$ **then** $support \leftarrow true$;
      end
      if $\neg support$ **then**
        $\mathcal{D}_j \leftarrow \mathcal{D}_j \backslash \{v\}$;
        if $\mathcal{D}_j = \varnothing$ **then return** `false`;
        $Q \leftarrow Q \cup \{(x_k, \mathcal{C}_l) | \mathcal{C}_l \in \mathcal{C} \wedge \mathcal{C}_l \neq \mathcal{C}_i \wedge x_j, x_k \in \mathcal{X}_l \wedge k \neq j\}$;
      end
    end
  end
  return P ;
end

par GAC du nœud racine de ce CSP, et nous allons centrer notre exemple sur ce nœud après filtrage.

Pour effectuer le branchement, il existe deux choix de variables x_1 et x_2 et, pour chacune de ces variables, deux choix de valeurs sont possibles. Étudions les deux branchements possibles en sélectionnant la variable x_1.

1. si nous branchons sur la variable x_1 en lui affectant la valeur 1, alors par filtrage sur la contrainte C_1 seule la valeur 3 est autorisée pour x_2. De plus, le triplet $(x_1 = 1, x_2 = 3, x_3 = 4)$ satisfait la contrainte C_2, et n'est donc pas solution du CSP (la figure 1.5 montre la construction de l'arbre de recherche jusqu'à la première solution) ;

2. si nous branchons sur la variable x_1 en lui affectant la valeur 2, alors par filtrage sur la contrainte C_1 seule la valeur 2 reste autorisée pour x_2. Cependant, le triplet $(x_1 = 2, x_2 = 2, x_3 = 4)$ ne satisfait pas la contrainte C_2, donc il ne s'agit pas d'une solution du CSP (figure 1.5).

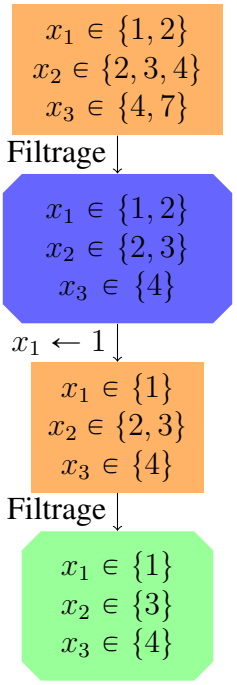

FIGURE 1.5 – Branchement sur x_1 avec la valeur 1 d'abord.

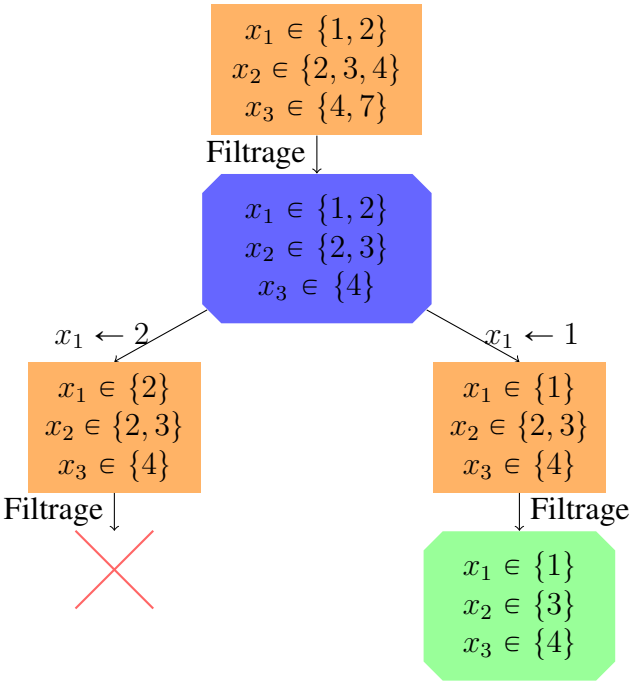

FIGURE 1.6 – Branchement sur x_1 avec la valeur 2 d'abord.

Par construction, toute heuristique de choix de valeur induit un ordre entre les affectations totales. L'objectif est donc de "positionner" les affectations solutions le plus à gauche dans le cadre d'une recherche en profondeur d'abord afin de réduire le temps d'obtention d'une première solution. Nous remarquons toujours que ces heuristiques sont sans incidence sur la taille de l'arbre de recherche. À l'inverse, les heuristiques de choix de variables influent sur la taille de l'arbre mais non pas sur le nombre de feuilles, ni leur positionnement relatif. L'objectif est ici de donner la priorité aux variables induisant le plus petit arbre.

△

1.2.2.4 Contraintes globales

La modélisation des CSP peut s'avérer fastidieuse et coûteuse lorsqu'elle est limité à l'use de contraintes en extension ou de contraintes primitives fondées sur des opérateurs arithmétiques ou logiques élémentaires. L'emploi de contraintes dites globales vise à remédier au problème.

D'une part, les contraintes globales sont des contraintes exprimées en intention et dont l'arité excède généralement deux. Certaines permettent de modéliser directement des relations complexes entre variables et typiques de certains domaines d'application (e.g. problématiques de sac à dos, couplage, etc.). D'autres facilitent l'expression de relations dont la décomposi-

tion de contraintes primitives est rédhibitoire (e.g. contrainte globale `all_different` en lieu et place d'un nombre quadratique de d'inégalités binaires). D'autre part, nous associons à ces contraintes des algorithmes de filtrage dédiés, appelés propagateurs, qui s'intègrent aisément dans les méthodes génériques de maintien de consistance (AC, BC, GAC, etc.) et contribuent à l'efficacité générale de la résolution CSP.

Dans les langages PPC, les contraintes globales sont implantées par un ou plusieurs propagateurs selon les des degrés de consistance. Ces propagateurs fonctionnent en autonomie et disposent de leur propre structure de données. Beldiceanu dresse un catalogue des contraintes globales communément utilisées [3].

Nous présentons deux propagateurs pour les contraintes globales `all_different` et `regular`. L'objectif est double : présenter des mécanismes de propagation dédiés et comprendre le fonctionnement de la contrainte `regular` que nous emploierons ultérieurement. Nous utiliserons les notations suivantes. Soit x une variable de domaine D, $dom(x) \subseteq D$ correspond au domaine de x à un instant de la résolution d'un CSP. Nous supposons D totalement ordonné et $min(x)$ et $max(x)$ représentent respectivement les valeurs minimum et maximum de l'ensemble $dom(x)$. Soit une contrainte c et I un intervalle sur D, $vars(c)$ correspond à l'ensemble des variables impliquées dans la contrainte c et $vars(c, I)$ correspond aux variables de $vars(c)$ dont les domaines sont inclus dans I ($vars(c, I) = \{x | x \in vars(c) \wedge dom(x) \subseteq I\}$).

1.2.2.4.1 Contrainte `all_different`

La contrainte `all_different` [32] impose un ensemble de variables d'être différentes deux à deux. Puget propose dans [56] différents propagateurs pour réaliser la consistance de bornes pour cette contrainte, c'est-à-dire que seules les valeurs minimales et maximales des domaines des contraintes sont altérées. Nous nous concentrerons sur le premier algorithme proposé, même s'il ne s'agit pas du plus efficace.

Le propagateur repose sur la notion d'intervalle de Hall. Un intervalle de Hall pour une contrainte portant sur un ensemble de variables $\mathcal{X}$ est un intervalle I tel que le nombre de variables de $\mathcal{X}$ dont les domaines sont inclus dans I est égal à la cardinalité de I. Dans le cas de la contrainte `all_different`, cela signifie que si I est intervalle de Hall, les variables dont le domaine est inclus dans I doivent prendre une valeur de I afin de respecter la contrainte, ce qui interdit aux autres variables de prendre une valeur de I.

Définition 6 (Intervalle de Hall)
Soit une contrainte C sur un ensemble de variables à domaine sur $\mathcal{D}$ et I un intervalle de $\mathcal{D}$ I est un intervalle de Hall si $|I| = |vars(c, I)|$.

L'algorithme 3 présente le fonctionnement du propagateur pour la mise à jour des bornes minimales. Les bornes maximales peuvent être obtenues via le même algorithme, en l'appliquant

aux variables opposées. Par exemple, si la contrainte `all_different` porte sur n variables $x_1, \ldots, x_n$, alors nous pouvons mettre à jour les bornes minimales des x_i en appliquant le propagateur sur les x_i et les bornes maximales sont mises à jour en appliquant le propagateur sur des variables z_i telles que $z_i = -x_i$ ($i \in [1, n]$).

Aussi, le propagateur traite-t-il ses variables une à une par ordre croissant de leurs valeurs maximales. Soit $\mathcal{X}$ le n-uplet de variables sur lequel s'applique la contrainte. $\mathcal{X} = (x_1, \cdots, x_n)$ est trié si pour tout $i < j \in [\![n]\!]$ $max(x_i) \leqslant max(x_j)$. Pour chaque variable x_i et x_j, l'algorithme maintient la valeur $u_j^i = min(x_j) + |\{k | k < i, min(x_k) \geqslant min(x_j)\}| - 1$. Cette valeur u_j^i permet de détecter les intervalles de Hall. En effet, si $u_j^i = max(x_i)$ alors $|\{k | k \leqslant max(x_i), min(x_k) \geqslant min(x_j)\}| = max(x_i) - min(x_j) + 1$, ce qui implique qu'il existe $max(x_i) - min(x_j) + 1$ variables dont le domaine est inclus dans l'intervalle $[min(x_j), max(x_i)]$. Il s'agit donc d'un intervalle de Hall. La détection d'un intervalle de Hall met à jour la borne minimale des variables n'appartenant pas à cet intervalle et dont le minimum est supérieur à la borne minimale de l'intervalle. En effet, pour ces variables le minimum ne peut pas être inférieur à $max(x_i)$. Dans le cas où u_j^i est supérieur à $max(x_i)$, il y a strictement plus de $max(x_i) - min(x_j) + 1$ variables dont le domaine est compris entre $[min(x_j), max(x_i)]$, donc la contrainte ne peut pas être satisfaite. L'algorithme 3 met en œuvre ce processus.

Exemple 5. Soient les variables $x_1 \in [3, 6]$, $x_2 \in [3, 4]$, $x_3 \in [2, 5]$, $x_4 \in [2, 4]$, $x_5 \in [3, 4]$, $x_6 \in [1, 6]$ et la contrainte `all_different`(x_1,x_2,x_3,x_4,x_5,x_6).

Dans un premier temps, le propagateur trie les x_i par ordre croissant des bornes maximales : $x'_1 = x_2$, $x'_2 = x_4$, $x'_3 = x_5$, $x'_4 = x_3$, $x'_5 = x_1$, $x'_6 = x_6$. La trace de l'exécution est présentée en figure 1.7. À la fin de l'exécution , les domaines des variables sont : $x_1 \in \{6\}$, $x_2 \in [3, 4]$, $x_3 \in \{5\}$, $x_4 \in [2, 4]$, $x_5 \in [3, 4]$, $x_6 \in [1, 6]$.

L'obtention des bornes finales, consiste à répéter l'opération sur les bornes maximales et minimales jusqu'à atteindre un point fixe. △

- $min[1] \leftarrow min(x_2) = 3, max[1] \leftarrow max(x_2) = 4$
- $min[2] \leftarrow min(x_4) = 2, max[2] \leftarrow max(x_4) = 4$
- $min[3] \leftarrow min(x_5) = 3, max[3] \leftarrow max(x_5) = 4$
- $min[4] \leftarrow min(x_3) = 2, max[4] \leftarrow max(x_3) = 5$
- $min[5] \leftarrow min(x_1) = 3, max[5] \leftarrow max(x_1) = 6$
- $min[6] \leftarrow min(x_6) = 1, max[6] \leftarrow max(x_6) = 6$
- $Insert(1)$
  - $u[1] \leftarrow 3$
- $Insert(2)$
  - $u[2] \leftarrow 2$
  - $min[1] \geqslant min[2] \rightarrow u[2] \leftarrow 3$
- $Insert(3)$
  - $u[3] \leftarrow 3$
  - $min[1] \geqslant min[3] \rightarrow u[3] \leftarrow 4$
  - $u[3] = max[1] = 4$ appelle $IncrMin(3, 4, 3)$ qui poste $x_1 \geqslant 5$
  - $min[2] \geqslant min[3] \rightarrow u[3] \leftarrow 4$
  - $u[2] = max[3] = 4$ appelle $IncrMin(2, 4, 3)$ qui poste $x_1 \geqslant 5$ et $x_3 \geqslant 5$
- $Insert(4)$
  - $u[4] \leftarrow 2$
  - $min[1] \geqslant min[4] \rightarrow u[4] \leftarrow 3$
  - $min[2] \geqslant min[4] \rightarrow u[4] \leftarrow 4$
  - $min[3] \geqslant min[4] \rightarrow u[4] \leftarrow 5$
  - $u[4] = max[4] = 4$ appelle $IncrMin(2, 5, 4)$ qui poste $x_1 \geqslant 6$
- $Insert(5)$
  - $u[5] \leftarrow 3$
- $Insert(6)$
  - $u[6] \leftarrow 1$

FIGURE 1.7 – Déroulement du propagateur de `all_different`.

Algorithme 3 – Propagateur pour la contrainte all_different

Entrées : x tableau contenant les n variables de la contrainte

Données : u, min et max tableaux d'entiers

```
1  begin
2      x ← Trier(x) ;                ▷ tri par ordre croissant des maxima des domaines
3      for i ← 1 to n do
4          min[i] ← min(x[i]);
5          max[i] ← max(x[i]);
6      end
7      for i ← 1 to n do Insert(i) ;
8  end
9  Insert(i) :
10 begin
11     u[i] ← min[i] ;
12     for j ← 1 to i − 1 do
13         if min[j] < min[i] then
14             u[j] ← u[j] + 1;
15             if u[j] > max[i] then Echec ;
16             if u[j] = max[i] then IncrMin(min[j],max[i],i) ;
17         end
18         else
19             u[i] ← u[i] + 1
20         end
21     end
22     if u[i] > max[i] then Echec ;
23     if u[i] = max[i] then IncrMin(min[i],max[i],i) ;
24 end
25 IncrMin(a,b,i) :
26 begin
27     for j ← i + 1 to n do
           ▷ Poster() modifie le domaine de x[j] en imposant que sa valeur minimum
           ▷ soit supérieure ou égale à b + 1.
28         if min[j] ⩾ a then Poster(x[j] ⩾ b + 1) ;
29     end
30 end
```

1.2.2.5 Contrainte `regular`

La contrainte `regular`, introduite par Pesant [55], force une séquence de taille donnée à respecter une expression régulière ou un automate. Elle prend en entrée une séquence composée de variables et un automate donné (l'automate ne sera pas modifié en cours de recherche). La contrainte `regular` construit une structure de données interne qui guide la propagation. Cette structure dépend de l'automate et par taille de séquence, mais si les mécanismes de création de propagation restent les mêmes d'une instance de `regular` à une autre. Un exemple d'utilisation de cette contrainte est proposé en figure 1.8.

```
% ab*
int: n;     % longueur de la sequence
int: Q = 3; % nombre d'etats de l'automate
int: S = 2; % nombre de symboles

set of int: ETATS = 1..Q;
set of int: SYMBOLES = 1..S;

int: q0 = 1;                                % etat initial
set of int: F = {Q};                        % etats finaux
array[ETATS,SYMBOLES] of set of int: d =  % transitions
  [|{2}, {}|
     {},{3}|
     {},{3}|];

array[1..n] of var SYMBOLES: sequence;

constraint regular_nfa(sequence, Q, S, d, q0, F);
```

FIGURE 1.8 – Utilisation de la contrainte `regular`

La construction de la structure de données est présentée pour un automate fini et déterministe (AFD). Si une expression régulière est passée en paramètre, il suffit de la transformer en AFD. Soit n la taille de la séquence et N le nombre d'états de l'automate qui représente le langage régulier auquel doit appartenir la séquence. Le propagateur va créer un réseau de n couches numérotées de 1 à n, chaque couche contient N sommets. La figure 1.10 montre un exemple de réseau utilisé par le propagateur de la contrainte. Ce réseau décrit les états atteignables par la séquence après la lecture de chaque symbole. La couche 1 ne contient que l'état initial de l'automate, la couche 2 contient les états pouvant être obtenus en lisant un symbole quelconque de l'alphabet - le réseau est moindre dans le propagateur car les domaines des variables sont pris en compte -, la couche k fait apparaître tous les états pouvant être atteints en ayant parcouru une séquence quelconque de $k-1$ symboles. Les arcs situés entre une couche k et une couche $k+1$ sont uniquement liés au domaine de la variable en position k de la séquence. En effet, ces arcs contiennent les symboles permettant la transition d'un état de l'automate à l'autre. Ils correspondent donc aux domaines des variables de la séquence. Les arcs entre les couches du

réseau sont construits en deux passes sur le réseau : la première crée les arcs qui peuvent être atteints à partir de l'état initial de l'automate, en partant de la couche 1 jusqu'à la couche n. La

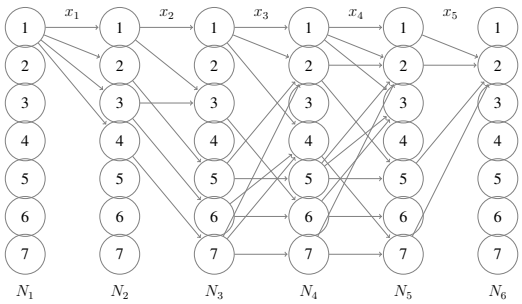

FIGURE 1.9 – Construction du graphe interne de la contrainte `regular` (étape 1 à 5)

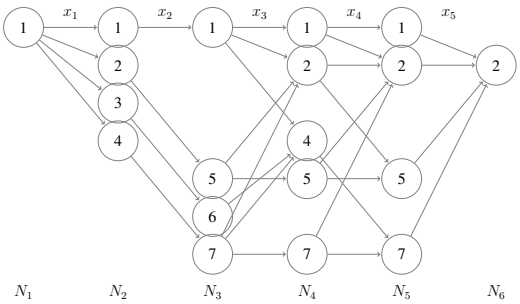

FIGURE 1.10 – Construction du graphe interne de la contrainte `regular` (étape 7 à 8)

seconde, dans l'autre sens, élimine les arcs qui ne peuvent pas aboutir à un état acceptant. La construction des arcs du réseau se déroule comme suit :

1. la construction démarre à la couche 1 sur le sommet de l'état initial ;
2. un arc est alors créé entre l'état initial de la couche 1 et ses successeurs possibles de la couche 2 ; un successeur est possible s'il existe une transition dans l'automate dont le symbole appartient au domaine de la variable de la séquence ;
3. nous avançons d'une couche dans le réseau ;
4. pour chaque sommet dont le degré entrant est nonnul, un arc est créé avec ses successeurs possibles de la couche suivante ;
5. l'étape 3 et 4 sont répétées jusqu'à atteindre la dernière couche ;
6. nous supprimons de la dernière couche les arcs entrant sur des sommets dont l'état n'est pas un état acceptant de l'automate ;
7. nous revenons une couche en arrière ;
8. nous retirons les arcs entrants des sommets qui n'ont aucun arc sortant car ceux-ci ne pourront pas atteindre un état final ;
9. les étapes 7 et 8 sont répétées jusqu'à atteindre la première couche.

Exemple 6. Soient une séquence de variables x_1, x_2, x_3, x_4 et x_5 de domaines respectifs $D_1 = \{a, b, c, o\}$, $D_2 = \{b, o\}$, $D_3 = \{a, c, o\}$, $D_4 = \{a, b, o\}$, $D_5 = \{a\}$ et l'automate défini par la figure 1.12.

La figure 1.9 montre la création du réseau lors des étapes 1 à 5. La figure 1.10 montre la création du réseau lors des étapes 7 à 8. La figure 1.11 montre la mise à jour du réseau après avoir retiré le symbole o du domaine de x_4. △

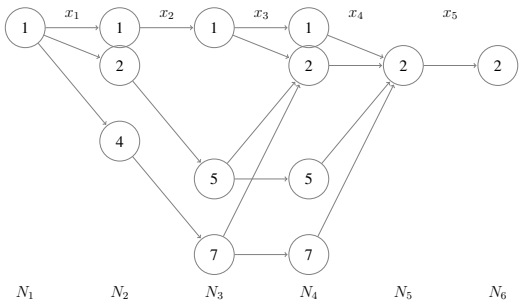

FIGURE 1.11 – Réduction du réseau après modification du domaine de x_4

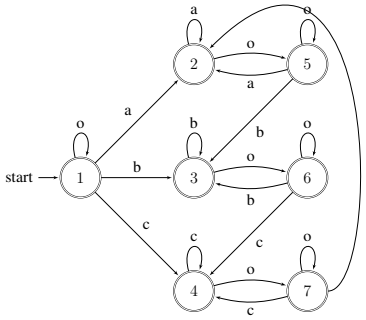

FIGURE 1.12 – Automate fourni à la contrainte `regular`

Finalement, pour chaque variable x_a de la séquence, a étant la position de la variable dans la séquence, son domaine correspond à l'union des symboles impliqués dans une transition à partir de la couche a à la couche $a + 1$.

L'algorithme vérifie si la réduction du domaine d'une variable n'entraîne pas l'élimination d'un arc du réseau. Si tel est le cas, alors il effectue une mise à jour du réseau qui supprime les arcs se trouvant séparés du reste du réseau. Eu égard au fait qu'un arc joigne deux nœuds entre eux, l'éviction se déroule dans les deux sens : à la fois vers l'avant du réseau (couches d'indices plus élevés) en démarrant du nœud cible de l'arc précédent et vers l'arrière en partant du nœud source. Lorsqu'une suppression intervient, cela affecte potentiellement le domaine d'une autre variable qui, à son tour, lance une phase d'élagage du réseau. Le propagateur atteint un point fixe dès qu'il n'altère plus les domaines des variables.

1.3 Fouille de données et apprentissage artificiel

Cette section aborde deux problématiques : la recherche d'itemsets et la recherche de motifs. Nous formalisions brièvement ces problèmes et présentons différentes approches que nous distinguons selon qu'elles reposent sur des algorithmes dédiés où des modélisations à base de contraintes sous-tendus par une algorithmique générique de type PPC ou (SAT).

1.3.1 Recherche d'itemsets

La recherche d'itemsets fréquents introduit par Argwal [1] est un problème central en fouille de données. Comme mentionné précédemment, l'exemple classique s'intéresse aux habitudes de consommations des clients d'une enseigne. En l'occurrence, la base de données correspond à

	A	B	C	D
T_1	0	1	1	1
T_2	0	1	0	1
T_3	0	1	1	1
T_4	1	0	1	1

TABLEAU 1.1 – Base de transactions sous forme matricielle

un ensemble de transactions constituées chacune d'un ensemble de produits achetés. L'objectif est alors de trouver les ensembles de produits dont la fréquence excède un seuil donné.

Une base constituée d'un ensemble $\mathcal{I}$ d'items, d'un ensemble $\mathcal{T}$ de transactions peut être représentée de deux façons : Soit par une matrice pseudo-boolénne $\mathcal{D}$ de taille $|\mathcal{I}| \times |\mathcal{T}|$ telle que $\mathcal{D}_{ti} = 1$ si l'item i appartient à la transaction t; soit par le multi-ensemble $\mathcal{T}$ d'itemsets tel que $t \subseteq \mathcal{I}$ pour tout $t \in \mathcal{T}$. La couverture d'un itemset I, notée $\varphi_{\mathcal{D}}(I)$, correspond à l'ensemble des transactions dans lesquelles. Il apparaît : $\varphi_{\mathcal{D}}(I) = \{t \in \mathcal{T} | \forall i \in I : \mathcal{D}_{ti} = 1\}$. Le support d'un itemset, noté $support_{\mathcal{D}}(I)$, correspond au nombre de transactions couvertes : $support_{\mathcal{D}}(I) = |\varphi_{\mathcal{D}}(I)|$.

Le problème de recherche d'itemsets fréquents consiste alors à déterminer l'ensemble des itemsets dont le support dépasse un seuil Θ donné selon l'ensemble $\{I | I \subseteq \mathcal{I}, support_{\mathcal{D}}(I) \geqslant \Theta\}$. Une variante consiste à calculer les itemsets fermés, c'est-à-dire les itemsets qui ne sont inclus dans aucun itemset de même support, soit l'ensemble $\{I | I \subseteq \mathcal{I}, support_{\mathcal{D}}(I) \geqslant \Theta, \psi_{\mathcal{D}}(\varphi_{\mathcal{D}}(I)) = I\}$ avec $\psi_{\mathcal{D}}(T) = \{i \in \mathcal{I} | \forall t \in T : \mathcal{D}_{ti} = 1\}$. $\psi_{\mathcal{D}}(T)$ contient l'intégralité des items communs aux transactions de T.

Exemple 7. Soient $\mathcal{I} = \{A, B, C, D\}$ un ensemble d'items et $t = \{T_1, T_2, T_3, T_4\}$ un ensemble de transactions composées d'items de $\mathcal{I}$. Le tableau 1.1 présente la matrice pseudo-booléenne modélisant la base de données. La valeur 1 indique qu'un item appartient à la transaction, tandis que la valeur 0 indique sa non-appartenance. Soit $\theta = 2$ le support minimum à respecter pour qu'un itemset soit solution, l'itemset $\{B, C, D\}$ est solution, car il couvre deux transactions T_1 et T_3. De plus, il est fermé, car sa seule extension possible impose l'ajout de l'item A, ce qui entrainerait une réduction de son support à 0. $\{B, D\}$ et $\{C, D\}$ sont deux itemsets fermés de support 3. $\{B\}$ et $\{C\}$ sont deux itemsets de support 3 mais non-fermés. En outre, tout itemset incluant A ne peut pas être solution.

△

1.3.1.1 [DÉDIÉ] APriori

L'algorithme APriori, développé par Agrawal et Srikantdre [2] est l'un des premiers traitant le problème de recherche d'itemsets fréquents. Nous n'en donnons ici qu'une description rapide, car le chapitre 5 le décrit plus en détail.

L'algorithme repose sur le parcours d'un treillis exploitant sur la relation d'anti-monotonicité suivante : si un itemset est infréquent alors ses super-itemsets (les itemsets qui l'incluent) le sont aussi. APriori identifie d'abord les itemsets singletons fréquents, créant ainsi la première couche du treillis. Les itemsets de chaque couche sont fusionnés deux à deux, afin de générer la couche suivante jusqu'à obtenir l'intégralité des itemsets fréquents. Avant chaque fusion, une étape de filtrage élimine ainsi les itemsets infréquents. De par son exploration complète du treillis des itemsets fréquents, APriori est à même d'identifier les itemsets fermés lors de son exécution. En effet, si tous les super-itemsets d'un itemset I ont un support inférieur au support de I, alors I est fermé.

1.3.1.2 [PPC] Méthode Guns

Guns, Njissen et De Raedt ont initié les premières approches concernant la recherche d'itemsets en programmation par contraintes [20, 12, 13, 52]. Ces articles présentent des modèles PPC pour la recherche d'itemsets fréquents et fermés, ainsi que des variantes, comme la recherche d'itemsets optimaux basés sur un critère de coût. En outre, l'article [20] traite de la classification supervisée entre deux ensembles de transactions. Nous présentons ici un des modèles proposés en expliquant succinctement le rôle des diverses contraintes utilisées lors la recherche d'itemsets.

Le modèle s'appuie sur un encodage de la base de transactions sous forme de matrice pseudo-booléenne D. À chaque transaction est associée une variable pseudo-booléenne ($\forall t \in \mathcal{T},\ T_t \in \{0,1\}$), ainsi qu'à chaque item ($\forall i \in \mathcal{I},\ I_i \in \{0,1\}$). L'affectation $T_t = 1$ indique que la transaction t est couverte par l'itemset solution, l'affectation I_i indique que l'item i appartient à l'itemset solution. Une solution est une affectation des variables satisfaisant les contraintes suivantes :

$$\forall t \in \mathcal{T} :\ T_t = 1 \leftrightarrow \sum_{i \in \mathcal{I}} (1 - \mathcal{D}_{ti}) I_i = 0 \tag{1.1}$$

$$\sum_{t \in \mathcal{T}} T_t \geqslant \Theta \tag{1.2}$$

$$\forall i \in \mathcal{I} : I_i = 1 \rightarrow \sum_{t \in \mathcal{T}} T_t \mathcal{D}_{ti} \geqslant \Theta \tag{1.3}$$

$$\forall i \in \mathcal{I} : I_i = 1 \leftrightarrow \sum_{t \in \mathcal{T}} T_t(1 - \mathcal{D}_{ti}) = 0 \tag{1.4}$$

La contrainte (1.1) stipule que chaque transaction couverte doit contenir l'ensemble des items de la solution. Les contraintes (1.2) et (1.3) forcent le respect du seuil de fréquence fixé (Θ). La contrainte (1.3) est une version réifiée de (1.2) - les deux sont présentées à titre de comparaison, mais une seule est requise. La contrainte (1.3) se traduit comme suit : si un item ne permet pas de couvrir θ transactions parmi celles couvertes ou pouvant encore être couvertes, alors la sélection de cet item, à ce stade de la recherche, n'aboutira jamais à une solution. Enfin, la dernière contrainte (1.4) force la fermeture, c'est-à-dire que si un item couvre l'ensemble des transactions T de la solution, alors celui-ci doit appartenir à l'ensemble des items I de la solution et réciproquement.

1.3.1.3 [PPC] k-patterns

De Raedt, Guns et Njissen abordent différents problèmes d'extraction de k-patrons sous l'angle de la PPC : k-term DNF, clustering, k-tiling [21]. Un k-patron est un ensemble de k itemsets (k est un paramètre fixé). Par exemple, le problème de k-term DNF consiste à trouver une formule booléenne prédictive, sous forme normale disjonctive, permettant de distinguer deux classes. Les clauses sont représentées par des itemsets où chaque item représente un atome. Signalons également le problème de k-clustering dans lequel k clusters sont formés à partir d'itemsets. Pour chaque type de problème, les auteurs s'intéressent à l'efficacité des modèles en fonction des contraintes requises.

1.3.2 Recherche de motifs séquentiels

La recherche de motifs sur données séquentielles recouvre de nombreux problèmes égard au caractère protéiforme du concept de motif : sous-chaîne commune à un jeu de séquences, sous-séquence commune, alignement, expression régulière, etc. Ces problèmes déclinent aussi en différentes variantes selon que nous cherchons un, plusieurs ou la totalité des motifs, selon que nous cherchons des motifs communs à l'ensemble des séquences ou spécifiques à des classes de séquences pré-établies, ou encore selon le type d'alphabet sous-jacent. Afin d'unifier la terminologie et de pouvoir mettre en perspective les différentes méthodes présentées, nous distinguons et formalisons deux types de motifs fondamentaux : les sous-séquences et les patron. Les premiers prennent uniquement en compte l'ordre d'apparition des caractères, tandis que les seconds ajoutent à cet ordre une contrainte d'espacement fixe entre les caractères consécutifs.

Étant donné un jeu de séquences B défini comme un multi-ensemble de chaînes de caractères construit su le même alphabet Σ, nous définissons sous-séquences et patrons comme suit :

s_1	A	B	C	D
s_2	A	C	B	D
s_3	A	A	B	D

TABLEAU 1.2 – Base de séquences

— **Sous-séquence :** Soient $s_1 =< i_1, \ldots, i_m >$ et $s_2 =< i'_1, \ldots, i'_n >$ deux séquences, s_1 est incluse dans s_2, dénoté $s_1 \preceq s_2$, s'il existe $1 \leqslant j_1 < j_2 < \ldots < j_m \leqslant n$ tel que $i_1 = i'_{j_1}, \ldots, i_m = i'_{j_m}$. s_1 est appelée sous-séquence de s_2 et s_2 super-séquence de s_1.

— **Patron :** Soient $s =< i_1, \ldots, i_n >$ une séquence et $p =< i'_1, \ldots, i'_m >$ une séquence sur l'alphabet $\Sigma \cup \{\#\}$, où $\#$ est un caractère joker qui n'appartient pas à Σ et qui peut se substituer à tout caractère de Σ dit caractère plein. p est un patron inclus dans s s'il existe $k \in \{0, \cdots, n-1\}$ tel que $\forall j \in [\![m]\!] : i'_j \in \{i_{j+k}, \#\}$.

Nous associons à ces deux types de motifs des définitions communes. Soit un motif m correspondant à une sous-séquence ou un patron, la couverture de m correspond à l'ensemble des séquences de B qui incluent m. Le support de m correspond à la cardinalité de la couverture donc au nombre de séquences incluant le motif et la fréquence le ratio entre le support et la cardinalité de B. Nous pouvons également définir un motif fermé comme un motif fréquent tel qu'il n'existe aucun motif de même fréquence l'incluant [2].

Exemple 8. Soient $\{s_1, s_2, s_3\}$ trois séquences sur l'alphabet $\Sigma = \{A, B, C, D\}$ définies dans le tableau 1.2. Soit $\theta = 2$ le support minimum à respecter pour qu'un motif soit solution. AB et ABD sont deux sous-séquences de support 3 et ABD est fermée, car elle ne peut ni être étendue par la gauche ou la droite sans perdre en support. AC est une sous-séquence fermée de support 2. $A\#\#D$ et $A\#B$ sont deux patrons fermés de supports respectifs 3 et 2. Ce sont les seuls patrons qui respectent la contrainte de support.

△

1.3.2.1 [PPC] Méthode Kemmar

Kemmar et al. présentent un modèle PPC pour extraire des patrons fréquents [30]. Il s'agit ici de patrons au sens de la définition proposée précédemment et soumis à deux contraintes. La première contrainte limite leur taille (jokers inclus) à un paramètre k. La seconde stipule qu'un jeu ne peut pas apparaître en première position de patron. L'approche garantit l'extraction de

2. L'inclusion entre patrons et l'inclusion entre sous-séquences expriment fondamentalement des relations de subsomption qui peuvent prendre différentes formes. Nous nous attardons aux chapitre 5 sur l'inclusion entre patrons.

l'ensemble des patrons fréquents. Dans le modèle, les patrons à déterminer sont représentés par un vecteur de variables dont le domaine est un alphabet incluant le caractère joker. Le joker sert également à compléter le vecteur lorsque celui-ci modélise un patron de taille inférieure à n. Comme les patrons recherchés doivent être fréquents, leur couverture est modélisée par un tableau de variables booléennes, indicées sur les identifiants de séquences qui enregistrent les séquences couvertes. L'article soumet d'autres propositions comme, par exemple, l'adjonction d'une contrainte sur l'allure des patrons par l'utilisation de la contrainte globale `regular`.

1.3.2.2 [PPC & SAT] Méthode Coquery

Coquery, Jabbour, Saïs et Salhi proposent deux approches pour la recherche de patrons fermés de support dépassant un seuil donné [8, 7]. Il s'agit ici de patrons constitués soit d'un unique caractère plein, soit d'une suite de caractères pleins et de jokers commençant et se terminant par un caractère plein. Les patrons appartiennent au langage $\Sigma \cup \Sigma(\Sigma \cup \#)^*\Sigma$, avec Σ un alphabet et $\#$ le caractère joker. L'extraction prend place au sein d'une même séquence et non dans une base de séquences, ce qui amène à redéfinir la notion de support. Cette fois-ci, le support correspond au nombre de répétitions du patron dans la séquence. La fermeture, quant à elle, s'exprime ainsi : un patron est fermé si, lorsqu'il est étendu à droite ou à gauche, son support diminue, c'est-à-dire qu'il ne peut pas être étendu en l'un de ses points d'ancrage. Deux modélisations sont décrites : l'une en SAT, l'autre hybridant SAT avec des contraintes.

1.3.2.3 [DÉDIÉ] PrefixSpan

Pei et al. proposent l'algorithme PrefixSpan qui est une amélioration de l'algorithme FreeSpan [53, 22]. Ces algorithmes se spécialisent dans le calcul d'ensembles de sous-séquences ou de sous-séquences d'itemsets, c'est-à-dire que les caractères sont représentés par des ensembles d'items et donc un caractère apparaît dans une séquence si celle-ci contient un sur-ensemble incluant le caractère. Une sous-séquence d'itemsets est de la forme $< a(abc)b(ac) >$ (les parenthèses signalent que plusieurs symboles se chevauchent). Le tableau 1.3 propose une représentation tabulaire de cette séquence.

e \ t	1	2	3	4
a	1	1	0	1
b	0	1	1	0
c	0	1	0	1

TABLEAU 1.3 – Représentation graphique de la séquence $< a(abc)b(ac) >$. Nous emploierons dorénavant le terme événement (colonne i) pour désigner les caractères. Les lignes représentent les événements et les colonnes représentent la chronologie dans laquelle les événement se sont déroulés. Une cellule de colonne t_i et de ligne e_j du tableau prend la valeur 1 si et seulement si l'événement e_j a lieu au temps t_i. Deux événements peuvent survenir au même instant.

Le processus de résolution commence par effectuer une analyse de la base de séquences afin d'énumérer les items fréquents. Pour chaque item fréquent, une base de séquences projetées est créée, cette base contient les postfixes des séquences ayant pour préfixe commun les items fréquents. Par exemple, la projection de $< a(abc)b(ac) >$ par a est $< (abc)b(ac) >$, et par b $< (a\_c)b(ac) >$ (le symbole _ signifie qu'un a et un c occupe la même localisation que b). Ces deux projections sont représentées sous forme tabulaire dans le tableau 1.4 et le tableau 1.5. Ce mécanisme se répète récursivement sur les bases projetées générant ainsi des sous-séquences fréquentes. Nous retrouverons cette notion de projection par la suite, notamment au chapitre 3.

e \ t	2	3	4
a	1	0	1
b	1	1	0
c	1	0	1

TABLEAU 1.4 – Représentation graphique de la séquence $< a(abc)b(ac) >$ après une projection sur a.

e \ t	2	3	4
a	1	0	1
b	0	1	0
c	1	0	1

TABLEAU 1.5 – Représentation graphique de la séquence $< a(abc)b(ac) >$ après une projection sur a.

1.3.2.4 [DÉDIÉ] WAP & PLWAP

WAP [54] et PLWAP [16] sont deux algorithmes cherchant des sous-séquences fréquentes. Les deux reposent sur le même principe, PLWAP étant une version modifiée de WAP. Ils compressent les séquences sous la forme d'un arbre qui, par la suite, sert à l'extraction des motifs. La construction d'une telle structure perd l'information de couverture mais conserve l'information de support, afin d'évaluer la fréquence des motifs. Nous noterons que cette structure peut, avec des modifications, être incorporée dans une contrainte globale.

```
include "regular_nfa.mzn";

int: Q = 5; % nombre d'etats de l'automate
int: S = 5; % nombre de symboles (1,2,3,4,5=joker)

set of int: ETATS = 1..Q;
set of int: SYMBOLES = 1..S;

int: q0 = 1;
set of int: F = {Q};

array[ETATS,SYMBOLES] of set of int: d1234 =
  [|{2},{3},{4},{5}, {}|
     {},{3},{4},{5},{5}|
     {}, {},{4},{5},{5}|
     {}, {}, {},{5},{5}|
     {}, {}, {}, {},{5}|];
array[ETATS,SYMBOLES] of set of int: d3412 =
  [|{4},{5},{2},{3}, {}|
    {4},{5}, {},{3},{5}|
    {4},{5}, {}, {},{5}|
     {},{5}, {}, {},{5}|
     {}, {}, {}, {},{5}|];

int: n = 4;     % longueur de la sequence
array[1..n] of var SYMBOLES: sequence;

constraint regular_nfa(sequence, Q, S, d1234, q0, F);
constraint regular_nfa(sequence, Q, S, d3412, q0, F);

solve satisfy;

output [show(sequence) ++ "\n"];
```

FIGURE 1.13 – Méthode Métivier sur un jeu de deux séquences

1.3.2.5 [PPC] Méthode Métivier

Toujours dans le cadre de la recherche de sous-séquences fréquentes et fermées, Métivier, Loudni et Charnois présentent un modèle PPC utilisant principalement la contrainte `regular` (voir figure 1.13). L'emploi de cette contrainte impose l'encodage des séquences de la base sous la forme d'automates. `regular` ici sert à imposer la présence du motif au sein des séquences qui sont modélisées par un vecteur de variables entières. Cependant, la fermeture doit être traitée via un problème d'optimisation avec ajouts de nogoods pour ne pas redécouvrir d'anciennes solutions. De la même manière que l'approche de Kemmar (section 1.3.2.1), la taille des séquences constitue un paramètre fixe et des jokers sont agrégés à la fin du vecteur dans le but de permettre l'extraction de motifs de longueur variable. Altérer le type de motifs extraits, par exemple imposer un écart minimum entre deux symboles, nécessite de modifier les automates en entrée. Nous constaterons que l'emploi contrainte `regular` s'avère coûteux en espace mémoire à cause de la taille de son encodage (voir section 1.2.2.5).

1.3.2.6 [PPC] Méthode Negrevergne

Negrevergne et Guns proposent une autre approche pour extraire les sous-séquences maximales [51]. Une sous-séquence commune est dite maximale si elle ne peut pas être étendue (ajout de caractères) sans enfreindre le seuil de fréquence imposé. Il s'agit d'un modèle similaire à celui proposé par Kemmar dans [30], à la différence du test d'inclusion des sous-séquences qui est effectué à l'aide de deux contraintes globales ad-hoc. La première vérifie uniquement l'inclusion tandis que la seconde explicite également l'enracinement des caractères de la sous-séquence extraite pour les séquences couvertes du jeu de données. La seconde est sous-sommée par la première et permet alors de contraindre l'espacement entre deux symboles, ce qui autorise également l'extraction de patrons. Afin d'améliorer les performances, les auteurs proposent deux moyens de simuler le mécanisme de projection de PrefixSpan (voir section 1.3.2.3), l'un prenant appui sur un mécanisme de branchement ad-hoc qui ne repose pas sur les heuristiques génériques et classiques de la PPC ; l'autre étant guidé par l'ajout de contraintes et de variables. Un second mécanisme de branchement est développé qui optimise le calcul des enracinements, afin de court-circuiter le processus classique d'énumération de tous les enracinements possibles.

1.3.2.7 [PPC] Méthode Kemmar 2

Kemmar, Loudni, Lebbah, Boizumault et Charnois proposent une seconde méthode pour résoudre le problème de sous-séquences communes [29] basée sur une contrainte globale dédiée. Cette contrainte encapsule la procédure PrefixSpan (voir section 1.3.2.3). Cependant, comme les autres méthodes PPC présentées dans cette section, la contrainte proposée reste moins générale que l'algorithme PrefixSpan car elle ne traite pas les séquences d'itemsets.

1.3.2.8 [PPC] Chaînes de caractères

Golden et Pang proposent une représentation des chaînes de caractères en PPC [17]. L'originalité de leur approche réside dans la création d'un nouveau type de variable qui permet de représenter des chaînes de caractères. Les domaines des chaînes de caractères sont des langages réguliers définis à l'aide d'automates ou d'expressions régulières. Sur cette base, les auteurs proposent différentes contraintes :

- $matches(string\ x, regexp\ re)$: x appartient au langage re (cette contrainte est notamment utilisée pour initialiser le domaine de x) ;
- $concat(string\ z, string\ x, string\ y)$: z est la concaténation de x et y ;
- $contains(string\ x, string\ y)$: y est une sous-chaîne de x.
- $length(string\ x, size\ n)$: la longueur de la chaîne x est égale à n.

Ces contraintes sont implantées sous la forme d'opérations sur les automates. Nous retrouvons par exemple l'intersection et la négation des automates. Les auteurs proposent également une approche pour tester si les domaines d'automates sont consistants ou non.

Exemple 9. La figure 1.15 présente un exemple d'utilisation de ces variables chaînes de caractères pour la résolution de la grille de mots-croisés donnée en figure 1.15. Dans le modèle, la liste des mots est enregistrée dans le tableau x. La contrainte en lignes 14-16 impose que chaque élément de x respecte l'expression régulière définie par la disjonction des mots de la liste. Les tableaux de variables chaînes b et c permettent de définir la grille. Le tableau c enregistre les cases de la grille où les mots se croisent : il s'agit donc de chaîne de caractères de taille 1 comme spécifié par la contrainte en lignes 23-24. Le tableau b, quant à lui, contient les cellules de la grille qui sont propres à un mot. La taille des éléments de b varie en fonction du nombre de cellules contiguës qui sont propres à un mot. Les contraintes en lignes 27-34 stipulent que chaque mot est la concaténation des éléments de b et c qui le constituent. Par exemple, $x[1]$ est la concaténation de $b[1]$, $c[1]$, $b[2]$ et $c[2]$.

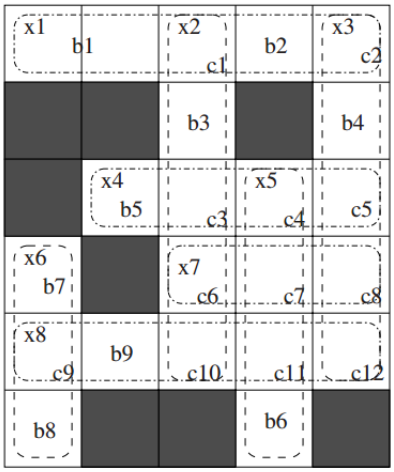

FIGURE 1.14 – Grille de mots-croisés

△

Cette méthode est coûteuse en mémoire et en temps de calcul, car la taille des automates croît avec le nombre de contraintes. Il n'y a pas non plus d'approche satisfaisante pour gérer la négation. En effet, la négation nécessite de transformer un automate non-déterministe en son complémentaire. Cependant, une telle opération peut résulter en un nombre exponentiel d'états, même si la complexité pour passer d'un automate déterministe à son complémentaire est linéaire en temps de calcul (inversion des états finaux).

```
int: mots = 8;
int: segs = 9;
int: pts  = 12;

set of int: MOTS = 1..mots;
set of int: SEGS = 1..segs;
set of int: PTS  = 1..pts;


array[MOTS] of var string: x;
array[SEGS] of var string: b;
array[PTS]  of var string: c;

constraint forall(m in MOTS)(
  matches(x[m], "(AFT)|(ALE)|(EEL)|(HEEL)|...|(TIE)")
);

constraint length(b[1],2);
constraint forall(m in SEGS where m > 1)(
  length(b[m], 1)
);

constraint forall(m in PTS)(
  length(c[m], 1)
);

constraint concat(x[1],b[1],c[1],b[2],c[2]);
constraint concat(x[2],c[1],b[3],c[3],c[6],c[10]);
constraint concat(x[3],c[2],b[4],c[5],c[8],c[12]);
constraint concat(x[4],b[5],c[3],c[4],c[5]);
constraint concat(x[5],c[4],c[7],c[11],b[6]);
constraint concat(x[6],b[7],c[9],b[8]);
constraint concat(x[7],c[6],c[7],c[8]);
constraint concat(x[8],c[9],b[9],c[10],c[11],c[12]);
```

FIGURE 1.15 – Modèle PPC pour la résolution d'une grille de mots-croisés

1.3.2.9 [PPC] Clustering

Dans un autre registre que la recherche de données séquentielles, Dao, Duong et Vrain traitent des problèmes de classification non-supervisée [14]. Dans cet article, le problème de k-clustering est modélisé en PPC. Étant donné un ensemble d'items et une fonction de distance entre deux items, l'objectif consiste à trouver k groupes d'items dont la distance inter-groupe est minimale et la distance extra-groupe est maximale. La première distance correspond à la distance maximale entre deux items d'un même groupe, alors que la seconde distance correspond à la distance minimale entre deux items de deux groupes différents. Le modèle proposé permet d'ajouter simplement des contraintes spécifiques, par exemple des contraintes qui imposent à deux objets d'appartenir au même groupe.

Autres

D'autres approches d'hybridation ont également proposé tel Métivier et al. [49] pour la recherche de patrons fréquents, Métivier et al. pour la recherche de sous-séquences fréquentes [50], où encore Khiari et al. [31] pour la recherche d'ensembles de n-patrons.

1.3.3 Comparaison des méthodes

Le tableau 1.6 compare les différentes méthodes présentées précédemment, à l'exception des méthodes de Dao (section 1.3.2.9), ainsi que celle de Guns (section 1.3.1.3) qui traitent d'autres problèmes que la recherche de données fréquentes. Nous différencions les méthodes selon les motifs calculés : itemsets, sous-séquences ou patrons. Nous les différencions également selon le type d'approche : algorithme dédié (ad-hoc), approche PPC ou SAT.

Quatre autres colonnes précisent d'autres caractéristiques. La fermeture indique si l'approche recherche des motifs fermés. La colonne *support variable* indique si le support des motifs dépend uniquement d'un seuil de fréquence. La colonne *taille variable* signale si une méthode peut calculer toute taille de motifs, c'est-à-dire si aucun paramètre de longueur maximale n'est requis. Finalement, la colonne *énumération* précise si une méthode peut énumérer l'ensemble des motifs. Nous les différencions enfin selon quatre critères : sie l'approche calcule des motifs fermés (colonne "Fermeture"), si le support des motifs dépend uniquement d'un seuil de fréquence (colonne "support variable"), si la méthode à la capacité de calculer des motifs de toute taille [3] et si elle a la capacité à énumérer l'ensemble des motifs (colonne "énumération des solutions").

Nous noterons que la méthode Negrevergne permet de calculer de patrons, même si elle n'y est pas dédiée. Les méthodes WAP et PLWAP peuvent calculer des motifs fermés, mais uniquement après extraction de tous les motifs. La méthode Métivier extrait des motifs fermés, bien qu'elle requière l'ajout de nogoods spécifiques nécessitant l'affectation quasi-complète des variables de décision. La méthode Coquery est la seule à ne pas avoir de support variable, car elle n'opère que sur une unique séquence. Cependant, elle calcule une fréquence basée sur la répétition des motifs au sein de cette même séquence.

1.3.4 Algorithmes sur les textes

Les algorithmes travaillant sur les textes sont nombreux et couvrent une multitude de domaines allant de l'alignement de séquences jusqu'à la recherche de sous-chaines communes en passant par la recherche de mots dans un dictionnaire. Nous abordons les algorithmes sur les

3. C'est-à-dire si aucune limite de taille n'est fixée sur les motifs.

	Itemset	Sous-séquence	Sous-séquence d'itemsets	Patron	Support variable	Fermeture	Taille variable	Énumération des solutions	Approche
APriori (1.3.1.1)	✓				✓	✓	✓	✓	ad-hoc
Guns (1.3.1.2)	✓				✓	✓	✓	✓	CSP
Kemmar (1.3.2.1)				✓	✓				CSP
Coquery (1.3.2.2)				✓		✓		✓	SAT
PrefixSpan (1.3.2.3)		✓	✓		✓		✓	✓	ad-hoc
SPAM		✓	✓		✓		✓	✓	ad-hoc
WAP & PLWAP (1.3.2.4)		✓			✓		✓	✓	ad-hoc
Métivier (1.3.2.5)		✓			✓	✓			CSP
Negrevergne (1.3.2.6)		✓		✓	✓				CSP
Kemmar (1.3.2.7)		✓			✓			✓	CSP

TABLEAU 1.6 – Résumé méthodes de classification

textes, car nous proposons en chapitre 5 d'extraire des motifs basés sur des chaînes de caractères.

M.Crochemore, dans son livre, présente non seulement un ensemble d'algorithmes travaillant sur des textes [9], mais encore la terminologie et les définitions classiques de la théorie des langages. Ainsi, il présente les notions de langages, automates, expressions régulières, sous-séquences, dictionnaires dans des textes. Sont présentes approches utilisées pour résoudre ces problèmes et propose différentes implantations en explicitant les avantages et inconvénients de ces dernières. À chaque algorithme est associée une preuve ainsi qu'une complexité en temps et en mémoire. En guise d'exemple, présentons le problème de recherche la plus longue sous-chaîne commune à un ensemble de séquences.

1.3.4.1 Plus longue sous-chaîne commune

L'objectif est de trouver la plus longue sous-chaîne commune à un ensemble de séquences. Deux approches sont proposées : l'une basée sur la programmation dynamique et l'autre sur la création d'un arbre de suffixe. La première méthode résout le problème en temps exponentiel la seconde en temps polynomial. Ce problème est sous-sommé par celui décrit dans ce chapitre qui consiste à extraire un ensemble de patrons avec des caractères jokers. La méthode jugée efficace pour extraire la plus longue sous-chaîne commune est l'utilisation d'un arbre de suffixes.

L'arbre de suffixes [60] est une manière compacte de représenter l'ensemble des suffixes d'un mot sous la forme d'un arbre-automate. C'est une structure de données utile pour l'in-

dexation qui permet de rechercher efficacement des patrons dans un texte. L'arbre de suffixes généralisés [41] adapte l'arbre des suffixes pour que celui-ci puisse travailler avec n séquences au lieu d'une seule. Un tel arbre peut donc détecter les suffixes de n séquences différentes. Il permet en outre de trouver la plus longue sous-chaîne commune entre plusieurs séquences avec une complexité temporelle de $O(NK)$ avec N la taille de l'alphabet, K le nombre de séquences. Il s'agit d'une méthode adaptée au langage fini à l'instar les automates finis déterministes, peut souffrir d'une croissance exponentielle de la taille de sa structure de données sous-jacente lorsque des jokers sont à considérer.

1.4 Unification des problèmes de fouilles

Il est possible d'unifier la terminologie entre problèmes de recherche d'itemsets et de sous-séquences communes. Pour cela nous allons prendre appui sur une représentation matricielle des données. Cette représentation est intuitive et déjà présentée précédemment dans la recherche d'itemsets fréquents. D'après celle-ci, les lignes dénotent les transactions et les colonnes les items de ces transactions. Nous pouvons représenter les instances de recherche de sous-séquences également sous forme matricielle. Par exemple, nous pouvons encoder une base de données de séquences sous forme matricielle en assimilant les caractères aux colonnes et les séquences aux lignes. Les cellules contiendront alors l'ensemble des localisations des caractères au sein des séquences.

Cette représentation permet ainsi de définir différents problèmes comme une recherche de sous-matrice cohérente. Une matrice est cohérente si, pour les lignes et colonnes sélectionnées de l'instance, le choix des valeurs est cohérent (présence d'itemset, localisations cohérentes des caractères au sein des séquences pour la recherche d'une sous-séquence fréquente).

Aussi nous proposons-nous d'étudier dans le prochain chapitre un modèle inspiré du cadre de la programmation par contrainte, afin de définir des problèmes basés sur une telle représentation matricielle. Dans le chapitre 3, nous élaborons des modèles basés sur la PPC afin de résoudre deux problèmes en particulier : la recherche de sous-séquences communes à un jeu de séquences et la recherche d'ordre partiels communs à un ensemble de séquences. Dans le chapitre 4, nous proposons un algorithme mémétique dédié au calcul d'ordre partiel commun. Finalement, au chapitre 5, nous développons deux algorithmes ad-hocs pour l'extraction de patrons communs à un jeu de séquences. Ces patrons servent de base aux instances utilisées lors de cette thèse.

2

Maximal Matrix Problem

Ce chapitre développe un formalisme dédié à l'unification et la formalisation de problèmes de fouille de données, parmi lesquels nous notons la recherche d'itemsets fréquents au sein d'une base de transition, la recherche de sous-chaînes fréquentes au sein d'un ensemble de séquences ou la recherche de sous-séquences fréquentes, etc. Nous pouvons également y modéliser des problèmes de caractérisation à base de motif (sous-chaîne ou sous-séquence) afin, par exemple, de distinguer un ensemble de séquences par rapport à un autre.

Comme nous avons pu le constater dans le précédent chapitre, des problèmes de fouille de données comme la recherche d'itemsets fréquents ou de sous-séquences fréquentes se ressemblent sur plusieurs aspects : les propriétés calculées (couvertures, support, fréquence, fermeture, etc.), une formulation des instances en deux dimensions (transactions et items ou séquences et caractères de l'alphabet). De ce constat, nous proposons une modélisation se basant sur la définition d'une structure matricielle définissant un espace de recherche des motifs. Cet espace de recherche caractérise ses solutions, dans un premier temps, par leur validité vis à vis d'un ensemble de prédicats. [1]. Dans un second temps, les solutions se caractérisent par un critère de maximalité par rapport à une relation d'extension paramétrable. Une matrice est donc maximale s'il n'existe aucune autre matrice solution qui l'étend. Nous proposons un encodage de cette structure matricielle par un formalisme, dénommé MMP, et qui repose sur une modélisation QCSP. Enfin, nous montrons que ce formalisme permet d'encoder directement des problèmes de caractérisation NP-complets.

1. Une liste de prédicats usuels est fourni dont un faisant le lien entre structure matriciel et un jeu de données.

Finalement, nous fournissons une extension à la structure matricielle, afin que celle-ci puisse considérer un n-uplet de matrices plutôt qu'une simple matrice. Cette extension facilite notamment la définition de certains prédicats [2].

2.1 Notations

Nous présentons ici définitions et notations de concepts mathématiques standards qui seront utilisés dans la suite de ce chapitre.

Pour tout entier strictement positif k, nous notons $[\![k]\!]$ l'ensemble $\{1, \cdots, k\}$.

Soit $f : A \to B$ une fonction définie d'un ensemble A vers un ensemble B. Nous appelons domaine de f l'ensemble A, et image de f, dénotée $f(A)$, l'ensemble des éléments de B images d'éléments de A par f : $f(A) = \{f(u)|u \in A\}$. Soit $X \subseteq A$, la restriction de f à X, notée $f|_X$, est la fonction $f|_X : X \to B$ telle que $f|_X(x) = f(x)$ $(x \in X)$.

Un ordre partiel sur A est une relation binaire sur A qui est réflexive, transitive et antisymétrique. Un ordre partiel strict sur A est une relation binaire sur A qui est irréflexive, transitive et antisymétrique. Si $A \subseteq \mathbb{N}$ et $<_B$ est un ordre total sur B, f est dite $<_B$-monotone si et seulement si $(x < y \to f(x) <_B f(y)$ $(x, y \in A))$.

Une relation d'équivalence sur A est une relation binaire sur A réflexive, transitive et symétrique. Si $f : A \to B$ est surjective, la relation d'équivalence induite par f, dénotée par $\equiv$, se définit par $x \equiv y$ si et seulement si $f(x) = f(y)$ $(x, y \in A)$. Pour tout $x \in A$, la classe d'équivalence de x pour $\equiv$, notée $[x]$, est définie par l'ensemble $\{y|x \equiv y\}$. L'ensemble quotient de A par $\equiv$, noté $A/\equiv$, est défini par l'ensemble $\{[x]|x \in A\}$. Une section s est une fonction injective $s : A/\equiv \to A$ telle que $[s(E)] = E$ $(E \in A/\equiv)$. Soit $R = \{r_1, \cdots, r_n\}$ un multi-ensemble de relations unaires sur un même ensemble S, nous notons par $R(x)$ la conjonction des relations de R appliquées à x, i.e. $R(x) \leftrightarrow \bigwedge_{i \in [\![n]\!]} r_i(x)$ $(x \in S)$. Soient R une relation unaire sur A $(R \subseteq A)$ et $\prec$ un ordre partiel strict sur A, R est dite$\prec$-monotone si et seulement si $((R(x) \wedge x \prec x') \to R(x')$ $(x, x' \in A))$; R est dite $\prec$-antimonotone si et seulement si $((\neg R(x) \wedge x \prec x') \to \neg R(x')$ $(x, x' \in A))$.

2.2 Calcul de matrices maximales

Une matrice est une application du produit cartésien de deux ensembles d'éléments appelés lignes et colonnes, respectivement, dans un ensemble d'éléments appelés coefficients. Nous nous intéressons ici aux matrices dont les lignes et colonnes identifient des objets (transactions,

2. Cette extension revêt un côté pratique dans l'expression des prédicats et des instances de MMP même si elle est dispensable théoriquement.

séquences, items, caractères, etc.). Aucun ordre n'est supposé a priori entre ces objets. Nous appelons type matriciel l'ensemble des matrices de dimension quelconque que nous pouvons construire à partir d'ensembles finis de lignes, colonnes et coefficients. Un type matriciel délimitera l'espace de recherche pour la résolution du problème considéré. Nous proposerons en section suivante un encodage de type matriciel par linéarisation sous forme d'un n-uplet de domaines finis, appelé domaine matriciel. Cette modélisation sera à la base du langage MMP.

Définition 7 (Type matriciel)

Soient L_1, L_2 et K des ensembles finis non-vides et $L = (L_1, L_2)$. Le type matriciel dénoté K_L est l'ensemble défini par

$$K_L = \{m | m \ : \ L_1^{(m)} \times L_2^{(m)} \rightarrow K \wedge \varnothing \subset L_i^{(m)} \subseteq L_i \ (i = 1..2)\}$$

Nous appelons lignes, colonnes, coefficients et matrices de K_L les éléments de L_1, L_2, K et K_L respectivement.

Lorsqu'un type matriciel K_L sera symbolisé par M, nous utiliserons la syntaxe $^{(M)}$ pour dénoter les objets liés à ce type : $L_1^{(M)}$ pour L_1, $L_2^{(M)}$ pour L_2, etc. De même, nous utiliserons l'annotation $^{(m)}$ pour faire référence à une matrice m.

Nous appelons dimension d'un type matriciel M la paire $d^{(M)} = (d_1^{(M)}, d_2^{(M)})$ telle que $d_i^{(M)} = |L_i^{(M)}|$ $(i = 1..2)$. Nous appelons dimension d'une matrice m la paire $d^{(m)} = (d_1^{(m)}, d_2^{(m)})$ telle que $d_i^{(m)} = |L_i^{(m)}|$ $(i = 1..2)$ et portée de m le produit cartésien $L_1^{(m)} \times L_2^{(m)}$.

Un problème de fouille vise à déterminer des matrices satisfaisant un certain nombre de propriétés (seuils de fréquence pour les itemsets ; couverture d'une sous-chaîne ; alignement d'une sous-séquence commune ; répétition, ordre et écart entre caractères d'une sous-séquence, etc.). Dans le cadre MMP, nous spécifions ces propriétés à l'aide de contraintes matricielles. Une contrainte sur un type matriciel dénote simplement un ensemble de matrices, i.e. une relation

Concept	Notation
Intervalle	$[\![n]\!]$
Domaine de f	A
Image de f	$f(A)$
Restriction de f à X	$f\|_X$
Équivalence	$x \equiv y$
Classe d'équivalence	$[x]$
Ensemble quotient	$A/\equiv$

TABLEAU 2.1 – Notations associées à une fonction $f : A \rightarrow B$ et $X \subseteq A$.

unaire sur le type matriciel. Ces contraintes matriciels prendront la forme de contraintes n-aires dans le langage MMP. L'objectif est donc pour un problème donné de déterminer une matrice qui soit consistante avec l'ensemble des contraintes imposées.

Nous recherchons typiquement des matrices consistantes maximales au sens où toute extension détermine une matrice inconsistante. Nous entendons par extension l'ajout de lignes, de colonnes et le choix de coefficients pour les lignes et colonnes ajoutées. Différents types d'extension peuvent être envisagées : ajout de lignes uniquement, ajout d'une seule ligne, etc. Dans la pratique, nous traitons séparément l'extension en ligne et l'extension en colonne : par exemple, la fréquence d'un itemset mesure son inextensibilité en ligne relativement à la contrainte de support, la maximalité d'un itemset fréquent mesure son inextensibilité en colonne relativement à la contrainte de seuil de fréquence.

Le maximalité d'une matrice consistante se définit donc comme l'inconsistance par extension, relativement à une relation d'extension choisie. Nous définissons donc une structure matricielle par l'adjonction d'une relation d'extension et d'un ensemble de contrainte à un type matriciel. L'adjonction d'une relation d'extension et d'un ensemble de contraintes sur un type matriciel caractérise ainsi une structure matricielle. Une relation d'extension doit simplement être subsumée par l'opération de restriction fonctionnelle appliquée aux matriciel pour obtenir la sémantique recherchée.

Parmi les contraintes imposées, le cadre MMP permet de distinguer un sous-ensemble de contraintes dites négatives. L'évaluation de la maximalité d'une matrice est restreinte à ces seules contraintes, i.e. nous devons simplement établir l'inconsistance des extensions avec les contrainte négatives. Ce choix répond à différents besoins. Il permet de s'abstenir de tester les contraintes monotones pour l'extension lors du calcul de maximalité. Par définition, toute extension de matrice satisfaisant une contrainte monotone ne peut la violer. Nous pouvons donc classer ces contraintes comme "positives" (i.e. non-négatives) dans une structure matricielle sans perte d'équivalence.

De même, il est vain d'imposer des contraintes négatives dont nous savons qu'elle garantisse la maximalité de toute matrice consistante. C'est le cas notamment des contraintes fonctionnelle fixant la valeur d'une propriété strictement monotone ou anti-monotone, par exemple, fixant la taille ou la portée de la matrice. Les utiliser comme contraintes rend le test de maximalité caduque. Nous recommanderons donc de classer comme positives toute contrainte monotone et toute contrainte fonctionnelle.

Définition 8 (Structure matricielle)
Une structure matricielle est un quadruplet $(M, \prec, C, C^-)$ tel que :

— M est un type matriciel ;

— $\prec$ est un ordre partiel strict sur M vérifiant $(\forall m, m' \in M \ : \ m \prec m' \rightarrow m'_{|L_1^{(m)} \times L_2^{(m)}} = m)$;

— C et C^- sont des ensembles de contraintes unaires sur M ($\forall c \in C, \ c \subseteq M$) [3] et tels que $\varnothing \subseteq C^- \subseteq C$.

$\prec$ est appelée relation d'extension et les éléments de C^- sont appelés contraintes négatives de C.

Par abus de notation, nous symboliserons une structure par son type matriciel M et nous annoterons par $^{(M)}$ les éléments de la structure.

Définition 9 (Matrices maximales d'une structure)
Soient $\mathcal{M} = (M, \prec, C, C^-)$ une structure matricielle, et $m \in M$. m est dite matrice maximale sur $\mathcal{M}$ si et seulement si m satisfait

$$(C(m) \wedge \forall m' \in M \ : \ m \prec m' \rightarrow \neg C^-(m'))$$

Nous notons $\texttt{MAX}(\mathcal{M})$ l'ensemble des matrices maximales sur $\mathcal{M}$.

Les sections suivantes présentent différents types matriciels, une forme d'extension et des contraintes matricielles pour des problèmes pratiques.

2.2.1 Extension incrémentale en ligne

Dans la pratique, on s'intéresse aux relations d'extension en ligne (i.e. à colonnes constantes) ou en colonne (i.e. à lignes constantes). Par exemple, la fréquence d'un itemset caractérise son extensibilité en ligne, sa fermeture caractérise son extensibilité en colonne.

3. Une contrainte dénote ici l'association d'une symbole et d'une relation unaire, différentes contraintes pouvant dénoter la même relation. Par abus de notation, nous assimilerons une contrainte à sa relation lorsqu'il n'y aura pas d'ambigüité.

Définition 10 (Extension en ligne)
Soient une structure matricielle $(M, \prec, C, C^-)$ et $k \in \mathbb{N}^+$. La k-extension en ligne, dénotée $\prec_k$, est la relation d'extension vérifiant

$$\forall m, m' \in M \ : \ m \prec_k m' \leftrightarrow$$
$$(L_2^{(m)} = L_2^{(m')}) \wedge (L_1^{(m)} \subset L_1^{(m')}) \wedge (|L_1^{(m')}| \leqslant |L_1^{(m)}| + k) \wedge (m'_{|L_1^{(m)} \times L_2^{(m)}} = m).$$

Nous appelons extension incrémentale en ligne la relation $\prec_1$.

Par la suite, nous allons présenter des prédicats qui sont monotones ou anti-monotones par extension en ligne. Lorsqu'une structure matricielle ne contient que des contraintes négatives anti-monotones en ligne, on peut se restreindre à une relation d'extension plus faible tout en préservant l'ensemble des matrices maximales.

Proposition : Soient $\mathcal{M} = (M, \prec_1, C, C^-)$ et $\mathcal{M}' = (M, \prec_k, C, C^-)$ deux structures matricielles telles que pour tout $c \in C^-$, c est $\prec_1$-antimonotone, et $k \geqslant 1$, alors $\texttt{MAX}(\mathcal{M}) = \texttt{MAX}(\mathcal{M}')$.

Démonstration.

(1) Par définition de MAX et du fait que C soit commun à $\mathcal{M}$ et $\mathcal{M}'$, nous avons $C(x)$ pour tout $x \in \texttt{MAX}(\mathcal{M}) \cup \texttt{MAX}(\mathcal{M}')$.

(2) Toute conjonction de contraintes $\prec$-monotones est trivialement $\prec$-monotone, donc la négation d'une disjonction de contraintes $\prec$-antimonotone est $\prec$-monotone, autrement dit, une conjonction de contraintes $\prec$-antimonotone est $\prec$-monotone. Donc C^- est $\prec$-antimonotone par hypothèse sur ses éléments.

(3) De (1) et du fait que $\prec_k$ subsume $\prec_1$, nous déduisons directement que $\texttt{MAX}(\mathcal{M}') \subseteq \texttt{MAX}(\mathcal{M})$.

(4) Supposons $\texttt{MAX}(\mathcal{M}) \nsubseteq \texttt{MAX}(\mathcal{M}')$. Il existe donc $x, z \in M$ tels que $C(x), (\forall y \succ_1 x \ : \neg C^-(y))$ et $(x \prec_k z \wedge C^-(z))$. Si $x \prec_1 z$ il y a contradiction. Donc il existe $y_1 \cdots y_n$ ($n \geqslant 1$) tels que $x \prec_1 y_1$, $y_i \prec_1 y_{i+1}$ ($1 \leqslant i < n-1$) et $y_n \prec_1 z$. Par $\prec_1$-antimonotonie de C^- (2), nous en déduisons par récurrence sur le chemin $xy_1 \cdots y_n z$ que $\neg C^-(z)$ ce qui est une contradiction. Donc $\texttt{MAX}(\mathcal{M}) \subseteq \texttt{MAX}(\mathcal{M}')$.

□

Lorsqu'une structure matricielle comprend des contraintes monotones pour l'extension incrémentale nous pouvons placer ces contraintes dans la classe positive tout en préservant l'ensemble des matrices maximales.

Proposition : Soient $\mathcal{M} = (M, \prec_1, C, C^-)$, $\mathcal{M}' = (M, \prec_1, C, C'^-)$ tels que $C'^- \subset C^-$, $C^- \backslash C'^- = \{c\}$ et c $\prec_1$-monotone, alors $\texttt{MAX}(\mathcal{M}) = \texttt{MAX}(\mathcal{M}')$.

Démonstration.
De manière évidente, $\texttt{MAX}(\mathcal{M}') \subseteq \texttt{MAX}(\mathcal{M})$. Soit $x \in \texttt{MAX}(\mathcal{M})$, x vérifie $C(x)$ et donc $c(x)$. Par $\prec_1$-monotonie de c, nous avons $(\forall y \succ_1 x \ : \ c(y))$ donc $(\forall y \succ_1 x \ : \ \neg C^-(y)) \leftrightarrow (\forall y_1 \succ x \ : \ \neg C'^-(y))$. Donc $x \in \texttt{MAX}(\mathcal{M}')$. □

Nous allons donc considérer des structures matricielles dont la relation d'extension est incrémentale en ligne et, par défaut, nous classerons positives les contraindre monotones et fonctionnelles.

2.2.2 Prédicat de base de données

Tout problème de fouille de données repose sur une base de données qui présuppose le choix d'un format relationnel. Nous formalisons la notion de base de données sur un type matriciel et nous présentons un prédicat de conformité. Nous illustrerons ensuite types et bases de données matricielles pour des problèmes usuels. Les définitions feront référence à une structure matricielle $M = K_L$.

Définition 11 (Base de données)
Une base de données sur M est une fonction $B : \underset{i=1..2}{\times} L_i^{(M)} \mapsto 2^{K^{(M)}}$.

Nous remarquons qu'une base de données permet l'affectation d'un domaine de coefficients vide pour n'importe quelle cellule.

Définition 12 (Prédicat de base de données)
Soit B une base de données sur M. Le prédicat de base de données, dénoté db_B, est défini par

$$\forall m \in M, \ \mathrm{db}_B(m) \leftrightarrow \bigwedge_{(i,j) \in \times L_1^{(m)} \times L_2^{(m)}} (k^{(m)}(i,j) \in B(i,j))$$

Toute matrice portant sur une cellule à domaine vide dans une base de données B sera inconsistante avec la contrainte db_B. De manière évidente, toute contrainte de base de données est $\prec_1$-antimonotone.

2.2.3 Type matriciel et prédicats pour fouille d'itemsets

Les problèmes de recherche d'itemsets (fréquents, clos, maximaux) se définissent à partir d'un ensemble d'items $\mathcal{I}$ et d'un ensemble de n transactions $\mathcal{T}$ tels que $\mathcal{T} = \{(i,t)|t \subseteq \mathcal{I} \wedge i \in [\![n]\!]\}$. On peut alors assimiler les lignes du type matriciel aux identifiants de transactions ($L_1^{(M)} = \mathcal{T}$), les colonnes aux items ($L_2^{(M)} = \mathcal{I}$) et le domaine de coefficients K à un ensemble singleton quelconque $\{u\}$. Il suffit alors d'associer à chaque cellule (item,transaction) dans la base de données matricielles B l'ensemble vide si l'item n'apparaît pas dans la transaction. Toute matrice portant sur cette cellule sera alors, par définition, non conforme à la base de données. La figure 2.1 illustre une base de données matricielle pour le type $\{u\}_{(\mathcal{T},\mathcal{I})}$.

	i_1	i_2	i_3	i_4
t_1	$\{u\}$	$\{u\}$	$\varnothing$	$\{u\}$
t_2	$\varnothing$	$\{u\}$	$\{u\}$	$\varnothing$
t_3	$\{u\}$	$\{u\}$	$\{u\}$	$\{u\}$
t_4	$\{u\}$	$\varnothing$	$\varnothing$	$\varnothing$

FIGURE 2.1 – Base de données matricielle pour un problème de recherche d'itemsets.

Le choix de cette modélisation garantit que toute matrice conforme à la base de données correspond à un itemset dont le support inclut nécessairement les transactions dans la portée de la matrice. L'objectif classique en fouille d'itemsets étant de déterminer des itemsets fréquents, c'est-à-dire dont la cardinalité du support excède un seuil prédéfini, nous introduisons un prédicat de fréquence minimale à cet effet.

Définition 13 (Prédicat de fréquence minimale)

Soit $f \in [\![d_1^{(M)}]\!]$. Le prédicat de fréquence minimale f, dénoté $\mathrm{freq}_{\{\geqslant,f\}}$, est défini par

$$\forall m \in M,\ \mathrm{freq}_{\{\geqslant,f\}}(m) \leftrightarrow f \leqslant d_1^{(m)}$$

De manière duale, on introduit un prédicat de cardinalité maximale portant sur les colonnes et permettant de contraindre la taille des itemsets.

Définition 14 (Prédicat de cardinalité maximale)

Soit $w \in [\![d_2^{(M)}]\!]$. Le prédicat de cardinalité maximale w, dénoté $\mathrm{card}_{\{w,\geqslant\}}$, est défini par

$$\forall m \in M,\ \mathrm{card}_{\{\leqslant,w\}}(m) \leftrightarrow w \geqslant d_2^{(m)}$$

De manière générale, nous noterons $\pi_{\{R,v\}}$ pour un prédicat de mesure π, une relation R et une valeur v tels que $\pi(m)Rv$. Les contraintes de fréquence minimale et de cardinalité maximale sont respectivement $\prec_1$-monotone et $\prec_1$-antimonotone.

2.2.4 Type matriciel et prédicat pour recherche de motifs sans répétitions

Les problèmes de recherche de motifs visent à extraire des ensembles de caractères communs à des séquences et soumis à diverses contraintes (séquencement, espacement, etc.) : sous-chaîne commune, alignement, sous-séquence commune, etc. Étant donné un jeu de séquences S construit sur un alphabet Σ, un type matriciel élémentaire consiste à assimiler les lignes aux séquences ($L_1^{(M)} = S$), les colonnes aux caractères de l'alphabet ($L_2^{(M)} = \Sigma$), et le domaine de coefficients à l'ensemble des localisations de caractère possibles dans les séquences ($K = [\![\max_{s \in S} |s|]\!]$). La figure 2.2 illustre une base de données matricielle pour le type $[\![\max_{s \in S} |s|]\!]_{(S,\Sigma)}$.

séquences	caractères			
	A	C	D	E
s_1 = AACCDDAACCDD	$\{1,2,7,8\}$	$\{3,4,9,10\}$	$\{5,6,11,12\}$	$\varnothing$
s_2 = DDCCAADDCDD	$\{5,6\}$	$\{3,4,9\}$	$\{1,2,7,8,10,11\}$	$\varnothing$
s_3 = AACCAADDEDD	$\{1,2,5,6\}$	$\{3,4\}$	$\{7,8,10,11\}$	$\{9\}$
s_4 = DDAADDAA	$\{3,4,7,8\}$	$\varnothing$	$\{1,2,5,6\}$	$\varnothing$

FIGURE 2.2 – Base de données matricielle pour un problème de recherche de motifs sans répétitions.

Le choix de cette modélisation garantit que toute matrice conforme à la base de données correspond à ensemble de caractères qui sont tous enracinés dans les séquences figurant dans la portée de la matrice. Lorsque l'on recherche des sous-chaînes ou sous-séquences communes on doit alors s'assurer que les localisations des caractères déterminent sur chaque séquence le même ordre total. Nous introduisons à cet effet un prédicat d'ordre total entre les colonnes d'une matrice qui force le même ordre total sur chaque ligne de la portée basé sur les coefficients choisis.

Définition 15 (Prédicat d'ordre total entre colonnes)

Soit $<$ un ordre total sur $K^{(M)}$. Le prédicat d'ordre total basé sur $<$, dénoté $\mathrm{total}_<$, est défini par

$$\forall m \in M,\ \mathrm{total}_<(m) \leftrightarrow \bigwedge_{j \neq j' \in L_2^{(m)}} \Big(\bigwedge_{i \in L_1^{(m)}} (k^{(m)}(i,j) < k^{(m)}(i,j')) \vee \bigwedge_{i \in L_1^{(m)}} (k^{(m)}(i,j) > k^{(m)}(i,j')) \Big)$$

De manière évidente, le prédicat d'ordre total entre colonnes est $\prec_1$-antimonotone.

2.2.5 Type matriciel et prédicat pour recherche de motifs avec répétitions

Le type matriciel présenté ci-dessus permet uniquement de calculer des motifs sans répétitions de caractères. Pour plus de flexibilité on peut associer à chaque colonne non plus un caractère de l'alphabet mais une occurrence de caractère, autrement dit, $L_2^{(M)}$ correspond à un multi-ensemble de caractères de l'alphabet. On peut définir $L_2^{(M)}$ comme suit :

— $L_2^{(M)} = \bigcup_{a \in \Sigma} E(a)$,

— $E(a) = \{(a,1), \cdots, (a,n_a)\}$ pour $a \in \Sigma$ et n_a un nombre d'occurrences de a paramétrable. [4]

La figure 2.3 illustre une base de données matricielle pour le type $[\![\max_{s \in S} |s|]\!]_{(S, \bigcup_{a \in \Sigma} E(a))}$ où $n_a \leqslant 2$ pour tout $a \in \Sigma$.

séquences	caractères						
	A	A	C	C	D	D	E
s_1 = AACCDDAACCDD	$\{1,2,7,8\}$	$\{1,2,7,8\}$	$\{3,4,9,10\}$	$\{3,4,9,10\}$	$\{5,6,11,12\}$	$\{5,6,11,12\}$	$\varnothing$
s_2 = DDCCAADDCDD	$\{5,6\}$	$\{5,6\}$	$\{3,4,9\}$	$\{3,4,9\}$	$\{1,2,7,8,10,11\}$	$\{1,2,7,8,10,11\}$	$\varnothing$
s_3 = AACCAADDEDD	$\{1,2,5,6\}$	$\{1,2,5,6\}$	$\{3,4\}$	$\{3,4\}$	$\{7,8,10,11\}$	$\{7,8,10,11\}$	$\{9\}$
s_4 = DDAADDAA	$\{3,4,7,8\}$	$\{3,4,7,8\}$	$\varnothing$	$\varnothing$	$\{1,2,5,6\}$	$\{1,2,5,6\}$	$\varnothing$

FIGURE 2.3 – Base de données matricielle pour un problème de recherche de motifs avec répétitions.

Sans autre prédicat, un motif (matrice) conforme à la base de données peut contenir plusieurs occurrences d'un même caractère (plusieurs colonnes équivalentes) qui ont la même localisation (coefficient) sur une séquence. Pour garantir que deux occurrences d'un même caractère au sein d'un motif ne coïncident pas sur une séquence, nous introduisons un prédicat de séquencement d'occurrences.

Définition 16 (Prédicat de séquencement entre colonnes équivalentes)

Soient $<$ un ordre total sur $K^{(M)}$, et $\equiv$ une relation d'équivalence sur $L_2^{(M)}$. Le prédicat d'équivalence entre colonnes basé sur $<$ et $\equiv$, dénoté $\text{equivalence}_{\{\equiv,<\}}$, est défini par

$$\forall m \in M,\ \text{equivalence}_{\{\equiv,<\}}(m) \leftrightarrow$$
$$\bigwedge_{j \neq j' \in L_2^{(m)}} \Big((j \equiv j') \rightarrow \big(\bigwedge_{i \in L_1^{(m)}} (k^{(m)}(i,j) < k^{(m)}(i,j')) \vee \bigwedge_{i \in L_1^{(m)}} (k^{(m)}(i,j) > k^{(m)}(i,j')) \big) \Big)$$

De manière évidente, toute contrainte de séquencement entre colonnes équivalentes est $<_1$-antimonotone.

4. Le nombre maximum d'occurrences d'un caractère a par séquence étant connu, il est inutile de choisir une valeur supérieure pour n_a.

2.2.6 Type matriciel pour recherche de motifs à base de patrons

Un type matriciel basé sur les caractères ou occurrences de caractère est d'une granularité très fine et peut induire des ensembles $L_2^{(M)}$ et $K^{(M)}$ de taille prohibitive [5]. Rien n'empêche de modéliser des patrons précalculés en les assimilant aux colonnes du type matriciel. Les coefficients peuvent, par exemple, représenter les positions de départ des patrons dans les séquences. Sur cette base, tous les prédicats présentés précédemment peuvent être utilisés pour rechercher des ensembles de patrons structurés. La figure 2.4 décrit une base de données pour une telle structure matricielle où les patrons sont de simples sous-chaînes.

séquences	patrons		
	AA	CC	DD
s_1 = AACCDDAACCDD	{1,7}	{4,9}	{5,11}
s_2 = DDCCAADDCDD	{5}	{3}	{1,7,10}
s_3 = AACCAADDEDD	{1,5}	{3}	{7,10}
s_4 = DDAADDAA	{3,7}	$\varnothing$	{1,5}

FIGURE 2.4 – Base de données matricielle pour un problème de recherche de motifs à base de patrons.

2.2.7 Caractérisation et prédicat de couverture minimale

Le prédicat de couverture force les matrices solutions à couvrir un ensemble de lignes prédéfini - il pourrait porter sur des colonnes. Il est utile dans des problèmes de caractérisation lorsque nous souhaitons différencier deux classes de lignes à l'aide d'une matrice solution. Par exemple, on recherchera un motif couvrant la classe de séquences positives et si possible ne couvrant pas la classe de séquences négatives.

Définition 17 (Prédicat de couverture minimale)

Soit $R \subseteq L_1^{(M)}$. Le prédicat de couverture minimale R, dénoté $\mathrm{cover}_{\{\supseteq,R\}}$, est défini par

$$\forall m \in M,\ \mathrm{cover}_{\{\supseteq,R\}}(m) \leftrightarrow R \subseteq L_1^{(m)}$$

De manière évidente, le prédicat de couverture minimale est $\prec_1$-monotone.

5. Dans le cas des séquences d'ADN, l'alphabet est de petite taille (4 caractères) mais les séquences extrêmement longues, les classes d'équivalences et les domaines de localisations dans la base de données seront donc de grande taille. Dans les séquences protéiques, on aura des classes d'équivalence en plus grand nombre (l'alphabet contient 20 acides aminés) mais de plus petite taille et avec des domaines de localisations plus petits. Pour les familles de protéines LEAP et SHSP considérées dans nos expérimentations, la taille des séquences varie d'environ 60 à 250 caractères.

2.2.8 Illustration

La figure 2.5 illustre un type matriciel pour la recherche d'ensembles de patrons discriminants. La base de données modélise un ensemble de patrons prélocalisés dans un jeu de séquences étiquetées positive ou négative - cet exemple s'inspire des expérimentations menées au chapitre 5. La base de données spécifie l'ensemble des localisations de départ du patron dans la séquence pour chaque paire (patron, séquence) possible. Un problème envisageable est alors de trouver un séquencement de patrons présent dans les séquences positives mais pas dans les séquences négatives. Pour ce faire, on impose la contrainte de conformité à la base de données B, la contrainte d'ordre total entre colonnes et la contrainte de couverture minimale des séquences positives S^+. Sachant que les deux premiers prédicats sont $\prec_1$-antimonotones et le dernier $\prec_1$-monotone, on se restreint à l'extension en ligne incrémentale et nous pouvons classer "positive" la contrainte de couverture. L'objectif est donc de calculer des matrices maximales pour la structure résultante. La structure matricielle associé au problème est la suivante :
$\mathcal{M} = (\llbracket 10 \rrbracket_{(\{s_1,s_2,s_3,s_4\},\{AA,CC,DD\})}, \prec_1, \{\mathrm{db}_B, \mathrm{total}_<, \mathrm{cover}_{\{S^+,\subseteq\}}\}, \{\mathrm{db}_B, \mathrm{total}_<\})$

séquences	+/-	patrons		
		AA	CC	DD
s_1 = AACCDDAACCDD	+	{1,7}	{4,9}	{5,11}
s_2 = DDCCAADDCDD	+	{5}	{3}	{1,7,10}
s_3 = AACCAADDEDD	-	{1,5}	{3}	{7,10}
s_4 = DDAADDAA	-	{3,7}	$\varnothing$	{1,5}

FIGURE 2.5 – Une base de séquences constituée des positions de départ de différents patrons.

La matrice de dimension $(2, 2)$, d'étiquettes $(\{s_1, s_2\}, \{\mathtt{CC}, \mathtt{DD}\})$ et de coefficients $(s_1, \mathtt{CC}) = 9$, $(s_1, \mathtt{DD}) = 5$, $(s_2, \mathtt{CC}) = 3$ et $(s_2, \mathtt{DD}) = 1$ est maximale : ses coefficients sont conformes aux localisations autorisées, elle couvre les lignes étiquetées positives, ses colonnes satisfont l'ordre $\{\mathtt{CC} > \mathtt{DD}\}$, aucune combinaison de localisations autorisées pour CC et DD ne satisfait cet ordre sur s_3, et CC n'admet aucune localisation sur s_4. Autrement dit, la matrice est consistante avec les contraintes de base de données, de couverture minimale positive et d'ordre total entre colonnes, toute extension sur s_3 viole la contrainte d'ordre total, et toute extension sur s_4 viole la contrainte de base de données. La matrice est donc consistante et maximale pour l'extension incrémentale en ligne, et donc, pour toute extension en ligne par antimonotonie de la classe de contraintes négatives.

2.2.9 Typologie des prédicats matriciels

Nous pouvons dresser une typologie des prédicats matriciels selon que leur sémantique dépende ou non des lignes (e.g., cardinalité), des colonnes (e.g., fréquence, couverture) des coefficients (cardinalité, fréquence, couverture), des étiquettes des lignes (e.g., ordre total, sé-

quencement entre colonnes équivalentes), etc. Nous remarquons en particulier que certains des prédicats présentés s'appliquent uniformément à toute portée et traitent lignes ou colonnes indistinctement. Par exemple, le prédicat d'ordre total détermine l'existence d'un ordre des coefficients colonnes de la portée d'une matrice quelle que soit la portée et en vérifiant les mêmes propriétés. Ce n'est pas le cas du prédicat $\mathrm{cover}_{\{\supseteq,R\}}$ dont l'interprétation est conditionnée à la portée de la matrice à évaluer ($R \subseteq d_1^{(M)}$ ou $R \nsubseteq d_1^{(M)}$). Même si l'usage de prédicats uniformes fait sens dans la pratique, le cadre MMP n'empêche pas l'usage de prédicats conditionnels. À titre d'exemple, nous pouvons imaginer un prédicat de parité qui sera satisfait par toute matrice à nombre de lignes paires, ou encore un prédicat "s'appliquant" à deux lignes particulières.

2.3 Le langage MMP

Type et structure matricielle fournissent une spécification abstraite de problèmes de fouille de données. Nous présentons ici un problème de décision, dénommé Maximal Matrix Problem (MMP), et le formalisme associé qui permet de modéliser toute structure matricielle afin d'en déterminer les matrices maximales. Nous donnerons en section suivante un encodage sous forme de CSP quatifié (QCSP).

La modélisation MMP repose sur la décomposition d'une matrice en cinq composants élémentaires représentant ses dimensions, lignes, colonnes et coefficients. Les composants sont chacun des n-uplets dimensionnés pour pouvoir représenter toute taille de matrice du type considéré. Nous associons donc à chaque type matriciel un n-uplet de domaines finis (un domaine par dimension, un domaine d'étiquettes par positions d'étiquetage possible, un domaine de coefficients par cellule possible). Nous associons donc à ce n-uplet de domaine, appelé domaine matriciel, une relation n-aire (par produit cartésien) servant de contrainte afin de mettre en bijection les "solutions" résultantes et le type matriciel.

Nous présentons à cet effet trois contraintes constitutives de toute instance MMP qui en permettent l'interprétation. La première est une contrainte de dimensionnement qui restreint l'interprétation des composantes étiquetage et coefficients d'un n-uplet du domaine matriciel à la dimension modélisée dans le n-uplet. La méthode consiste simplement à projeter chaque composante sur ses premières valeurs. Toute contrainte sur le domaine matriciel modélisant la relation d'extension doivent donc être interprétée par dimensionnement.

La seconde contrainte pré-imposée est une contrainte d'injection qui empêche le choix d'une même étiquette dans chaque composante dimensionnée. Il s'agit donc de rendre les composants d'étiquetage injectives. Nous garantissons alors les n-uplets satisfaisant cette contrainte modélisent exactement les matrices du type. Plus précisément, chaque matrice se modélise par une classe de n-uplets qui sont équivalents modulo permutation matricielle en ligne ou en colonne.

La troisième contrainte est optionnelle mais permet de casser ces symétries. Il s'agit d'une

contrainte de sectionnement qui fixe et trie les étiquettes hors portée ainsi que les coefficients hors portée. Cette contrainte repose sur le choix arbitraire d'un ordre sur chaque ensemble d'étiquettes et d'une valeur par défaut de coefficient.

Nous présentons ci-dessous ces contraintes et l'isomorphisme résultant entre structure matricielle et instance de MMP. Les définitions feront référence à un type matriciel M.

Définition 18 (Domaine matriciel)

Le domaine matriciel de type M, dénoté $D^{(M)}$, est le n-uplet

$$D^{(M)} = (D_1, D_2, L_{11}, \cdots, L_{1d_1}, L_{21}, \cdots, L_{2d_2}, K_1, \cdots, K_{d_1.d_2})$$

de taille $t = 2 + d_1^{(M)} + d_2^{(M)} + d_1^{(M)} d_2^{(M)}$ tel que

- $D_i = [\![d_i^{(M)}]\!]$ $(i = 1..2)$;
- $L_{ij} = L_i^{(M)}$ $(i = 1..2, j \in [\![d_i^{(M)}]\!])$;
- $K_k = K^{(M)}$ $(k \in [\![d_1^{(M)}.d_2^{(M)}]\!])$.

On note $\Delta^{(M)} = \underset{i \in [\![t]\!]}{\times} D_i^{(M)}$.

Nous associons à tout élément x de $\Delta^{(M)}$ le triplet, dénoté $(d^{(x)}, l^{(x)}, k^{(x)})$, obtenu par projection de x sur les composantes dimension, portée, et coefficients, et $d^{(x)}$ et $l^{(x)}$ par les paires $(d_1^{(x)}, d_2^{(x)})$ et $(l_1^{(x)}, l_2^{(x)})$ par projection sur chacune des dimensions du type. À noter que la linéarisation des composants étiquetage et coefficients suppose un système d'indexation couplant paires d'indices (i, j) et indices k de coefficients. Nous nous cantonnerons ici à utiliser la notation conventionnelle $k^{(x)}(i, j)$, indépendant de l'implantation choisie.

Toute matrice m de M se définit par son graphe, i.e. par l'ensemble des paires $((l_1, l_2), k)$ telles que $k = m(l_1, l_2)$ pour tout $(l_1, l_2) \in L_1^{(m)} \times L_2^{(m)}$. En indexant arbitrairement les étiquettes de $L_i^{(m)}$ sur l'intervalle $[\![d_i^{(m)}]\!]$ $(i = 1..2)$, m se définit de manière équivalente par l'ensemble des paires $((i_1, i_2), k)$ pour toute paire (i_1, i_2) d'indices dans $[\![d_1^{(m)}]\!] \times [\![d_2^{(m)}]\!]$. Nous pouvons donc représenter toute matrice m par un élément x de $\Delta^{(m)}$ en fixant $d^{(x)}$ à $d^{(m)}$, en modélisant la réciproque de l'indexation de $L_i^{(m)}$ par la projection de $l_i^{(x)}$ sur $[\![d_i^{(x)}]\!]$, puis en modélisant la matrice des coefficients $((i_1, i_2), k)$ de m par la projection de $k^{(x)}$ sur $[\![d_1^{(x)}]\!] \times [\![d_2^{(x)}]\!]$. La modélisation consiste donc à conditionner l'interprétation de $l^{(x)}$ et $k^{(x)}$ à $d^{(x)}$.

L'indexation des étiquettes étant arbitraire, une même matrice m se modélise sur $\Delta^{(M)}$ par autant d'éléments x que le nombre de permutations possibles en ligne et en colonne sur la portée $[\![d_1^{(m)}]\!] \times [\![d_2^{(m)}]\!]$. Qui plus est, elle laisse libre choix quant à l'affectation des valeurs d'étiquette ou de coefficients hors portée de chaque n-uplet x. Une matrice m caractérise donc une classe d'équivalence sur $\Delta^{(M)}$ qui regroupe tous les éléments identiques par restriction de l'interprétation à la portée de m et modulo la permutation des lignes et colonnes dans la portée.

Réciproquement, tout élément de x de $\Delta^{(M)}$ dont l'étiquetage $l_i^{(x)}$ restreint à $[\![d_i^{(x)}]\!]$ $(i = 1..2)$ est injectif, détermine une matrice m par composition fonctionnelle de $k^{(x)}$ à la paire d'indexations correspondantes. Un type matriciel M est donc en bijection avec le quotient de $\Delta^{(M)}$ restreint aux éléments à étiquetage injectif par conditionnement pour la relation d'équivalence sus-mentionnée. Nous présentons ci-dessous la contrainte $I^{(M)}$ d'indexation et la surjection associée de $I^{(M)}$ vers M.

Lemme 1 (Indexation conditionnelle)**.** *Soit* $I^{(M)} = \{x | x \in \Delta^{(M)}$ *et* $l_i^{(x)} | [\![d_i^{(x)}]\!]$ *injective* $(i = 1..2)\}$. *Soit* $x \in I^{(M)}$, $i \in \{1, 2\}$, *la fonction* $g_i^{(x)} : l_i^{(x)}([\![d_i^{(x)}]\!]) \to [\![d_i^{(x)}]\!]$ *telle que* $g_i^{(x)}(a) = k \leftrightarrow l_i^{(x)}(k) = a$ *(*$k \in [\![d_i^{(x)}]\!]$, $a \in l_i^{(x)}([\![d_i^{(x)}]\!])$*) existe, est unique et bijective.*

Démonstration. Trivialement, toute fonction $f : A \to B$ injective détermine la bijection $g : f(A) \to A$ telle que $g(b) = a \leftrightarrow f(a) = b$ $(b \in f(A), a \in A)$. Pour $x \in I^{(M)}$, $i \in \{1, 2\}$, la fonction $l_i^{(x)}$ restreinte à $[\![d_i^{(x)}]\!]$ est injective par définition de $I^{(M)}$ et le résultat en découle. □

Lemme 2 (Composition de matrice par indexation conditionnelle)**.** *La fonction, dénotée* $k \circ g^{(M)}$, *telle que* $k \circ g^{(M)} : I^{(M)} \to M$ *et* $k \circ g^{(M)}(x) = k^{(x)} \circ (g_1^{(x)}, g_2^{(x)})$ *(*$x \in I^{(M)}$*) est surjective.*

Démonstration.
(1) Soit $x \in I^{(M)}$. Nous notons $L_i^{(x)} = l_i([\![d_i(x)]\!])$ $(i = 1..2)$. Il découle du Lemme 1 que la fonction $g : \underset{1 \leqslant i \leqslant 2}{\times} L_i^{(x)} \to \underset{1 \leqslant i \leqslant 2}{\times} [\![d_i^{(x)}]\!]$ telle que $g^{(x)}(l_1, l_2) = (g_1^{(x)}(l_1), g_2^{(x)}(l_2))$ est bien définie et bijective. Donc $k \circ g^{(M)}(x) = k^{(x)} \circ g^{(x)}$ est une fonction de $(\underset{1 \leqslant i \leqslant 2}{\times} L_i^{(x)} \to K^{(M)})$ bien définie. Puisque $L_i^{(x)} \subseteq L_i^{(M)}$ et $L_i^{(x)} \neq \varnothing$ $(d_i^{(x)} \geqslant 1)$ pour $i = 1..2$, nous en déduisons que $k \circ g^{(M)}$ est bien définie.
(2) Soit $m \in M$. Soit $<$ une paire $(<_1, <_2)$ telle que $<_i$ est un ordre total (arbitraire) sur $L_i^{(M)}$ $(i = 1..2)$. Il existe donc une unique fonction $l_i^{(x)} : [\![d_i^{(m)}]\!] \to L_i^{(m)}$ qui est $<_i$-monotone $(i = 1..2)$ et par conséquent bijective. Soit $k : \underset{1 \leqslant i \leqslant 2}{\times} [\![d_i^{(M)}]\!] \to K^{(M)}$ telle que $k(i_1, i_2) = k^{(m)}(l_1(i_1), l_2(i_2))$ alors $k \circ (l_1^{-1}, l_2^{-1}) = m$. Pour tout $x \in D^{(M)}$ tel que :

- $d^{(x)} = d^{(m)}$,
- $l_i^{(x)} | [\![d_i^{(x)}]\!] = l_i$ $(i = 1..2)$,
- $k^{(x)} | \underset{1 \leqslant i \leqslant 2}{\times} [\![d_i^{(x)}]\!] = k$,

nous avons $x \in I^{(M)}$ et $k \circ g^{(M)}(x) = m$. Donc $k \circ g^{(M)}$ est surjective.

□

La fonction $k \circ g^{(M)}$ induit donc une relation d'équivalence sur $I^{(M)}$ et permet d'établir une bijection entre le quotient de $I^{(M)}$ et le type matriciel M.

Corollaire 1 (Stabilité de composition par permutation)**.** *Soit $\equiv^{(M)}$ la relation d'équivalence sur $I^{(M)}$ induite par $k \circ g^{(M)}$:*

$$x \equiv^{(M)} y \leftrightarrow k \circ g^{(M)}(x) = k \circ g^{(M)}(y)\ (x, y \in I^{(M)}).$$

La fonction $f^{(M)} : M \to I^{(M)}/\equiv$ définie par

$$f^{(M)}(m) = [x] \leftrightarrow m = k \circ g^{(M)}(x)\ (m \in M,\ x \in I^{(M)})$$

est bijective.

Démonstration. $k \circ g^{(M)}$ étant surjective (Lemme 2), $\equiv^{(M)}$ est bien une relation d'équivalence et $f^{(M)}$ est bien définie et bijective. □

Il en découle que tout système de représentants des classes de $\equiv^{(M)}$ est en bijection avec M. Nous définissons ci-dessous une contrainte $S^{(M)}_{<w}$ de sectionnement possible.

Lemme 3. *Soit $M = K_L$ un type matriciel, $<_i$ un ordre total sur $L_i^{(M)}$ ($i = 1..2$), $<= (<_1, <_2)$ et $w \in K$. La fonction $s^{(M)}_{<w} : I^{(M)}/\equiv^{(M)} \to I^{(M)}$ définie par*

$$\forall x, y \in I^{(M)} : s^{(M)}_{<w}([x]) = y \leftrightarrow \begin{cases} y \in [x] \\ l_i^{(y)} | [\![d_i^{(y)}]\!] \textit{ est } <_i \textit{-monotone } (i = 1..2) \\ l_i^{(y)} | ([\![d_i^{(M)}]\!] \backslash [\![d_i^{(y)}]\!]) \textit{ est } <_i \textit{-monotone } (i = 1..2) \\ l_i^{(y)}(d_i^{(M)}) = L_i^{(M)}\ (i = 1..2) \\ \forall j \in [\![d_i^{(M)}]\!]^2,\ (j_1 > d_1^{(y)} \vee j_2 > d_2^{(y)}) \to k^{(y)}(j) = w \end{cases}$$

est une section. On notera $S^{(M)}_{<w}$ l'image de $s^{(M)}_{<w}$ ($S^{(M)}_{<w} = s^{(M)}_{<w}(I^{(M)}/\equiv^{(M)})$).

La composée $s^{(M)}_{<w} \circ f^{(M)}$ détermine donc la bijection $F : 2^M \to 2^{S^{(M)}_{<w}}$ telle que $F(X)$ est l'image de X par $s^{(M)}_{<w} \circ f^{(M)}$. L'ensemble $2^{S^{(M)}_{<w}}$ des relations unaires sur $S^{(M)}_{<w}$ est stable par intersection, union, et par complément dans $S^{(M)}_{<w}$. F constitue donc un isomorphisme entre $(2^M, \cup, \cap, C)$ et $(2^{S^{(M)}_{<w}}, \cup, \cap, C_{S^{(M)}_{<w}})$ et par extension entre les algèbres de relations n-aires.

Nous pouvons donc coupler instances de MMP et structures matricielles par cet isomorphisme en associant les relations d'extension et les contraintes deux à deux. Nous présentons ci-dessous le problème MMP de manière indépendante (par abus de notation, nous continuons d'utiliser $s_{<w}^{(M)} \circ f^{(M)}$ en lieu et place de F).

Définition 19 (MMP)

Une instance de MMP est une paire $P = (D^{K_{L<w}}, \sqsubset, C, C^-)$ telle que

- K_L est un type matriciel,
- $<= (<_1, <_2)$ et $<_i$ un ordre total sur L_i $(i = 1..2)$,
- $w \in K$,
- $\sqsubset$ est l'image par $s_{<w}^{(K_L)} \circ f^{(K_L)}$ d'une relation d'extension sur K_L,
- C est un ensemble de contraintes unaires sur $\Delta^{(K_L)}$ tel que pour tout $c \in C$, $c \subseteq S_{<w}^{(K_L)}$,
- $\varnothing \subseteq C^- \subseteq C$.

On appelle solution de P tout élément $x \in K_L$ vérifiant

$$S_{<w}^{(K_L)}(x) \wedge C(x) \wedge (\forall y \in S_{<w}^{(K_L)} \ : \ x \sqsubset y \rightarrow \neg C^-(y)).$$

On dénote $Sol(P)$ l'ensemble des solutions de P. On dit que P est satisfiable si et seulement si $Sol(P) \neq \varnothing$.

À noter que la formule caractérisant une solution de MMP utilise la négation, i.e. le complément dans $\Delta^{(M)}$, pour déterminer la maximalité. Cette formulation équivaut à prendre le complément dans $S_{<w}^{(M)}$ puisque $\neg C^-(y)$ est évalué pour tout $y \in S_{<w}^{(M)}$. Le résultat suivant établit donc qu'une instance de MMP et la structure correspondante ont mêmes ensembles de solutions par $S_{<w}^{(M)} \circ f^{(M)}$.

Proposition : Soient M une structure matricielle et P une instance de MMP telles que

- $\sqsubset = s_{<w}^{(M)} \circ f^{(M)}(\prec^{(M)})$,
- $C = \{c_1, \cdots, c_n\}$, $C^{(M)} = \{c_1^{(M)}, \cdots, c_n^{(M)}\}$,
- $\forall i \in [\![n]\!] : c_i = s_{<w}^{(M)} \circ f^{(M)}(c_i^{(M)})$ et $(c_i \in C^- \leftrightarrow c_i^{(M)} \in C^{-(M)})$.

M et P vérifient $s_{<w}^{(M)} \circ f^{(M)}(MAX(M)) = Sol(P)$. Par la suite, nous noterons directement une instance MMP $(D^{K_{L<w}}, \sqsubset, C, C^-)$ par sa structure matricielle $(K_{L_{<w}}, \prec, C, C^-)$.

2.4 Modélisation QCSP pour structure matricielle

Tout instance de MMP se formule sous la forme d'une instance de CSP quantifié (QCSP) en explicitant la formule logique en caractérisant les solutions. Cette modélisation QCSP comprend différents éléments (fragments) : la représentation de la matrice solution à déterminer par le biais de variables existentiellement quantifiées modélisant ses dimensions, étiquettes et coefficients, la représentation des matrices utilisées pour le test de maximalité par le biais de variables universellement quantifiées, la représentation des contraintes prédéfinies (contraintes d'injection et de section), la relation d'extension et enfin les contraintes matricielles propres au MMP.

Nous présentons ces fragments tour à tour qui reposent sur l'usage de tableaux de variables à domaines finis pour représenter les composants $d_1^{(x)}$, $d_2^{(x)}$, $l_1^{(x)}$, $l_2^{(x)}$ et $k^{(x)}$ de matrices x. Nous utilisons la notation $X[R] \in V$ pour définir un tableau X de variables à domaine fini V indexées sur l'intervalle R.

Nous modélisons ici une instance de MMP $(K_{L_{<w}}, \prec_1, C, C^-)$ où $\prec_1$ est la relation d'extension incrémentale en ligne. Nous notons $P = [\![|L_1|]\!]$ et $Q = [\![|L_2|]\!]$.

Les tableaux de variables existentiellement quantifiées d_1, d_2, l_1, l_2 et k modélisent la matrice solution x à déterminer (2.1). Le tableau de variables g modélise la fonction d'indexation $g^{(x)}$. e_1 et e_2 sont des tableaux de variables auxiliaires booléennes, chaque variable dénotant pour une ligne donnée (e_1) ou pour une colonne donnée (e_2), si elle figure (valeur $true$) ou non dans la portée de la matrice x (2.2). Les mêmes types de tableaux sont utilisés pour les variables universellement quantifiées qui modélisent les matrices y utilisées pour déterminer la maximalité de la matrice x (2.3, 2.4).

$$\exists d_1 \in P, d_2 \in Q, l_1[P] \in L_1, l_2[Q] \in L_2, k[P,Q] \in K \tag{2.1}$$
$$e_1[L_1] \in \mathbb{B}, e_2[L_2] \in \mathbb{B}, g_1[L_1] \in P, g_2[L_2] \in Q \tag{2.2}$$
$$\forall d'_1 \in P, d'_2 \in Q, l'_1[P] \in L_1, l'_2[Q] \in L_2, k'[P,Q] \in K \tag{2.3}$$
$$e'_1[L_1] \in \mathbb{B}, e'_2[L_2] \in \mathbb{B}, g'_1[L_1] \in P, g'_2[L_2] \in Q \tag{2.4}$$

Les contraintes (2.5, 2.6) modélisent le couplage entre l'indexation g_i et l'étiquetage l_i de x et y sur chaque dimension i.

$$\bigwedge_{i\in L_1,p\in P} l_1[p] = i \leftrightarrow g_1[i] = p \qquad \bigwedge_{i\in L_1,p\in P} l'_1[p] = i \leftrightarrow g'_1[i] = p \quad (2.5)$$

$$\bigwedge_{j\in L_2,q\in Q} l_2[q] = j \leftrightarrow g_2[j] = q \qquad \bigwedge_{j\in L_2,q\in Q} l'_2[q] = j \leftrightarrow g'_2[j] = q \quad (2.6)$$

Les contraintes (2.7, 2.8) définissent les variables d'exclusion e_1 et e_2 : toute ligne ou colonne hors portée est exclue.

$$\bigwedge_{i\in L_1} e_1[i] \leftrightarrow g_1[i] > d_1 \qquad \bigwedge_{i\in L_1} e'_1[i] \leftrightarrow g'_1[i] > d'_1 \quad (2.7)$$

$$\bigwedge_{j\in L_2} e_2[j] \leftrightarrow g_2[j] > d_2 \qquad \bigwedge_{j\in L_2} e'_2[j] \leftrightarrow g'_2[j] > d'_2 \quad (2.8)$$

Les contraintes (2.9, 2.10, 2.11, 2.12, 2.13) modélisent la contrainte de section $S^{(M)}_{<w}$ qui ordonne l'ensemble des lignes (respectivement, colonnes) selon $<_1$ (respectivement, $<_2$) dans la portée, et l'ensemble des lignes (respectivement, colonnes) selon $<_1$ (respectivement, $<_2$) hors portée et qui fixent les coefficients hors portée à w.

$$alldifferent(l_1) \qquad alldifferent(l'_1) \quad (2.9)$$

$$alldifferent(l_2) \qquad alldifferent(l'_2) \quad (2.10)$$

$$\bigwedge_{p<p'\in P} ((p' \leqslant d_1) \vee (p > d_1)) \rightarrow l_1[p] < l_1[p'] \qquad \bigwedge_{p<p'\in P} ((p' \leqslant d'_1) \vee (p > d'_1)) \rightarrow l'_1[p] < l'_1[p'] \quad (2.11)$$

$$\bigwedge_{q<q'\in Q} ((q' \leqslant d_2) \vee (q > d_2)) \rightarrow l_2[q] < l_2[q'] \qquad \bigwedge_{q<q'\in Q} ((q' \leqslant d'_2) \vee (q > d'_2)) \rightarrow l'_2[q] < l'_2[q'] \quad (2.12)$$

$$\bigwedge_{p\in P,q\in Q} (p > d_1 \vee q > d_2) \rightarrow k[p,q] = w \qquad \bigwedge_{p\in P,q\in Q} (p > d'_1 \vee q > d'_2) \rightarrow k'[p,q] = w \quad (2.13)$$

La contrainte (2.14) modélise la consistance de la matrice x avec la conjonction des contraintes matricielles. Chaque contrainte $c(d_1, d_2, l_1, l_2, k)$ doit se conformer à la sémantique des

contraintes matricielles (indépendance à l'ordre entre étiquettes de lignes et de colonnes) et à la contrainte de dimensionnement ($l^{(x)}$ et $k^{(x)}$ s'interprètent relativement à $d^{(x)}$). Il en va de même pour la contrainte d'extension comme le montre sa formulation (antécédent de la clause (2.15)).

$$\bigwedge_{c \in C} c(d_1, d_2, l_1, l_2, k) \tag{2.14}$$

La contrainte (2.15) modélise la clause d'exclusion $(x \sqsubset_1 y \rightarrow \neg C^-(y))$.

$$\begin{aligned} \bigwedge_{i \in L_1} & \left(\begin{array}{l} d'_1 = d_1 + 1 \wedge e_1[i] \wedge \neg e'_1[i] \wedge \bigwedge\limits_{i' \in L_1} (e_1[i'] \rightarrow e'_1[i']) \\ \wedge d'_2 = d_2 \wedge \bigwedge\limits_{j \in L_2} (e'_2[j] \leftrightarrow e_2[j]) \\ \wedge \bigwedge\limits_{i' \in L_1, j \in L_2} (i \neq i' \wedge \neg e_1[i] \wedge \neg e_2[j]) \rightarrow k[g_1[i'], g_2[j]] = k'[g'_1[i'], g'_2[j]] \end{array} \right) \\ & \longrightarrow \bigvee_{c \in C^-} \neg c(d'_1, d'_2, l'_1, l'_2, k') \end{aligned} \tag{2.15}$$

y est une extension incrémentale en ligne de x si elle étend x d'une ligne - $(d'_1 = d_1 + 1) \wedge \bigwedge\limits_{i' \in L_1} (e_1[i'] \rightarrow e_1[i'])$ -, que cette ligne a pour étiquette i - $e_1[i] \wedge \neg e'_1[i]$ -, que y a les mêmes colonnes que x - $d'_2 = d_2 \wedge \bigwedge\limits_{j \in L_2} (e'_2[j'] \leftrightarrow e_2[j])$ -, et qu'elle préserve tout coefficient de x - $\bigwedge\limits_{i' \in L_1, j \in L_2} (i \neq i' \wedge \neg e_1[i] \wedge \neg e_2[j]) \rightarrow k[g_1[i'], g_2[j]] = k'[g'_1[i'], g'_2[j]]$. Pour cette dernière équation, l'utilisation de g_1, g_2, g'_1 et g'_2 est requise et dissimule des contraintes de type *element*. Il s'agit ici d'une forme Skolémisée où la contrainte `element` est interprétée comme une fonction sur n variables (la variable indice et les variables d'un tableau g_i) et est substituée à la variable résultat.

2.5 Complexité

Comme montré en section précédente, le problème MMP est la classe de QCSP n-aires restreints à une alternance de quantificateurs existentiels et universels et au langage de contraintes incluant les contraintes prédéfinies de section et d'extension et toute contrainte matricielle sur un type matriciel quelconque. La complexité d'une classe d MMP dépend donc de la classe de contraintes matricielles considérée et du type matriciel sous-jacent. Ces choix conditionnent la complexité des deux sous-problèmes à résoudre : la détermination d'une matrice consistante et, pour la donnée d'une matrice consistante, la détermination de sa maximalité.
Nous reformulons à cet effet la caractérisation logique des solutions d'une instance P de MMP

de signature $(M, \prec_1, C, C^-)$.[6]
Pour tout $x \in \Delta^{(M)},\ x \in Sol(P)$ si et seulement si $POS_C(x) \wedge NEG_{\sqsubset_1,C^-}(x)$ où POS_C et NEG se définissent pour un ensemble de contraintes matricielles C par :

1. $POS_C(x) \leftrightarrow S^{(M)}_{\prec w}(x) \wedge C(x)$,
2. $NEG_{\sqsubset_1,C}(x) \leftrightarrow (\forall y \in \Delta^{(M)}\ :\ (POS_C(x) \wedge \sqsubset_1 y) \rightarrow \neg C(y))$.

Pour certaines classes de structures matricielles, le sous-problème $NEG_{\sqsubset_1 C^-}$ est trivialement décidable. C'est le cas notamment des structures à une ligne ($D^{(M)} = 1$) ou sans contraintes négatives ($C^- = \varnothing$), des structures incluant la contrainte de couverture $\mathrm{cover}_{\{\supseteq, L_1^{(M)}\}}$, et des structures ne comportant que des contraintes $\prec_1$-monotones comme contraintes négatives. Pour chacune de ces classes, le problème $NEG^{(x)}_{\sqsubset_1 C^-}$ est satisfiable quelle que soit la matrice x. Elles dispensent donc du test de maximalité et l'instance MMP se ramène à la résolution de $(\exists x \in \Delta^{(M)}\ :\ POS(x))$, c'est-à-dire à la résolution d'une instance de CSP.

Plus généralement, certaines classes de contraintes caractérisent un problème de maximalité $NEG_{\sqsubset_1,C}$ qui est polynomial ($NEG_{\sqsubset_1,C} \in P$). C'est le cas en particulier des classes de contraintes négatives limitées aux seuls prédicats de base de données et d'ordre total entre colonnes.

Proposition : Soit $C = \{\mathrm{db}_B\},\ x \in \Delta^{(M)}$.

$$NEG_{\sqsubset_1,C^-}(x) \leftrightarrow (POS_C(x) \rightarrow \bigwedge_{i \in L_1^{(M)}} (i \notin L_1^{(x)} \rightarrow \bigvee_{j \in L_2^{(M)}} (j \in L_1^{(x)} \wedge B(i,j) = \varnothing)))$$

Démonstration. La reformulation découle directement de là décomposabilité du test du prédicat db_B par cellule : tester si une ligne hors portée de x est non conforme à la base de données revient à tester s'il existe une cellule sur cette ligne pour laquelle la base de données n'autorise aucun coefficient. □

Dans le cas du prédicat d'ordre total combiné au prédicat de base de données, exclure une ligne qui est hors portée d'une matrice consistante revient à montrer qu'il est impossible de trouver des coefficients sur les cellules de cette ligne qui ordonnent les colonnes conformément à l'ordre imposé par la matrice et aux domaines autorisés par la base de données. Résoudre ce problème consiste à résoudre le complémentaire d'un CSP dont les variables correspondent aux colonnes du type matriciel, dont les domaines sont les domaines autorisés par la base de données pour les variables-colonnes dans la portée de la matrice ou le domaine de coefficients $K^{(M)}$ pour les autres, et dont les contraintes sont des contraintes binaires d'inégalité entre toute paire de variables-colonnes ordonnées par la matrice. Ce type de CSP se résoud en temps polynomial par

6. Nous, nous limiterons ici à la l'extension incrémentale en ligne.

calcul de bornes-consitance et il en va donc de même pour la classe de co-CSP correspondante et donc du problème $NEG_{\sqsubset_1,C}$ pour $C = \{\mathrm{db}_B, \mathrm{total}_<\}$.

Proposition : Soit $C = \{\mathrm{db}_B, \mathrm{total}_<\}$, $x \in \Delta^{(M)}$.
$NEG_{\sqsubset_1,C} \leftrightarrow (POS_C(x) \rightarrow (\bigwedge_{i \in L_1^{(M)}} (i \notin L_1^{(x)} \rightarrow coCSi(X', D', C'))))$ où $coCSP_i(X', D', C')$ est vrai si et seulement si l'instance du CSP (X', D', C') est insatisfiable avec :

— $X' = L_2^{(M)}$,

— $D'(x') = K$ ($x' \in X'$),

$$-C'(X') \leftrightarrow \bigwedge_{y \in X'} (y \in L_2^{(x)} \rightarrow \mathrm{db}_B(i, y))$$
$$\wedge \bigwedge_{y_1, y_2 \in X'} \Big(\big(k^{(x)}(1, g_2^{(x)}(y_1)) < k^{(x)}(1, g_2^{(x)}(y_2)) \wedge y_1 \in L_2^{(x)} \wedge y_2 \in L_2^{(x)}\big) \rightarrow y_1 < y_2 \Big)$$

Démonstration. La reformulation découle directement de la sémantique de db_B et $\mathrm{total}_<$. □

Ces résultats montrent donc qu'il suffit pour ces classes de contraintes de résoudre un (co-)CSP polynomial pour chaque test de maximalité. D'un point de vue opérationnel, il suffit pour résoudre une instance de MMP d'utiliser un algorithme CSP pour résoudre POS_C sur M et de lui adjoindre un algorithme CSP polynomial dédié pour résoudre $NEG_{\sqsubset,C^-}$ qui est appelé pour chaque matrice consistante obtenue. Cette approche utilise donc "l'algorithme de maximalité" comme une boite noire (un oracle) et ne l'invoque qu'à la découverte d'une matrice consistante (approche de type *générer-et-tester*).

Pour une approche plus efficace et plus intégrée, nous pouvons tenter de modéliser l'exécution de l'algorithme de maximalité (e.g., le renfoncement de nœud-consistance dans le cas $\{\mathrm{db}_B\}$, le renforcement de bornes-consistance dans le cas de $\{\mathrm{total}_<, \mathrm{db}_B\}$) sous la forme d'un CSP. Autrement dit, nous simulons en CSP les règles de calcul déterministes de l'algorithme et nous caractérisons état acceptant et non-acceptant. Le CSP résultant peut alors être intégré au CSP modélisant POS_C et permettre ainsi un couplage plus serré des deux calculs de consistance et de maximalité. Nous proposons au chapitre 3 une telle approche.

Nous notons, enfin, que la polynomialité du sous-prolbème de maximalité $NEG_{\sqsubset_1,C}$ ne garantit pas celle de la classe de MMP qui dépend du langage de contraintes positives utilisé. Le résultat suivant montre, par exemple, que la recherche d'un itemset (ou motif) de taille majorée et excluant un ensemble prédéfini de transactions (ou séquences) est NP-complet. La preuve s'obtient par réduction du problème de recouvrement d'ensembles.

Théorème 1. *Soit $\mathcal{C}$ la classe d'instances $P = (K_{L_{<w}}, \prec_1 C, C^-)$ de* MMP *telles que $C = \{\mathrm{db}_B,\ \mathrm{card}_{\{\geqslant,s\}},\ \mathrm{freq}_{\{\leqslant,f\}}\}$ et $C^- = \{\mathrm{db}_B,\ \mathrm{card}_{\{\geqslant,s\}}\}$. $\mathcal{C}$ est NP-complet.*

Démonstration.

(A) Nous montrons que $\mathcal{C}$ est dans NP.

Vérifier pour une instance P de $\mathcal{C}$ qu'une "matrice" m de $K_{L_{<w}}$ est consistante consiste à tester les dimensions de m avec les contraintes de cardinalité et fréquence, et chaque cellule de m avec la base de données. Le deux premiers tests sont en $O(|L_1|.|L_2|)$ dû au conditionnement de m, le test de base de données est en $O(|L_1|.|L_2|.|K|)$. Si m est consistante, vérifier qu'elle est maximale peut s'effectuer comme suit. La contrainte de cardinalité est satisfaite est puisque $\prec_1$-monotone. Si la fréquence de m atteint f, le test est positif et la vérification est terminée. Sinon nous testons ligne par ligne et cellule par cellule, si une ligne hors portée de m est conforme à la base. Le cas échéant, les test négatif et la vérification terminée. Sinon, le test s'arrête lorsque toutes les lignes hors portée ont été testées et la maximalité est avérée dans ce cas. En pire cas, le test de maximalité est $O(|L_1|.|L_2|.|L_2|.|K|)$. Donc la vérification prend un temps polynomial en la taille de P. Une machine non déterministe peut donc deviner une matrice m et vérifier qu'elle est solution en temps polynomial. Donc $P \in NP$.

(B) Nous établissons une réduction *many-one* polynomiale du problème de recouvrement d'ensemble vers $\mathcal{C}$. Soit R une instance (U, S, s) de recouvrement d'ensemble [7] et $f(R) = P = (K_{L_{<w}}, \prec_1 C, C^-)$ tel que :

— $L_1 = U \cup F,\ F \cap U = \varnothing$ et $|F| = f$;

— $L_2 = S$;

— $K = \{w\}$;

— $\forall i \in L_1,\ j \in L_2\ :\ i \in j \rightarrow K$;

— $\forall i \in L_1,\ j \in L_2\ :\ i \notin j \rightarrow \varnothing$.

Supposons que R est satisfaite et $T \subseteq S$ est solution. Soit $m \in \Delta^{(K_L)}$ tel que $L_1^{(m)} = F$, $L_2^{(m)} = T$ et $k^{(m)}(i, j) = w \neq \varnothing$ pour tout $i \in L_1^{(m)}$ et $j \in L_2^{(m)}$. m satisfait $\mathrm{card}_{\{\geqslant,s\}}$, db_B et $\mathrm{freq}_{\{\leqslant,f\}}$ est donc consistante. T étant solution de R, $\bigcup_{t \in T} t = U$ ce qui implique

7. Un problème de recouvrement d'ensemble peut être défini par un triplet (U, S, s) tel que $s \in \mathbb{N}$, U un ensemble fini et $S \subseteq 2^U$. L'objectif est de trouver un sous-ensemble T de S de cardinalité inférieure ou égale à s tel que $\bigcup_{t \in T} t = U$.

$(\forall i \in U,\ \exists j \in T\ :\ B(i,j) = \varnothing)$ donc $(\forall m' \in \Delta_{<w}^{(K_L)}\ :\ m \prec_1 m' \rightarrow \neg \mathrm{db}_B(m'))$ donc m est maximale et $f(R) = P$ satisfiable.

Réciproquement, supposons que P est satisfiable et m solution de P. Puisque $|F| = f$ et du fait que toute matrice n'incluant aucune ligne de F ne peut être maximale (les lignes de F sont conformes à la base de données), nous avons $L_1^{(m)} = F$. Par définition de la base de données, pour tout $i \notin F$, il existe $j \in l_2^{(m)}([\![d_2^{(m)}]\!]) = F$ tel que $B(i,j) = \varnothing$, i.e., $i \in j$. Donc $U \subseteq \bigcup_{j \in l_2^{(m)}([\![d_2^{(m)}]\!])} j \subseteq U$. Donc $l_2^{(m)}([\![d_2^{(m)}]\!]) \leqslant s$, $l_2^{(m)}([\![d_2^{(m)}]\!])$ est une solution de R. Donc R est satisfiable.

□

2.6 Matrices maximales n-aires : définitions et prédicats

Nous proposons d'étendre la définition de structure matricielle pour que celle-ci puisse considérer des n-uplets de types matriciels. Un tel changement permet de définir plus aisément certains types de prédicats (e.g., prédicat d'ordre partiel entre les colonnes). Nous ne fournirons par la linéarisation de cette structure matricielle, celle-ci consiste simplement à linéariser les n matrices du n-uplet.

Définition 20 (Structure matricielle n-aire)

Une structure matricielle n-aire est un quadruplet $(M, \prec, C, C^-)$ tel que :

— $M = M^{(1)} \times \cdots \times M^{(n)}$ est un produit cartésien de types matriciels ($n \in \mathbb{N}$) ;

— $\prec \subseteq M \times M$ est une relation d'extension définie par n relations d'extension telle que
$m \prec m' \leftrightarrow \big((\bigwedge_{i \in [\![n]\!]} m^{(i)} \preccurlyeq^{(i)} m'^{(i)}) \wedge m \neq m'\big)$ [8] $(m, m' \in M)$;

— C et C^- sont des ensembles de contraintes sur M ($\forall c \in C,\ c \subseteq M$), et tels que $\varnothing \subseteq C^- \subseteq C$.

C^- est appelée ensemble des contraintes négatives de C.

Une matrice $m \in M$ est dite matrice maximale sur $\mathcal{M}$ si et seulement si m satisfait

$$(C(m) \wedge \forall m' \in M\ :\ m \prec m' \rightarrow \neg C^-(m'))$$

Nous notons $\mathtt{MAX}(\mathcal{M})$ l'ensemble des matrices maximales sur $(M, \prec, C, C^-)$.

8. Soit m et m' deux matrices, $m \preccurlyeq m'$ signifie que m étend m' ou que $m = m'$.

Pour simplifier les notations, les contraintes de structure matriciels n-aire sont ici définies sur des n-uplets de matrices, mais, en pratique, une contrainte ne peut porter que sur quelques matrices du n-uplet. De ce fait, lorsque nous définirons un prédicat matriciel nous lui associerons une liste d'indices de matrices afin de préciser sur quelles matrices la contrainte porte. Par exemple, $c^{[1,2]}$ signifie que la contrainte c porte exclusivement sur les matrices 1 et 2 et sera définie relativement à cet ordre.

2.6.1 Prédicats d'ordre binaire

Nous introduisons, d'abord, une formulation binaire au prédicat d'ordre total entre les colonnes présenté dans le cadre de structure matricielle unaire. Ce prédicat binaire est équivalent à sa version unaire et force les colonnes d'un matrice à être ordonnées. Contrairement à sa version unaire, il explicite l'ordre total dans une matrice auxiliaire. Donc, ce prédicat porte sur deux matrices dont les dimensions sont liées (même nombre de colonnes).

Définition 21 (Prédicat d'ordre total entre colonnes)

Soient $M = M^{(1)} \times M^{(2)}$ un type matriciel binaire défini tel que :

— L_1 et L_2 deux ensembles d'étiquettes quelconques ;

— $M^{(1)} = \mathbb{N}_{(L_1,L_2)}$;

— $M^{(2)} = L_{1_{(\{1\},L_2)}}$.

Soit $<$ l'ordre total usuel sur les entiers. Le prédicat binaire d'ordre total basé sur $<$, dénoté $\mathrm{total}_{<}^{[1,2]}$, est défini par

$$\forall m \in M,\ \mathrm{total}_{[1,2],<}(m) \leftrightarrow$$
$$\bigwedge_{i \in L_1^{(m^{(2)})}, j \neq j' \in L_2^{(m^{(2)})}} ((k^{(m^{(2)})}(1,j) < k^{(m^{(2)})}(1,j')) \leftrightarrow (k^{(m^{(2)})}(i,j') < k^{(m^{(2)})}(i,j')))$$
$$\wedge \bigwedge_{j \neq j' \in L_2^{(m^{(1)})}} (k^{(m^{(2)})}(1,j) \neq k^{(m^{(2)})}(1,j'))$$

Nous introduisons, maintenant, le prédicat binaire d'ordre partiel entre les colonnes d'une matrice. Ce prédicat suppose un type matriciel binaire qui adjoint à une matrice principale une seconde matrice d'adjacence permettant de modéliser tout graphe entre ses colonnes. Le prédicat impose que le graphe d'une matrice binaire (i.e., sa seconde matrice) soit conforme à l'ordre partiel induit sur sa matrice principale (i.e., sa première matrice) [9]. Par exemple, le choix

9. Chaque séquence dans la portée d'un motif détermine un ordre partiel entre les colonnes fondé sur leurs localisations dans la séquence. Nous appelons ordre induit d'un motif la conjonction des ordres partiels induits par les séquences, autrement dit, la relation maximale qui subsume les ordres induits par les séquences. Cette relation est unique et constitue un ordre partiel.

d'une matrice d'adjacence vide ou de l'ordre partiel induit lui même déterminent des matrices consistantes. Pour garantir la validité de ce prédicat, il est nécessaire que la seconde matrice soit carrée et que son nombre de colonnes et son nombre de lignes soient égales au nombre de colonne de la première matrice. Le domaine de coefficients du type matriciel associé à la seconde matrice est de type booléen.

Définition 22 (Prédicat d'ordre partiel entre colonnes)

Soient $M = M^{(1)} \times M^{(2)}$ un type matriciel binaire défini tel que :

— L_1 et L_2 deux ensembles d'étiquettes quelconques ;

— $M^{(1)} = \mathbb{N}_{(L_1,L_2)}$;

— $M^{(2)} = \mathbb{B}_{(L_2,L_2)}$.

Soit $<$ l'ordre total usuel sur les entiers. Le prédicat binaire d'ordre partiel basé sur $<$, dénoté $\mathrm{partial}_{<}p^{[1,2]}$, est défini par

$$\forall m \in M,\ \mathrm{partial}_{[1,2],<}(m) \rightarrow \bigwedge_{i \in L_1^{(m^{(1)})}, j \neq j' \in L_2^{(m^{(1)})}} (k^{(m^{(2)})}(j,j') \rightarrow (k^{(m^{(1)})}(i,j) < k^{(m^{(1)})}(i,j')))$$

2.6.2 Structure matricielle pour la recherche d'ordre partiel fréquent

Afin de définir une structure matricielle pour la recherche d'ordre partiel nous pouvons nous appuyer sur le type matriciel dédié à la recherche d'ensemble de caractères (section 2.2.4). Nous associons à ce premier type matriciel $M^{(1)}$ un second type matriciel $M^{(2)}$ modélisant toute matrice d'adjacence (graphe) entre colonnes de $M^{(1)}$. Étant donné un jeu de séquences S construit sur un alphabet Σ, le type matriciel élémentaire pour $M^{(1)}$ consiste à assimiler les lignes aux séquences ($L_1^{(M^{(1)})} = S$), les colonnes aux caractères de l'alphabet ($L_2^{(M^{(1)})} = \Sigma$), et le domaine de coefficients à l'ensemble des localisations de caractère possibles dans les séquences ($K^{(M^{(1)})} = [\![\max_{s \in S} |s|]\!]$). Comme $M^{(2)}$ représente un graphe entre les colonnes de $M^{(2)}$, alors ses lignes et colonnes doivent coïncider avec les colonnes de $M^{(1)}$ ($L_1^{(M^{(2)})} = L_2^{(M^{(1)})}$ et $L_2^{(M^{(2)})} = L_2^{(M^{(1)})}$) et le domaine de coefficients doit indiquer la présence ou absence d'ordre entre deux colonnes de $M^{(1)}$ par une valeur booléenne ($K^{(M^{(2)})} = \mathbb{B}$). La relation d'extension matricielle se base sur une extension en ligne de $M^{(1)}$. Pour $M^{(2)}$ nous pouvons utiliser la relation d'extension généraliste. Toutefois, le prédicat d'ordre partiel, de par sa sémantique, impose l'extension incrémentale conjointe en ligne et colonne (i.e., ajout d'une ligne et d'une colonne simultanément). C'est donc cette seconde relation d'extension que nous retiendrons. (nombre de colonnes de $m^{(1)}$ différent du nombre de lignes ou du nombre de colonnes de $m^{(2)}$). En ce

qui concerne les prédicats, nous considérons le prédicat de base de données basée sur la base de séquences S, le prédicat de fréquence minimale f et le prédicat d'ordre partiel basé sur le relation d'ordre usuelle sur les entiers $<$. Soit la structure matricielle binaire $(M, \prec, C, C^-)$ définie telle que

— $M^{(1)} = \mathbb{N}_{(L_1,L_2)}$;

— $M^{(2)} = \mathbb{B}_{(L_2,L_2)}$;

— $M = M^{(1)} \times M^{(2)}$;

— $m \prec m' \leftrightarrow (m^{(1)} \prec_1^{(1)} m'^{(1)} \wedge m^{(2)} \prec_{[1,1]}^{(2)} m'^{(2)})\ (m, m' \in M)$ [10] ;

— $C = \{\mathrm{db}_S^{[1]}, \mathrm{freq}_{\{\geqslant,f\}}^{[1]}, \mathrm{partial}_{<}^{[1,2]}\}$

— $C^- = \{\mathrm{db}_S^{[1]}, \mathrm{partial}_{<}^{[1,2]}\}$.

10. La notation $\prec_{[1,1]}$ fait référence à la relation d'extension incrémentale conjointe en ligne et colonne.

3

Résolution de MMP

3.1 Introduction

Ce chapitre présente la résolution par programmation par contraintes de deux classes d'instances de MMP : la recherche de sous-séquences communes à un jeu de séquences S et la recherche d'ordres partiels basés sur les localisations de caractères et communs à un jeu de séquences [1]. Nous considérons, ici, des motifs (sous-séquence ou ordre partiel) constitués d'un ensemble de caractères (ou patrons) communs à un ensemble de séquences et contraints par une relation d'ordre total ou partiel. Soient S^+ et S^- deux sous-ensembles de séquences, dites positives et négatives respectivement, partitionnant S, nous recherchons sur le jeu de séquences des motifs communs aux séquences positives et absents des séquences négatives. Pour chaque catégorie de motif, nous proposons ou plusieurs modélisations reposant sur la programmation par contraintes et une présentation du problème sur la forme de MMP.

La modélisation et la résolution de ces problèmes se décomposent en deux parties : la première consiste à extraire un motif commun des séquences positives et la seconde à tester l'exclusion des séquences négatives en fonction du motif.

Pour le test d'exclusion d'une séquence nous proposons une contrainte globale pour chaque type de motifs (sous-séquence ou ordre partiel) dont l'efficacité est avérée par des expérimen-

1. Les séquences considérées dans ce chapitre sont des séquences d'ensembles (itemsets) de caractères (i.e. deux caractères différents de l'alphabet peuvent coïncider sur une même séquence) et non des séquences de caractères.

tations sur des jeux de séquences protéiques.

3.2 Ordre Total

Cette section présente une classe de MMP binaire dédié à l'extraction de motifs totalement ordonnés, dont nous fournirons une résolution par PPC. Nous considérons une base de séquences classées positives et négatives. L'objectif est de trouver un motif optimal défini par un multi-ensemble de caractères (i.e. un caractère peut être répété) consistant avec la base de séquences et un ordre total entre ces caractères. Le motif solution doit être fréquence au sein des séquences négatives et ne doit pas couvrir les séquences négatives ou en couvrir un minimum. Pour une séquence donnée, l'exclusion s'établit soit par l'absence d'un caractère du motif sur la séquence, soit par l'impossibilité de reproduire l'ordre total sur la séquence.

Nous étudierons d'abord une version sans séquence négative et sans répétition de caractères, puis nous ajouterons la répétition de caractères. Finalement, nous introduirons la classe négative avec le critère de minimisation des classes négatives.

3.2.1 Sous-séquences fréquentes sans répétition de caractères

Ce premier modèle est la variante la plus simple du problème introduit précédemment. Notre objectif vise à extraire un une sous-séquence fréquente dans un ensemble de séquences

Nous considérons une base B de séquences construites sur un alphabet Σ, qui est donnée sous la forme d'une matrice dont les lignes sont indicées sur un ensemble L_1 et les colonnes sont indicées sur un ensemble L_2 et f un seuil de fréquence donné. Ayant pour objectif de calculer des motifs n'autorisant pas la répétition de caractères nous définissons L_2 comme l'ensemble des caractères de l'alphabet Σ. Notre problème peut alors s'exprimer comme l'instance MMP $(M, <, C, C^-)$ telle que le domaine matriciel $M = (M^{(1)}, M^{(2)})$ avec $M^{(1)} = \mathbb{N}_{(L_1,L_2)}$ et $M^{(2)} = [\![|L_2|]\!]_{(L_2,\{1\})}$, $<= (<^{(1)}, <^{(2)}) = (<_1, =)$, $C^+ = \mathcal{C}\backslash C^-$, $C^+ = \{\text{freq}^{[1]}_{\geqslant,f}\}$ et $C^- = \{\text{db}^{[1]}_B, \text{total}^{[1,2]}_<\}$. Ce MMP a pour solution des matrices de la forme $m = (m^{(1)}, m^{(2)})$ telles que les lignes de $m^{(1)}$ correspondent aux séquences couvertes par le motif et les colonnes de $m^{(1)}$ correspondent aux caractères du motif. Les coefficients de la matrice $m^{(1)}$ correspondent aux enracinements, cohérents avec B, des caractères au sein des séquences couvertes. La matrice $m^{(2)}$, quant à elle, représente la relation d'ordre total (représentée sous la forme d'une permutation) entre les colonnes de $m^{(1)}$ et basée sur les coefficients de $m^{(1)}$. La contrainte de fréquence impose de couvrir au moins f lignes.

En d'autres termes, ce modèle vise à résoudre le problème de recherche de sous-séquences sans répétition de caractères et fréquentes au sein d'un ensemble de séquences. Dans la section suivante nous incorporerons la répétition de caractères.

Le fonctionnement de notre modèle s'inspire des techniques de projection employées notamment dans [53, 29, 51] (Pei, Kemmar, Negrevergne). Notre objectif vise à reproduire la sous-séquence extraite en sélectionnant sur chaque séquence les localisations cohérentes minimales (les plus à gauche) pour chaque caractère de la sous-séquence. À cette fin, le modèle réordonne la base de données en fonction de la permutation des colonnes, induite par la sous-séquence. En effet, comme le motif contient au plus une occurrence de chaque caractère de l'alphabet, il est constitué alors une permutation d'un sous-ensemble de cet alphabet. Cette réorganisation de la base de données, basée sur la permutation, permet ainsi de choisir efficacement les localisations minimales pour tout caractère.

Modèle PPC : Nous présentons ci-dessous le modèle PPC dédié à la résolution du MMP présenté précédemment.

Données : Les données se limitent à une base de données B, un nombre de caractères $colonnes$, un nombre de séquences $lignes$ et un support minimal (le lien entre support et fréquence est direct). Les séquences sont représentées par une matrice B dont les cellules représentent les localisations des caractères sur chaque séquence. Celles-ci sont représentées par un ensemble d'entiers. À partir des données fournies précédemment nous créons un nouvel ensemble $DUMMIES$ ainsi qu'une nouvelle matrice d'ensembles BD. Cette nouvelle matrice est simplement la matrice B enrichie des valeurs de $DUMMIES$, c'est-à-dire que chaque cellule de BD est obtenue à partir de l'union de $DUMMIES$ et de la cellule de B de même indice. L'ensemble $DUMMIES$, quant à lui, empêche l'échec de la résolution du modèle PPC dû à l'épuisement de l'ensemble du domaine de certaines variables [2].

Variables : Les variables de décisions $l1$, $l2$ et v correspondent respectivement à : un vecteur de booléens indiquant les séquences couvertes, un vecteur de booléens indiquant les caractères de la sous-séquence et une matrice correspondant aux localisations sélectionnées pour chaque caractère sur chaque séquence. Les deux variables $card$ et $supp$ sont respectivement la cardinalité de la sous-séquence (i.e. sa longueur) ainsi que le support (i.e. le nombre de séquences positives la contenant). Le vecteur $rangs$ identifie, pour chaque caractère, son rang dans la sous-séquence extraite (i.e. sa localisation dans la sous-séquence) et le vecteur $derniers$ enregistre pour chaque séquence la localisation du dernier caractère de la sous-séquence [3]. Nous trouvons également deux matrices vs et BS qui correspondent respectivement aux matrices v et B dont les lignes ont été réordonnées en fonction de la permutation induite par le vecteur $rangs$ ce qui implique que toutes les valeurs de $rangs$ doivent être différentes.

2. L'utilisation de cet ensemble sera explicitée dans la section "Contraintes" de notre modèle.

3. Le vecteur $rangs$ contient un rang différent pour chaque caractère de l'alphabet, mais seuls les $card$ premiers caractères constituent la sous-séquence, le dernier caractère de la sous-séquence est alors celui d'indice $card$.

Contraintes : La première contrainte (ligne 19) restreint les domaines des cellules de la matrice v aux ensembles contenus dans les cellules de BD de mêmes indices. L'objectif est de lier les domaines des variables de v aux localisations des caractères dans les séquences de la base. Si aucune localisation n'est possible, alors nous utiliserons des valeurs de $DUMMIES$ qui, par définition, ne sont présentes dans aucune cellule de B, afin d'éviter l'épuisement du domaine d'une des variables de v. La contrainte ligne 21 calcule le support de la sous-séquence, support correspondant aux séquences couvertes (vecteur $l1$). À la ligne suivante, nous imposons le respect du support minimal. La contrainte lignes 23 et 24 indique que chaque ligne de v doit être ordonnée (triée par ordre croissant) avec la même permutation. Cette permutation est représentée par le vecteur $rangs$ et sert, dans la contrainte suivante, à ordonner les lignes de vs. Les lignes de vs sont donc ordonnées par ordre croissant. La contrainte lignes 27 et 28 réordonne la base de séquences BD en fonction de la permutation $rangs$ (ici les lignes ne sont pas ordonnées par ordre croissant car BD et BS contiennent des ensembles d'entiers et non des valeurs entières).

À ce stade nous avons une sous-séquence représentée par le vecteur $rangs$ indiquant l'ordre d'apparition des caractères dans cette sous-séquence. La variable $card$, quant à elle, indique que seuls les caractères dont le $rangs$ est inférieur ou égal à $card$ appartiennent à la sous-séquence. Ces deux variables ($rangs$ et $card$), définissent une sous-séquence dont nous ne connaissons pas encore la couverture. Afin de déterminer cette dernière, nous parcourons tout caractère de la sous-séquence de la gauche vers droite, en sélectionnant, pour chaque séquence de B, la localisation minimale et cohérente du caractère. Si, pour une séquence s de B donnée, la sélection des localisations minimales des caractères est impossible, alors la sous-séquence ne peut être reproduite sur s. Cette opération est réalisée par la contrainte ligne 29 et celle des lignes 30 et 31. En effet, vs correspond à la matrice v dont les colonnes sont triées en fonction de l'ordre d'apparition des caractères dans la sous-séquence. Il en est de même pour BS vis-à-vis de BD. Par exemple, $vs[2, 1]$ correspond à la localisation du premier caractère de la sous-séquence dans la séquence 2. Ainsi, la première de ces deux contraintes sélectionne pour chaque séquence la localisation minimale pour le premier caractère de la sous-séquence, et la seconde sélectionne pour chacun des autres caractères sa localisation minimale en garantissant la cohérence par rapport à la localisation du caractère le précédant. Dans le cas où la sous-séquence ne peut être reproduite sur une séquence, c'est-à-dire qu'il n'existe pas dans B de valeur cohérente pour reproduire la sous-séquence, des valeurs de $DUMMIES$ seront sélectionnées comme affectation des variables de vs (par définition, toute valeur de $DUMMIES$ est strictement plus grande que toute valeur des ensembles contenus dans B). De plus, $DUMMIES$ contient suffisamment de valeurs (autant que le nombre de caractères de l'alphabet) pour pouvoir reproduire n'importe quelle sous-séquence, ce qui garantit que tout caractère n'appartenant pas à la sous-séquence peut tout de même trouver des localisations - ces dernières pouvant être fictives.

Les trois dernières contraintes définissent les vecteurs $rangs$, $l1$ et $l2$. Pour toute séquence, $rangs$ localise le dernier caractère de la sous-séquence (lignes 34 et 35). Localiser le dernier caractère de la sous-séquence, en examinant la colonne de vs correspondant au dernier caractère, permet de déduire le vecteur $l1$ (correspondant à la couverture de la sous-séquence). En effet, une séquence n'est pas couverte si la localisation du dernier caractère de la sous-séquence appartient à $DUMMIES$ (lignes 35 et 36)[4]. Le dernier rang de la séquence, quant à lui, correspond à la cardinalité de celle-ci. Ainsi, pour obtenir $l2$ il suffit de regarder les caractères dont le rang est suffisamment faible pour appartenir à la séquence (lignes 33).

```
1  int: colonnes, lignes, supp_min;
2
3  set of int: COLONNES = 1..colonnes;
4  set of int: LIGNES   = 1..lignes;
5
6  array[LIGNES,COLONNES] of set of int: B;
7
8  int: MAX_B = max([max(B[i,j])| i in LIGNES, J in COLONNES]);
9  set of int: DUMMIES = (MAX_B+1)..(MAX_B+1+COLONNES);
10 array[LIGNES,COLONNES] of set of int: BD
11   = [B[i,j] union DUMMIES | i in LIGNES, j in COLONNES];
12
13 array[COLONNES]  of var bool: l1, l2;
14 array[COLONNES]  of var COLONNES: rangs;
15 array[LIGNES]    of var int : derniers;
16 array[LIGNES,COLONNES] of var int : v, vs;
17 array[LIGNES,COLONNES] of var set of int : BS;
18 var int: card, supp;
19
20 constraint forall(i in LIGNES, j in COLONNES)(v[i,j] in BD[i,j]);
21 constraint card >= 1;
22 constraint supp = sum([bool2int(l1[i])| i in LIGNES]);
23 constraint supp >= supp_min;
24 constraint forall(i in LIGNES)(
25   arg_sort([v[i,j]| j in COLONNES], rangs));
26 constraint forall(i in LIGNES, j in COLONNES)(
27   element(rang[j],[v[i,jj] | jj in COLONNES], vs[i,j]));
28 constraint forall(i in LIGNES, j in COLONNES)(
29   element(rangs[j], [BS[ii,j]| ii in LIGNES], BD[i,j]));
30 constraint forall(i in LIGNES)(vs[i,1] = min(BS[i,1]));
31 constraint forall(i in LIGNES, j in 2..lignes)(
32   next_greater_element(vs[i,j-1], vs[i,j], BS[i,j]));
33 constraint forall(j in COLONNES)(l2[j] <-> (rang[j] <= card));
34 constraint forall(i in LIGNES)(
35   element(card, [vs[i,j]| j in COLONNES], derniers[i]));
36 constraint forall(i in LIGNES)(
37   l1[i] <-> (dernier[i] not in DUMMIES));
```

FIGURE 3.1 – Sous-séquences fréquentes sans répétition de caractères

4. i.e. l'ordre n'a pu être répliqué en sélectionnant les localisations minimales et cohérentes. Si, pour une séquence, ce dernier caractère appartient à $DUMMIES$ cela signifie que les valeurs de B ne suffisent pas pour reproduire le motif. Et, si ce denier caractère n'appartient pas à $DUMMIES$ cela exhibe un choix de localisations pour chaque caractère qui permet de répliquer la sous-séquence.

arg_sort : Cette contrainte prend en paramètre deux vecteurs de variables x et $perm$ s'assurant que la permutation définie par $perm$ permet d'ordonner, par ordre croissant, les variables de x. Soit n la taille des vecteurs x et $perm$, pour tout $i \in [1, n-1]$ nous avons $x[perm[i]] < x[perm[i+1]]$.

```
1 set of int: RANGE;
2
3 array[RANGE] of var int : V;
4 var int: x, y;
5
6 constraint member(x, V);
7 constraint forall(i in RANGE)(v[i] < y+1 <-> v[i] < x);
```

FIGURE 3.2 – next_greater_than

next_greater_element : Cette contrainte s'applique à deux variables entières x et y, et une variable ensembliste V. Cette contrainte affecte à x la plus petite valeur de v qui est strictement supérieure à y. Cette contrainte peut altérer les bornes inférieures de V et de x, et la borne supérieure de y. Dans le catalogue de contraintes globales [3], celle-ci prend en paramètre un tableau de variables triées. Néanmoins, dans le modèle présenté ci-dessus ce tableau est substitué par une variable ensembliste. Pour un ensemble V fourni en tant que donnée et non comme variable ensembliste, cette contrainte peut se reformuler comme dans la figure 3.2. Ce dernier modèle poste une contrainte de domaine (ligne 6), puis poste un ensemble d'inégalités servant à encadrer la variable x par des valeurs de V. Reprenons la contrainte présentée dans la figure 3.1, celle-ci dispose d'une variable ensembliste employée dans un contexte spécifique. En effet, BS sert uniquement à réordonner la base de données B, et avant que cette réorganisation ne survienne, le domaine de ces variables ensemblistes reste peu informatif (i.e. il représente à peu près tout ensemble possible). Le degré de propagation de cette contrainte est donc faible dans le modèle (figure 3.1), celle-ci n'intervient que lorsque les variables ensemblistes de BS sont pleinement déterminées - ce qui se produit lors de l'affectation des variables de $rangs$.

Branchement : Concernant le branchement, comme présenté en figure 3.3, il s'agit de sélectionner les caractères de la sous-séquence un à un en partant de la gauche (vecteur $rangs$). Une fois les caractères sélectionnés, il ne reste plus qu'à déterminer la longueur de la sous-séquence en choisissant l'instanciation de la variable $card$. L'instanciation des variables restantes peut être garantie en forçant l'arc consistance des autres contraintes. En effet, le fait de sélectionner systématiquement les localisations minimales des caractères assure le choix de ces localisations une fois le choix des caractères de la sous-séquence effectué (affectation des variables $rangs$).

```
1 solve :: seq_search([
2   int_search(rangs, input_order, indomain_min, complete),
3   int_search([card], input_order, indomain_min, complete)]) satisfy;
```

FIGURE 3.3 – Heuristique de branchement

Limites : Le niveau de propagation de ce modèle est faible car le choix des localisations minimales dépend des BS. De ce fait, le modèle n'est pas efficient et ne peut aucunement prétendre à un quelconque passage à l'échelle en terme de nombre de séquences et de caractères de l'alphabet.

3.2.2 Sous-séquence fréquente avec répétition

Ce deuxième problème vise à extraire une sous-séquence fréquente avec répétition de caractères dans un ensemble de séquences

Nous considérons une base B de séquences construites sur un alphabet Σ, qui est donnée sous la forme d'une matrice dont les lignes sont indicées sur un ensemble L_1 et les colonnes sont indicées sur un ensemble L_2 et f un seuil de fréquence donné.

Ayant pour objectif de calculer des motifs autorisant la répétition de caractères , nous définissons L_2 comme l'ensemble des occurrences possibles des caractères de Σ sur L_1, et non pas simplement comme l'ensemble Σ. Formellement L_2 est l'ensemble $\bigcup_{a\in\Sigma} E(a)$ où $E(a) = \{(a,1),\cdots,(a,n_a)\}$ et n_a dénote le nombre maximum d'occurrences du caractère par séquence. Nous noterons $\equiv$ la relation d'équivalence entre éléments de L_2. (i.e. partageant le même caractère). Avec ces données nous définissons le problème comme l'instance MMP $(M, \prec, C, C^-)$ tel que le domaine matriciel $M = (M^{(1)}, M^{(2)})$ avec $M^{(1)} = \mathbb{N}_{(L_1,L_2)}$ et $M^{(2)} = [\![|L_2|]\!]_{(L_2,\{1\})}$, $\prec = (\prec^{(1)}, \prec^{(2)}) = (\prec_1, =)$, $C^+ = \mathcal{C}\backslash C^-$, $C^+ = \{\text{freq}^{[1]}_{\geqslant,f}\}$ et $C^- = \{\text{db}^{[1]}_B, \text{total}^{[1,2]}_{\prec}, \text{equivalence}^{[1]}_{\{\equiv,<j\}}\}$. Ce MMP a pour solution des matrices de la forme $m = (m^{(1)}, m^{(2)})$ telles que les lignes de $m^{(1)}$ correspondent aux séquences couvertes par le motif et les colonnes de $m^{(1)}$ correspondent aux caractères du motif. Les coefficients de la matrice $m^{(1)}$ correspondent aux enracinements, cohérents avec B, des caractères au sein des séquences couvertes. La matrice $m^{(2)}$, quant à elle, représente la relation d'ordre total (représentée sous la forme d'une permutation) entre les colonnes de $m^{(1)}$ et basée sur les coefficients de $m^{(1)}$. La contrainte de fréquence impose de couvrir au moins f lignes. La contrainte d'équivalence impose un ordre total entre les colonnes de $m^{(1)}$ appartenant à une même classe d'équivalence. Dans ce MMP cette contrainte est subsumée par la contrainte d'ordre totale qui impose un ordre total sur toute les colonnes.

En d'autres termes, ce modèle vise à résoudre le problème de recherche de sous-séquences fréquentes au sein d'un ensemble de séquences et avec répétition de caractères

Modèle PPC : Nous présentons ci-dessous le modèle PPC dédié à la résolution du MMP présenté précédemment.

```
1 include "element.mzn";
2
3 predicate next_greater_element(var int: y, var int: x, set of int: v)
4   = assert((card(v)>0), "error",
5     (x in v) /\ forall(e in v)(e < (y+1) <-> e < x));
```

FIGURE 3.4 – Sous-séquence fréquente avec répétition (préambule)

Données : Nous retrouvons, dans ce modèle, la base de données B avec ses lignes et colonnes, ainsi que la longueur maximale ($RANGS$) de la sous-séquence à extraire.

Variables : La combinaison des variables $ordre$ et $d2$ définit la sous-séquence extraite. $ordre$ contient la séquence dont seuls les $d2$ premiers caractères forment le motif. v contient les localisations des caractères de l'ordre dans les séquences. Les colonnes de v correspondent aux caractères (comme $ordre$) du motif, et les lignes aux séquences. v_proj enregistre pour chaque caractère, chaque rang et chaque séquence, la localisation minimale du caractère dans la séquence cohérente avec le préfixe de longueur $rang-1$ de la sous-séquence extraite. Par exemple, $v\_proj[1,2,4] = 10$ signifie que si le symbole 4 est sélectionné pour le rang 1, alors sa localisation dans la séquence 2 serait 1 (ce qui se traduit par $v[1,2] = 4$ et $ordre[1] = 4$). cov et $d1$ correspondent respectivement à la couverture et au support du motif. r_cov et r_supp représentent l'évolution de la couverture et du support après chaque nouvelle projection. cov_proj et $supp\_proj$ fournissent l'évolution du support et de la couverture pour tous les caractères (i.e. les éléments de r_cov et r_supp sont déterminés à partir de ceux de cov_proj et $supp\_proj$). r_symbs utilise des informations de $supp\_proj$ afin de détecter les caractères encore fréquents après projection. Il indique à l'aide de variables booléennes, les caractères valides pour chaque localisation du motif. Le dernier tableau de variables r_ordre détecte si une projection est encore possible. Si aucune projection n'est possible, alors r_ordre prend la valeur $false$, sinon le choix s'effectuera lors du branchement.

Contraintes : Les deux premières contraintes lignes 35-36 calculent la longueur de la sous-séquence extraite et imposent que celle-ci contienne au moins un caractère. La contrainte lignes 39-40 calcule les localisations des caractères sur toute les séquences pour le premier rang du motif. La contrainte suivante lignes 41-42 traite le même objectif que la précédente mais pour les rangs suivants. Cette fois-ci, la contrainte emploie la contrainte globale `next_greater_element`. Pour chaque séquence et chaque rang, cette dernière contrainte utilise la localisation du précédent caractère dans la séquence (i.e. du dernier caractère du préfixe),

```
7  int: cols;
8  int: ligs;
9  int: long;
10 int: support;
11
12 set of int: COLONNES = 1..cols;
13 set of int: LIGNES   = 1..ligs;
14 set of int: RANGS = 1..long;
15 array[LIGNES,COLONNES] of set of int: B;
16
17 int: MAX_B = max([max(B[i,j])| i in LIGNES, j in COLONNES]);
18 set of int: DUMMIES = (MAX_B+1)..(MAX_B+1+cols);
19 array[LIGNES,COLONNES] of set of int: BD
20   = [B[i,j] union DUMMIES | i in LIGNES, j in COLONNES];
21
22 array[RANGS]  of var int: ordre; % symboles de la sous-sequence
23 array[LIGNES,RANGS] of var int : v; % localisations des symboles sur les sequences
24 array[LIGNES,RANGS] of var bool : r_cov; % couvertures des prefixes
25 array[LIGNES] of var bool : cov; % couverture de la sous-sequence
26 array[RANGS] of var int : r_supp; % supports des prefixes
27 array[LIGNES,RANGS,COLONNES] of var int : v_proj; % localisations des symboles apres
       projection
28 array[LIGNES,RANGS,COLONNES] of var bool : cov_proj; % couvertures des symboles apres
       projection
29 array[RANGS,COLONNES] of var int : supp_proj; % supports des symboles apres projection
30 array[RANGS,COLONNES] of var bool : r_symbs; % symboles frequents au rang r
31 array[RANGS] of var bool : r_ordre; % appartenance d'un rang a la sous-sequence
32 var int: d2;
33 var int: d1;
```

FIGURE 3.5 – Sous-séquence fréquente avec répétition (données et variables)

localisation qui est enregistrée dans v. Cet enregistrement intervient lignes 54-55 en sélectionnant, pour chaque cellule de la matrice v, les valeurs obtenues par projection (v_proj).

Les contraintes lignes 43-46 calculent, pour chaque caractère, sa couverture après projection. La couverture s'obtient en analysant les localisations des caractères obtenues par projection (un caractère n'est pas couvert s'il doit sélectionner une localisation dans $DUMMIES$). La couverture dépend également de la couverture du préfixe (la couverture est anti-monotone). La contrainte suivante lignes 47-48 calcule les supports après projection à partir des couvertures précédemment déterminées. Les caractères fréquents après projection sont alors déterminés grâce aux supports obtenus précédemment (lignes 49-50).

La contrainte lignes 52-53 se comporte de deux façon différentes : si pour un rang donné du motif un caractère doit être sélectionné (i.e. la séquence doit être étendue vers la droite), alors seul les caractères encore fréquents après projection peuvent être sélectionnés. Au contraire, si le motif ne doit pas être étendu, alors la contrainte aux lignes 52-53 et celle à la ligne 54 imposent le choix du caractère 1. Dans la contrainte lignes 52-53 r_ordre apparaît à deux endroits, ce qui n'est pas dérangeant car il est attendu que cette contrainte ne se déclenche qu'après le choix d'une valeur pour r_ordre.

Les trois dernières contraintes (lignes 56-61) font le lien entre les localisations, couvertures et supports obtenus par projection pour chaque caractère avec les localisations, couvertures et supports du caractère effectivement sélectionné dans la sous-séquence.

```
35 constraint d2 = sum([bool2int(r_ordre[rang])|rang in RANGS]);
36 constraint d2 >= 1 /\ r_ordre[1];
37 constraint element(d2, r_supp, d1);
38
39 constraint forall(i in LIGNES, j in COLONNES)(
40   v_proj[i,1,j] = min(B[i,j]));
41 constraint forall(i in LIGNES, j in COLONNES, rang in 2..long)(
42   next_greater_element(v[i,rang-1], v_proj[i,rang,j], BD[i,j]));
43 constraint forall(i in LIGNES, j in COLONNES)(
44   cov_proj[i,1,j] <-> v_proj[i,1,j] <= MAX_B);
45 constraint forall(i in LIGNES, j in COLONNES, rang in 2..long)(
46   cov_proj[i,rang,j] <-> (r_cov[i,rang-1] /\ v_proj[i,rang,j] <= MAX_B));
47 constraint forall(j in COLONNES, rang in RANGS)(
48   supp_proj[rang,j] = sum([bool2int(cov_proj[i,rang,j]) | i in LIGNES]));
49 constraint forall(j in COLONNES, rang in RANGS)(
50   r_symbs[rang,j] <-> (supp_proj[rang,j] >= support));
51
52 constraint forall(rang in RANGS)(
53   element(ordre[rang],[r_ordre[rang] /\ r_symbs[rang,j] | j in COLONNES],r_ordre[rang
     ]));
54 constraint forall(rang in RANGS)( not r_ordre[rang] -> ordre[rang] = 1);
55 constraint forall(rang in 2..long)(not r_ordre[rang-1] -> not r_ordre[rang])
56 constraint forall(i in LIGNES, rang in RANGS)(
57   element(ordre[rang],[v_proj[i,rang,j] | j in COLONNES],v[i,rang]));
58 constraint forall(rang in RANGS)(
59   element(ordre[rang],[supp_proj[rang,j] | j in COLONNES],r_supp[rang]));
60 constraint forall(i in LIGNES, rang in RANGS)(
61   element(ordre[rang],[cov_proj[i,rang,j] | j in COLONNES],r_cov[i,rang]));
```

FIGURE 3.6 – Sous-séquence fréquente avec répétition (contraintes)

Branchement : La figure 3.7 présente l'heuristique de branchement employée. Elle sélectionne les caractères du motif du rang le plus faible vers le rang le plus élevé. Chaque caractère est sélectionné en deux temps : le premier choix concerne la présence d'un caractère à un rang donné et le deuxième la sélection du caractère.

```
63 array[R] of ann : branch_o = [int_search([ordre[r]], input_order, indomain_min,
     complete)| r in R];
64 array[R] of ann : branch_a = [bool_search([r_ordre[r]], input_order, indomain_min,
     complete)| r in R];
65 array[R] of ann : brancho_ao = [seq_search([branch_a[r],branch_o[r]])| r in R];
66 solve :: seq_search(brancho_oa) satisfy;
```

FIGURE 3.7 – Sous-séquence fréquente avec répétition (branchement)

3.2.3 Sous-séquence fréquente avec répétition et semi-fermée

Ce section présente une modification apportée au modèle PPC de la précédente section par l'ajout d'une contrainte de pseudo-fermeture. Une séquence est dite fermée si elle est fréquente et si toute super-séquence (une séquence l'incluant) est soit non-fréquente, soit moins fréquente. Nous favoriserons la sélection d'une super-séquence tant que celle-ci reste fréquente. Afin d'y parvenir nous construisons la sous-séquence au fur et à mesure par extension de ses préfixes. Autrement dit, nous étendons la sous-séquence en cours de construction tant qu'elle peut être étendue avec un caractère qui permet de respecter le seuil de fréquence.

La figure 3.8 présente les modifications à réaliser sur le modèle précédent pour intégrer cette contrainte. Il s'agit de changer l'heuristique de branchement, en enlevant la possibilité de sélectionner un caractère à un rang donné, et d'imposer, à l'aide d'une nouvelle contrainte, la sélection d'un caractère fréquent après projection.

```
1 constraint forall(rang in RANGS)(sum(j in COLONNES)([bool2int(r_symbs[r,j])) > 0 <->
      r_ordre[rang]);
2 solve :: seq_search(ordre) satisfy;
```

FIGURE 3.8 – Sous-séquence fréquente avec répétition (pseudo-fermeture)

3.2.4 Sous-séquence fréquente et discriminante

Ce quatrième problème vise à extraire une sous-séquence fréquente avec répétition de caractères dans un ensemble de séquences positives et absent de séquences négatives.

Nous considérons une base B de séquences construites sur un alphabet Σ, qui est donnée sous la forme d'une matrice dont les lignes sont indicées sur un ensemble L_1 et les colonnes sont indicées sur un ensemble L_2 et f un seuil de fréquence donné. L_1 est l'ensemble des identifiants de séquences que nous supposons bi-partitionné en deux sous-ensembles L_1^+ et L_1^- dites respectivement positives et négatives. L_2 est l'ensemble des occurrences possibles des caractères de Σ sur L_1^+ et défini par l'ensemble $\bigcup_{a\in\Sigma} E(a)$ où $E(a) = \{(a,1), \cdots, (a,n_a)\}$ et n_a dénote le nombre maximum d'occurrences du caractère par séquence. Nous noterons $\equiv$ la relation d'équivalence entre éléments de L_2. (i.e. partageant le même caractère). Avec ces données nous définissons le problème comme l'instance MMP $(M, \prec, C, C^-)$ tel que le domaine matriciel $M = (M^{(1)}, M^{(2)})$ avec $M^{(1)} = \mathbb{N}_{(L_1,L_2)}$ et $M^{(2)} = [\![|L_2|]\!]_{(L_2,\{1\})}$, $\prec = (\prec^{(1)}, \prec^{(2)}) = (\prec_1, =)$, $C^+ = \mathcal{C}\backslash C^-$, $C^+ = \{\text{freq}^{[1]}_{\geqslant,f}\}$ et $C^- = \{\text{db}^{[1]}_B, \text{total}^{[1,2]}_{<}, \text{equivalence}^{[1]}_{\{\equiv,<j\}}, \text{cover}^{[1]}_{\{\subseteq,L_1^+\}}\}$.

En d'autres termes, ce modèle vise à résoudre le problème de recherche de sous-séquences avec répétition de caractères, fréquentes au sein des séquences positives et absentes des sé-

quences négatives (la contrainte de couverture impose de ne couvrir que des séquences positives).

Modèle PPC : Le modèle PPC proposé ne résout pas ce MMP mais une variante optimisation de celui-ci. L'optimisation, non intégrée au problème MMP qui est un problème de satisfaction, porte sur la minimisation du nombre de séquences négatives exclues par le motif, ce critère remplace donc la contrainte imposant la non-couverture de séquences négatives.

La figure 3.9 présente les modifications à apporter aux modèles précédents afin d'introduire ces séquences négatives. Il faut notamment introduire deux ensembles de séquences disjoints NEG et POS dans les données. Le support calculé en cours de construction (cov_proj) porte uniquement sur les séquences positives (lignes 3 et 4). Le support global (r_cov) porte sur toutes les séquences (lignes 5 et 6). Ainsi, le critère d'optimisation ne requiert plus que la minimisation du support global (support des séquences positives et négatives).

```
1 set of int: NEGS;
2 set of int: POS = diff(B,NEGS);
3 constraint forall(j in POS, rang in RANGS)(
4   supp_proj[rang,j] = sum([bool2int(cov_proj[i,rang,j]) | i in LIGNES]));
5 constraint forall(rang in RANGS)(
6   r_suppr[rang] = sum(i in LIGNES)(bool2int(r\_cov[i,rang])));
7
8 solve minimize d1;
```

FIGURE 3.9 – Ordre Total Exclusion

3.2.5 Contrainte globale dédiée à l'exclusion

Le calcul des enracinements successifs s'avère utile mais il reste coûteux en espace mémoire. Une manière d'évincer ce problème consiste à employer une contrainte globale qui reproduit ce mécanisme. Deux contraintes peuvent être réalisées, l'une avec un rôle de propagation sur la partie positive et l'autre dédiée à l'exclusion sur la partie négative. Non seulement certaines variables mais encore l'ensemble $DUMMIES$ tout entier deviennent inutiles (figure 3.5). Concernant la partie positive, nous retrouvons les contraintes de Kemmar et Guns, même si celles-ci fonctionnent uniquement sur des séquences, alors que nos modèles sont aptes à traiter séquences et séquences d'itemsets (deux caractères distincts peuvent avoir la même localisation sur une même séquence). Aussi présentons-nous, ci-dessous, la contrainte globale dédiée à l'exclusion de séquences (négatives). La section "expérimentations" atteste d'autant l'intérêt d'employer une telle contrainte.

3.2.5.1 Déroulement

La description du propagateur est divisée en trois parties : la structure (les variables), l'initialisation et la propagation lors de la réduction des domaines des variables. Les phases d'initialisation et de propagation retournent l'état de la recherche. Nous trouvons trois états distincts : $ECHEC$ indiquant que la propagation entraîne l'épuisement du domaine d'une des variables [5], $FINI$ indiquant que le propagateur ne pourra plus jamais inférer de nouvelles informations lors de la recherche, et $POINT\_FIXE$ indiquant que le propagateur ne peut plus inférer d'information dans l'état actuel de la recherche, mais il pourra éventuellement réduire le domaine d'autres variables lors des prochaines affectations de variables liées au propagateur.

Dans la structure nous différencions les variables propres à la structure et les variables issues du modèle PPC par l'introduction du mot-clé var, à l'instar des modèles minizinc.

Parmi les deux principales procédures, nous trouvons la phase d'initialisation et la phase de propagation. La première est appelée à la création du modèle et la seconde l'est à chaque modification du domaine d'une des variables représentant le motif (ajout de caractère). Ces deux procédures renvoient l'état dans lequel se trouve la recherche ($FINI$, $ECHEC$ ou $POINT\_FIXE$).

Structure : La structure du propagateur, présentée dans l'algorithme 6, contient, entre autres, toutes les variables de décisions du modèle PPC représentant le motif, à savoir $ordre$ et $taille$. $ordre$ représente l'ordre d'apparition des caractères dans la sous-séquences et $taille$ dénombre le nombre de caractères du motif (i.e. seuls les $taille$ premiers caractères du vecteur $ordre$ constituent la sous-séquence). Le vecteur $ordre$ est indicé sur l'ensemble $[\![long]\!]$, où $long$ fixe la longueur maximale de la sous-séquence, et les variables du vecteur $ordre$ prennent leurs valeurs dans l'ensemble $SYMB$ ($long = |SYMB|$). Nous trouvons un tableau contenant les localisations possibles pour chaque caractère $LOCS$ (il s'agit d'un tableau à une dimension, qui représente une ligne de B, c'est-à-dire une séquence). La variable de décision $excl$ exprime si la sous-séquence définie par $ordre$ et $taille$ peut être reproduite sur la séquence représentée par $LOCS.\ excl$ prend la valeur $true$ si le motif ne peut pas être reproduit et la valeur $false$ sinon. Finalement, nous définissons trois variables propres au propagateur : $locmin$ qui correspond à la plus petite valeur de localisation possible de $LOCS$ (si elle existe), $rmin$ qui correspond au nombre de variables déjà considérées par le propagateur (il s'agit nécessairement des $rmin$ premiers caractères de la sous-séquence) et loc qui correspond au choix d'une localisation pour tout caractère de la sous-séquence (il s'agit de la localisation minimale et cohérente).

5. L'épuisement du domaine d'une variable ne survient jamais explicitement dans le propagateur présenté. Cependant, un échec peut survenir lorsque le propagateur décide de l'exclusion ou non d'une séquence ce qui entraîne la modification du domaine de la variable de décision représentant l'exclusion d'une séquence, pouvant ainsi entraîner l'épuisement du domaine de cette variable.

Algorithme 4 – Vérificateur Ordre Total (Structure)

```
1 struct {
        set of int :                SYMB ;
        int :                       long ;
        set of 1..long :            RANGS ;
        array[RANG] of var SYMB :   ordre ;
        var int :                   taille ;
2       array[SYMB] of set of int : LOCS ;
        var bool :                  excl ;
        array[0..long] of int :     loc ;
        int :                       locmin ;
        int :                       rmin ;
3 } VerificateurTotal;
```

Initialisation : La phase d'initialisation (algorithme 5) commence par affecter à la structure les différents paramètres fournis au propagateur. Cette création n'est pas détaillée, il s'agit simplement d'affecter aux champs de la structure les paramètres de même nom. Ensuite, nous affectons à $rmin$ la valeur 1 pour indiquer qu'aucun caractère de la sous-séquence n'a encore été considéré, et à $locmin$ soit la plus petite valeur de $LOCS$, soit la valeur 0 dans le cas où les ensembles de $LOCS$ sont vides. Finalement, l'initialisation se termine par le déclenchement de la phase de propagation.

Algorithme 5 – Vérificateur Ordre Total (Initialisation)

Entrée : Vérificateur v
Sortie : État de la propagation

```
1  poster(SYMB, long, ordre, taille, LOCS, excl) :
2  begin
3      v ← créerVérficateur(SYMB, long, ordre, taille, LOCS, excl) ;
4      v.rmin ← 1 ;
5      if ∀s ∈ v.SYMB B[s] == ∅ then
6          v.locmin ← 0 ;
7      else
8          v.locmin ← min({l|s ∈ v.SYMB ∧ l ∈ v.LOCS[s]}) ;
9      end
10     foreach r ∈ 0..v.long do v.loc[r] ← v.locmin − 1;
11     return propager(v) ;
12 end
```

Propagation : La phase de propagation est présentée dans l'algorithme 6. La première ligne vérifie que le motif n'est pas vide, en effet s'il est vide il ne peut exclure aucune séquence [6]

6. L'instruction **return** suivi d'un **if** signifie que l'algorithme renvoie la valeur située juste après le **return** uniquement si celle-ci respecte la condition du **if** (ici, si la valeur est différente de $CONTINUE$).

(algorithme 7). Le second test concerne la base de localisation, si celle-ci est vide et que le motif contient au moins un caractère, alors la séquence sera nécessairement exclue (algorithme 7).

L'instruction suivante tente de localiser les caractères au sein des séquences (algorithme 8). La boucle principale de cet algorithme parcourt toutes les variables qui constituent la sous-séquence en commençant par celle la plus à gauche et qui n'était pas affectée ($rmin$) lors des précédents appels de l'algorithme de propagation, jusqu'à la variable la plus à droite qui appartient nécessairement à la sous-séquence ($min(taille)$). Pour commencer, un ensemble B' des localisations autorisées est créé. Ce dernier contient pour une variable de rang r les valeurs de $LOCS[ordre[r]]$ le caractère r de la sous-séquence est connu. Dans le cas où B' n'est pas connu B' contient les valeurs des $LOCS$ dont les indices appartiennent au domaine de $ordre[r]$. Dans les deux cas nous filtrons les valeurs de B' vis-à-vis de la localisation choisie pour le précédent caractère de la sous-séquence [7]. (c'est pour cela que loc est indicé sur l'ensemble $\{0, \cdots, long\}$ et non $\{1, \cdots, long\}$ comme l'est $ordre$). Si B' est vide cela signifie que le motif ne peut pas être reproduit. Dans le cas où B' n'est pas vide la valeur de $loc[r]$ est mise à jour avec la plus petite valeur de B', et si $ordre[r]$ est connu et que r correspond au premier rang non encore affecté ($rmin$) alors $rmin$ est incrémenté.

Enfin, le propagateur vérifie si la sous-séquence est pleinement déterminée. Si cela s'avère être la cas, alors la séquence ne peut pas être exclue. En effet l'instruction précédente tente de reproduire la sous-séquences et interrompt la propagation si la séquence est exclue.

Algorithme 6 – Vérificateur Ordre Total (Prop)

Entrée : Vérificateur v
Sortie : État de la propagation

```
1 propager(v) :
2 begin
3     return tailleNonNulle(v) if differentDe(CONTINUE);
4     return absenceSymboles(v) if differentDe(CONTINUE);
5     return verifierOrdre(v) if differentDe(CONTINUE);
6     return testerFin(v) if differentDe(CONTINUE);
7     return POINT_FIXE ;
8 end
```

3.3 Ordre Partiel

Dans cette section, nous présentons le modèle dédié à l'extraction d'ordre partiel fréquent. Soit une base de séquences pré-classées en séquences positives et séquences négatives telle que tout caractère de l'alphabet apparaît dans toute séquence positive. L'objectif est de trouver un

7. B' n'a pas besoin d'être déterminé intégralement il suffit d'ordonner les localisations de $LOCS$ et de choisir la première valeur valide

Algorithme 7 – Vérificateur Ordre Total (Prop)

Entrée : Vérificateur v
Sortie : État de la propagation

```
1  tailleNonNulle(v) :
2  begin
3      if max(v.taille) ⩽ 0 then
4          v.excl ← false ;
5          return FINI ;
6      end
7      return CONTINUE ;
8  end
9  absenceSymboles(v) :
10 begin
11     if min(v.taille) > 0 ∧ ∀s ∈ v.SYMB B[s] == ∅ then
12         v.excl ← true ;
13         return FINI ;
14     end
15     return CONTINUE ;
16 end
```

Algorithme 8 – Vérificateur Ordre Total (Vérifier Ordre)

Entrée : Vérificateur v
Sortie : État de la propagation

```
1  verifierOrdre(v) :
2  begin
3      foreach r ∈ v.rmin..min(size) do
4          B' ← ∅ ;
5          if assigné(ordre[r]) then
6              B' ← {l|l ∈ v.LOCS[val(v.ordre[r])] ∧ l > v.loc[r − 1]};
7          end
8          else
9              B' ← {l'|d ∈ dom(v.ordre[r]) ∧ l' ∈ v.LOCS[d] ∧ l' > v.loc[r − 1]} ;
10         end
11         if B' == ∅ then
12             v.excl ← true ;
13             return FINI ;
14         end
15         v.loc[r] ← min(B') ;
16         if estAssigné(v.ordre[r]) ∧ r == v.rmin then
17             v.rmin ← v.rmin + 1 ;
18         end
19     end
20     return CONTINUE ;
21 end
```

Algorithme 9 – Vérificateur Ordre Total (Tester Fin de la Propagation)

Entrée : Vérificateur v
Sortie : État de la propagation

```
1 verifierOrdre(v) :
2 begin
3     if estAssigné(v.taille) ∧ v.rmin > val(v.taille) then
4         v.excl ← false ;
5         return FINI ;
6     end
7     return CONTINUE
8 end
```

motif optimal défini par un ensemble de caractères consistants avec la base de séquences et un ordre partiel commun sur les séquences positives. L'optimalité porte sur la minimisation du nombre de séquences négatives exclues par le motif - pour une séquence, l'exclusion s'établit soit par l'absence d'un caractère du motif sur la séquence, soit par inconsistance de l'ordre partiel sur la séquence.

Nous réutilisons les mêmes données que pour la section "Sous-séquence fréquente et discriminante" à l'exception du seuil de fréquence qui n'est plus requis et de L_2 qui sera égale à Σ (plus de répétition des caractères).

Avec ces données nous définissons le problème comme l'instance MMP $(M, \prec, C, C^-)$ tel que le domaine matriciel $M = (M^{(1)}, M^{(2)})$ avec $M^{(1)} = \mathbb{N}_{(L_1,L_2)}$ et $M^{(2)} = \mathbb{B}_{(L_2,L_2)}$, $\prec = (\prec^{(1)}, \prec^{(2)}) = (\prec_1, =)$, $C^+ = \mathcal{C} \backslash C^-$, $C^+ = \{\text{cover}^{[1]}_{\{\supseteq, L_1^+\}}\}$ et $C^- = \{\text{db}^{[1]}_B, \text{partial}^{[1,2]}_{\prec}\}$. À ce MMP nous allons ajouter un critère d'optimisation consistant à sélectionner la matrice minimisant le nombre de lignes couvrant L_1^-.

Modèle PPC : Le modèle figure 3.10 présente le modèle final contrairement à la section portant sur les sous-séquences fréquentes. La section suivante introduit la contrainte globale dédiée à l'exclusion de séquences avec un ordre partiel.

Nous noterons que ce modèle utilise des mécanismes similaires à la projection basée sur les préfixes pour tester l'exclusion de séquences.

Données : Ici, nous retrouvons les mêmes données que pour l'ordre total, à savoir une base de séquences B contenant les localisations (représentées sous la forme d'ensembles d'enracinements) de $cols$ caractères sur $ligs$ séquences. Nous retrouvons aussi une classe positive et une classe négative, ainsi que DB qui correspond à la base B dont les cellules sont enrichies d'en ensemble de localisations fictive ($DUMMIES$). Toutefois, le seuil de fréquence disparaît, imposant à l'ordre partiel de couvrir l'intégralité des séquences positives.

```
1 include "element.mzn";
2
3 predicate next_greater_element(var int: y, var int: x, set of int: v)
4   = assert((card(v)>0), "error",
5     (x in v) /\ forall(e in v)(e < (y+1) <-> e < x));
6
7 int: cols;
8 int: ligs;
9
10 set of int: COLONNES = 1..cols;
11 set of int: LIGNES   = 1..ligs;
12 set of int: NEGS;
13 set of int: POS = diff(B,NEGS);
14 array[LIGNES,COLONNES] of set of int: B;
15
16 int: MAX_B = max([max(B[i,j])| i in LIGNES, j in COLONNES]);
17 set of int: DUMMIES = (MAX_B+1)..(MAX_B+1+cols);
18 array[LIGNES,COLONNES] of set of int: BD
19   = [B[i,j] union DUMMIES | i in LIGNES, j in COLONNES];
```

FIGURE 3.10 – Ordre Partiel (Données)

Variables : Les deux variables décrivant l'ordre partiel sont $ordre$ et $caractres$. $caractres$ représente les caractères présents dans l'ordre partiel, tandis que $ordre$ définit la relation d'ordre partiel entre ces caractères [8]. La matrice v représente les localisations choisies pour chaque caractère sur l'ensemble des séquences (positives et négatives). La matrice v_pred_max sert à déterminer, pour les séquences négatives, la localisation maximale des caractères précédents un caractère j. Par exemple, supposons que nous ayons le caractère a ainsi que le caractère b qui précèdent c et que, par un certain mécanisme (explicité dans la section "Contraintes"), nous ayons attribué à une séquence s la valeur 3 à $v[s,a]$ et 5 à $v[s,b]$, alors nous attribuerions

```
25 array[COLONNES]  of var bool: symboles;
26 array[COLONNES,COLONNES]  of var bool: ordre;
27 array[LIGNES,COLONNES] of var int: v;
28 array[LIGNES,COLONNES] of var int: v_pred_max;
29 var int: d2;
```

FIGURE 3.11 – Ordre Partiel (Variables)

la valeur 5 à $v[s,c]$ (maximum entre 3 et 5). Finalement, la variable $d2$ compte le nombre de caractère de l'ordre.

Contraintes : Les contraintes Lignes 27 et 28 imposent la sélection d'au moins un caractère. La contrainte à la ligne 80 force les cellules de v, couvrant la classe positive, à appartenir à la base de données non-enrichie par $DUMMIES$ car toutes les séquences doivent être couvertes (à noter que tous les caractères doivent être présents dans toutes les séquences sinon cette

8. Dans l'ordre extrait, il est envisageable d'obtenir un caractère sans aucune relation d'ordre avec les autres.

contrainte entrainerait un échec du modèle au regard de l'épuisement du domaine d'une des variables de v). La contrainte aux lignes 80 et 81 indique que si un arc est présent entre un caractère $j1$ et un caractère $j2$, alors sur toutes les séquences de la classe positive la localisation de $j1$ doit précéder celle de $j2$. En outre, les caractères $j1$ et $j2$ doivent faire partie de l'ordre.

```
27 constraint d2 = sum([bool2int(symboles[j])|j in COLONNES]);
28 constraint d2 >= 1;
29
30 constraint forall(i in POS, j in COLONNES)(v[i,j] in B[i,j]);
31 constraint forall(i in POS, j1,j2 in COLONNES)(
32   ordre[j1,j2] -> (symboles[j1] /\ symboles[j1] /\ (v[i,j1] < v[i,j2]));
33
34 constraint forall(i in NEGS, j in COLONNES)(v[i,j] in BD[i,j]);
35 constraint forall(i in NEGS, j in COLONNES)(symboles[j] -> v[i,j] >= B_MAX);
36 constraint forall(i in NEGS, j2 in COLONNES)(
37   v_pred_max[i,j2] = max([bool2int(ordre[j1,j2])*v[i,j2] | j1 in COLONNES));
38 constraint forall(i in NEGS, j2 in COLONNES)(
39   next_greater_element(v_pred_max[i,j], v[i,j], BD[i,j]));
```

FIGURE 3.12 – Ordre Partiel (Contraintes)

Pour la partie négative, nous devons ajouter une contrainte de domaine (ligne 84) par rapport à la base enrichie des localisations de $DUMMIES$ afin de vérifier pour un caractère isolé (sans arc entrant ou sortant) s'il est effectivement présent dans les séquences négatives. La contrainte lignes 36 et 37 enregistre pour chaque caractère x et séquence s, la localisation la plus grande sur s, parmi les localisations des caractères qui précède le caractère x dans l'ordre. La contrainte lignes 38 et 39 fonctionne en adéquation avec la précédente et sélectionne la localisation minimale et cohérente pour chaque caractère sur chaque séquence. Les deux contraintes précédentes fonctionnent comme suit : lorsqu'un caractère ne dispose pas de prédécesseur dans l'ordre partiel, alors le tableau généré par la première contrainte ne contiendra que des 0 dus à l'expression `bool2int(ordre[j1,j2])` et donc le maximum sera 0. Une fois ce 0 déterminé la contrainte suivante peut se déclencher et donc choisir la plus petite valeur de BD (0 n'est pas une localisation valide). Une fois les caractères sans prédécesseur dans l'ordre déduit, ceux qui leur succèdent peuvent alors être déterminés, ce qui permet ainsi de déterminer la localisation maximale parmi les prédécesseurs des caractères de rang 2 dans l'ordre partiel [9] permettant ainsi de choisir une localisation pour ces caractères de rang 2. Les deux étapes précédentes sont répétées jusqu'à détermination totale des localisations des caractères de l'ordre.

Branchement : La stratégie de branchement consiste à d'abord sélectionner les caractères puis les arcs entre ces caractères. Pour le choix des valeurs de localisations, il faut simplement garantir qu'une affectation est consistante. À cet effet, nous forçons la borne-consistance des

9. Un caractère est de rang $n > 1$ s'il existe un caractère de rang $n-1$ et qu'il n'existe pas de caractère de rang $n' > n-1$. Un caractère est de rang 1 s'il n'a pas de prédécesseur.

contraintes d'inégalité (lignes 31-32), puisqu'une fois le choix des arcs et des caractères effectués le modèle se réduit à un ensemble d'inégalités, qui permet à la borne-consistance nous garantit, dans ce cas, de garantir l'existence d'au moins une affectation consistante des variables de localisations (e.g., sélection du minimum des domaines une fois la borne-consistance appliquée).

3.3.1 Contrainte globale dédiée à l'exclusion

Comme pour le modèle dédié au calcul de sous-séquences, nous proposons une contrainte globale dédiée à la mesure d'exclusion des séquences négatives. Nous utilisons le même schéma de présentation que celui de la contrainte globale précédente, à savoir une présentation de la structure, de la phase d'initialisation puis de la phase de propagation. Les phases d'initialisation et de propagation renvoient l'état de la recherche parmi les trois possibilités suivantes : $ECHEC$, $FINI$ et $POINT\_FIXE$.

3.3.1.1 Déroulement

Le vérificateur dédié à l'ordre partiel vérifie la cohérence de l'ordre sur une séquence mais il propage également les propriétés intrinsèques à celui-ci : la propriété d'antisymétrie ($a[i,j] \rightarrow a[j,i]$) et la cohérence des arcs ($a[i,j] \rightarrow c[i] \wedge c[j]$). En outre, il fonctionne de façon similaire au vérificateur décrit pour les sous-séquences en choisissant, pour chaque caractère de l'ordre partiel, une localisation minimale et cohérente.

Algorithme 10 – Vérificateur Ordre Partiel (Structure)

```
1 struct {
      set of int :                        SYMB;
      array[SYMB] of var bool :           c;
      array[SYMB,SYMB] of var bool :      a;
      array[SYMB] of bool :               caff;
      array[SYMB,SYMB] of bool :          aaff;
      int :                               ncaffs;
2     int :                               naaffs;
      array[SYMB] of set of int :         LOCS;
      var bool :                          excl;
      array[SYMB] of set of SYMB :        succs;
      int :                               locmax;
      array[SYMB] of int :                locs;
3 } VerificateurPartiel;
```

Structure : Dans la structure algorithme 10 nous trouvons toutes les variables de décisions représentant l'ordre, à savoir c et a qui définissent respectivement les caractères de l'ordre ainsi

que la relation d'ordre entre ces derniers. c et a sont indicés par l'ensemble des caractères $SYMBS$. Les tableaux de variables $caff$ et $aaff$ représentent respectivement les variables de décisions c et a déterminées lors de précédents appels à la fonction de propagation. Ces deux derniers tableaux sont accompagnés de deux variables $ncaffs$ et $naaffs$ qui comptabilisent alors le nombre de variables de décisions déjà affectées. Le tableau des localisations $LOCS$ correspond aux localisations des caractères de l'alphabet sur la séquence testée, la variable d'exclusion $excl$ indique si la séquence est exclue ou non. Le tableau $locs$ contient les localisations minimales choisies pour chaque caractère de l'ordre partiel sur la séquence. La variable $locmax$ correspond à la plus grande valeur de localisation possible de $LOCS$ (si elle existe). Finalement, nous trouvons le tableau $succs$ qui est une représentation du graphe, induit par a, sous la forme d'une liste d'adjacences.

Initialisation : Comme pour le vérificateur présenté pour l'exclusion de sous-séquence la première étape consiste à construire (algorithme 11) la structure associée au propagateur. Si l'ordre partiel est pleinement déterminé et ne dispose d'aucun arc et caractère, alors nous considérons qu'il n'y pas d'exclusion. Avant toute propagation, nous ne disposons d'aucune infor-

Algorithme 11 – Vérificateur Ordre Partiel (Initialisation)

Entrée : Vérificateur v
Sortie : État de la propagation

```
1  poster(SYMB, c, a, LOCS, excl) :
2  begin
3      v ← créerVérficateur(SYMB,c,a,LOCS,excl) ;
4      foreach s ∈ v.Symb do v.caff[s] ← false;
5      foreach s1, s2 ∈ v.Symb do v.aaff[s1, s2] ← false;
6      foreach s ∈ v.Symb do v.succs[s] ← ∅;
7      (v.ncaffs, v.aaffs) ← (0, 0) ;
8      v.locmax ← max_{s∈v.SYMB} (max(v.LOCS[s])) ;
9      foreach s ∈ v.Symb do
10         if v.Locs[s] == ∅ then
11             v.locs[s] = v.locmax + 1 ;
12         else
13             v.locs[s] = min(v.Locs[s]);
14         end
15     end
16     return propager(v) ;
17 end
```

mation liée à l'affectation des variables de décisions définissant l'ordre partiel (c et a). Donc, les variables recensant les cellules de c et a affectées lors de précédentes propagations ($caff$ et $aaff$) sont toutes mises à $false$. Ensuite, nous calculons la localisation maximale qui servira de repère pour exclure une séquence dès qu'un caractère ne peut sélectionner une localisation

cohérente et minimale. D'où l'initialisation du tableau $locs$ soit avec les localisations minimales des caractères pour ceux qui disposent de localisations, soit avec la valeur $locmax + 1$ pour les autres. Finalement, l'initialisation se termine par le déclenchement de la phase de propagation.

Propagation : La phase de propagation est présentée dans l'algorithme 12.

Comme pour le propagateur dédié à la recherche de sous-séquences, les deux premières instructions vérifient l'existence d'un motif ainsi que celle de localisations dans la base de séquences (algorithme 13).

Algorithme 12 – Vérificateur Ordre Partiel (Propagation)

Entrée : Vérificateur v
Sortie : État de la propagation

```
1  propager(v) :
2  begin
3      return ordreVide(v) if differentDe(CONTINUE);
4      return absenceSymboles(v) if differentDe(CONTINUE);
5      return verifierNouveauxSymboles(v) if differentDe(CONTINUE);
6      (etat, modifies) ← verifierNouveauxArcs(v) ;
7      return etat if differentDe(CONTINUE);
8      return exclusionPossible(v, modifies) if differentDe(CONTINUE);
9      return POINT_FIXE ;
10 end
```

La troisième instruction appelle l'algorithme 14 qui vérifie d'abord les caractères appartenant au motif en regardant dans le tableau c. Pour chaque caractère dont l'appartenance ou non à l'ordre est connue, nous testons si son affectation n'est pas survenue lors d'une propagation antérieure. Si tel est le cas et que le caractère appartient à l'ordre, nous vérifions qu'il dispose d'une localisation cohérente. Dans le cas où un des caractère ne dispose d'aucune localisation cohérente la séquence est exclue et la phase de propagation s'interrompt. Si le caractère a été nouvellement affecté et qu'il n'appartient pas à l'ordre nous interdisons simplement les arcs constitués du caractère [10].

Une fois l'appartenance des caractères à l'ordre testée, la quatrième instruction de la propagation appelle l'algorithme 15 qui teste la présence des arcs. Comme pour les caractères, seuls les arcs nouvellement assignés sont traités. En outre, si un caractère d'un nouvel arc ne dispose d'aucune localisation la séquence est exclue et la propagation s'interrompt. Pour tout nouvel arc de l'ordre nous forçons la présence des caractères qui le constituent et nous interdisons la présence de l'arc opposé (cf. note de bas de page 8). Chaque caractère lié à un nouvel arc est inséré une unique fois dans la file $modifies$ ce qui sert à modifier la localisation minimale des caractères liés entre eux par une relation d'antécédence à un autre caractère. Finalement,

10. Cela peut mener à un échec du modèle dans le cas où il serait sémantiquement faux dû à la présence d'arcs pour des caractères en dehors de l'ordre.

les nouveaux arcs permettent de générer au fur et à mesure le graphe représentant l'ordre partiel sous la forme d'une liste d'adjacences ($succs$). Cet algorithme renvoie en outre l'état de la propagation, ainsi que les caractères liés à un nouvel arc.

Le prochain algorithme (algorithme 16) se sert des caractères liés à un nouvel arc, précédemment déterminés, afin de tenter de leur affecter une localisation cohérente minimale (lignes 3 à 19). À cet effet, le processus parcourt récursivement [11] les caractères de $modifies$ et change pour chacun de ces caractères les valeurs minimales de leurs successeurs. Si un successeur ne dispose plus d'une localisation cohérente, alors la séquence est exclue et la propagation s'interrompt. Sinon, la boucle s'interrompt lorsque les localisations minimales des caractères ne sont plus modifiées [12].

Algorithme 13 – Vérificateur Ordre Partiel (Domaine Vide)

Entrée : Vérificateur v
Sortie : Etat de la propagation

```
1  ordreVide :
2  begin
3      if
4          ∀s ∈ v.SYMB ¬assignedvalOr(v.c[s], true) ∧
           ∀s1, s2 ∈ v.SYMB ¬assignedvalOr(v.a[s1, s2], true)
5      then
6          v.excl ← false ;
7          return FINI ;
8      end
9      return CONTINUE ;
10 end
11 absenceSymboles :
12 begin
13     if v.caffs + v.naffs = 3 ∗ |v.SYMB| ∧ ∀s ∈ v.SYMB v.B[s] == ∅ then
14         v.excl ← true ;
15         return FINI ;
16     end
17     return CONTINUE ;
18 end
```

Finalement, notre algorithme 16 teste si les arcs non affectés pourront entraîner l'exclusion de la séquence dans un hypothétique futur appel de la fonction de propagation (lignes 20 à 29). L'exclusion reste possible s'il existe un arc dont la localisation minimale du caractère antécédent est supérieure à la localisation du caractère lui succédant. Si aucune exclusion n'est possible, alors la propagation s'arrête là et la séquence n'est pas exclue par le motif.

11. La récursion est simulée par une file.
12. La complexité est dans le pire cas quadratique en fonction du nombre de caractères.

Algorithme 14 – Vérificateur Ordre Partiel (Nouveaux caractères)

Entrée : Vérificateur v
Sortie : Etat de la propagation

```
1  ordreVide :
2  begin
3      foreach s ∈ v.Symb do
4          if estAssigné(c[s]) ∧ ¬v.caff[s] then
5              v.caff[s] ← true ;
6              v.ncaffs ← v.ncaffs + 1 ;
7              if val(v.c[s]) ∧ v.locs[s] > v.locmax then
8                  v.excl ← true ;
9                  return FINI ;
10             end
11             else if ¬val(v.c[s])
12              then
13                 foreach s' ∈ v.Symb do (v.a[s, s'], v.a[s', s]) ← (false, false);
14             end
15         end
16     end
17     return CONTINUE ;
18 end
```

Algorithme 15 – Vérificateur Ordre Partiel (Nouveaux Arcs)

Entrée : Vérificateur v
Sortie : Couple état de la propagation, ensemble de caractères

```
1  verifierNouveauxArcs :
2  begin
3      foreach s1 ≠ s2 ∈ v.Symb do
4          if estAssigné(v.a[s1, s2]) ∧ ¬v.aaff[s1, s2] then
5              v.aaff[s1, s2] ← true ;
6              v.naaffs ← v.naaffs + 1 ;
7              if val(v.a[s1, s2]) then
8                  (v.c[s1], v.c[s2]) ← (true, true) ;
9                  v.a[s2, s1] ← false ;
10                 v.aaff[s2, s1] ← true ;
11                 v.naaffs ← v.naaffs + 1 ;
12                 (v.caff[s1], v.caff[s2]) ← (true, true) ;
13                 if v.locs[s1] > v.locmax ∨ v.locs[s2] > v.locmax then
14                     v.excl ← true ;
15                     return (FINI, ∅) ;
16                 end
17                 if s1 ∉ modifies then
18                     modifies ← ajoutFin(modifies, s1) ;
19                 end
20                 if s2 ∉ modifies then
21                     modifies ← ajoutFin(modifies, s2) ;
22                 end
23                 v.succs[s1] ← v.succs[s1] ∪ {s2} ;
24             end
25         end
26     end
27     return (CONTINUE, modifies) ;
28 end
```

Algorithme 16 – Vérificateur Ordre Partiel (Exclusion Possible)

Entrée : Vérificateur v, $modifies$ file de caractères
Sortie : État de la propagation
Données : $excl$ booléen signalant si la séquence peut encore être exclue

```
1  exclusionPossible :
2  begin
3      modifies ← trier_en_fonction_des_arcs(modifies, v.c, v.a) ;
4      while ¬vide(modifies) do
5          s1 ← tete(modifies) ;
6          modifies ← enleverDebut(modifies) ;
7          foreach s2 ∈ v.succs[s1] do
8              L2 ← {l2|l2 ∈ v.LOCS[S2] ∧ l2 > v.locs[s1]} ;
9              if L2 == ∅ then
10                 v.excl ← true ;
11                 etat ← FINI ;
12                 return v ;
13             end
14             v.locs[s2] ← min(L2) ;
15             if s2 ∉ modifies then
16                 modifies ← ajoutFin(modifies, s2) ;
17             end
18         end
19     end
20     exclusionPossible ← false ;
21     foreach s1 ≠ s2 ∈ v.Symb do
22         if ¬v.aaff[s1, s2] then
23             exclusionPossible ← exclusionPossible ∨ v.locs[s1] ⩾ v.locs[s2] ;
24         end
25     end
26     if ¬exclusionPossible then
27         v.excl ← false ;
28         etat ← FINI ;
29     end
30     return CONTINUE ;
31 end
```

3.3.2 Expérimentations

Toutes nos expérimentions ont été conduites sur deux jeux de séquence protéiques (encodées sur l'alphabet des acides aminés, soit 20 caractères) : Late Embryogenesis Abundant Proteins [25, 26] (LEAP) et Small Heat Shock Proteins [28, 27] (SHSP) contenant respectivement 1371 séquences classées [13] et 3765 séquences.

Pour chaque jeu, les séquence ont été préalablement réparties en classes disjointes en fonction de propriétés biologiques : 13 pour LEAP et 15 pour SHSP. La cardinalité de ces classes varie d'une dizaine de séquences (LEAP 12) jusqu'à environ deux cents séquences (SHSP 9). Les séquences, quant à elles, contiennent de 65 à 500 acides aminés (caractères). La partie expérimentale du chapitre 5 décrit plus en détail ces séquences, notamment parce que les algorithmes qui y sont présentés travaillent directement sur la représentation sous forme de séquences d'acides aminés, contrairement aux expérimentations conduites dans ce chapitre et le suivant. Le chapitre 5 propose un pré-traitement de ces jeux de séquences afin d'extraire, pour chaque classe, des patrons communs à l'intégralité de la classe ainsi que leurs positions de départ dans les séquences. Ce pré-traitement transforme ces jeux de séquences en un ensemble de patrons par classe avec leur localisations possibles dans toutes les séquences du jeu. La figure 3.13 illustre le pré-traitement qui est appliqué aux instances (dans cet exemple, il s'agit de séquences quelconques et non de séquence protéiques). Dans ce chapitre et le suivant nous allons donc travailler sur ces patrons et non sur les caractères. La figure 3.14 propose un visuel des motifs extraits.

		S^+		S^-	
Patron	Séquence :	AABXXCYD	CZDKKAABCZD	AABX	CCFDE
AAB C#D		AAB C#D	AAB C#D C#D	AAB	C#D
AAB C#D		$\{1\}$ $\{6\}$	$\{6\}$ $\{1, 9\}$	$\{1\}$ $\varnothing$	$\varnothing$ $\{2\}$

FIGURE 3.13 – Extraction des patrons à partir de séquences. Soient S^+ un ensemble de séquences, positives, et S^- un ensemble de séquences, négatives. Soient AAB et $C\#D$ deux patrons extraits à partir des séquences de S^+. Les instances sont alors formées des localisations des patrons au sein des séquences de S^+ et S^-. Une instance est alors une matrice d'ensembles d'entiers (potentiellement vide pour les séquences négatives) comme présenté dans les deux dernières lignes du tableau.

Les instances sur lesquelles les expérimentations portent se caractérisent à la fois par un identifiant de classe (numéro) et d'un paramètre nommé "écart" qui est propre au pré-traitement des séquences [14]. Nous noterons LEAP 10 :8 pour signifier l'instance de LEAP de classe 10

13. Il existe des séquences non-classées dans ces jeux de séquences mais celles-ci ne sont pas prises en compte.
14. Ce pré-traitement extrait des patrons sous la forme de chaînes de caractères contenant des jokers (i.e. un

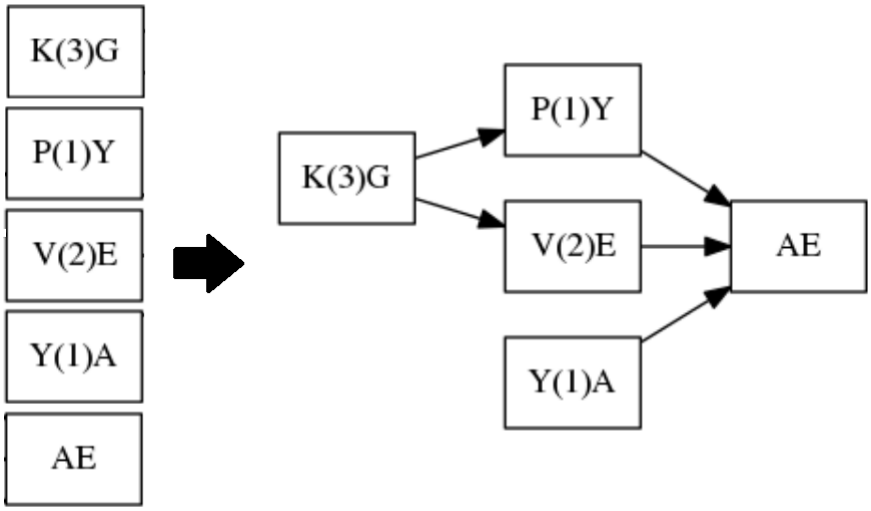

FIGURE 3.14 – Construction d'un ordre partiel à partir de patrons

et d'écart 8. À noter que certaines classes de protéines sont présentes en plusieurs exemplaires (LEAP 1, 2 et 10) tandis que d'autres sont absentes (LEAP 4,6,7,8,9 et 13 ; SHSP 1,6,11,12,18 et 20). Les absentes le sont soit par absence de patrons obtenus lors du pré-traitement, soit par un nombre trop faible de patrons (e.g. un unique patron). Néanmoins, même si ces classes sont absentes des instances leurs séquences sont prises en compte pour l'exclusion. Les classes présentes en plusieurs exemplaires le sont car elles proposent des instances plus complexes à résoudre en fonction du paramètre d'écart maximal.

Chaque instance se décompose en deux ensembles de séquences, celles dites positives (dont le nombre est signalé par la colonne $|S^+|$) et les négatives (dont le nombre est signalé par la colonne $|S^-|$). Les positives correspondent aux séquences de la classe liée à l'instance les négatives sont toutes les autres séquences du jeu.

Cette partie présente trois tableaux comparatifs. Le premier tableau compare différentes approches sur ces instances :

— recherche de motifs non ordonnés [15] (Recouvrement),

— recherche de sous-séquences semi-fermées (Ordre Total)

— recherche de motifs partiellement ordonnés (Ordre Partiel).

Le troisième problème subsume les deux premiers et bénéficie.

Étant donné qu'il s'agit de données réelles, l'exclusion totale des séquences négatives peut ne pas être possible. Nous utiliserons donc les variantes optimisation des modèles qui consistent à maximiser le nombre de séquences négatives exclues. Pour ce faire, nous imposons un critère

joker peut se substituer à n'importe quel autre caractères), l'écart correspond alors au nombre maximum de jokers entre deux caractères non-jokers d'un patron (voir chapitre 5).

15. Il s'agit ici d'un problème de recouvrement d'ensemble minimum dont le modèle n'est pas présenté, car ce problème est uniquement utilisé en terme de capacité d'exclusion des séquences

de minimisation sur le nombre de caractères des motifs. L'objectif correspond alors à une fonction lexicographique qui dans un premier temps, minimise le nombre de séquences négatives exclues, puis minimise la taille des motifs. Une autre différence, par rapport aux modèles dédiés à l'extraction de l'ordre partiel, est l'ajout de la répétition entre caractères.

Le premier (tableau 3.1) présente la comparaison entre les trois problèmes ("Recouvrement", "Ordre Total" et "Ordre Partiel"), le second (tableau 3.2) évalue le recours à des contraintes globales dédiées à l'exclusion pour les problèmes "Ordre Total" et "Ordre Partiel".

Les programmes ont été développés en C++ avec la bibliothèque Gecode (4.0) (Schulte et al. [**?**]) sur "Intel(©) Xeon(©) E5439 - 2.83GHz quad-core (4)".

Le tableau 3.1 est composé des colonnes suivantes :

— le nom de l'instance (colonnes Instance et écart) ;

— le nombre de séquences positives (colonne $|\mathcal{S}^+|$) ;

— le nombre de séquences négatives (colonne $|\mathcal{S}^-|$) ;

— le nombre de patrons (colonne d_2) ;

— le nombre moyen de localisations pour les patrons sur la classe positive (colonne $\bar{D}$) ;

— le nombre minimum de localisations (colonne $D_\vee$) ;

— le nombre maximum de localisations (colonne $D_\wedge$) ;

— le nom des méthodes (Recouvrement, Ordre Total, Ordre Partiel) ;

— pour chaque méthode :

  — le nombre de séquences négatives non-exclues (colonne excl) ;

  — la cardinalité du motif (colonne card) ;

  — le temps d'exécution en milisecondes (colonne t(ms)) ;

  — le nombre d'arcs pour l'ordre partiel (colonne arcs).

15. La recherche de sous-séquence utilise la contrainte de semi-fermeture qui impose de sélectionner des caractères tant que la sous-séquence peut être étendue en restant fréquente. De ce fait, la minimisation de cardinalité ne fournit pas nécessairement les sous-séquences les plus courtes. Néanmoins, le nombre de solutions à examiner est moindre ce qui permet de mener à complétion tous les calculs pour la recherche de sous-séquences.

Instance	écart	$\lvert S^+\rvert$	$\lvert S^-\rvert$	d_2	$\bar{D}$	$D_\vee$	$D_\wedge$	Recouvrement excl	Recouvrement card	Recouvrement t (ms)	Ordre Total excl	Ordre Total card	Ordre Total t (ms)	Ordre Partiel excl	Ordre Partiel card	Ordre Partiel arcs	Ordre Partiel t (ms)
LEAP																	
1	10	208	1163	20	3.2	1	60	86	6	108781	86	4	43784	62	7	9	3600029
1	16	208	1163	29	3.2	1	60	41	16	3600000	66	5	150526	50	5	6	3600000
2	8	92	1279	25	4.3	1	26	12	10	3600000	29	3	64493	168	4	3	3600000
2	15	92	1279	38	4.3	1	26	8	18	3600000	29	3	300536	162	5	3	3600000
2	20	92	1279	45	4.1	1	26	7	23	3600000	12	6	1177304	160	6	4	3600000
3	0	34	1337	9	3.5	1	24	0	1	4	0	4	57	0	1	0	76
5	2	66	1305	10	2.7	1	12	0	4	22	0	4	267	0	3	3	935
10	8	78	1293	21	3.1	1	8	67	14	329361	120	4	12288	61	8	12	3600001
10	15	78	1293	30	2.9	1	8	27	11	3600000	22	4	59520	18	7	9	3600012
10	20	78	1293	38	2.8	1	8	14	20	3600000	16	4	174245	10	7	3	3600018
11	1	35	1336	19	2.7	1	10	1	5	25081	5	4	4906	0	5	2	23135
12	0	17	1354	5	1.0	1	2	0	2	25	0	4	24	0	2	1	37
SHSP																	
2	6	109	3656	14	2.0	1	5	47	13	13667	314	4	7533	0	8	10	758886
3	2	64	3701	13	2.2	1	8	1	4	2911	0	3	3342	0	3	3	2540
4	0	23	3742	10	1.7	1	11	3	2	900	0	4	1011	0	3	3	1116
5	0	25	3740	12	1.6	1	6	0	3	29	0	6	1314	0	2	1	1175
7	10	100	3665	11	1.8	1	6	118	9	1951	718	2	2079	105	10	13	407787
8	1	30	3735	15	1.6	1	4	2	7	27342	2	4	9119	8	7	10	3600000
9	6	238	3527	?	2.5	1	7	82	6	73	217	3	663	8	6	9	2433
13	3	156	3609	11	2.2	1	6	1	6	1535	9	3	2304	0	5	5	8719
14	0	84	3681	20	1.4	1	5	0	1	31	0	6	747	0	1	0	1830
15	0	26	3739	20	1.6	1	5	0	1	29	0	1	651	0	1	0	1426
16	9	75	3690	6	1.4	1	5	0	4	30	6	3	285	0	4	3	631
17	3	88	3677	9	1.5	1	3	30	9	570	14	4	1286	0	5	6	3517
19	1	23	3742	24	2.1	1	8	0	5	5066	0	4	5647	0	4	5	173161
21	2	131	3634	6	2.4	1	10	0	4	23	0	3	130	0	3	3	415
22	0	15	3750	36	2.6	1	9	0	2	83	0	2	3660	0	2	1	18178
23	4	60	3705	6	1.7	1	5	6	5	55	9	3	506	1	5	6	605

TABLEAU 3.1 – Résultats pour l'exclusion

Ce premier tableau compare les trois problèmes entre elles. Nous constatons d'abord que seul le calcul d'ordre total garantit l'optimum global dans la limite de temps de 1 heure. L'ordre partiel est celui qui a le plus de difficultés à terminer dans le temps imparti. Néanmoins, en termes d'exclusion l'ordre partiel a tendance à trouver de meilleurs résultats dans l'heure excepté pour les instances LEAP 1 :16, LEAP 2 :8, LEAP 2 :20 et SHSP 8 :1. Ces mauvais résultats sont imputables à la stratégie de branchement utilisée, qui devrait favoriser la relance sur la

recherche complète, à cause de la faible structuration du problème de recherche d'ordre partiel. Dans le chapitre 4, d'autres stratégies montrent que l'ordre partiel peut obtenir de meilleurs résultats sur toutes les instances.

Nous remarquons également que pour certaines instances, l'ordre total obtient de meilleurs résultats que le recouvrement d'ensembles et inversement. Ce qui signifie que l'une ou l'autre des approches considérées séparément, n'est pa capables de fournir les meilleurs solutions.

En terme de cardinalité, le recouvrement par ensemble semble être la méthode qui fournit les motifs les plus grands. Entre ordre total et ordre partiel, les résultats sont proches cependant d'une cardinalité supérieure pour l'ordre partiel a tendance à signifier une exclusion plus forte (LEAP 1 :10, LEAP 10 :8, LEAP 10 :15, LEAP 10 :20, LEAP 11 :1, LEAP 7 :10, SHSP 14 :0, etc.).

Le tableau 3.2 atteste de l'intérêt d'utiliser des propagateurs pour le test d'exclusion. Ce tableau est composé des colonnes suivantes :

— le nom de l'instance (colonnes Instance et écart) ;

— le nom des méthodes (Ordre Total, Ordre Partiel) ;

— "Prop" et "No Prop" indiquent respectivement l'utilisation du propagateur et sa non utilisation ;

— pour chaque méthode :

  — le nombre de séquences négatives non-exclues (colonne excl) ;

  — la cardinalité du motif (colonne card) ;

  — le temps d'exécution en milisecondes (colonne t(ms)) ;

  — le gain pour les variantes avec propagateur (colonne gain), il s'agit du ratio entre le temps d'exécution sans propagateur sur le temps d'exécution avec propagateur.

		Ordre Total						Ordre Partiel					
		Prop			No Prop			Prop			No Prop		
Instance	écart	excl	card	t (ms)	excl	card	t (ms)	excl	card	arcs	t (ms)	gain	excl	card	arcs	t (ms)	gain	excl	card	arcs	t (ms)
LEAP																					
1	10	86	4	43784	86	4	103718	62	7	9	3600029	1*	62	7	9	3600005	1*	62	7	9	3600005
1	16	66	5	150526	66	5	403001	50	5	6	3600000	1*	50	5	5	3600060	1*	50	5	5	3600060
2	8	29	3	64493	29	3	154094	168	4	3	3600000	1*	168	4	3	3600047	1*	168	4	3	3600047
2	15	29	3	300536	29	3	805399	162	5	3	3600000	1*	162	5	3	3600029	1*	162	5	3	3600029
2	20	12	6	1177304	12	6	3048549	160	6	4	3600000	1*	160	6	4	3600050	1*	160	6	4	3600050
3	0	0	4	57	0	4	379	0	1	0	76	6.7	0	1	0	315	6.7	0	1	0	315
5	2	0	4	267	0	4	1035	0	3	3	935	4.3	0	3	3	2340	4.3	0	3	3	2340
10	8	120	4	12288	120	4	35882	61	8	12	3600001	1*	61	8	12	3600001	1*	61	8	12	3600001
10	15	22	4	59520	22	4	204901	18	7	9	3600012	1*	33	8	10	3600003	1*	33	8	10	3600003
10	20	16	4	174245	22	4	360000	10	7	3	3600018	1*	10	7	3	3600089	1*	10	7	3	3600089
11	1	5	4	4906	5	4	34241	0	5	2	23135	155.6**	0	5	5	345942	155.6**	0	5	5	345942
12	0	0	4	24	0	4	96	0	2	1	37	7.3	0	2	1	157	7.3	0	2	1	157
SHSP																					
2	6	314	4	7533	314	4	24258	0	8	10	758886	1.7	2	9	14	3600197	1.7	2	9	14	3600197
3	2	0	3	3342	0	3	20210	0	3	3	2540	4.1	0	3	3	8724	4.1	0	3	3	8724
4	0	0	4	1011	0	4	3038	0	3	3	1116	5.2	0	3	3	3229	5.2	0	3	3	3229
5	0	0	6	1314	0	6	5191	0	2	1	1175	15.5	0	2	1	10281	15.5	0	2	1	10281
7	10	718	2	2079	718	2	12271	105	10	13	407787	8.8	105	10	13	3600006	8.8	105	10	13	3600006
8	1	2	4	9119	2	4	43431	8	7	10	3600000	1	8	7	10	3600028	1	8	7	10	3600028
9	6	217	3	663	217	3	1711	8	6	9	2433	1.4	8	6	9	10063	1.4	8	6	9	10063
13	3	9	3	2304	9	3	11068	0	5	5	8719	9.2	0	5	3	22926	9.2	0	5	3	22926
14	0	0	6	747	0	6	4282	0	1	0	1830	6.1	0	1	0	6204	6.1	0	1	0	6204
15	0	0	1	651	0	1	6902	0	1	0	1426	6.4	0	1	0	5645	6.4	0	1	0	5645
16	9	6	3	285	6	3	1068	0	4	3	631	4.1	0	4	3	1267	4.1	0	4	3	1267
17	3	14	4	1286	14	4	5158	0	5	6	3517	10.9	0	5	6	6607	10.9	0	5	6	6607
19	1	0	4	5647	0	4	216648	0	4	5	173161	20*	0	4	3	804441	20*	0	4	3	804441
21	2	0	3	130	0	3	453	0	3	3	415	4.1	0	3	3	903	4.1	0	3	3	903
22	0	0	2	3660	0	2	64389	0	2	1	18178	4.5	0	2	1	175652	4.5	0	2	1	175652
23	4	9	3	506	9	3	1623	1	5	6	605	5.6	1	5	7	1633	5.6	1	5	7	1633

TABLEAU 3.2 – Utilisation des contraintes globales

Dans la colonne gain, la présence d'une étoile indique que l'approche sans propagateur n'a pas atteint l'optimal global en une heure, donc que le gain indiqué est incorrect et plus faible que la réalité.

Pour l'ordre total, toutes les instances testées fournissent l'optimum global. Cependant,

l'emploi du propagateur apporte un facteur compris entre deux et trois, allant jusqu'à dix, en termes de vitesse de calcul. Sur l'ordre partiel la différence est plus marquée. En effet, certaines instances arrivent à complétion uniquement lors de la présence du propagateur (LEAP 11 :1, SHSP 19, SHSP 25). Pour les instances qui arrivent à complétion avec et sans propagateur, le gain en temps de calcul est significatif. Pour le reste des instances, qui n'arrivent à complétion pour aucun des modèles, l'utilisation du propagateur fournit de meilleurs résultats. Ce dernier point s'explique en raison d'une exploration plus importante de l'arbre de recherche grâce au gain de vitesse fournie par le propagateur.

Le gain de temps s'explique facilement par un nombre de contraintes et de variables fortement diminuées. En effet, l'absence de propagateur impose la création de nombreuses variables intermédiaires afin de tenter de recréer un ordre sur les séquences négatives. Le propagateur, quant à lui, nécessite beaucoup moins de variables internes. En outre, le propagateur de l'ordre partiel permet de court-circuiter certains tests sous certaines conditions. Par exemple, lorsque toutes les données indéterminées ne peuvent jamais permettre d'exclure la séquence peu importe les combinaisons choisies. De plus, le propagateur n'a plus besoin de réaliser de calcul lorsqu'une exclusion intervient alors que le modèle utilisant des variables intermédiaires continue d'affecter ces variables.

4

Méthode approchée pour le calcul de motifs partiellement ordonnés

4.1 Introduction

Ce chapitre propose une approche mémétique pour résoudre le problème d'extraction d'ordre partiel (modélisé par un graphe) présenté au chapitre 3. En effet, ce problème comme nous l'avons montré ne permet pas d'effectuer une propagation efficace, lors de la résolution PPC, sur les arcs et nœuds à sélectionner. Nous savons toutefois que vérifier qu'un graphe constitue bien un ordre partiel s'effectue en temps polynomial. Nous proposons donc un algorithme mémétique qui explore un espace de graphes et s'appuie sur une résolution PPC pour vérifier la propriété d'ordre. Afin de cantonner la recherche à l'espace des ordres partiels, l'algorithme mémétique proposé utilise un modèle PPC lors des phase d'initialisation et d'intensification. Le principe consiste à opérer sur des individus correspondant à des graphes de colonnes et à déléguer la preuve de consistance et le calcul d'exclusivité à un algorithme CSP.

4.2 Algorithmes génétiques et mémétiques

Les algorithmes génétiques ([11, 10, 42]) sont des algorithmes évolutionnaires qui s'inspirent de la sélection naturelle (Eiben [15], Mitchell [42], Holland [24]). Les algorithmes génétiques sont employés en optimisation combinatoire afin d'obtenir dans un temps raisonnable des solutions de bonne qualité mais potentiellement sous-optimales. En effet, de telles approches ne prouvent pas l'optimalité des solutions contrairement aux méthodes exactes de type recherche arborescente. Un algorithme génétique manipule un ensemble de solutions appelé population qu'il tente de faire converger vers les meilleures solutions possibles. Schématiquement, un algorithme génétique construit une population initiale qu'il fait évoluer de génération en génération jusqu'à ce qu'un critère d'arrêt prédéfini soit satisfait. Chaque itération (i.e. chaque renouvellement de génération) consiste à modifier et renforcer tout ou une partie de la population courante. Deux opérations sont utilisées à cet effet : l'opération de mutation et l'opération de croisement (Schaffer [58]). L'opération de mutation s'applique à un seul individu à la fois et consiste à le modifier. L'opération de croisement consiste à fusionner deux individus pour n'en produire qu'un. La nature de ces opérations dépend fondamentalement du type de représentation utilisée pour les individus (1-flip pour mutation sur vecteur de bits etc.).

Les opérations de croisement et mutation sélectionnent les individus sur lesquels opérer ou muter puis, après croisement, éliminent parents ou descendants afin de maintenir une taille de population constante. Différents critères de sélection peuvent être utilisés. Parmi les plus courants, citons la sélection aléatoire et la sélection par tournoi (Blickle et Thiele [5]) etc. Pour le contrôle de la taille de la population, citons le remplacement des parents par les fils et la sélection élitiste. La première génération est construite ex-nihilo et peut s'obtenir, par exemple, par génération des individus avec un algorithme glouton stochastique [62]. Enfin, la condition d'arrêt peut prendre différentes formes : nombre de générations fini, limite de temps de calcul, seuil de qualité sur les individus, etc. La figure 4.1 schématise le déroulement d'un algorithme génétique.

Les algorithmes mémétiques sont une extension des algorithmes génétiques qui ajoutent une étape d'intensification après la phase de mutation (Hao [23], Moscato [48, 47]). L'intensification vise à améliorer chaque enfant obtenu et s'appuie communément sur des méthodes de recherche locale.

À noter que la démultiplication des paramètres est une caractéristique des algorithmes mémétiques et génétiques : nombre de générations à fixer, heuristique de sélection des individus, probabilité de muter un individu, etc. De ce fait, ces algorithmes nécessitent une phase de paramétrage importante pour concilier qualité des solutions d'une part et efficacité de la résolution d'autre part. Nous discuterons du paramétrage de l'algorithme mémétique proposé dans ce chapitre mais nous n'aborderons pas ici la problématique du paramétrage automatique (e.g., par

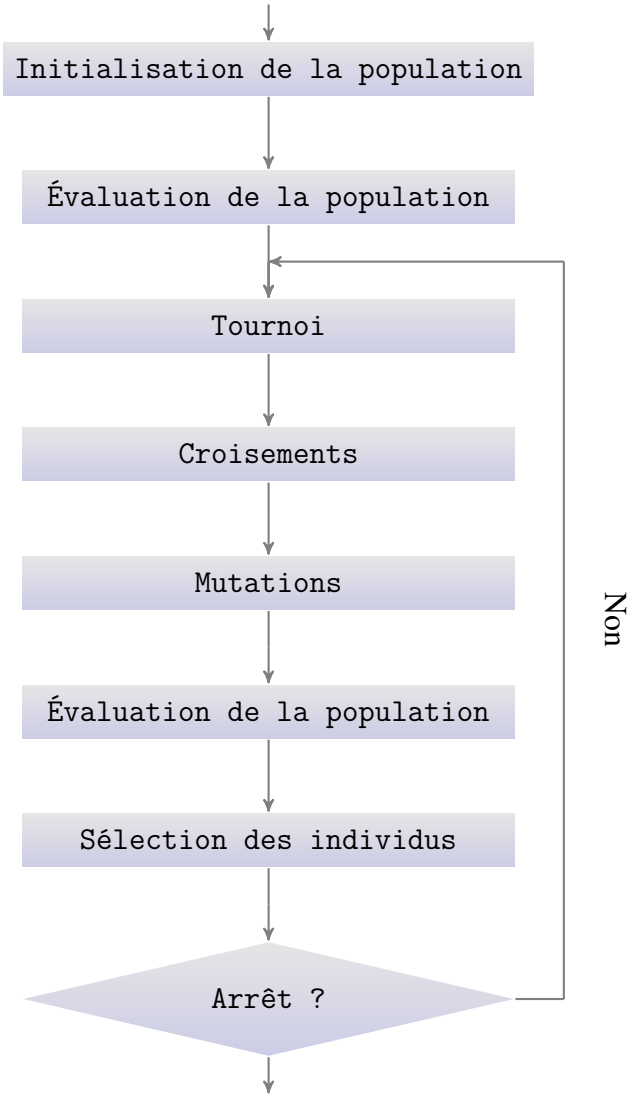

FIGURE 4.1 – Schéma général d'un algorithme génétique

apprentissage artificiel).

4.3 Calcul de motifs partiellement ordonnés maximaux pour l'exclusion

Cette section présente le problème, modélisé sous la forme d'un MMP, résolu par l'algorithme mémétique développé en section suivante.

Il s'agit ici du problème abordé en section 3.3 du chapitre 3. Nous noterons que, chaque caractère pris séparément détermine (au moins) un motif singleton consistant puisque nous nous restreignons, par hypothèse, aux jeux qui enracinent chaque caractère de l'alphabet dans les sé-

quence positives. De plus, tout motif englobant l'ensemble des colonnes et ordonné conformément à l'ordre induit [1] est aussi consistant par construction de la base matricielle (tout caractère peut être répété au sein d'un motif sans violer la contrainte de couverture positive). En particulier, les motifs globaux non ordonnés (modulo l'ordre total sur les occurrences de caractères répétés) sont consistants et leur score d'exclusivité fournit une borne inférieure. Cette borne est directement calculable puisqu'il suffit de comptabiliser les séquences négatives exclues par absence de localisation d'un caractère ou par nombre insuffisant d'occurrences. Nous développons en section suivante une approche mémétique pour le résoudre.

4.4 Approche mémétique

Une stratégie possible pour le calcul de motif ($m^{(1)}$, $m^{(2)}$) consiste à déterminer $m^{(2)}$ avant $m^{(1)}$, autrement dit choisir un graphe dirigé sur un sous-ensemble des colonnes (choix de $m^{(2)}$) puis à s'assurer de l'existence de localisations de ces colonnes qui soient compatibles avec les séquences positives et ce graphe (choix de $m^{(1)}$). L'algorithme mémétique que nous proposons repose sur cette décomposition. Le principe consiste à opérer sur des individus correspondant à des graphes de colonnes et à déléguer la preuve de consistance et le calcul d'exclusivité à un algorithme CSP. Nous noterons qu'un individu peut ne pas être consistant au sens où son graphe ($m^{(2)}$) ne correspond à aucun motif ($m^{(1)}$, $m^{(2)}$) consistant. Afin de cantoner l'exploration à l'espace des individus consistants, l'algorithme CSP est utilisé comme méthode d'initialisation (création d'individus consistants ex-nihilo) et comme méthode d'intensification (amélioration d'individus consistants obtenus par croisement). L'opération de croisement étant conçue comme l'intersection des graphes parents, tout descendant est nécessairement consistant. L'intensification menée par résolution CSP à partir d'un descendant consiste alors à explorer l'espace de graphes borné inférieurement par ce descendant et supérieurement par l'union des graphes parents.
Nous présentons d'abord le modèle d'individus utilisé avant de détailler l'algorithme.

4.4.1 Représentation des individus

Un individu modélise un graphe (N, A) sur un sous-ensemble des colonnes de L_2 ($N \subseteq L_2$) pour lequel existe un motif consistant ($m^{(1)}$, $m^{(2)}$) tel que $m^{(2)} = (N, A)$. Un individu est affecté d'un âge et d'un score. L'âge d'un individu représente le nombre d'itérations depuis

1. Chaque séquence dans la portée d'un motif détermine un ordre partiel entre les colonnes fondé sur leurs localisations dans la séquence. Nous appelons ordre induit d'un motif la conjonction des ordres partiels induits par les séquences, autrement dit, la relation maximale qui subsume les ordres induits par les séquences. Cette relation est unique et constitue un ordre partiel.

sa création. L'âge est utilisé pour éliminer les individus trop anciens dans un objectif de renouvellement de la population afin de diversifier la recherche. Le score d'un individu (N, A) correspond à la valeur de $f(m)$ pour $m = (m^{(1)}, (N, A))$ [2]. Un individu peut donc être décrit par un triplet $((N, A), age, score)$. D'un point de vue pratique, N peut être implanté par un vecteur de booléens ($true$ pour indiquer la présence d'un arc) et A par une matrice de booléen (matrice d'adjacence).

4.4.2 Algorithme général

L'algorithme 17 présente l'algorithme mémétique employé. La population initiale de taille fixe est d'abord générée (ligne 2) et le meilleur individu est enregistré dans s (ligne 3). Lors de cette phase d'initialisation les scores des individus sont calculés et leur âge est fixé à la valeur 1. La population est ensuite mise à jour autant de fois que le nombre de générations à évaluer (lignes 4 - 13). Dans la boucle principale, la première étape consiste d'abord à incrémenter l'âge de tous les individus (ligne 5) ; puis à isoler un sous-ensemble de la population pour les opérations de croisement dont la taille est paramétrée et au sein de laquelle des paires d'individus sont sélectionnées aléatoirement (ligne 6). Les deux étapes suivantes (lignes 7 - 8) génèrent de nouveaux individus, à partir des individus précédement appariés, par une opération de croisement suivie d'une phase d'intensification. Une fois les nouveaux individus générés, nous vérifions si le meilleur individu précédemment enregistré a été amélioré (ligne 9). Ensuite, une sélection est effectuée entre les individus croisés et leur descendance afin de conserver une population de taille fixe (ligne 10). Une fois cette sélection effectuée, sont supprimés de la population, avec une probabilité déterminée, les individus dont l'âge a dépassé la limite (ligne 11). Ces individus sont remplacés en réutilisant l'algorithme d'initialisation. (ligne 12).

Les sections suivantes présentent successivement les opérations d'initialisation, de croisement, d'intensification et de sélection.

4.4.2.1 Initialisation

L'opération d'initialisation vise à créer la première génération d'individus. Chaque individu est extrait d'une solution $(m^{(1)}, m^{(2)})$ du MMP obtenue par résolution du CSP présenté en section 3.3 du chapitre 3. De cette solution $(m^{(1)}, m^{(2)})$, nous ne conservons que l'ordre partiel $m^{(2)}$ sous forme d'un graphe (N, A). L'ensemble des nœuds N correspond aux colonnes de $m^{(2)}$ ($N = l_2^{(m^{(2)})}(m^{(2)})$), et l'ensemble des arcs A aux cellules de $m^{(2)}$ dénotant la présence d'un arc ($A = \{(i,j) | i \in l_1^{(m^{(2)})}(m^{(2)}), j \in l_2^{(m^{(2)})}(m^{(2)}), v^{(m^{(2)})}(i,j) = true\}$).

L'heuristique de choix de variables utilisée pour la résolution du CSP consiste à instancié d'abord les variables nœuds (présence ou absence du nœud), puis les variables d'arcs (présence

2. Par définition, $f(m) = f(m')$ si $m^{(2)} = m'^{(2)}$.

Algorithme 17 – Algorithme Génétique : Ordre Partiel

-Entrée : $S = L_1^+ \cup L_1^-$ ensemble de séquences
-Entrée : $taille$ nombre d'individus
-Entrée : $generationmax$ nombre de générations
-Entrée : $agemax$ âge maximum des individus
-Entrée : $pagemax$ probabilité de supprimer un individu une fois qu'il atteint l'âge maximum
-Entrée : $croisements$ nombre de croisements
-Entrée : $tempsGen$ temps alloué à la résolution du CSP pour la phase d'initialisation
-Entrée : $tempsIntens$ temps alloué à la résolution du CSP pour la phase d'intensification
-Entrée : $intensOptim$ seuil autorisant une intensification optimale
-Entrée : $relances$ nombre de relances du CSP autorisée
-Sortie : s meilleur individu trouvé
-Donnée : Pop population d'individus
-Donnée : $Tournois$ paires d'individus sélectionnés pour croisement
-Donnée : $Descendance$ individus générés par croisement
-Donnée : $GraphesCrois$ ensembles des ensembles de graphes obtenus par croisements

1 **begin**
2 $Pop \leftarrow$ `initialisation`$(S, taille, tempsGen, relances)$;
3 $s \leftarrow$ `meilleure`(Pop) ;
4 **foreach** $i \in [\![generationmax]\!]$ **do**
5 **foreach** $sol \in Pop$ **do** `augmenter_age`(sol);
6 $Tournois \leftarrow$ `tournois_aleatoire`$(Pop, croisements)$;
7 $GraphesCrois \leftarrow$ `croisement`$(Tournois)$;
8 $Descendance \leftarrow$ `intensification`$(S, GraphesCrois, tempsIntens, intensOptim, relances)$;
9 $s \leftarrow$ `meilleure`$(\{s\} \cup Descendance)$;
10 $Pop \leftarrow (Pop \backslash Tournois) \cup$ `selection`$(Tournois, Descendance)$;
11 $Pop \leftarrow$ `suppression_age`$(Pop, agemax, pagemax)$;
12 $Pop \leftarrow Pop \cup$ `initialisation`$(S, taille - |Pop|, tempsGen, relances)$;
13 **end**
14 **return** s ;
15 **end**

ou absence dans le graphe). À noter que l'affectation des variables de localisation n'est pas nécessaire sous condition d'un maintien de borne-consistance en cours de recherche (chapitre 3). Afin de diversifier la population, l'ordre d'instanciation des variables dans chacun des deux groupes est aléatoire ainsi que le choix de valeur pour chaque variable nœud et chaque variable arc. Ces heuristiques reposent sur une distribution uniforme.

Nous imposons d'autre part une limite de temps à la recherche de solution. Cette dernière n'examine qu'une partie de l'espace de recherche, d'où elle extrait la meilleure solution s'il en existe. Si aucune solution n'est obtenue dans le temps imparti, alors la recherche est relancée selon un capital de relances alloué. Ce capital de relance est global à la phase d'intensification. Si ce capital est consommé avant d'avoir pu générer le nombre d'individus requis, alors l'algorithme est interrompu.

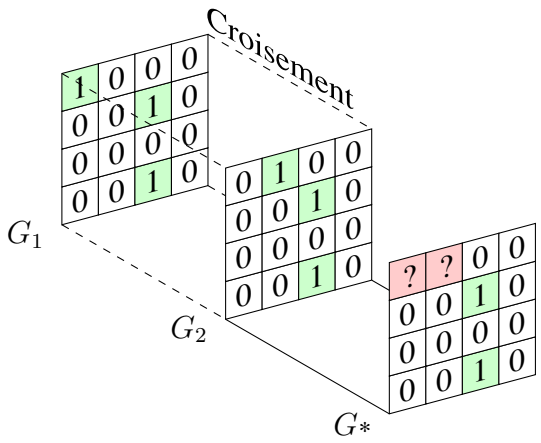

FIGURE 4.2 – Ensemble de graphes G^* résultant d'un croisement (G_1, G_2).

4.4.2.2 Croisement et intensification

L'étape de croisement est précédée d'une sélection de paires d'individus, sur lesquelles porteront les croisements, parmi la population. Le croisement consiste à combiner les graphes des individus appariés. L'opération ne donne pas un graphe à proprement parler mais un ensemble de graphes qui détermine l'espace de recherche du CSP lors de l'étape d'intensification. L'ensemble de graphes correspond précisément aux graphes minorés par l'intersection des graphes parents et majorés par leur union. Formellement, le croisement d'une paire d'individus (G_1, G_2) génère l'ensemble de graphes G^* défini par $\{(N, A)|(N_1 \cap N_2 \subseteq N \subseteq N_1 \cup N_2) \wedge (A_1 \cap A_2 \subseteq A \subseteq A_1 \cup A_2)\}$. Nous noterons que l'intersection des graphes G_1 et G_2, lorsqu'elle est non vide, détermine un individu consistant.

La phase d'intensification vise à déterminer le meilleur individu consistant dans G^*. À cet effet, cette opération emploie le CSP utilisé lors de la phase d'initialisation, amendé d'une contrainte d'appartenance à G^*. Cette contrainte peut s'encoder dans le CSP en imposant la présence des nœuds de $N_1 \cap N_2$ et les arcs de $A_1 \cap A_2$, et en excluant les nœuds n'appartenant pas à $N_1 \cup N_2$ et les arcs n'appartenant pas à $A_1 \cup A_2$. La figure 4.2 illustre ce processus sur les graphes qui sont représentés sous la forme d'une matrice d'adjacence. Concernant la résolution du CSP, nous conservons les heuristiques de choix de valeurs et de variables utilisées pour l'étape d'intensification. Concernant le temps de calcul imparti, nous utilisons trois paramètres. Le premier est un seuil sur le nombre de variables de décision indéterminées du CSP. En deçà de ce seuil, nous procédons à une résolution optimale (recherche de la solution optimale). Sinon, nous procédons à une résolution approchée en s'appuyant sur une limite de temps de calcul et un capital de relances, comme lors de la phase d'initialisation.

4.4.2.3 Sélection

La sélection porte sur l'ensemble des individus utilisés pour le croisement et leur descendance. Seuls les individus de meilleurs scores sont retenus. En cas d'égalité, la priorité est donnée aux individus les plus jeunes. Cette phase de sélection préserve la taille de la population en sélectionnant un nombre d'individus égal au nombre d'individus croisés.

4.4.3 Expérimentations

Nous présentons ici une comparaison expérimentale du modèle PPC proposé en section 3.3 au chapitre 3 et de deux paramétrages différents de l'algorithme mémétique sur les jeux de données biologiques LEAP et SHSP décrits en section 3.3.2 au chapitre 3. Pour rappel, chacun de ces jeux est prépartionné en classes de séquences : 13 pour LEAP et 25 pour SHSP. L'objectif visé est, pour chaque classe, de calculer un ordre partiel commun aux séquences positives excluant le maximum de séquences négatives, et de taille minimal. Chaque instance de test se définit par la donnée d'une classe et d'un seuil d'écart maximale. Nous rapportons dans le tableau 4.1 les résultats relatifs à :

— l'exécution du modèle PPC ;

— l'exécution de trois paramétrages pour l'algorithme mémétique (*GEN1* et *GEN2*).

Les algorithmes mémétiques ont été exécutés à 100 reprises sur chaque instance et nous reportons dans le tableau :

— la moyenne des temps d'exécution (colonne t) ;

— la nombre minimum de séquences négatives exclues (colonne min) ;

— le nombre moyen de séquences négatives exclues (colonne moy) ;

— l'écart type du nombre de séquences négatives exclues (colonne $ec.t$).

Le modèle PPC a lui aussi été exécuté à 100 reprises avec les mêmes stratégies de branchement que celles qui sont appliquées au sein de l'algorithme mémétique (sélection des variables et des valeurs).

Le paramétrage de *GEN1* est le suivant :

— nombre de générations : 150 ;

— taille de la population : 100 ;

— âge seuil [3] d'un individu : 70 ;

— probabilité de supprimer un individu ayant atteint l'âge limite : 50% ;

3. Un individu ne peut être retiré de la population avant d'avoir atteint cet âge, mais il peut l'être suite à la sélection post-tournois.

— nombre de paires d'individus croisés : 5 ;

— temps alloué au modèle PPC pour la phase d'initialisation : 2s ;

— temps alloué au modèle PPC pour la phase d'intensification : 2s ;

— nombre de relances maximal : $+\infty$.

Le paramétrage de *GEN2* est similaire à celui de *GEN1* cependant les phases d'initialisation et d'intensification s'arrêtent à la première solution trouvée :

— nombre de générations : 150 ;

— taille de la population : 100 ;

— âge seuil d'un individu : 70 ;

— probabilité de supprimer un individu ayant atteint l'âge limite : 50% ;

— nombre de paires d'individus croisées : 5 ;

— temps alloué au modèle PPC pour la phase d'initialisation : 2s ;

— temps alloué au modèle PPC pour la phase d'intensification : 2s.

— nombre de relances maximal : $+\infty$;

— les phase d'initialisation et d'intensification arrêtent leur modèle PPC à la première solution rencontrée.

Le paramétrage de *GEN3* est le suivant :

— nombre de générations : 110 ;

— taille de la population : 100 ;

— âge seuil d'un individu : 45 ;

— probabilité de supprimer un individu ayant atteint l'âge limite : 50% ;

— nombre de paires d'individus croisées : 9 ;

— temps alloué au modèle PPC pour la phase d'initialisation : 6s ;

— temps de base t_b alloué au modèle PPC pour la phase d'intensification : 130s.

— temps réellement alloué t_a pour la phase d'intensification :

  — I_{max} : nombre de générations (110),

  — I : génération en cours de construction,

  — $t_a = t_b * e^{ln(0.1).(1-\frac{I}{I_{max}})}$.[4]

— nombre de relances maximal : $+\infty$;

4. L'objectif de *GEN3* est de laisser de plus en plus de temps de calcul au modèle PPC lors des phases d'intensification, en commençant à 10% du temps de base jusqu'à 100% de celui-ci.

Tous les algorithmes ont un temps d'exécution limité à de deux heures et trente minutes. Les paramètres des algorithmes mémétiques ont été déterminés expérimentalement, en essayant d'allouer autant de temps aux deux approches. Toutefois, la prédiction du temps d'exécution pour les algorithmes mémétiques est compliquée, donc le temps alloué à la résolution PPC pure a été favorisée par rapport aux versions mémétiques.

La première comparaison analyse la qualité des solutions entre la résolution uniquement PPC et la résolution par algorithme mémétique. La deuxième comparaison analyse l'influence de la phase d'intensification sur la qualité des solutions.

Les programmes ont été développés en C++ avec la bibliothèque Gecode (4.0) (Schulte et al. [**?**]) sur "Intel(©) Xeon(©) E5439 - 2.83GHz quad-core (4)".

Nous retrouvons ici des colonnes identiques à celles présentes dans le chapitre 3 à savoir, le nombre de séquences positives (colonne $|\mathcal{S}^+|$), le nombre de séquences négatives (colonne $|\mathcal{S}^-|$), le nombre de patrons (colonne d_2), le nombre moyen de localisations pour les patrons sur la classe positive ($\bar{D}$), le nombre minimum de localisations (colonne $D_\vee$), le nombre maximum de localisations (colonne $D_\wedge$).

								PPC			GEN 1			GEN 2			GEN 3		
Instance	écart	$\lvert S^+\rvert$	$\lvert S^-\rvert$	d_2	$\bar{D}$	$D_\vee$	$D_\wedge$	min	moy	ec. t.	t (ms)	min	moy	ec. t.	t (ms)	min	moy	ec. t.	t (ms)	min	moy	ec. t.	t (ms)
LEAP	1	10	208	1163	20	3.2	1	60	46	66.28	10.35	7200000	38	48.15	3.25	438230	37	46.91	3.59	406198	36	43.92	4.16	1172138
	1	16	208	1163	29	3.2	1	60	25	33.97	4.89	7200001	19	19.8	1.19	1300627	19	20.25	1.57	935187	19	19.74	1.37	3765938
	2	8	92	1279	25	4.3	1	26	5	8.90	2.58	7200000	5	5.95	0.19	2204716	5	5.95	0.19	1952250	5	5.92	0.25	7750751
	2	15	92	1279	38	4.3	1	26	6	7.26	0.96	7200003	5	5	0	2527866	5	5	0	2727738	5	5	0	7207275
	2	20	92	1279	45	4.1	1	26	5	5.81	0.60	7200002	3	3	0	2749146	3	3	0	2205653	3	3	0	8011345
	3	0	34	1337	9	3.5	1	24	0	0	0	52	0	0	0	21136	0	0	0	12629	0	0	0	20330
	5	2	66	1305	10	2.7	1	12	0	0	0	396	0	0	0	26165	0	0	0	22153	0	0	0	35680
	10	8	78	1293	21	2.0	0	53	36	49.33	5.83	7200000	30	33.04	0.78	476788	31	33.07	0.77	431492	30	32.69	0.99	1102346
	10	15	78	1293	30	2.9	1	8	7	17.01	5.40	7200000	7	7	0	786800	7	7	0	580044	7	7	0	2175462
	10	20	78	1293	38	2.8	1	8	6	10.28	2.16	7200000	5	5.03	0.17	1004826	5	5.02	0.15	624845	5	5.03	0.17	2988248
	11	1	35	1336	19	2.7	1	10	0	0.03	0.17	20120	0	0	0	180918	0	0	0	183020	0	0	0	649040
	12	0	17	1354	5	1.0	1	2	0	0	0	27	0	0	0	7098	0	0	0	5068	0	0	0	7985
SHSP	2	6	109	3656	14	2.0	1	5	0	1.38	3.50	251806	0	0.38	0.72	243968	0	0.32	0.63	243921	0	0.02	0.14	663516
	3	2	64	3701	13	2.2	1	8	0	0	0	2160	0	0	0	179030	0	0	0	175547	0	0	0	423716
	4	0	23	3742	10	1.7	1	11	0	0	0	1056	0	0	0	91837	0	0	0	59971	0	0	0	95351
	5	0	25	3740	12	1.6	1	6	0	0	0	580	0	0	0	101721	0	0	0	69292	0	0	0	100168
	7	10	100	3665	11	1.8	1	6	105	105	0	270942	105	105	0	7545462	105	105	0	5585503	105	105	0	5514325
	8	1	30	3735	15	1.6	1	4	0	0.09	0.39	15585	0	0	0	279350	0	0	0	350057	0	0	0	829427
	9	6	238	3527	?	2.5	1	7	8	8	0	1401	8	8	0	198921	8	8	0	174342	8	8	0	240234
	13	3	156	3609	11	2.2	1	6	0	0	0	3190	0	0	0	166335	0	0	0	145128	0	0	0	241430
	14	0	84	3681	20	1.4	1	5	0	0	0	1767	0	0	0	212950	0	0	0	183233	0	0	0	261107
	15	0	26	3739	20	1.6	1	5	0	0	0	2099	0	0	0	205129	0	0	0	177906	0	0	0	245065
	16	9	75	3690	6	1.4	1	5	0	0	0	260	0	0	0	46240	0	0	0	38557	0	0	0	54877
	17	3	88	3677	9	1.5	1	3	0	0	0	1333	0	0	0	137658	0	0	0	118774	0	0	0	174571
	19	1	23	3742	24	2.1	1	8	0	0	0	46488	0	0	0	370352	0	0	0	328934	0	0	0	1331797
	21	2	131	3634	6	2.4	1	10	0	0	0	231	0	0	0	34331	0	0	0	28347	0	0	0	38329
	22	0	15	3750	36	2.6	1	9	0	0	0	13087	-	-	-	-	-	-	-	-	0	0	0	978081
	23	4	60	3705	6	1.7	1	5	1	1	0	601	1	1	0	106373	1	1	0	98891	1	1	0	138245

TABLEAU 4.1 – Comparaisons expérimentales pour le score d'exclusion induit par un ordre partiel.

Les algorithmes mémétiques montrent leur efficacité sur le problème. En effet, ils obtiennent de meilleurs résultats en moyenne et sont plus robustes que le modèle PPC seul. Cette efficacité peut s'expliquer par le fait que le problème de recherche d'ordre partiel est peu structuré. En outre le gain apporté par la tentative d'optimisation de la résolution PPC lors des phases d'initialisation et d'intensification semble négligeable pour les paramétrages proposés lorsque nous comparons les résultats de *GEN1* et *GEN2*. Sur les instances les plus difficiles (LEAP 1 :10, LEAP 1 :16, LEAP 2 :8, LEAP 2 :15, LEAP 2 :20, LEAP 10 :8, LEAP 10 :15, LEAP 10 :20) leurs temps moyens sont similaires. En termes d'exclusion, les résultats de *GEN1* et *GEN2* sont aussi similaires. Ils améliorent tous les deux le résultat de l'autre sur une instance (LEAP 10 :8 pour *GEN1* et LEAP 1 :10 pour *GEN2*). *GEN3* propose les meilleurs résultats en termes d'exclusion sur toutes les instances, notamment sur LEAP 1 :10. *GEN3* nécessite en moyenne plus de temps de calcul, ceci s'explique par le fait que sa phase d'initialisation et sa phase d'intensification durent plus longtemps en moyenne. D'ailleurs, comme la phase d'initialisation de *GEN3* est plus longue, cela permet d'obtenir des résultats pour SHSP 22 :0 contrairement à *GEN1* et *GEN2*.

Pour la résolution PPC pure, nous pouvons remarquer que pour les instances LEAP 11 :1, SHSP 2 :6 et SHSP 8 :1 le score moyen n'est pas nul malgré le temps d'exécution moyen faible. Ce phénomène survient lorsque l'heuristique de branchement aléatoire sélectionne de nombreux nœuds mais que ces nœuds ont un "pouvoir d'exclusion" faible et cantonerront la recherche des ordres partiels à un espace de recherche peu intéressant.

5

Deux algorithmes dédiés à l'extraction de motifs séquentiels

5.1 Introduction

Le langage MMP permet de calculer des motifs communs et exclusifs sur des jeux de séquences préclassées. La forme des motifs requis par un MMP se personnalise en posant des contraintes sur les représentations matricielles (e.g. nombre de colonnes, ordre entre colonnes, écart de coefficients entre deux colonnes, etc.). En particulier, le cadre MMP permet le calcul de sous-chaînes communes, de sous-séquences communes, et différentes variantes soumises à contraintes.

Le choix du domaine matriciel conditionne la nature des motifs calculés et influence l'efficacité de la résolution. Par défaut, pour des motifs sur données séquentielles, le domaine matriciel apparie colonnes et caractères de l'alphabet. Dans ce cas, tout motif est nécessairement sans répétition de caractères. Afin de permettre plus de flexibilité, nous pouvons associer aux colonnes du domaine matriciel des occurrences de caractères. Ceci permet de calculer des motifs plus généraux autorisant la répétition. Cette représentation reste cependant d'une granularité très fine et peut induire des tailles d'instances prohibitives [1]. Elle pose donc des problèmes de passage à

1. La taille d'une instance sera d'autant plus importante que le ratio entre taille de l'alphabet et taille de séquences sera faible. C'est le cas notamment des séquences d'ADN (alphabet de 4 caractères). Concernant les

l'échelle et rend d'autant plus difficile l'énumération exhaustive de motifs.

Pour atténuer ces difficultés, une approche consiste à se doter d'un domaine matriciel de moins fine granularité en associant non plus les colonnes aux caractères mais à des patrons pré-calculés. Cette approche suppose de définir d'une part le type de patron recherché, et d'autre part, de mettre en œuvre une méthode permettant de pré-calculer tous les enracinements possibles de tous les patrons. Afin de concilier efficacité et expressivité, nous extrayons des patrons qui correspondent à des sous-séquences communes à écart invariable entre caractères d'une séquence à l'autre. Ces patrons sont une forme d'alignement localisé qui généralisent les sous-chaînes communes, et du fait de la contrainte d'alignement sont en moindre nombre que les sous-séquences communes.

Nous imposons aussi des contraintes d'extensibilité sur les patrons communs ainsi que des contraintes de taille et d'écart afin d'extraire des patrons maximaux qui soient informatifs et en nombre restreint. Qui plus est, nous pouvons ajouter une contrainte de maximalité sur le nombre de caractères non substituables afin de chercher les patrons les plus informatifs.

Nous proposons sur cette base deux approches au calcul de patrons maximaux - une inspirée d'APriori et une de la programmation dynamique - et nous montrons l'intérêt de ces approches dans le cadre d'expérimentations menées sur des jeux de séquences protéiques. Nous formalisons au préalable le problème du calcul de patrons maximaux.

5.2 Calcul de patrons maximaux

Comme évoqué plus haut, nous nous intéressons au calcul de patrons correspondant à des sous-séquences communes où l'écart entre caractères consécutifs est invariable d'une séquence à l'autre. Un patron peut donc se représenter par une chaîne de caractères constituée de caractères non substituables (appelés caractères solides) et de caractères substituables (appelés jokers et dénotés par un même symbole $\#$), par exemple $ACC\#\#A\#B$. Nous définissons alors l'inclusion d'un patron dans une séquence par l'existence d'une position dans la séquence (appelée enracinement) à partir de laquelle patron et séquence s'accordent sur les caractères solides du patron. Autrement dit, un patron est inclus dans une séquence s'il existe une substitution des jokers qui détermine une sous-chaîne de la séquence. Par exemple, $A\#B$ est inclus dans $AABB$ avec pour domaine d'enracinements l'ensemble des positions $\{1, 2\}$.

Le calcul de patrons communs à une classe de séquences et de leurs domaines d'enracinement fournit donc une base sous forme matricielle directement exploitable pour une résolution MMP. Toutefois, sans contraintes supplémentaires, le nombre de patrons communs à un jeu

séquences protéiques, l'alphabet contient 20 acides aminés et la taille des séquences varie selon les familles de protéines. Pour LEAP et SHSP considérées dans nos expérimentations, la taille varie d'environ 60 à 250 caractères.

de séquences peut s'avérer rédhibitoire [2]. Pour cette raison, nous limitons le calcul à un sous-ensemble des patrons communs dits maximaux et nous proposons à cet effet deux définitions alternatives. La première se fonde sur une généralisation de la relation d'inclusion aux patrons et dénote l'impossibilité d'étendre un patron par ajout de caractères ou substitution de jokers. Nous rechercherons alors les éléments maximaux pour cette relation parmi l'ensemble des patrons communs. La seconde définition caractérise les patrons communs à enracinements maximaux. Un enracinement de patron sur une séquence est dit maximal s'il est impossible d'étendre le patron à partir de cet enracinement. L'objectif est alors de calculer les domaines d'enracinements maximaux sur chaque séquence pour un patron donné. Nous introduisons à cet effet la notion de bloc sur laquelle s'appuie les algorithmes proposés : un bloc sur un ensemble de séquences définit l'association d'un enracinement sur chaque séquence à un patron donné. Nous disons alors d'un bloc b qu'il enracine un bloc b' s'il est possible sur au moins une séquence d'étendre le patron de b' en celui de b en respectant le décalage d'enracinements.

Le problème est donc à calculer tous les blocs maximaux pour l'enracinement. Par définition, l'ensemble des blocs maximaux d'un patron correspond au produit cartésien des domaines des enracinements maximaux du patron sur les séquences. L'approche que nous proposons consiste donc à identifier les patrons communs, à éliminer ceux qui n'ont pas de blocs maximaux, et pour les autres à factoriser leur ensemble de blocs maximaux sous forme de produit cartésien. Nous noterons que, par définition, tout bloc de patron maximal pour l'inclusion est nécessairement maximal pour l'enracinement. Le calcul des blocs maximaux pour ces patrons s'en trouve donc simplifié et nous présenterons une simplification de l'algorithme de calcul de blocs maximaux pour le cas des patrons maximaux pour l'inclusion.

Nous imposons par ailleurs des contraintes de solidité minimale et d'écart maximum sur les patrons où la solidité représente le nombre de caractères solides et l'écart le nombre maximum de jokers séparant deux caractères solides. Ces contraintes restreignent d'avantage le nombre de patrons solutions et facilitent ainsi le passage à l'échelle. Au delà des considérations liées à l'efficacité, toutes ces contraintes - récurrence, maximalité, solidité, écart - visent à produire des patrons qui soient porteurs d'une information la plus pertinente possible pour l'utilisateur.

Nous formalisons ci-dessous les notions de patron, bloc et maximalité pour les relations d'inclusion et d'enracinement. Les définitions suivantes feront référence à deux alphabets Σ et $\Sigma' = \Sigma \cup \{\#\}$ où $\#$ est un caractère n'appartenant pas à Σ, que nous appelons joker.

Un patron est une suite de caractères de Σ' qui commence et se termine par un caractère solide.

Définition 23 (Patron)

Un patron est un mot du langage $\Sigma \cup \Sigma.(\Sigma'^*).\Sigma$.

2. Par exemple, les patrons communs à une séquence constituée de tous les caractères de Σ sont au nombre de $2^{|\Sigma|}$.

Une séquence est une suite de caractères non vide sur l'alphabet Σ. Les séquences sont donc un cas particulier de patrons sans joker.

Définition 24 (Séquence)
Une séquence est un mot du langage Σ^+.

Pour un patron x, $|x|$ dénote la taille de x, c'est-à-dire le nombre de caractères de x (y compris les jokers) ; x_i dénote le i-ème caractère de x ($i \in [\![|x|]\!]$) [3].

Nous définissons la solidité d'un patron comme son nombre de caractères solides.

Définition 25 (Solidité de patron)
Soit p un patron. La solidité de p, noté $\pi(p)$, est définit par son nombre de caractères solides, $\pi(p) = |\{p_i | i \in [\![|p|]\!] \wedge p_i \neq \#\}|$.

Nous définissons l'écart d'un patron comme le plus grand nombre de caractères jokers consécutifs y apparaissant.

Définition 26 (Écart de patron)
Soit p un patron. L'écart de p, noté $\gamma(p)$, est défini par $\gamma(p) = max(\{n | n \in \{0, \cdots, |p|\} \wedge \exists k \in [\![|p| - n + 1]\!],\ \forall j \in [\![n]\!],\ p_{k+j-1} = \#\})$

Un patron p est inclus dans un patron p' s'il existe un décalage pour lequel tous les caractères solides de p concordent avec ceux de p'.

Définition 27 (Inclusion entre patrons)
Soient $p, p' \in \Sigma.(\Sigma')^*.\Sigma$ tels que $|p| \leqslant |p'|$ et $k \in \{0, \cdots, |p'| - |p|\}$. p est inclus dans p' en position k, dénoté $p \leqslant_k p'$, si et seulement si $\forall i \in [\![|p|]\!] : p_i \in \{\#, p'_{i+k}\}$.

Nous noterons $E(p, p')$ l'ensemble $\bigcup_{k \in \mathbb{N}} \{k | p \leqslant_k p'\}$ appelé l'ensemble des enracinements du patron p dans le patron p'.

Définition 28 (Inclusion générale entre patrons)
Soient p et p' deux patrons. p est inclus dans p', dénoté $p \leqslant p'$, si et seulement si $E(p, p') \neq \varnothing$. p est strictement inclus dans p', dénoté $p < p'$, si et seulement si $p \leqslant p'$ et $p \neq p'$.

Trivialement, $\leqslant$ (respectivement $<$) est un ordre partiel (respectivement un ordre partiel strict) sur l'ensemble des patrons construits sur Σ. À noter que les séquences étant des cas particuliers de patrons, ces relations s'appliquent également entre patrons et séquences.

Un bloc représente l'enracinement d'un patron dans un ensemble de séquences.

3. Pour tout $n \in \mathbb{N}^+$, $[\![n]\!]$ représente l'ensemble $\{1, \cdots, n\}$.

Définition 29 (Bloc)

Soit $\varnothing \subset S \subseteq 2^{\Sigma^+}$, un bloc sur S est un triplet (S, p, e) tel que :

— $p \in \Sigma \cup \Sigma.(\Sigma'^*).\Sigma$;

— $e \ : \ S \to \mathbb{N}$;

— $\forall s \in S \ : \ p \leqslant_{e(s)} s$.

Nous notons $(S^{(b)}, p^{(b)}, e^{(b)})$ le triplet associé à un bloc b et dans le cas particulier où S est un singleton $\{s\}$, nous donnerons le triplet sous la forme $(s, p, e^{(b)}(\{s\}))$.

Par extension, solidité et écart de patrons s'étendent aux blocs.Nous noterons $\pi(b) = \pi(p^{(b)})$ et $\gamma(b) = \gamma(p^{(b)})$. Nous appelons k-patron tout patron de solidité k ($k \in \mathbb{N}$). Nous appelons k-bloc tout bloc dont le patron est un k-patron.

Les définitions suivantes feront référence à un ensemble non-vide S de séquences sur Σ. Nous notons $B(S)$ l'ensemble des blocs sur S. $B_{pg}(S) = \bigcup_{b \in B(S)} \{b | \pi(b) \geqslant p \wedge \gamma(b) \leqslant g\}$ dénote alors l'ensemble de blocs sur S de solidité supérieure ou égale à p et d'écart inférieur ou égal à g.

Nous introduisons la relation d'enracinement entre blocs d'un même ensemble de séquences qui prolonge la relation d'inclusion entre patrons. Précisément, un bloc en enracine un autre s'il existe une séquence sur laquelle son patron inclut le patron du second sur la base de leurs positionnements dans la séquence.

Définition 30 (Enracinement de bloc)

Soient b, $b' \in B_{pg}(S)$. b est enraciné dans b', dénoté $b \subset b'$, si et seulement si
$\exists s \in S, \ \exists k \in \mathbb{N} \ : \ k \leqslant |p^{(b')}| - |p^{(b)}| \ \wedge \ k = e^{(b)}(s) - e^{(b')}(s) \wedge p^{(b)} <_k p^{(b')}$.

L'enracinement entre deux blocs suppose donc l'inclusion entre leurs patrons, autrement dit, l'inclusion subsume l'enracinement. À noter que l'enracinement ne constitue pas un ordre partiel strict sur $B_{pg}(S)$ dans le cas général où S contient plusieurs séquences, la transitivité n'étant pas assurée. La propriété d'ordre partiel strict n'est en effet garantie que pour les ensembles de blocs sur une seule séquence, i.e. $S = \{s\}$ pour $s \in \Sigma^+$. Dans ce cas, nous noterons que tout bloc b sur $\{s\}$ peut être identifié par l'ensemble des positions de ses caractères solides dans s. La relation d'enracinement sur $B_{pg}(\{s\})$ correspond alors à l'inclusion ensembliste sur les parties de $[\![|s|]\!]$. L'ensemble des blocs sur $B_{pg}(\{s\})$ muni de la relation $\subset$ constitue donc un demi-treillis supérieur borné par le bloc $(\{s\}, s, 0)$. Sur cette base, nous définissons l'opération de fusion entre blocs sur $\{s\}$, dénotée $\bigsqcup$, comme le bloc correspondant à l'union des ensembles de positions de leurs caractères solides sur s.

Nous nous intéresserons donc aux blocs maximaux sur S pour l'enracinement ($\subset$) et aux blocs maximaux pour l'inclusion ($<$). En outre, nous nous limiterons aux blocs qui satisfont un seuil de solidité minimum et un seuil d'écart maximum donnés. Nous formalisons ces en-

sembles de blocs comme suit. Soient $p, g \in \mathbb{N}$,

— $BME_{pg}(S) = \bigcup_{b \in B_{pg}(S)} \{b | \forall b' \in B_{pg}(S) : b \not\subset b'\}$ dénote l'ensemble de blocs maximaux pour l'enracinement sur $B_{pg}(S)$.

— $BMI_{pg}(S) = \bigcup_{b \in BME_{pg}(S)} \{b | \forall b' \in BME_{pg}(S) : b \not\lessdot b'\}$ dénote l'ensemble de blocs maximaux pour l'inclusion sur $B_{pg}(S)$ [4].

De manière évidente, $BMI_{pg}(S) \subseteq BME_{pg}(S) \subseteq B_{pg}(S) \subseteq B(S)$.

Les algorithmes que nous proposons calculent $BMI_{pg}(S)$ ou $BME_{pg}(S)$ en les factorisant au lieu d'en donner une représentation énumérée. L'intérêt de la factorisation est d'une part de réduire l'espace requis pour l'encodage des ensembles et d'autre part d'en fournir une représentation directement exploitable comme base de données dans le cadre d'une résolution MMP subséquente. Le principe de la factorisation est d'identifier les classes de blocs équivalents [5] au sein de l'ensemble à calculer, et à associer à chacun l'ensemble des enracinements de ses blocs sur chaque séquence. Nous formalisons la factorisation d'un ensemble de blocs dans le cas général par une fonction [6]

Définition 31 (Facorisation)

Soit $B \subseteq B(S)$, la factorisation de B dénoté $f(B)$ se définit par :

$$f(B) = \bigcup_{[b] \in B|\sim} \{(p^{(b)}, E : S \to 2^{\mathbb{N}}) | E(s) = \{e^{(b^*)}(s) | b' \in [b]\} (s \in S\}.$$

Nous proposons donc de calculer $f(BMI_{pg}(S))$ et $f(BME_{pg}(S))$. Pour chacun des ensembles $BMI_{pg}(S)$ et $BME_{pg}(S)$, le produit cartésien des domaines d'enracinement sur l'ensemble des séquences par classe d'équivalence coïncide en effet avec l'ensemble des blocs de la classe.

Le tableau 5.1 résume les définitions présentées dans cette section et la section suivante présente une première approche au calcul des factorisations de l'ensemble des blocs maximaux pour l'enracinement sur un jeu de séquences S, de solidité maximum 2 et d'écart majoré par un paramètre utilisateur g.

4. En effet, un bloc est maximal pour l'inclusion sur $B_{pg}(S)$ si et seulement si il est maximal sur $BME_{pg}(S)$.

5. Nous définissons la relation d'équivalence entre blocs d'un ensemble de séquences S induite par l'égalité entre patrons : $b \sim b' \leftrightarrow p^{(b)} = p^{(b')}$ pour $b, b' \in B(S)$. Nous notons $B_{pg}(S)| \sim$ l'ensemble quotient associé à $\sim$ et $[b]$ la classe d'équivalence de $b \in B_{pg}(S)$.

6. L'opération consistant à échanger les enracinements de deux blocs équivalents sur une même séquence produit un bloc équivalent. Cette opération préserve aussi la maximalité lorsque les blocs sont tous deux maximaux pour l'inclusion (ou bien maximaux pour l'enracinement). Autrement dit, les ensembles $BMI_{pg}(S)$ et $BME_{pg}(S)$ sont chacun stables par produit cartésien des domaines d'enracinement par classe d'équivalence, i.e. $\bigcup_{(p,E) \in f(B)} \{(p, e) | e \in \times_{s \in S} E(s)\} = B$ si $B = BMI_{pg}(S)$ ou $B = BME_{pg}(S)$.

Dénomination	Notation
Patron	p
Séquence	s
Solidité	$\pi(p)$
Écart maximal	$\gamma(p)$
Ensemble des enracinements de p dans p'	$E(p, p')$
Inclusion entre patrons	$<$
Bloc	(S, p, e)
Enracinement entre blocs	$\subset$
Ensemble des blocs	$B(S)$
Ensemble des blocs de solidité min. p et d'écart maX. g	$B_{pg}(S)$
Ensemble des blocs maximaux pour l'inclusion	$BMI_{pg}(S)$
Ensemble des blocs maximaux pour l'enracinement	$BME_{pg}(S)$

TABLEAU 5.1 – Synthèse

5.3 Factorisation de blocs maximaux par parcours de treillis

Nous présentons ici un algorithme, appelé `BlocMaxE`, dédié au calcul de la factorisation $f(BME_{pg}(S))$. `BlocMaxE` peut calculer de manière concomitante la factorisation $f(BMI_{pg}(S))$ et est paramétrable à cet effet. `BlocMaxE` décompose le calcul en deux étapes : la première calcule un sur-ensemble des patrons correspondants aux classes de blocs dans $BME_{pg}(S)$, la seconde élimine les patrons sans correspondance et factorise les autres, filtrage et factorisation étant menés simultanément.

Le calcul de patrons lors de la première étape suppose de générer tous les patrons communs aux séquences de S afin d'identifier le sur-ensemble recherché. Pour ce faire, `BlocMaxE` génère tous les blocs d'une séquence pré-sélectionnée, et tente pour chacun si le patrons associé est commun ou non. Comme mentionné en section 5.2, l'ensemble des blocs d'une séquence ordonné par enracinement est isomorphe aux ensembles de position possible sur la séquence ordonnée par l'inclusion ensembliste. Nous pouvons donc manier l'ensemble de blocs d'une opération de fusion qui correspond à l'union des ensembles de positions à caractère solide.

`BlocMaxE` génère ce treillis par couche de solidité croissante en fusionnant les blocs de chaque couche deux à deux. Tout bloc obtenu par fusion détermine un patron dont l'inclusion dans les séquences de S est immédiatement testée. Si le patron n'est pas commun, le bloc est éliminé et les blocs l'enracinant ne seront pas générés. Ce filtrage, est correct et exploite l'antimonotonie de l'inclusion de patron relativement à l'enracinement de blocs [7]. La génération

7. Tout bloc enracinant un bloc dont le patron n'est pas commun correspond nécessairement à un patron qui n'est pas commun.

du treillis de blocs à patrons communs d'apparente donc au calcul d'itemsets fréquents effectué par l'algorithme APriori.

`BlocMaxE` intègre à la génération du treillis une procédure de filtrage permettant d'une part d'identifier les classes de $f(BMIO_{pg}(S))$, d'autre part d'éliminer des patrons ne pouvant figurer dans $f(BLE_{pg}(S))$. Autrement dit, ce filtrage encadre l'ensemble des classes de $f(BME_{pg}(S))$. Tout d'abord, `BlocMaxE` élimine tout bloc (à patron commun) ayant été fusionné. Ces blocs étant enracinés dans des blocs à patrons communs de solidité supérieure, ils ne peuvent figurer dans aucune des classes de $f(BME_{pg}(S))$. Parmi les blocs non fusionnés, `BlocMaxE` identifie ceux dont le patron est maximal pour l'inclusion. Il suffit en effet pour chacun de ces blocs de tester l'inclusion de leur patron dans les patrons communs de solidité immédiatement supérieure. La complétude du test est garantie par transitivité de l'inclusion. Ce test permet, de manière symétrique, d'identifier tous les patrons communs qui incluent au moins un patron de bloc non filtré. La donnée de ces patrons est indispensable à l'étape de factorisation afin d'en garantir la correction [8]. `BlocMaxE` exécute la procédure de filtrage de manière rétro-active à chaque génération de couche en testant les blocs ayant servi à la génération avec ceux nouvellement générés.

Au final, la première étape de calcul produit donc l'ensemble des patrons correspondant x classes de f$f(BMI_{pg}(S))$, un sur-ensemble des patrons correspond aux classes de $f(BME_{pg}(S))$ et un ensemble de patrons communs dominant ces derniers. La seconde étape de factorisation extrait d'abord le domaine d'enracinement de chaque patron maximal sur chaque séquence. Par définition, ces domaines fournissent les classes d'équivalence de $f(BMI_{pg}(S))$. Elle procède de manière similaire pour le patrons candidats de $f(BME_{pg}(S))$. Pour chacun, elle détermine sur chaque séquence le domaine d'enracinement maximaux en testant chaque enracinement avec ceux des patrons dominants. Les patrons à domaine vide sont alors éliminés.

Avant de présenter `BlocMaxE` dans le détail, nous donnes une brève description d'APriori et d'une adaptation d'APriori au calcul d'itemsets maximaux sur lequel se fonde la première étape du calcul de patron.

5.3.1 Algorithme APriori

Le problème de recherche d'itemsets fréquents se définit à partir d'un ensemble d'items $\mathcal{I}$ et d'un ensemble de n transactions $\mathcal{T}$ tels $\mathcal{T} = \{(i,t)|t \subseteq \mathcal{I} \wedge i \in [\![n]\!]\}$. La couverture d'un itemset I, notée $\varphi_{\mathcal{T}}(I)$, correspond à l'ensemble des transactions dans lesquelles il apparait : $\varphi_{\mathcal{T}}(I) = \{(i,t) \in \mathcal{T}|I \subseteq t\}$. Le support d'un itemset, noté $support_{\mathcal{T}}(I)$, correspond au nombre de transactions couvertes : $support_{\mathcal{T}}(I) = |\varphi_{\mathcal{T}}(I)|$. Le problème de recherche d'itemsets fréquents consiste alors à trouver l'ensemble des itemsets dont le support dépasse un seuil Θ donné, soit

8. L'absence de ces blocs implique que les algorithmes présentés dans Lesaint [33] et Vigneron [59] sont incorrects.

l'ensemble $\{I | I \subseteq \mathcal{I} \wedge support_{\mathcal{T}}(I) \geqslant \Theta\}$.

Différents algorithmes de recherche d'itemsets fréquents maximaux ont été proposés, e.g. MAFIA [6], GenMax [19] et APriori [1]. Cette section présente ce dernier, dont l'algorithme d'extraction de blocs maximaux s'inspire.

L'algorithme APriori détermine l'ensemble des itemsets fréquents. Il parcourt le treillis des itemsets en largeur d'abord en générant les itemsets fréquents couche par couche où chaque couche correspond à une cardinalité d'itemset donnée. Pour cela il commence par générer la couche des itemsets singletons, les couches suivantes sont alors obtenues par fusion des itemsets de la couche précédente. La fusion correspond à l'union d'itemsets de même cardinalité et dont le résultat doit avoir la cardinalité de ses opérandes incrémentée de un. La propriété d'anti-monotonie du support d'itemset - tout super itemset d'un itemset infréquent est infréquent - permet un filtrage des itemsets obtenus par fusion.

L'algorithme 18 présente le processus de calcul des itemsets fréquents. Il débute en générant la première couche des itemsets singletons (ligne 3). À chaque itération, la nouvelle couche est créée d'abord en générant une liste de candidats possibles à partir des items fréquents de la couche précédente (ligne 6), puis en filtrant les candidats ne respectant pas le seuil de fréquence (ligne 7). L'étape de génération n'est pas explicitée ici mais son implantation influence les performances de l'algorithme.

Algorithme 18 – APriori

Entrée : $\mathcal{I}$ ensemble d'itemsets, $\mathcal{T}$ ensemble de transactions , Θ entier
Sortie : Ensemble des itemsets fréquents
Donnée : Ensemble d'itemsets ($i \in [\![\mathcal{T}]\!]$)
Donnée : F_i ensemble d'itemsets fréquents de cardinalité i
Donnée : C_i ensemble d'itemsets candidats de cardinalité i

```
1  begin
2      F_1 ← ∅ ;
3      foreach i ∈ I : support_T({i}) ⩾ Θ do F_1 ← F_1 ∪ {{i}} ;
4      k ← 2;
5      while F_{k-1} ≠ ∅ do
6          C_k ← {I | ∃U, V ∈ F_{k-1} ∧ I = U ∪ V ∧ |I| = k};
7          F_k ← {c | c ∈ C_k ∧ support_T(c) ⩾ Θ};
8          k ← k + 1;
9      end
10     return ⋃_{i∈[[k-1]]} F_i;
11 end
```

Exemple 10. Soient $\mathcal{I} = \{a, b, c, d\}$ et $\mathcal{T} = \{\{a, b, c, d\}, \{a, b, d\}, \{a, b\}, \{a, c\}, \{a, d\}, \{a, c, d\}, \{a\}\}$ et $\Theta = 2$. La base de transactions est représentée dans le tableau 5.2 et le treillis associé en

figure 5.1. Un nœud barré horizontalement indique que l'itemset a été filtré du à une fréquence trop faible de l'itemset tandis qu'un nœud barré avec une croix signifie que l'itemset n'a pas été généré du fait de l'existence d'un sous-itemset infréquent.

Le processus d'exploration du treillis est décrit ci-dessous :

1. La première étape génère la première couche du treillis constituée des itemsets singletons.
2. La couche des itemsets à deux éléments est générée à partir des itemsets singletons fréquents. Comme tous les items sont fréquents, tous les itemsets à deux éléments sont générés. La phase de filtrage élimine ensuite l'itemset $\{b, c\}$.
3. Les itemsets à 3 éléments sont générés à partir de ceux à deux éléments et comme $\{b, c\}$ est infréquent, aucun itemset contenant b et c n'est généré. La phase de filtrage n'élimine aucun candidat.
4. La dernière étape ne génère pas l'itemset à 4 éléments car des sous-itemsets sont infréquents.

$\mathcal{T}$	Itemsets
t_1	$\{ \ a \ , b \ , c \ , d \ \}$
t_2	$\{ \ a \ , b \ \ , d \ \}$
t_3	$\{ \ a \ , b \ \ \ \}$
t_4	$\{ \ a \ \ , c \ \ \}$
t_5	$\{ \ a \ \ \ , d \ \}$
t_6	$\{ \ a \ \ , c \ , d \ \}$
t_7	$\{ \ a \ \ \ \ \}$

TABLEAU 5.2 – Base de transactions $\mathcal{T}$

Les itemsets fréquents sont alors $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{a, b\}$, $\{a, c\}$, $\{a, d\}$, $\{b, d\}$, $\{c, d\}$, $\{a, b, d\}$ et $\{a, c, d\}$.

△

5.3.2 Adaptation d'APriori pour le calcul d'itemsets maximaux

L'algorithme 19 propose une modification de l'algorithme APriori pour extraire des itemsets maximaux. Un itemset est dit maximal s'il est fréquent et si tous ses super-itemsets sont

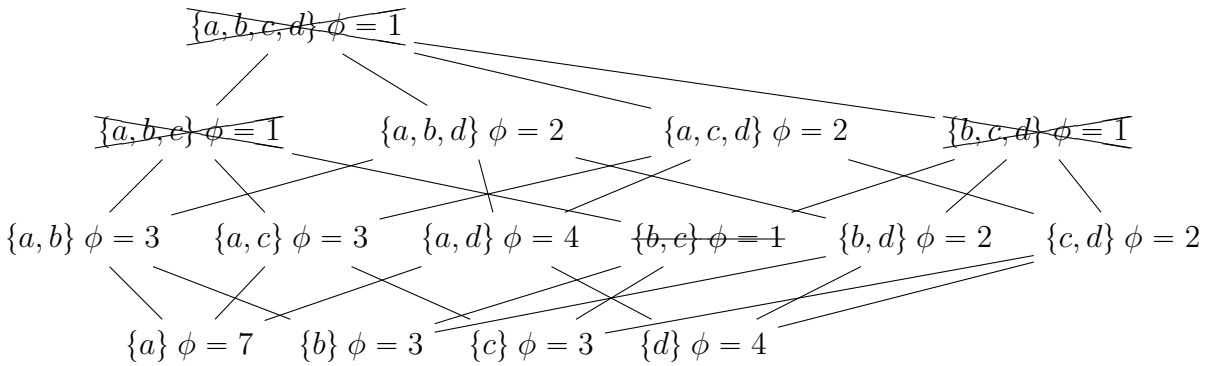

FIGURE 5.1 – Treillis d'itemsets

infréquents. `BlocMaxE` fonctionne sur le même principe que cette version d'APriori.

Algorithme 19 – APrioriMax

Entrée : $\mathcal{I}$ ensemble d'itemsets, $\mathcal{T}$ ensemble de transactions, Θ entier

Sortie : ensemble des itemsets maximaux

Donnée : M_i ensemble d'itemsets maximaux de cardinalité c, tel que $2^{i-1} \leqslant c < 2^i$

Donnée : B_i ensemble d'itemsets fréquents de cardinalité 2^i

Donnée : C_i ensemble d'itemsets candidats de cardinalité c, tel que $2^{i-1} < c \leqslant 2^i$

Donnée : F_i ensemble d'itemsets fréquents de cardinalité c, tel que $2^{i-1} < c \leqslant 2^i$

Donnée : L_i ensemble d'itemsets maximaux de cardinalité c, tel que $2^{i-1} \leqslant c \leqslant 2^i$

```
1  begin
2      M_0 ← ∅ ;
3      B_0 ← ∅ ;
4      foreach i ∈ I : support_T({i}) ⩾ Θ do B_0 ← B_0 ∪ {{i}} ;
5      k ← 1;
6      while B_{k-1} ≠ ∅ do
7          C_k ← {I | ∃U, V ∈ B_{k-1} : U ≠ V ∧ I = U ∪ V};
8          F_k ← {I | I ∈ C_k ∧ support_T(I) ⩾ Θ};
9          L_k ← {l | l ∈ F_k ∪ B_{k-1} ∧ (∀J ∈ F_k ∪ B_{k-1} : J = I ∨ I ⊄ J)};
10         M_k ← {I | I ∈ L_k ∧ |I| < 2^k} ;
11         B_k ← {I | I ∈ L_k ∧ |I| = 2^k} ;
12         k ← k + 1 ;
13     end
14     return ⋃_{i∈⟦k-1⟧} M_k;
15 end
```

Théorème 2. *APrioriMaX calcule l'ensemble des itemsets maximaux de* $(\mathcal{I}, \mathcal{T}, \Theta)$.

Démonstration.

1. Terminaison.

   La propriété $B_k \subseteq \{I | I \subseteq \mathcal{I} \wedge |I| = 2^k\}$ est un invariant de l'algorithme : la propriété est clairement vérifiée pour $k = 0$ (ligne 4) et à chaque itération (ligne 11) de la boucle (lignes 6-13) pour $k > 0$. Puisque $\{I | I \subseteq \mathcal{I} \wedge log_2(|I|) = log_2(|\mathcal{I}|) + 1\} = \varnothing$, il existe $k_{max} \in \{0, \cdots, log_2(|\mathcal{I}|) + 1\}$ tel que $B_{kmax} = \varnothing$ et $B_k \neq \varnothing$ ($k < k_{max}$), i.e., l'algorithme termine après k_{max} itérations de la boucle (lignes 6- 13).

2. Nous montrons par récurrence que $B_k = \{I | I \subseteq \mathcal{I} \wedge |I| = 2^k \wedge support_T(I) \geqslant \Theta\}$ pour tout $k \in \{0, \cdots, k_{max} - 1\}$.

   — $k = 0$. La propriété est clairement vérifiée lors de l'initialisation de B_0 en lignes (3) et (4).

   — Nous supposons la propriété vérifiée pour une valeur $k > 0$.

   (⊇) Soit $I \subseteq \mathcal{I}$, $|I| = 2^{k+1}$ et I fréquent. Il existe donc $J, J' \subseteq \mathcal{I}$ tels que $J \cup J' = I$, $|J| = |J'| = 2^k$ et $J, J' \in F_{k+1}$ (ligne 7-8). En outre, tout itemset de cardinalité maximum sur un ensemble d'itemsets S est nécessairement maximal pour l'inclusion sur S. D'autre part, $L_{k+1} \subseteq F_{k+1} \cup B_k$ et $\max_{K \in L_{k+1}} |K| \leqslant 2^{k+1}$ par définition de C_{k+1} et F_{k+1} (lignes 7-8). Puisque $J \cup J' \in F_{k+1}$ et $|J \cup J'| = 2^{k+1}$, $J \cup J'$ est maximal pour l'inclusion sur $F_{k+1} \cup B_k$, i.e. $I \in L_{k+1}$ (ligne 9) et $I \in B_{k+1}$ (ligne 11).

   (⊆) Soit $I \in B_{k+1}$. $I \in L_{k+1}$ (ligne 11) donc $I \in F_{k+1} \cup B_k$ (ligne 9). Puisque $|I| = 2^{k+1}$ et $\max_{K \in B_k} |K| < 2^{k+1}$ par hypothèse, $I \in F_{k+1}$. Puisque $F_{k+1} \subseteq C_{k+1} \subseteq \mathcal{I}$, nous en déduisons $I \in \{I' | I' \subseteq J \wedge |I'| = 2^{k+1} \wedge support_T(I') \geqslant \Theta\}$.

3. Soit $M = \{I | I \subseteq \mathcal{I} \wedge support_T(I) \geqslant \Theta \wedge (\forall J \supset I : support_T(J) < \Theta)\}$ l'ensemble des itemsets maximaux. Nous montrons par récurrence que $M_k = \{I | I \in M \wedge 2^{\max(0,k-1)} \leqslant |I| < 2^k\}$ pour $k \in \{0, \cdots, k_{max}\}$.

   — $k = 0$. La propriété est clairement vérifiée par $M_0 = \varnothing$ (ligne 2).

   — Nous supposons la propriété vérifiée pour une valeur $k > 0$.

($\supseteq$) Soit $I \in M$ tel que $2^k \leqslant |I| < 2^{k+1}$. Si $|I| = 2^k$ alors $I \in B_k$ puisque I fréquent (invariant (2)). Puisque I est maximal, $I \in L_{k+1}$ (ligne 9) et donc $I \in M_{k+1}$ (ligne 10). Si $|I| > 2^k$ alors $I \in F_{k+1}$ (selon le même argument que celui développé en (2)) puisque I fréquent. Puisque I est maximal, nous en déduisons $I \in M_{k+1}$.

($\subseteq$) Soit $I \in M_{k+1}$. $I \in L_{k+1}$ donc I maximal pour l'inclusion sur L_{k+1} et I fréquent. Si $I \notin M$ alors il existe $J \supset I$ tel que $|J| = |I| + 1$ et J fréquent, donc $2^{k-1} < |J| \leqslant 2^k$. Du fait de (2) et par construction de F_{k+1}, $J \in F_{k+1}$ ce qui contredit $I \in L_{k+1}$ (car $I \subset J$) donc $I \in M$.

□

Filtrage (des itemsets fréquents ou non-maximaux) mis à part, le principe d'APrioriMax consiste à générer la couche des itemsets singletons puis à générer chaque nouvelle couche par union deux à deux des itemsets de taille maximum de la couche précédente jusqu'à épuisement de ces éléments. Chaque itération, sauf la dernière, multiplie donc par 2 la taille maximum d'itemsets et chaque couche non-terminale de niveau k ($k \geqslant 1$) contient l'ensemble des itemsets de taille t vérifiant $2^k < t \leqslant 2^{k+1}$ puisque tout itemset vérifiant cette propriété s'exprime comme l'union de deux itemsets de taille 2^k. L'algorithme permet donc un parcours exhaustif du treillis d'itemsets, l'opération de génération des candidats (ligne 7) se chargeant d'éliminer les doublons au sein de chaque couche.

Le filtrage des itemsets infréquents s'applique à chaque itération et porte sur l'ensemble C_k des itemsets générés pour la couche k (ligne 7). Ces itemsets sont générés par fusion des itemsets de taille maximum (ensemble B_{k-1}) de la couche précédente. Ce filtrage produit l'ensemble F_k des itemsets fréquents de la couche k (ligne 8) dont on extrait l'ensemble B_k des itemsets de taille 2^k (ligne 9 et 11). Cet ensemble, selon qu'il est vide ou non (ligne 6), conditionne l'arrêt de la recherche ou la génération de la couche suivante. Du fait de l'antimonotonie de la contrainte de seuil minimal de fréquence pour l'inclusion et de la décomposabilité de tout itemset de la couche supérieure par fusion d'itemsets de B_{k-1}, B_k correspond exactement à l'ensemble des itemsets fréquents de taille 2^k et F_k correspond à l'ensemble des itemsets fréquents de taille t telle que $2^{k-1} < t \leqslant 2^k$. Enfin, l'algorithme calcule pour chaque couche les itemsets maximaux pour l'inclusion au sein de l'ensemble $B_{k-1} \cup F_k$ des itemsets fréquents de taille t telle que $2^{k-1} \leqslant t \leqslant 2^k$. Le calcul (ligne 9) produit l'ensemble L_k dont nous extrayons l'ensemble M_k d'itemsets de taille strictement inférieure à 2^k (ligne 10). Du fait de la propriété de décomposabilité sus-mentionnée, tout élément de M_k est nécessairement maximal parmi les itemsets fréquents. Réciproquement, tout itemset maximal de taille t telle que $2^{k-1} \leqslant t < 2^k$

est fréquent - donc appartient à $B_{k-1} \cup F_k$ - et est maximal sur $B_{k-1} \cup F_k$ donc appartient à M_k.

5.3.3 Calcul des blocs maximaux pour l'enracinement

Cette section présente l'algorithme `BlocMaxE` pour la factorisation de blocs maximaux, pour l'enracinement ou l'inclusion, et soumis à contrainte de solidité minimale 2, et à une contrainte d'écart maximal paramétré.

L'algorithme 20 décrit la procédure principale de `BlocMaxE`. L'algorithme sélectionne d'abord une séquence s du jeu de séquences S (ligne 2). Il appelle ensuite l'algorithme `generation_blocs_maximaux_candidats` (ligne 3) qui calcule l'ensemble de blocs enracinés dans s, dont les patrons sont inclus dans toutes les séquences de S, et qui vérifient les contraintes de solidité et d'écart. Enfin l'appel à l'algorithme `factorisation_blocs_maximaux` (ligne 4) calcule la factorisation de blocs maximaux pour l'inclusion et l'enracinement de S à partir de cet ensemble de blocs.

Algorithme 20 – BlocMaxE

Entrée : S ensemble de séquences, g écart maximal
Sortie : $(BMI, BME) = (f(BMI_{2g}(S)), f(BME_{2g}(S)))$
Donnée : s une séquence
Donnée : MI ensemble de blocs sur S
Donnée : ME ensemble de blocs sur S
Donnée : I ensemble de blocs sur S

```
1 begin
2     s ← selection_sequence(S) ;
3     (MI, ME, I) ← generation_blocs_maximaux_candidats(S, s, g) ;
4     (BMI, BME) ← factorisation_blocs_maximaux(MI, ME, I) ;
5     return (BMI, BME) ;
6 end
```

L'algorithme `generation_blocs_maximaux_candidats` (algorithme 21) calcule l'ensemble des blocs maximaux sur $\{s\}$ et dont les patrons sont inclus dans toutes les séquences de S. L'algorithme commence par générer les 2-blocs de $\{s\}$, d'écart maximal g et inclus dans toutes les séquences de S (ligne 3). Il génère ensuite itérativement les couches suivantes (lignes 5-14). Chaque couche est générée par fusion des blocs de la couche précédente dont le nombre de caractères solides est maximum (ligne 6). Nous mettons de côté les blocs nouvellement générés dont la solidité est maximale (ligne 7). Nous identifions parmi les autres blocs et ceux utilisés pour la fusion (B_{k-1}) ceux qui sont maximaux pour l'enracinement en vérifiant qu'il n'existe aucun bloc généré les enracinant (ensemble M_k, ligne 8). Nous identifions ensuite parmi ces blocs maximaux ceux dont le patron n'est strictement inclus dans aucun patron des blocs (ensemble MI_k, ligne 9). Lors de cette opération, sont également conservés parmi les blocs géné-

rés ceux dont le patron inclut un des blocs maximaux pour l'enracinement (I_k). L'algorithme cumule dans les ensembles ME, MI et I les ensembles de blocs M_k (blocs candidats à la maximalité pour l'enracinement), MI_k (blocs maximaux pour l'inclusion) et I_k (blocs générés ne figurant pas dans M_k mais incluant des blocs de M_k) à chaque itération (ligne 10-12). L'algorithme termine en retournant le triplet (ME, MI, I) (ligne 15).

Algorithme 21 – `generation_blocs_maximaux_candidats`

Entrées : S ensemble de séquences, s séquence de S, g écart maximal
Sortie : (MI, ME, I)
Donnée : B_k ensemble de blocs de $\{s\}$ de patron commun à S ($k \geqslant 1$)
Donnée : C_k ensemble de blocs de $\{s\}$ de patron commun à S ($k \geqslant 2$)
Donnée : MI_k ensemble de blocs de $\{s\}$ de patron commun à S ($k \geqslant 2$)
Donnée : ME_k ensemble de blocs de $\{s\}$ de patron commun à S ($k \geqslant 2$)
Donnée : I_k ensemble de blocs de $\{s\}$ de patron commun à S ($k \geqslant 2$)

```
1  begin
2      (MI, ME, I) ← (∅, ∅, ∅);
3      B_1 ← generation_2_blocs(S, s, g) ;
4      k ← 2 ;
5      while B_{k-1} ≠ ∅ do
6          C_k ← fusion_blocs(S, g, B_{k-1}) ;
7          B_k ← {b | b ∈ C_k ∧ π(b) = 2^k} ;
8          ME_k ← blocs_e_maximaux_candidats(B_{k-1} ∪ C_k \ B_k, C_k) ;
9          (MI_k, I_k) ← blocs_i_maximaux(ME_k, C_k) ;
10         MI ← MI ∪ MI_k ;
11         ME ← ME ∪ ME_k ;
12         I ← I ∪ I_k ;
13         k ← k + 1 ;
14     end
15     return (MI, ME, I);
16 end
```

L'algorithme `generation_2_blocs` (algorithme 22) calcule les 2-blocs d'une séquence s dont les patrons sont inclus dans toutes les séquences de S et dont l'écart est inférieur ou égal à g. Pour cela, il crée les 2-patrons inclus dans S en énumérant, dans la séquence s, toutes les combinaisons de deux caractères espacés d'au plus g caractères. Chaque patron généré est alors testé pour la relation d'inclusion avec l'ensemble des séquences de S.

Algorithme 22 – `generation_2_blocs`

Entrées : S ensemble de séquences, s séquence de S, g écart maximal
Sortie : $B = \{b | b \in B(\{s\}) \wedge \pi(b) = 2 \wedge \gamma(b) \leqslant g \wedge (\forall s' \in S \ : \ p^{(b)} \leqslant s')\}$
Donnée : g écart
Donnée : d, f enracinements
Donnée : p patron

```
1  begin
2      B ← ∅ ;
3      foreach i ∈ 0..g_max do
4          foreach d ∈ [[|s| − (i + 1)]] do
5              f ← d + (i + 1) ;
6              p ← concatener(s_d, repeter(#, i), s_f) ;
7              if inclus(p, S) then B ← B ∪ {({s}, p, d − 1)};
8          end
9      end
10     return B ;
11 end
```

Algorithme 23 – `inclus`

Entrées : p patron, S ensemble de séquences
Sortie : $true$ si p est inclus dans toutes les séquences de S, $false$ sinon
Donnée : s séquence
Donnée : b booléen

```
1  begin
2      b ← true ;
3      foreach s ∈ S do  b ← b ∧ (p ⩽ s) ;
4      return b ;
5  end
```

L'algorithme `fusion` (algorithme 24) effectue la fusion entre deux blocs d'une même séquence. La fusion est relative au décalage d'enracinement des blocs (ligne 5). Comme l'opération est commutative, ordonner les blocs par leur enracinement n'invalide pas le résultat (lignes 2 - 4). Le bloc résultant sera enraciné à la même position que le bloc d'enracinement minimum (ligne 6). La ligne 7 complète le patron du bloc d'enracinement minimum avec des caractères jokers afin de l'agrandir jusqu'à la longueur du patron du bloc résultant. Finalement, les deux chaînes sont fusionnées en priorisant les caractères solides sur les jokers (lignes 8 - 10).

Algorithme 24 – `fusion`

Entrées : b_1, b_2 blocs
Condition : b_1 et b_2 enracinés sur une unique et même séquence ($S^{(b_1)} = S^{(b_1)}$)
Sortie : Bloc
Donnée : b bloc, g écart, s séquence
Donnée : e, e_1 et e_2 enracinements
Donnée : p, p_1 et p_2 patrons

1 **begin**
2 $(s, p_1, e_1) \leftarrow b_1$;
3 $(s, p_2, e_2) \leftarrow b_2$;
4 **if** $e_2 < e_1$ **then** permuter((p_1, e_1),(s_2, e_2));
5 $g \leftarrow e_2 - e_1$;
6 $e \leftarrow e_1$;
7 $p \leftarrow \texttt{concatener}(p_1, \texttt{repeter}(\#, \max(0, g + |p_2| - |p_1|)))$;
8 **foreach** $i_2 \in [\![|p_2|]\!]$ **do**
9 **if** $p_{i_2+g} = \#$ **then** $p_{i_2+g} \leftarrow p_{2_{i_2}}$;
10 **end**
11 **end**
12 $b \leftarrow (s, p, e)$;
13 **return** b

L'algorithme `fusion_blocs` (algorithme 25) fusionne chaque paire de blocs de l'ensemble B donné en entrée (ligne 3). Pour chaque paire, il fait appel à l'algorithme `fusion` (ligne 4), puis vérifie que le bloc retourné est inclus dans S, qu'il satisfait la contrainte d'écart maximal et qu'il n'a pas déjà été calculé auparavant (ligne 5). Seuls les candidats vérifiant ces conditions sont retournés (ligne 6).

L'algorithme `blocs_e_maximaux_candidats` (algorithme 26) détecte parmi un ensemble de blocs B ceux qui ne sont enracinés dans aucun bloc d'un autre ensemble de blocs C. Pour cela, l'algorithme initialise ME avec l'ensemble des blocs de B (ligne 2) et retire au fur et à mesure les blocs enracinés dans des blocs de C (ligne 6).

L'algorithme `blocs_i_maximaux_candidats` (algorithme 27) détecte pour un ensemble de blocs B ceux dont les ne patrons sont pas inclus dans les patrons des blocs d'un ensemble de blocs C. Il conserve également des blocs de C ceux dont les patrons incluent des patrons de blocs de B.

Algorithme 25 – `fusion_blocs`

Entrées : S ensemble de séquences, B ensemble de blocs, g écart maximal
Condition : $\emptyset \subset B \subseteq B(\{s\})$ pour $s \in S$
Sortie : $C = \{b | b = b_1 \sqcup b_2 \wedge b_1, b_2 \in B \wedge b_1 \neq b_2 \wedge \gamma(b) \leqslant g \wedge (\forall s' \in S \ : \ p^{(b)} \leqslant s')\}$
Donnée : b_1, b_2, b^* blocs

```
1  begin
2      C ← ∅ ;
3      foreach b1, b2 ∈ B : e^(b1)(s) < e^(b2)(s) do
4          b* ← fusion(b1, b2) ;
5          if gap(b*) ⩽ g ∧ b* ∉ C ∧ inclus(p^b*, S) then
6              C ← C ∪ {b*} ;
7          end
8      end
9      return C ;
10 end
```

Algorithme 26 – `blocs_e_maximaux_candidats`

Entrée : B ensemble de blocs sur s ($s \in S$)
Entrée : C ensemble de blocs sur s
Sortie : $ME = \{b | b \in B \wedge (\forall b' \in C \ : \ b \not\sqsubset b')\}$
Donnée : b, b' blocs sur s

```
1  begin
2      ME ← B ;
3      foreach b ∈ B do
4          foreach b' ∈ C do
5              if enraciné(b, b') then
6                  ME ← ME\{b};
7                  break ;
8              end
9          end
10     end
11     return ME ;
12 end
```

L'algorithme `factorisation_blocs_maximaux` (algorithme 28) calcule la factorisation des blocs de S maximaux pour l'enracinement (BME). Cette factorisation produit un ensemble de couples associant un patron et des ensembles (non vides) d'enracinements de ce patron sur toutes les séquences de S. L'algorithme factorise d'abord l'ensemble LI des blocs à patrons maximaux pour l'inclusion (lignes 2-11). Pour ce faire, il extrait les patrons des blocs MI en ne conservant qu'une occurrence de patron (ligne 3). Il calcule ensuite les domaines d'enracinement de ces patrons sur chacune des séquences (ligne 4-11). L'algorithme factorise ensuite l'ensemble des blocs maximaux pour l'enracinement (ligne 12-39). A l'instar de la factorisation des blocs maximaux pour l'inclusion, l'approche consiste à extraire les patrons des blocs

Algorithme 27 – `blocs_i_maximaux_candidats`

Entrée : B ensemble de blocs sur s ($s \in S$)
Entrée : C ensemble de blocs sur s
Sortie : $(MI, I) = (\{b | b \in B \wedge (\forall b' \in C \ : \ p^{(b)} \not< p^{(b')})\}, \{b' | b' \in C \wedge (\exists b \in B \ : \ p^{(b)} < p^{(b')})\})$

```
1  begin
2      MI ← B ;
3      I ← ∅ ;
4      foreach b ∈ B do
5          foreach b' ∈ C do
6              if inclus(b, b') then
7                  MI ← MI\{b};
8                  I ← I ∪ {b'} ;
9              end
10         end
11     end
12     return (MI, I) ;
13 end
```

de l'ensemble ME donné en entrée (ligne 13) et à déterminer pour chacun son domaine d'enracinement sur chaque séquence. Toutefois, certains de ces patrons peuvent n'avoir aucun bloc maximal sur certaines séquences. Il s'agit donc d'identifier ces patrons afin de les éliminer de l'ensemble réponse. Par construction, il suffit pour chaque patron et sur chacune des séquences, de s'assurer de l'existence d'un bloc qui ne soit enraciné dans aucun des blocs associés aux patrons des blocs de l'ensemble I.

L'algorithme extrait donc les patrons des blocs de I au préalable (ligne 14). Pour chaque patron associé aux blocs candidats de ME (boucle 15-39), et pour chacune des séquences (boucle 18-37), il calcule le domaine d'enracinement du patron (ligne 19) et pour chacun des blocs correspondants (boucle 20-32), nous testons la maximalité relativement aux patrons de I incluant le patron (boucle 21-31) et procédons à l'élimination du bloc le cas échéant (ligne 25-27). Tout patron sans bloc maximal sur une séquence est alors éliminé (lignes 33-35 et 38).

Par défaut, l'algorithme BMaxE calcule les BMI. Si nous souhaitons nous limiter au calcul des BMI, il n'est pas nécessaire de calculer la factorisation des BME. Il suffit donc d'adapter l'algorithme BlocMaxE (algorithme 20) en vidant les ensemble ME et MI avant l'appel de la

fonction de factorisation (ligne 4).

Algorithme 28 – `factorisation_blocs_maximaux`

Entrée : MI ensemble de blocs sur s ($s \in S$)
Entrée : ME ensemble de blocs sur s ($s \in S$)
Entrée : I ensemble de blocs sur s ($s \in S$)
Sortie : $(BMI, BME) = (f(MI), \bigcup_{b \in ME} \{(p^{(b)}, E) | E : S \rightarrow 2^{\mathbb{N}} \wedge \forall s \in S,\ \forall e \in E(s),\ \forall b' \in I,\ \forall b'' \in B(s,S)\ :\ p^{(b')} = p^{(b'')} \rightarrow (\{s\}, p^{(b)}, e) \not\sqsubset b''\}$

```
1  begin
2      BMI ← ∅ ;
3      PMI ← patrons(MI) ;
4      foreach p ∈ PMI do
5          E ← ∅ ;
6          foreach s ∈ S do
7              E_s ← enracinements(p, s) ;
8              E ← E ∪ {(s, E_s)};
9          end
10         BMI ← BMI ∪ {(p, E)} ;
11     end
12     BME ← ∅ ;
13     PME ← patrons(ME) ;
14     PI ← patrons(I) ;
15     foreach p_1 ∈ PME do
16         max ← true ;
17         E ← ∅;
18         foreach s ∈ S do
19             E_1s ← enracinements(p_1, s);
20             foreach e_1 ∈ E_1s do
21                 ok ← true ;
22                 foreach p_2 ∈ PI : sol(p_2) > sol(p_1) do
23                     E_2s ← enracinements(p_1, s);
24                     foreach e_2 ∈ E_2s do
25                         if enracine({s}, p_1, e_1), ({s}, p_2, e_2) then
26                             E_1s ← E_1s\{e_1} ;
27                             ok ← false ;
28                             break ;
29                         end
30                     end
31                     if not ok then break;
32                 end
33             end
34             if E_1s = ∅ then
35                 max ← false ;
36                 break ;
37             end
38         end
39         if max then BME ← BME ∪ {(p, E)} ;
40     end
41     return (BMI, BME) ;
42 end
```

5.3.3.1 Complexité de `BlocMaxE`

Nous établissons ici la complexité en pire cas en temps de l'algorithme BlocMaxE sur la base des différentes étapes et procédures utilisées pour l'algorithme. Nous utilisons les notations suivantes : L dénote la taille de la plus grande séquence de S, e l'écart maximal autorisé et N le nombre maximal de blocs d'une couche du treillis (i.e. une couche est constituée de blocs de même solidité) [9].

Nous allons supposer que nous manipulons des arbres binaires équilibrés comme structure de données pour stocker les différents blocs. Pour un ensemble de N éléments avec un opérateur de comparaison entre deux éléments s'exécutant en linéaire $O(L)$, alors l'insertion, la suppression et la recherche d'un élément sont toutes $O(L.log_2(N))$.

Le test d'inclusion d'un patron dans une séquence ou un autre patron est en $O(L^2)$ et le test d'égalité entre deux patrons est en $O(L)$.

5.3.3.1.1 Génération des 2-blocs :

L'algorithme `generation_2_blocs` considère au plus $(e+1).L$ 2-blocs, soit le produit de l'écart maximal et de la longueur de la séquence sélectionnée. Vérifier si un 2-patron est inclus dans une séquence s'effectue en temps linéaire (il suffit seulement de tester les caractère extrêmes des 2-blocs pour tous les décalages), donc le test d'inclusion s'effectue en $O(|S|.L)$ pour toutes les séquences. La concaténation [10] présente dans l'algorithme s'effectue en temps linéaire également ($O(L)$). Dans cet algorithme, nous générons au plus L^2 blocs et la comparaison entre ces blocs s'effectue en temps constant [11] alors l'insertion s'effectue en $O(log_2(L^2)) = O(log_2(L))$. L'algorithme est donc en $O(e.L(L.|S| + L + log_2(L))) = O(e.L^2.|S|)$.

5.3.3.1.2 Fusion :

L'algorithme `fusion` ne dépend que de la longueur du patron le plus long qui puisse être généré, cette taille ne peut pas excéder celle de la plus petite séquence, donc l'algorithme est en $O(L)$.

5.3.3.1.3 Fusion des blocs :

L'algorithme `fusion_blocs` s'exécute à chaque itération de la boucle principale de `BlocMaxE`. Pour une couche donnée du treillis, la fusion des blocs requiert de tester toutes les paires de

9. N est borné par la taille de la couche médiane du treillis, soit par $\binom{O(L}{\lfloor L/2 \rfloor)}$.
10. En pratique, créer un 2-blocs sous la forme d'une chaîne de caractères complétée avec des jokers n'est pas requis.
11. Il suffit de comparer la taille des patrons et leurs caractères extrêmes.

blocs, le nombre de blocs est majoré par N, donc le nombre de paires est majoré par N^2. L'opération de fusion est réalisée en $O(L)$. Le test d'inclusion d'un bloc dans une séquence s'effectue en temps quadratique [12] en fonction de la taille du plus grand patron, soit une complexité pour le test d'inclusion sur toutes les séquences de $O(|S|.L^2)$. Le nombre de blocs à considérer pour une couche est de $N.\lceil(L/2)\rceil$ [13]. L'insertion et le test d'inclusion s'effectuent en $O(L.log_2(N.\lceil(L/2)\rceil) = O(L.log_2(L.N))$. Donc, la complexité de cet algorithme est $O(N^2.(L + |S|.L^2 + L.log_2(L.N)) = O(N^2.(L + |S|.L^2 + L.log_2(L) + L.log_2(N)))$ donc $O(L.N^2.(|S|.L + log_2(N)))$.

5.3.3.1.4 Blocs maximaux pour l'enracinement :

L'algorithme `blocs_e_maximaux_candidats` parcourt toutes les paires possibles de blocs dont le nombre est majoré par N^2 (ligne 3-4), et teste l'enracinement (sur une séquence) entre ces blocs appariés. Sur une unique séquence le test d'enracinement s'effectue en temps linéaire (ligne 5) - le décalage à considérer entre deux blocs est connu. La complexité de cet algorithme est donc $O(L.N^2)$.

5.3.3.1.5 Blocs maximaux pour l'inclusion :

L'algorithme `blocs_i_maximaux_candidats` parcourt toutes paires possibles de blocs (N^2) (lignes 4-5), et teste l'inclusion entre ces blocs appariés. Le test d'inclusion s'effectue en temps quadratique par rapport à la taille du patron le plus long associé à un bloc, soit $O(L^2)$ (ligne 6). La complexité de cet algorithme est donc $O(L^2.N^2)$.

5.3.3.1.6 Génération des blocs maximaux candidats :

L'algorithme `blocs_maximaux_candidats` fait appel à l'algorithme `generation_2_blocs` (ligne 3) dont la complexité est $O(e.L^2.|S|)$ et donc $O(L^3.|S|)$ (par définition $e < L$). L'opérateur de fusion double la taille des patrons (associés aux blocs) de taille maximale à chaque nouvelle itération, donc le nombre d'itérations est majoré par $log_2(L)$ (ligne 5-14). Les complexités des algorithmes de fusion de blocs, de détermination des blocs maximaux pour l'enracinement et de détermination des blocs maximaux pour l'inclusion sont respectivement $O(L.N^2.(|S|.L + log_2(N)))$ (ligne 8), $O(L.N^2)$ et $O(L^2.N^2)$ (ligne 9). Le nombre d'éléments contenus dans MI, I et ME avant leurs mises à jours (lignes 10-11) est majoré par $L.\lceil(L/2)\rceil.N$. Cette

12. Contrairement à la génération des 2-blocs, les patrons des blocs contiennent ici plus de deux caractères.

13. À chaque itération nous doublons le nombre de caractères solides, donc le nombre de couches générées. De ce fait, la dernière itération contient jusqu'à deux fois plus de couches que toutes celles générées précédemment et le nombre de couches de cette dernière itération est donc majoré par $\lceil\frac{L}{2}\rceil$.

dernière borne majore également le nombre d'éléments des MI_k, I_k et ME_k. La comparaison entre deux blocs est réalisée en temps linéaire, donc la complexité de ces insertions est $O(L.\lceil(L/2)\rceil.N.log_2(\lceil(L/2)\rceil.N+\lceil(L/2)\rceil)) = O(L^2.N.log_2(L.N))$. La complexité de la boucle principale (ligne 5-14) est alors de $O(L.N^2 + L^2.N^2 + L.N^2.(|S|.L + log_2(N))+ L^2.N.log_2(L.N)) = O(L.N^2.(|S|.L + log_2(N)) + L^2.N.log_2(L.N)) = O(L.N.(N.|S|.L + N.log_2(N) + L.log_2(L) + L.log_2(N))) = O(L.N.(N.|S|.L + N.log_2(N) + L.log_2(L)))$. Nous en déduisons que la complexité de l'algorithme de génération des blocs maximaux candidats est $O(L.N.(N.|S|.L + N.log_2(N) + L.log_2(L)) + |S|.L^3)$.

5.3.3.1.7 Factorisation des blocs maximaux :

L'algorithme `factorisation_blocs_maximaux` s'effectue en deux temps : le calcul des blocs maximaux pour l'inclusion puis le calcul des blocs maximaux pour l'enracinement. Les blocs maximaux pour l'enracinement contiennent ceux maximaux pour l'inclusion, donc la complexité des enracinements pour les blocs maximaux pour l'inclusion est trivialement bornée par celle du calcul des enracinements pour les blocs maximaux pour l'enracinement. Récupérer les patrons des blocs maximaux candidats et ceux incluant les patrons des blocs maximaux revient à insérer $L.N$ (N blocs par couche au plus et L couches au plus) patrons dans un ensemble, soit une complexité de $O(L.N.L.log_2(N.L))$ (ligne 14). Calculer la factorisation des blocs maximaux nécessite de considérer le produit cartésien des patrons des blocs maximaux candidats et les patrons des blocs incluant ceux-ci. Ce produit cartésien est majoré par $L^2.N^2$ (lignes 15 et 22). Ensuite, pour chaque paire, composée d'un patron candidat p et d'un patron p', il faut tester pour chaque enracinement du patron candidat p sur chaque séquence, si ce candidat n'est pas enraciné dans p'. Le nombre de localisations pour chaque patron sur chaque séquence est au plus de L, donc le nombre de combinaisons d'enracinements entre deux patrons sur toutes les séquences est borné par $|S|.L^2$ (lignes 19-20 et 23-24). Tester l'enracinement s'effectue en temps linéaire $O(L)$ (ligne 25). L'insertion s'effectue en $O(L.log_2(N.L))$. La complexité de la boucle principale est alors de $O(|S|.L^2.L^2.N^2.(L + L.log_2(N.L))$ (nombre de paires de patrons, nombre d'enracinements à tester, et test d'inclusion) (ligne 26). Nous en déduisons que la complexité de l'algorithme est de $O(|S|.L^5.N^2.log_2(N.L))$.

5.3.3.1.8 `BlocMaxE`

`BlocsMaxE` séquence la génération des blocs candidats puis leur factorisation des précédents blocs. La complexité de `BlocMaxE` est alors de

$$
\begin{aligned}
&O(L.N.(N.|S|.L + N.log_2(N) + L.log_2(L)) + |S|.L^3 + |S|.L^5.N^2.log_2(N.L)) \\
&= O(L.N.(N.|S|.L + N.log_2(N) + L.log_2(L)) + |S|.L^5.N^2.log_2(N.L)) \\
&= O(L^2.N^2.|S| + L.N^2.log_2(N) + L^2.log_2(L).N + |S|.L^5.log_2(L).N^2 + |S|.L^5.N^2.log_2(N)) \\
&= O(|S|.L^3.log_2(L).N^2 + |S|.L^5.N^2.log_2(N)) \\
&= O(|S|.L^5.N^2.log_2(N.L)).
\end{aligned}
$$

L'étape de factorisation est donc dominante en pire cas et détermine la complexité de `BlocMaxE`.

5.3.3.2 Factorisation des patrons maximaux (avec inclusion uniquement)

La complexité de l'algorithme `factorisation_blocs_maximaux` change légèrement si nous ne considérons que les blocs à patrons maximaux. En effet, il ne faut considérer que la première boucle qui détermine seulement des enracinements pour un ensemble de blocs sur S séquence de taille au plus L. Donc la complexité de cet algorithme est de $O(L.N.|S|.L^2)$

5.3.3.2.1 Algorithme général (avec inclusion uniquement)

La complexité de `BlocsMaxE` est correspond à la somme des complexité de générations des blocs candidats et de la factorisation (avec inclusion uniquement) des précédents blocs. La complexité de cet algorithme est alors de

$$
\begin{aligned}
&O(L.N.(N.|S|.L + N.log_2(N) + L.log_2(L)) + |S|.L^3 + |S|.L^3.N) \\
&= O(L^2.N^2.|S| + L.N^2.log_2(N) + L^2.log_2(L).N + |S|.L^3 + |S|.L^3.N) \\
&= O(L^2.N^2.|S| + L.N^2.log_2(N) + |S|.L^3.N) \\
&= O(L.(L.N^2.|S| + N^2.log_2(N) + |S|.L^2.N)).
\end{aligned}
$$

5.4 Approche dynamique

Nous présentons une alternative inspirée de la programmation dynamique pour le calcul des patrons maximaux pour l'inclusion.

L'algorithme que nous proposons procède par décomposition en résolvant le problème sur des ensembles de séquences croissants. Il exploite le fait que tout patron maximal sur un en-

semble de séquences S est nécessairement inclus dans un patron maximal de tout sous-ensemble (non-vide) de S. Le principe consiste donc à calculer l'ensemble des patrons maximaux sur deux séquences choisies, puis, à chaque itération, à ajouter une nouvelle séquence et à extraire les patrons maximaux sur l'ensemble de séquence résultant à partir de l'ensemble de patrons précédent. Le processus se termine lorsque toutes les séquences du jeu considéré ont et intégrées. L'opération élémentaire de calcul de patrons maximaux repose sur une méthode d'alignement dédiée.

Nous établissons tout d'abord la propriété de décomposition sus-mentionnée avant de décrire l'algorithme `BlocMaxIDyn` en section suivante.

Pour tout ensemble non-vide S de séquences ($\varnothing \subset S \subseteq 2^{\Sigma^+}$), nous notons $P(S)$ l'ensemble des patrons communs aux séquences de S (i.e., inclus dans chaque séquence de S) et $PM(S)$ l'ensemble des patrons maximaux pour l'inclusion sur $P(S)$:

— $P(S) = \bigcup_{p \in \Sigma \cup (\Sigma.(\Sigma')^*.\Sigma)} \{p | \forall s \in S \ : \ p \leqslant s\}$;

— $PM(S) = \bigcup_{p \in P(S)} \{p | \forall p' \in P(S) \ : \ p \not< p'\}$.

La règle de décomposition sur laquelle repose l'algorithme stipule que tout patron maximal sur un ensemble de séquences est inclus dans au moins un patron maximal sur un sous-ensemble non-vide de séquences quel que soit le sous-ensemble choisi.

Théorème 3. *Soient S, S' tels que $\varnothing \subset S \subseteq S' \subseteq 2^{\Sigma^+}$. Pour tout $p' \in PM(S')$, il existe $p \in PM(S)$ tel que $p' \leqslant p$.*

Démonstration. Soit $\varnothing \subset S \subseteq S' \subseteq 2^{\Sigma^+}$ et $p' \in PM(S')$. $p' \in P(S')$ par définition de $PM(S')$ donc $p' \in P(S)$ par définition de $P(S')$. $\leqslant$ étant un ordre partiel sur l'ensemble fini $P(S)$, tout élément de $P(S)$ est soit maximal pour $<$, soit inclus strictement dans un élément maximal pour $<$. Par conséquent, il existe $p \in PM(S)$ tel que $p' \leqslant p$. □

5.4.1 Algorithme

L'algorithme `BlocMaxIDyn` calcule les patrons maximaux d'un ensemble S de séquences de façon incrémental. Il commence par calculer les patrons maximaux communs à deux séquences, puis réitère ce calcul en adjoignant une nouvelle séquence à chaque étape jusqu'à couvrir intégralement S. L'ordre dans lequel les séquences sont sélectionnées influence l'efficacité de l'algorithme.

Le calcul de patrons maximaux entre deux séquences procède par alignement en considérant tous les décalages de positionnement possibles entres les deux séquences. Pour chaque décalage considéré, la méthode d'alignement consiste à détecter les positions communes (relativement

au décalage) sur lesquelles les séquences partagent le même caractère et les positions où il n'y a pas concordance ou bien s'il ne peut y avoir concordance (position non commune). Chaque alignement produit donc une chaîne constituée des caractères solides correspondant à une position concordante et du caractère joker pour chacune des positions non concordantes. Si cette chaîne contient au moins un caractère solide définit un patron commun aux deux séquences et qui est maximal relativement au décalage choisi. Il suffit alors d'extraire parmi tous les patrons communs calculés ceux qui sont maximaux pour l'inclusion.

La procédure d'alignement décrite ci-dessus peut être généralisée afin d'intégrer une contrainte d'écart maximal sur les patrons. Dans ce cas, la procédure peut générer plus d'un patron par décalage : dès qu'un patron d'écart maximal est atteint, il est mémorisé et l'alignement reprend à partir du dernier caractère solide afin de détecter les patrons restants s'il en existe.

Cette procédure est aussi réutilisée pour l'extraction de patrons maximaux lors de l'ajout d'une séquence. Il s'agit en effet dans ce cas de déterminer pour chaque patron maximal précalculé, les patrons maximaux qui ne sont pas inclus.

L'algorithme `BlocMaxIDyn` (algorithme 29) présente la procédure générale. Pour commencer l'algorithme sélectionne deux séquences (ligne 3). Une fois la sélection effectuée, il extrait les patrons maximaux de ces deux séquences (ligne 4). Ensuite, pour chaque séquence restante (lignes 5 - 10) un nouvel ensemble de patrons maximaux est calculé à partir de ceux précédemment calculés et de la séquence (ligne 9). Finalement, il factorise les blocs associés à ces patrons maximaux (ligne 11).

Algorithme 29 – `BlocMaxIDyn`

Entrée : S ensemble de séquences
Sortie : PM ensemble des blocs à patrons maximaux sur S
Données : s, s_1 et s_2 séquences
Données : S' ensemble de séquences

```
1  begin
2      PM ← ∅;
3      (s1, s2) ← paire_sequences(S);
4      PM ← patrons_maximaux_2_patrons(s1, s2);
5      S ← S\{s1, s2};
6      S' ← {s1, s2} ;
7      foreach s ∈ S do
8          S' ← S' ∪ {s} ;
9          PM ← patrons_maximaux(s, S', PM);
10     end
11     (BMI, _) ← factorisation_blocs_maximaux(PM, ∅) ;
12     return BMI ;
13 end
```

L'algorithme `patrons_maximaux_2_patrons` (algorithme 30) calcule les patrons maximaux entre deux séquences. Pour ce faire, il considère l'ensemble des décalages possibles entre

les deux séquences (ligne 2) et détermine pour chaque décalage les patrons maximaux associés à chacun (ligne 3). Finalement, il ne conserve uniquement que les patrons non inclus dans les autres (ligne 5).

L'algorithme `blocs_i_maximaux_candidats` est similaire à celui employé dans l'algorithme `BlocMaxE` de la section précédente. La seule différence est que les blocs y sont remplacés par des patrons et les ensembles B et C (passés en paramètres) sont identiques, ce qui a pour effet de ne conserver que les patrons maximaux de B. L'algorithme `factorisation_blocs_maximaux` est appelé avec des patrons et l'ensemble des blocs maximaux candidats est fixé à $\varnothing$, ce qui restreint l'exécution à la première boucle qui calcule les enracinements des blocs maximaux pour l'inclusion.

Algorithme 30 – `patrons_maximaux_2_patrons`

Entrée : s_1 séquence ou patron, s_2 séquence
Sortie : PM ensemble des patrons maximaux sur $\{s_1, s_2\}$
1 $PM \leftarrow \varnothing$;
2 **foreach** $(o_1, o_2) \in [\![|s_1|]\!] \times [\![|s_2|]\!]$ **do**
3 $\quad PM \leftarrow PM \cup$ `patrons_maximaux_decalage`(s_1, o_1, s_2, o_2) ;
4 **end**
5 $PM \leftarrow$ `blocs_i_maximaux_candidats`(PM, PM) ;
6 **return** PM ;

L'algorithme `patrons_maximaux` (algorithme 31) présente comment sont obtenus les patrons maximaux entre une séquence et un ensemble de patrons. Deux cas peuvent survenir : soit le patron est inclus dans la séquence (ligne 4) et donc il appartient aux patrons maximaux, soit il faut extraire les patrons maximaux entre le patron et la séquence (ligne 5). La procédure `patrons_maximaux_2_patrons` est réutilisée dans ce cas.

Algorithme 31 – `patrons_maximaux`

Entrée : s séquence, PM ensemble de patrons
Sortie : PM' patrons maximaux entre s et les patrons de PM
Donnée : p patron
1 **begin**
2 $\quad PM' \leftarrow \varnothing$;
3 $\quad$ **foreach** $p \in PM$ **do**
4 $\quad\quad$ **if** $p \leqslant s$ **then** $PM' \leftarrow PM' \cup \{p\}$;
5 $\quad\quad$ **else** $PM' \leftarrow PM' \cup$ `patrons_maximaux_2_patrons`(p, s) ;
6 $\quad$ **end**
7 $\quad PM' \leftarrow$ `supprimer_doublons`(PM') ;
8 $\quad PM' \leftarrow$ `blocs_i_maximaux_candidats`(PM', PM') ;
9 $\quad$ **return** PM' ;
10 **end**

L'algorithme `patrons_maixmaux_decalage` (algorithme 32) présente comment calculer

un ensemble de patrons maximaux à partir de deux patrons. Premièrement, il détermine la longueur maximale sur laquelle les deux patrons peuvent être comparés (ligne 2). Puis l'algorithme parcourt, sur cette longueur, les caractères des patrons, en veillant à prendre en compte les décalages respectifs o_1 et o_2 (lignes 7 - 8). Si les caractères concordent pour les deux patrons (ligne 8), alors deux cas sont possibles : soit un patron était déjà en cours de construction, soit cela dénote le départ d'un nouveau patron (ligne 9). Dans les deux cas, la longueur du patron est incrémentée (ligne 13), la localisation du dernier caractère solide est modifiée et le nombre de jokers consécutifs est réinitialisé à 0. Dans le cas où un nouveau patron est détecté, alors nous le signalons par la mise à $true$ de la variable $correspondance$.

Si les caractères ne coïncident pas, alors l'écart entre le dernier caractère solide et l'hypothétique prochain caractère solide est incrémenté. Si cet écart dépasse l'écart maximal autorisé, alors nous signalons la fin de la construction du patron par la mise à $false$ de la variable $correspondance$ (ligne 21).

Finalement, si le patron n'est plus en construction (soit parce qu'un patron a été parcouru entièrement, soit à cause d'un écart trop élevé entre deux caractères solides), alors celui-ci est enregistré dans le cas où il est valide (lignes 24 - 28), c'est-à-dire s'il contient au moins deux

caractères solides.

Algorithme 32 – `patrons_maximaux_decalage`

Entrée : p_1 et p_2 patrons, o_1 et o_2 entiers strictement positifs, g_{max} écart maximal
Sortie : P ensemble de patrons
Données : $correspondance$ booléen indiquant une région détectée, l_{max} longueur de chevauchement entre p_1 et p_2 pour un décalage, d décalage, p patron, deb_i et fin_i entiers délimitant une région, l longueur d'une région, g écart, $solide$ localisation

```
1  begin
2      l_max ← min(|p1| − o1, |p2| − o2) ;
3      P ← ∅ ;
4      if l_max < 1 then return P;
5      (solide, l) ← (0, 0) ;
6      (correspondance, g) ← (false, 0) ;
7      foreach d ∈ 0...l_max do
8          if p1[o1 + d] = p2[o2 + d] ∧ p1[o1 + d] ≠ # then
9              if ¬correspondance then
10                 correspondance ← true ;
11                 l ← 0 ;
12             end
13             l ← l + 1 ;
14             g ← 0 ;
15             solide ← l ;
16         else if correspondance ∧ g < g_max
17          then
18             g ← g + 1 ;
19             l ← l + 1 ;
20         else
21             correspondance ← false ;
22         end
23         if (¬correspondance ∨ d = l_max) ∧ solide > 1 then
24             (deb1, fin1) ← (o1 + l − solide, o1 + l);
25             (deb2, fin2) ← (o2 + l − solide, o2 + l);
26             p ← creer_patron(p1[deb1..fin1], p2[deb2..fin2]) ;
27             P ← P ∪ {p};
28             (l, solide) ← (0, 0) ;
29         end
30     end
31     return P ;
32 end
```

L'algorithme `creer_patron` (algorithme 33) détaille la génération d'un patron après détermination d'une fenêtre maximale de caractères contiguës. Eu égard au fait que l'algorithme s'applique à deux patrons, il s'applique également aux séquences. Ici, contrairement à la fusion de deux blocs dans l'algorithme `BlocMaxE` les caractères jokers sont favorisés par rapport aux

caractères solides.

Algorithme 33 – `creer_patron`

Entrée : s_1, s_2 séquences ou patrons
Condition : $|s_1| = |s_2| \wedge |s_1| \geqslant 2 \wedge s_1[1] = s_2[1] \wedge s_1[|s_1|] = s_2[|s_2|]$
Sortie : p patron
Donnée : i entier

```
1 begin
2     p ← ε ;                                   ▷ ε représente la chaîne vide
3     for i ∈ [[|s1|]] do
4         if s1[i] = s2[i] then p ← concatener(p, s1[i]) ;
5         else p ← concatener(p, #) ;
6     end
7     return p ;
8 end
```

5.4.1.1 Complexité

Comme dans le cas de `BlocMaxE` nous utilisons ici les symboles L, e et M qui correspondent respectivement à la longueur de la plus grande séquence, l'écart maximal considéré et le nombre maximum de patrons maximaux considérés lors d'une itération.

5.4.1.2 Création patron (algorithme 33)

La complexité en temps de l'algorithme `creer_patron` est linéaire en fonction de la plus grande séquence donc en $O(L)$.

5.4.1.3 Patrons maximaux entre deux patrons pour un décalage donné (algorithme 32)

L'algorithme `patrons_maximaux_decalage` génère l'ensemble des patrons en une seule passe. Cette dernière considère au plus L positions (ligne 6). La création des patrons s'effectue en temps linéaire, néanmoins un patron peut être construit au fur et à mesure du parcours des caractères donc la création peut être supposée de coût nul (ligne 26). Cette phase génère au plus L patrons. La comparaison entre deux patrons s'effectue en temps linéaire en fonction de leur taille, donc l'insertion (ligne 27) s'effectue en $O(L.log_2(L))$. La complexité de l'algorithme est donc de $O(L^2.log_2(L))$.

5.4.1.4 Patrons maximaux entre deux patrons (algorithme 30)

Nous n'allons pas utiliser la complexité de l'algorithme `patrons_maximaux_decalage`, mais nous allons supposer que celui-ci est directement intégrer au code de l'algorithme

`patrons_maximaux_2_patrons`. En raisonnant ainsi, nous savons que nous devons considérer L^2 décalages entre les deux patrons (leur taille est majorée par L). En effet, pour chacun de ces décalages le nombre de caractères à considérer varie et il y a au plus L^2 comparaisons à réaliser [14]. S'il y a au plus L^2 comparaisons, alors il y a au plus L^2 patrons à générer et la complexité de l'insertion de l'algorithme `patrons_maximaux_decalage` devient alors $O(L.L^2.log_2(L^2)) = O(L^3.log_2(L))$ (il s'agit du produit de l'insertion $L^2.log_2(L^2)$ et du coût de comparaison L). La suppression des non-inclus revient à tester ces L^2 patrons entre eux et à les tester pour l'inclusion (qui s'effectue en $O(L^2)$), donc la complexité de suppression des non-inclus est en $O(L^2.L^4)$. Nous en déduisons que la complexité de cet algorithme est de $O(L^6)$.

5.4.1.5 Patrons maximaux entre une séquence et un ensemble de patrons (algorithme 31)

La complexité de la boucle de l'algorithme `patrons_maximaux` est donnée par le produit du nombre maximum de patrons M et le maximum entre du coût d'extraction des patrons maximaux entre deux patrons et du test d'inclusion (lignes 3-6). Ces deux opérations sont respectivement en $O(L^2)$ et $O(L^6)$, la complexité de la boucle est donc de $O(L^6.M)$. Le nombre de patrons de PM' est majoré par M plus le nombre de nouveaux potentiellement générés qui sont au nombre de $M.L^2$ (produit du nombre de patrons de la précédente itération et du nombre de patrons générés par alignement de deux patrons). Donc, l'insertion (ligne 5) de L^2 nouveaux patrons dans un tel ensemble s'effectue en $O(L^2.L.log_2(M.L^2 + L^2)) = O(L^3.log_2(M.L))$. La complexité de la boucle (ligne 3-6) est alors de $O(L^6.M)$. Comme nous avons au plus $M.L^2$, la complexité du test d'inclusion est de $O(M^2.L^4)$ (voir complexité de `blocs_i_maximaux_candidats`). Nous en déduisons que la complexité de cet algorithme est de $O(L^4.M^2 + L^6.M)$.

5.4.1.6 Factorisation des patrons maximaux (algorithme 29)

L'appel de `factorisation_blocs_maximaux` annule de fait la seconde boucle de cet algorithme qui calcule les patrons maximaux pour l'enracinement. Sa première boucle détermine seulement des enracinements pour un ensemble de patrons sur $|S|$ séquences de taille au plus L. La complexité l'appel à `factorisation_blocs_maximaux` est donc de $O(M.|S|.L^2)$

14. Pour un décalage de 1 vers la "droite" ou vers la "gauche" entre deux séquences de même taille nous perdons une comparaison par rapport à l'absence de décalage. Le nombre de comparaisons à effectuer pour tous les décalages est de $\sum_{i=0}^{L-2} i < L^2$.

5.4.1.7 `BlocMaxIDyn`

L'algorithme `BlocMaxIDyn` a pour complexité la somme de la complexité de la boucle principale (lignes 7-10) et de l'étape de factorisation. La complexité de la boucle domine, trivialement, celle du calcul des patrons maximaux entre deux séquences. En outre, sa complexité est de $O(|S|.(L^4.M^2 + L^6))$ (nombre d'itérations multiplié par la complexité d'une itération) et celle de la factorisation de $O(M.|S|.L^2)$, donc nous en déduisons que la complexité de l'algorithme est de $O(|S|.L^2(L^2.M^2 + L^4))$ qui est la complexité de la boucle principale.

5.4.2 Expérimentations

Nous présentons ici une comparaison expérimentale des algorithmes BlocMaxE et BlocMaxIDyn sur les jeux de données biologiques LEAP et SHSP décrits en section 3.3.2 du chapitre 3. Pour rappel, chacun de ces jeux est prépartionné en classes de séquences : 13 pour LEAP et 25 pour SHSP. L'objectif visé est, pour chaque classe, de calculer les ensembles de blocs maximaux et de blocs à patrons maximaux pour différents seuil d'écart maximal. Chaque instance de test se définit donc par la donnée d'une classe et d'une valeur de seuil. Nous rapportons dans différents tableaux les résultats relatifs à :

— l'exécution de BlocMaxE pour le calcul des blocs maximaux pour l'enracinement sur toutes les classes pour une plage réduite d'écart maximal (tableau 5.3) ;

— l'exécution de BlocMaxE et BlocMaxIDyn pour le calcul de blocs maximaux pour l'inclusion sur toutes les classes pour une plage réduite d'écart maximal (tableau 5.4) ;

— l'exécution de BlocMaxE et BlocMaxIDyn sur deux classes choisies pour certaines valeurs d'écart maximales possibles (tableau 5.5).

Les résultats sont basés sur une implantation en C++ de BlocMaxE et BlocMaxIDyn et des exécutions sur architecture constituée d'un processeur intel core i5 de quatre cœurs (1.2GHz) et 8Go de RAM.

Le tableau 5.3 consigne chaque instance (nombre de séquences, taille des séquences, etc.) et les résultats obtenus (nombre de patrons extraits, nombre de localisations extraites, etc.). La sémantique des colonnes est la suivante :

— Classe : la classe de protéine de l'instance ;

— écart : l'écart maximal autorisé ;

— S^+ : le nombre de séquences positives ;

— S^- : le nombre de séquences négatives ;

— $\bar{s^+}$: la taille moyenne des séquences positives ;

— $\bar{s^-}$: la taille moyenne des séquences négatives ;

— $s^+_\vee$: la taille minimale des séquences positives ;

— $s^-_\vee$: la taille minimale des séquences négatives ;

— $s^+_\wedge$: la taille maximale des séquences positives ;

— $s^-_\wedge$: la taille maximale des séquences négatives ;

— $|BME(S)|$: le nombre de patrons associés aux blocs maximaux pour l'enracinement extrait ;

— $\bar{E}(p, s^+)$: le nombre moyen de localisations des patrons sur les séquences positives ;

— $\bar{E}(p, s^-)$: le nombre moyen de localisations des patrons sur les séquences négatives ;

— $E_\vee(p, s^+)$: le nombre minimum de localisations des patrons sur les séquences positives ;

— $E_\vee(p, s^-)$: le nombre minimum de localisations des patrons sur les séquences négatives ;

— $E_\wedge(p, s^+)$: le nombre maximum de localisations des patrons sur les séquences positives ;

— $E_\wedge(p, s^-)$: le nombre maximum de localisations des patrons sur les séquences négatives ;

— BlocMaxE (ms) : le temps moyen d'exécution en milliseconde de l'algorithme BlocMaxE.

Le tableau 5.4 consigne chaque instance (nombre de séquences, taille des séquences, etc.) et les résultats obtenus (nombre de patrons extraits, nombre de localisations extraites, etc.). La sémantique des colonnes différentes du tableau précédent est la suivante :

— Classe : la classe de protéine de l'instance ;

— écart : l'écart maximal autorisé ;

— $|PMI(S)|$: le nombre de patrons maximaux pour l'inclusion extrait ;

— $\bar{E}(p, s^+)$: le nombre moyen de localisations des patrons sur les séquences positives ;

— $\bar{E}(p, s^-)$: le nombre moyen de localisations des patrons sur les séquences négatives ;

— $E_\vee(p, s^+)$: le nombre minimum de localisations des patrons sur les séquences positives ;

— $E_\vee(p, s^-)$: le nombre minimum de localisations des patrons sur les séquences négatives ;

— $E_\wedge(p, s^+)$: le nombre maximum de localisations des patrons sur les séquences positives ;

— $E_\wedge(p, s^-)$: le nombre maximum de localisations des patrons sur les séquences négatives ;

— BlocMaxE (ms) : le temps moyen d'exécution en milliseconde de l'algorithme BlocMaxE ;

— BlocMaxIDyn (ms) : le temps moyen d'exécution en milliseconde de l'algorithme BlocMaxIDyn.

Sur les instances choisies l'extraction des blocs s'effectue en moins d'une seconde. Sur ces classes de protéines les deux méthodes restent rapides. En outre, l'approche dynamique n'apporte aucun gain en termes de performance pour l'extraction des patrons à blocs maximaux. Une grande partie du temps de calcul est alloué à l'extraction des localisations des patrons des blocs maximaux.

En terme de patrons différents obtenus nous pouvons constater que le calcul de blocs maximaux apportent peu de nous patrons par rapport au calcul de patrons maximaux et nécessitent un temps de calcul plus élevé.

	Classe	écart	$\lvert S^+\rvert$	$s^+_\perp$	$s^+_\vee$	$s^+_\wedge$	$\lvert S^-\rvert$	$s^-_\perp$	$s^-_\vee$	$s^-_\wedge$	$\lvert BM(S)\rvert$	$\bar{E}(p,s^+)$	$E_\vee(p,s^+)$	$E_\wedge(p,s^+)$	$\bar{E}(p,s^-)$	$E_\vee(p,s^-)$	$E_\wedge(p,s^-)$	BME (ms)
LEAP	1	10	208	190.4	117	507	1163	199.2	66	843	20	3.2	1	60	1.5	0	43	43
	1	16	208	190.4	117	507	1163	199.2	66	843	29	3.2	1	60	1.4	0	43	83
	2	8	92	230.0	140	338	1279	195.6	66	843	26	4.3	1	26	1.2	0	38	157
	2	15	92	230.0	140	338	1279	195.6	66	843	39	4.2	1	26	1.2	0	38	227
	2	20	92	230.0	140	338	1279	195.6	66	843	46	4.1	1	26	1.1	0	38	256
	3	0	34	121.3	86	186	1337	199.9	66	843	9	3.5	1	24	1.4	0	32	9
	5	2	66	108.0	83	217	1305	202.5	66	843	11	2.7	1	12	1.3	0	65	11
	10	8	78	144.4	88	173	1293	201.1	66	843	22	3.1	1	8	2.0	0	53	33
	10	15	78	144.4	88	173	1293	201.1	66	843	31	2.8	1	8	1.9	0	79	54
	10	20	78	144.4	88	173	1293	201.1	66	843	39	2.8	1	8	1.8	0	79	76
	11	1	35	249.0	159	288	1336	196.6	66	843	22	2.9	1	10	1.0	0	42	21
	12	0	17	94.3	71	117	1354	199.2	66	843	5	1.0	1	2	0.6	0	19	3
SHSP	2	6	109	156.2	129	175	3656	172.2	65	498	14	2.0	1	5	0.9	0	15	30
	3	2	64	197.7	165	328	3701	171.3	65	498	13	2.2	1	8	0.8	0	15	25
	4	0	23	142.8	119	211	3742	171.9	65	498	10	1.7	1	11	0.8	0	14	19
	5	0	25	194.2	172	238	3740	171.6	65	498	12	1.6	1	6	0.6	0	8	20
	7	10	100	234.1	157	266	3665	170.1	65	498	11	1.8	1	6	0.9	0	15	22
	8	1	30	143.2	115	178	3735	172.0	65	498	15	16	1	4	0.9	0	9	29
	9	5	238	134.1	100	175	3527	174.3	65	498	7	2.5	1	7	0.8	0	14	14
	13	3	156	210.8	103	271	3609	170.1	65	498	11	2.2	1	6	0.6	0	9	21
	14	0	84	152	141	187	3681	172.2	65	498	20	1.4	1	5	0.6	0	10	37
	15	0	26	176.4	108	262	3739	171.7	65	498	20	1.6	1	5	0.5	0	14	33
	16	9	75	146.0	114	230	3690	172.3	65	498	6	1.4	1	5	0.4	0	6	10
	17	3	88	106.6	102	143	3677	173.3	65	498	9	1.5	1	3	0.7	0	11	17
	19	1	23	210.8	194	220	3742	171.5	65	498	25	2.1	1	8	0.9	0	17	49
	21	2	131	294.8	161	344	3634	167.3	65	498	6	2.4	1	10	0.4	0	9	10
	22	0	15	378.5	316	453	3750	170.9	65	498	37	2.6	1	9	0.7	0	18	70
	23	4	60	97.8	65	144	3705	172.9	74	498	6	1.7	1	5	0.5	0	6	11
	24	2	107	161.4	134	208	3658	172.1	65	498	7	1.7	1	6	0.4	0	9	12
	25	2	57	167.1	146	277	3708	171.8	65	498	17	2.3	1	7	0.7	0	7	18

TABLEAU 5.3 – Blocs maximaux

Classe	écart	$\|PM(S)\|$	$\bar{E}(p,s^+)$	$E_\vee(p,s^+)$	$E_\wedge(p,s^+)$	$\bar{E}(p,s^-)$	$E_\vee(p,s^-)$	$E_\wedge(p,s^-)$	BMI (ms)	BlocMaxIDyn (ms)
LEAP										
1	10	20	3.2	1	60	1.5	0	43	20	15
1	16	29	3.2	1	60	1.4	0	43	33	47
2	8	25	4.3	1	26	1.1	0	38	35	28
2	15	38	4.3	1	26	1.2	0	38	53	48
2	20	45	4.1	1	26	1.1	0	38	65	61
3	0	9	3.5	1	24	1.4	0	32	6	7
5	2	10	2.7	1	12	1.0	0	23	7	9
10	8	21	3.1	1	8	2.0	0	53	19	20
10	15	30	2.9	1	8	1.9	0	79	32	38
10	20	38	2.8	1	8	1.8	0	79	46	37
11	1	19	2.7	1	10	0.8	0	42	12	14
12	0	5	1.0	1	2	0.6	0	19	2	8
SHSP										
2	6	14	2.0	1	5	0.9	0	15	24	31
3	2	13	2.2	1	8	0.8	0	15	20	29
4	0	10	1.7	1	11	0.8	0	14	15	16
5	0	12	1.6	1	6	0.6	0	8	16	17
7	10	11	1.8	1	6	0.9	0	15	21	26
8	1	15	1.6	1	4	0.9	0	9	23	23
9	5	7	2.5	1	7	0.6	0	14	19	16
13	3	11	2.2	1	6	0.6	0	9	18	22
14	0	20	1.4	1	5	0.6	0	10	28	29
15	0	20	1.6	1	5	0.5	0	14	27	26
16	9	6	1.4	1	5	0.4	0	6	9	13
17	3	9	1.5	1	3	0.7	0	11	13	17
19	1	24	2.1	1	8	0.9	0	17	38	38
21	2	6	2.4	1	10	0.4	0	9	7	18
22	0	36	2.6	1	9	0.7	0	18	52	44
23	4	6	1.7	1	5	0.5	0	6	8	18
24	2	7	1.7	1	6	0.4	0	9	10	16
25	2	16	2.3	1	7	0.7	0	7	27	41

TABLEAU 5.4 – Blocs à patrons maximaux

Le tableau 5.5 reporte l'évolution du nombre de patrons maximums rencontrés lors de la résolution pour deux classes de SHSP (14 et 15). Pour certaines instances (SHSP 4 :5, SHSP 5 :3, SHSP :4 et SHSP :5) les résultats n'ont pas aboutis pour l'algorithme BlocMaxE, donc il ne s'agit pas nécessairement au nombre de patrons de la couche la plus large qui puisse existe. Dans ce tableau nous ne reportons que le nombre de patrons rencontrés.

Nous nous rendons compte que pour ces deux instances le nombre de patrons considérés

pour l'approche treillis augmente très rapidement et bien plus rapidement que l'approche par programmation dynamique. D'ailleurs, à partir de 4000 blocs à considérer pour la fusion l'approche par treillis ne retourne plus de solution dans un temps raisonnable. En outre pour la classe 4 de SHSP les blocs à patrons maximaux ne peuvent plus être obtenus dans un temps raisonnable lorsque l'écart maximal autorisée dépasse 4. Pour la classe 5, ils ne peuvent plus être obtenus à partir d'un écart maximal de 2.

	écart	BlocMaxE	BlocMaxIDyn
SHSP 4	0	43	33
	1	72	57
	2	102	91
	3	157	121
	4	569	142
	5	4304	162
SHSP 5	0	39	34
	1	48	64
	2	697	93
	3	4136	108
	4	16329	127
	5	66265	165

TABLEAU 5.5 – Évolution du nombre maximum de blocs rencontrés lors de la recherche en fonction de l'écart maximal

Conclusion générale

Résumé

Cette thèse présente différentes manières d'utiliser le paradigme de programmation par contraintes afin de l'appliquer à la caractérisation de données biologiques. À cet effet, nous avons proposé le cadre MMP pour exprimer divers problèmes de fouilles de données en utilisant des contraintes, à l'instar du cadre CSP. D'un point de vue pratique, nous avons proposé divers modèles pour résoudre des problèmes de recherche de sous-séquences fréquentes ou d'ordres partiels communs au sein de jeux de séquences. Les limites de nos modèles dédiés à l'extraction d'ordres partiels, nous ont conduit à développer un algorithme mémétique. Enfin, parallèlement à cela, nous avons développé un pré-traitement de séquences protéiques, sous forme de séquences d'acides aminés, afin d'en extraire des patrons.

Dans le chapitre 2, nous avons défini le cadre MMP dont l'objectif est d'exprimer divers problèmes de fouilles de données sous forme matricielle. Nous suggérons également un ensemble de contraintes sur matrices facilitant la définition de problèmes. Parmi ces contraintes, nous en signalons particulièrement une qui lie instance de problèmes et matrices du MMP. D'autres contraintes servent à restreindre les dimensions ou les cellules des matrices solutions. À partir des différentes contraintes proposées, nous caractérisons trois problèmes de fouilles de données avec le formalisme MMP : la recherche d'itemsets fréquents, la recherche de sous-séquences fréquentes et la recherche d'ordres partiels. Pour le problème de recherche d'itemsets fréquents et la recherche de sous-séquences fréquentes nous fournissons une réduction de ces instances de MMP au problème de recouvrement par ensembles.

Dans le chapitre 3, nous définissons différents modèles de programmation par contraintes pour résoudre plusieurs instances de MMP ; instances classées en deux catégories : la recherche de sous-séquences fréquentes et la recherche d'ordre partiels communs. Ces deux problèmes travaillent sur un ou deux ensembles de séquences : un de séquences dites positives et un autre de séquences négatives. La solution à ces problèmes consiste alors à trouver des motifs (sous-séquence ou ordre partiel basés sur la localisation des caractères) fréquents dans les (ou communs aux) séquences positives et absents dans les séquences négatives. Néanmoins, d'un point

de vue pratique, l'absence n'est pas systématiquement envisageable. Afin de résoudre cela, une variante d'optimisation visant à minimiser la couverture des séquences négatives lui a été préférée. Pour ces deux problèmes (sous-séquences et ordres partiels), nous avons élaboré deux contraintes globales dédiées au test de couverture des motifs extraits à partir des séquences positives, sur les séquences négatives. D'un point de vue expérimental, nous avons attesté de la pertinence d'employer de telles contraintes globales. En effet, elles limitent le nombre de propagateurs déclenchés lors de la résolution, déduisant donc plus rapidement la couverture des séquences négatives. Par ailleurs, ces contraintes globales réduisent le nombre de variables de décisions à définir et rendent ainsi les modèles plus maniables. Pour autant, le problème d'ordre partiel que nous avons traité étant peu structuré par nature (i.e. l'absence d'un arc entre deux nœuds n'implique pas l'impossibilité d'un arc entre ceux-ci), la recherche arborescente sous-jacente à la résolution par PPC est inapte à éliminer de nombreux nœuds. Afin d'obtenir tout de même des solutions satisfaisantes à ce problème, nous avons élaboré un algorithme mémétique qui arbore des solutions approchées, de meilleure qualité que la résolution par PPC pure.

Cet algorithme mémétique dédié au calcul d'ordre partiel est présenté en chapitre 4. Celui-ci est construit autour du modèle de programmation par contraintes développé pour l'extraction d'ordres partiels. L'utilisation du modèle PPC sert à la fois à la recherche de solutions certifiées (qui respectent la contrainte d'ordre partiel), mais également à l'étape d'intensification propre aux algorithmes mémétiques. D'un point de vue expérimental, cette approche s'est montrée plus robuste que les stratégies de branchement et de sélection, aléatoires pour la résolution du modèle PPC pur.

Le chapitre 5, évoque la genèse des instances employées dans les chapitres précédents. À cet égard, deux algorithmes sont mis en exergue pour l'extraction de patrons (avec caractères jokers) communs à un ensemble de séquences. Nous avons également rectifié une méthode que nous avions proposé dans [33] et [59]. Les deux algorithmes reposent sur des approches profondément différentes pour résoudre un même problème. L'un de ces algorithmes s'inspire de l'approche par treillis d'APriori et l'autre s'appuie sur la programmation dynamique. D'un point de vue expérimental, ces deux algorithmes sont très proches en terme de temps de calcul. Cependant, celui basé sur la construction d'un treillis semble plus aisément sujet à des cas particuliers qui l'empêchent d'atteindre la solution en un temps raisonnable. Malgré cela, ce dernier reste plus général et autorise le calcul des deux formes de patrons proposées.

Perspectives

Les perspectives portent sur le MMP et la résolution de problème de fouilles de données par la programmation par contraintes. Nous n'évoquerons pas les algorithmes d'extraction qui ont servi à la création des instances et qui ont été intégrées dans une approche plus globale dans

les articles [35], [33], [34] et [59].

Premièrement, il pourrait être intéressant d'examiner plus en profondeur le problème de recherche d'ordre partiel, et notamment de l'étendre afin d'y substituer la contrainte imposant l'appartenance à toutes les séquences positives par un seuil de couverture minimale sur ces séquences positives. À cet égard, nous pouvons citer Morchen [46] qui traite le même genre de motif (ordre partiel) mais avec un seuil de fréquence. Relâcher ce seuil permettrait éventuellement d'extraire des ordres partiels couvrant moins de séquences positives, mais excluant plus de séquences négatives. Cela autoriserait aussi l'extraction d'ordres partiels peu fréquents qui pourraient potentiellement être employés dans des tâches de classification non-supervisée.

D'ailleurs, si nous voulions appliquer des méthodes de classification non-supervisée aux données biologiques (séquences protéiques) considérées lors de cette thèse, cela nécessiterait de modifier les algorithmes d'extraction de patrons (chapitre 5). En effet, actuellement, les algorithmes proposés pour le calcul de patrons supposent des classes de séquences connues a priori, et par définition ils ne considèrent que les patrons communs à toutes les séquences d'une classe. D'une part, pour la classification non-supervisée, il semble illusoire de forcer la présence de patrons sur toutes les séquences d'une jeu de séquences protéiques. D'autre part, autoriser des patrons avec une fréquence d'inclusion faible (enracinement) au sein des séquences risque d'accroître déraisonnablement le nombre de patrons maximaux.

Une première approche pour réaliser une procédure de classification supervisée sur ce jeu de séquences protéiques consisterait à : répartir aléatoirement les séquences de protéines entre différentes classes, puis extraire de chaque classe les patrons communs et maximaux. Si pour certaines classes aucun patron ne peut être extrait, alors il faut réarranger les classes (e.g., par des échange de séquences entre classes). Une fois que toutes les classes disposent de patrons communs, alors nous pourrions lancer une phase de calcul d'exclusivité basée sur un calcul d'ordres partiels communs. Si cette deuxième phase fournit des résultats peu satisfaisants, alors il faudrait de nouveaux réarranger les classes et donc recommencer à partir de l'extraction de patrons communs.

Finalement, un autre aspect qui pourrait être étudié plus en profondeur concerne le développement de propagateurs dédiés à certains problèmes de fouilles de données qui pourraient s'inspirer de méthodes déjà existantes. Mabroukeh dresse une taxonomie [36] des algorithmes travaillant sur données séquentielles et classe les différents algorithmes en fonction du type de recherche utilisée (arborescente en largeur d'abord, arborescente en profondeur d'abord, recherche arborescente hybride). Les méthodes à base de recherche arborescentes en profondeur d'abord ou hybrides semblent plus enclines à s'intégrer à des propagateurs. Par exemple, pour la recherche de sous-séquences fréquences les mécanismes de l'algorithme PLWAP [16] s'intègrent, avec quelques modifications, à un propagateur.

Liste des tableaux

1.1 Base de transactions sous forme matricielle 25
1.2 Base de séquences 28
1.3 Représentation graphique de la séquence $< a(abc)b(ac) >$. Nous emploierons dorénavant le terme événement (colonne i) pour désigner les caractères. Les lignes représentent les événements et les colonnes représentent la chronologie dans laquelle les événement se sont déroulés. Une cellule de colonne t_i et de ligne e_j du tableau prend la valeur 1 si et seulement si l'événement e_j a lieu au temps t_i. Deux événements peuvent survenir au même instant. 30
1.4 Représentation graphique de la séquence $< a(abc)b(ac) >$ après une projection sur a. 30
1.5 Représentation graphique de la séquence $< a(abc)b(ac) >$ après une projection sur a. 30
1.6 Résumé méthodes de classification 36

2.1 Notations associées à une fonction $f : A \rightarrow B$ et $X \subseteq A$. 41

3.1 Résultats pour l'exclusion 96
3.2 Utilisation des contraintes globales 98

4.1 Comparaisons expérimentales pour le score d'exclusion induit par un ordre partiel. 111

5.1 Synthèse 119
5.2 Base de transactions $\mathcal{T}$ 122
5.3 Blocs maximaux 147
5.4 Blocs à patrons maximaux 148
5.5 Évolution du nombre maximum de blocs rencontrés lors de la recherche en fonction de l'écart maximal 149

Table des figures

1.1 Modèle PPC associé à l'exemple 1. 10
1.2 Arbre développé par la méthode *générer-et-tester* pour le CSP de l'exemple 1 13
1.3 Arbre développé par la méthode *tester-et-générer* pour le CSP de l'exemple 1 13
1.4 Arbre de recherche développé par maintien d'arc-consistance sur le CSP de l'exemple 1. 15
1.5 Branchement sur x_1 avec la valeur 1 d'abord. 17
1.6 Branchement sur x_1 avec la valeur 2 d'abord. 17
1.7 Déroulement du propagateur de `all_different`. 20
1.8 Utilisation de la contrainte `regular` 22
1.9 Construction du graphe interne de la contrainte `regular` (étape 1 à 5) 23
1.10 Construction du graphe interne de la contrainte `regular` (étape 7 à 8) 23
1.11 Réduction du réseau après modification du domaine de x_4 24
1.12 Automate fourni à la contrainte `regular` 24
1.13 Méthode Métivier sur un jeu de deux séquences 31
1.14 Grille de mots-croisés 33
1.15 Modèle PPC pour la résolution d'une grille de mots-croisés 34

2.1 Base de données matricielle pour un problème de recherche d'itemsets. 46
2.2 Base de données matricielle pour un problème de recherche de motifs sans répétitions. 47
2.3 Base de données matricielle pour un problème de recherche de motifs avec répétitions. 48
2.4 Base de données matricielle pour un problème de recherche de motifs à base de patrons. 49
2.5 Une base de séquences constituée des positions de départ de différents patrons. 50

3.1 Sous-séquences fréquentes sans répétition de caractères 71
3.2 next_greater_than 72
3.3 Heuristique de branchement 73

3.4 Sous-séquence fréquente avec répétition (préambule) 74
3.5 Sous-séquence fréquente avec répétition (données et variables) 75
3.6 Sous-séquence fréquente avec répétition (contraintes) 76
3.7 Sous-séquence fréquente avec répétition (branchement) 76
3.8 Sous-séquence fréquente avec répétition (pseudo-fermeture) 77
3.9 Ordre Total Exclusion 78
3.10 Ordre Partiel (Données) 84
3.11 Ordre Partiel (Variables) 84
3.12 Ordre Partiel (Contraintes) 85
3.13 Extraction des patrons à partir de séquences. 93
3.14 Construction d'un ordre partiel à partir de patrons 94

4.1 Schéma général d'un algorithme génétique 103
4.2 Ensemble de graphes G^* résultant d'un croisement (G_1, G_2). 107

5.1 Treillis d'itemsets 123

Bibliographie

[1] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD Record*, volume 22, pages 207–216. ACM, 1993. (pages 24 et 121).

[2] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994. (page 26).

[3] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global constraint catalog, (revision a), 2012. (pages 18 et 72).

[4] Christian Bessiere. Arc-consistency and arc-consistency again. *Artificial intelligence*, 65(1) :179–190, 1994. (page 14).

[5] Tobias Blickle and Lothar Thiele. A comparison of selection schemes used in genetic algorithms, 1995. (page 102).

[6] Douglas Burdick, Manuel Calimlim, and Johannes Gehrke. Mafia : A maximal frequent itemset algorithm for transactional databases. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 443–452. IEEE, 2001. (page 121).

[7] Emmanuel Coquery, Said Jabbour, Lakhdar Sais, and Yakoub Salhi. A SAT-Based approach for discovering frequent, closed and maximal patterns in a sequence. In *ECAI*, pages 258–263, 2012. (page 29).

[8] Emmanuel Coquery, Said Jabbour, and Lakhdar Saïs. A constraint programming approach for enumerating motifs in a sequence. In *2011 IEEE 11th International Conference on Data Mining Workshops (ICDMW)*, pages 1091–1097, 2011. (page 29).

[9] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007. (page 36).

[10] Lawrence Davis. Handbook of genetic algorithms. 1991. (page 102).

[11] Kenneth Alan De Jong. Analysis of the behavior of a class of genetic adaptive systems. 1975. (page 102).

[12] Luc De Raedt, Tias Guns, and Siegfried Nijssen. Constraint programming for itemset mining. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 204–212. ACM, 2008. (page 26).

[13] Luc De Raedt, Tias Guns, and Siegfried Nijssen. Constraint programming for data mining and machine learning. In *AAAI*, 2010. (page 26).

[14] Khanh-Chuong Duong, Christel Vrain, et al. A declarative framework for constrained clustering. In *Machine Learning and Knowledge Discovery in Databases*, pages 419–434. Springer, 2013. (page 34).

[15] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*, volume 53. Springer, 2003. (page 102).

[16] Christie I Ezeife and Yi Lu. Mining web log sequential patterns with position coded pre-order linked wap-tree. *Data Mining and Knowledge Discovery*, 10(1) :5–38, 2005. (pages 30 et 153).

[17] Keith Golden and Wanlin Pang. Constraint reasoning over strings. In *Principles and Practice of Constraint Programming–CP 2003*, pages 377–391. Springer, 2003. (page 32).

[18] Solomon W Golomb and Leonard D Baumert. Backtrack programming. *Journal of the ACM (JACM)*, 12(4) :516–524, 1965. (page 11).

[19] Karam Gouda and Mohammed J Zaki. Genmax : An efficient algorithm for mining maximal frequent itemsets. *Data Mining and Knowledge Discovery*, 11(3) :223–242, 2005. (page 121).

[20] Tias Guns, Siegfried Nijssen, and Luc De Raedt. Itemset mining : A constraint programming perspective. *Artificial Intelligence*, 175(12–13) :1951–1983, August 2011. (page 26).

[21] Tias Guns, Siegfried Nijssen, and Luc De Raedt. k-pattern set mining under constraints. *IEEE Transactions on Knowledge and Data Engineering*, 25(2) :402–418, 2013. (page 27).

[22] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Freespan : frequent pattern-projected sequential pattern mining. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 355–359. ACM, 2000. (page 29).

[23] Jin-Kao Hao. Memetic algorithms in discrete optimization. *Handbook of memetic algorithms*, pages 73–94, 2012. (page 102).

[24] John Henry Holland. *Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992. (page 102).

[25] Gilles Hunault and Emmanuel Jaspard. LEAPdb : a database for the late embryogenesis abundant proteins. *BMC Genomics*, 11(1) :221, April 2010. PMID : 20359361. (page 93).

[26] Gilles Hunault and Emmanuel Jaspard. The late embryogenesis abundant proteins DataBase. `http://forge.info.univ-angers.fr/~gh/Leadb/index.php`, 2013. (page 93).

[27] Gilles Hunault and Emmanuel Jaspard. The small heat shock proteins database. sHSPdb. `http://forge.info.univ-angers.fr/~gh/Shspdb/index.php`, 2013. (page 93).

[28] Emmanuel Jaspard, David Macherel, and Gilles Hunault. Computational and statistical analyses of amino acid usage and physico-chemical properties of the twelve late embryogenesis abundant protein classes. *PLoS ONE*, 7(5) :e36968, May 2012. (page 93).

[29] Amina Kemmar, Samir Loudni, Yahia Lebbah, Patrice Boizumault, and Thierry Charnois. Prefix-projection global constraint for sequential pattern mining. *arXiv preprint arXiv :1504.07877*, 2015. (pages 32 et 69).

[30] Amina Kemmar, Willy Ugarte, Samir Loudni, Thierry Charnois, Yahia Lebbah, Patrice Boizumault, and Bruno Cremilleux. Mining relevant sequence patterns with cp-based framework. In *2014 IEEE 26th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 552–559. IEEE, 2014. (pages 28 et 32).

[31] Mehdi Khiari, Patrice Boizumault, and Bruno Crémilleux. Constraint programming for mining n-ary patterns. In *Principles and Practice of Constraint Programming–CP 2010*, pages 552–567. Springer, 2010. (page 35).

[32] Jena-Lonis Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial intelligence*, 10(1) :29–127, 1978. (page 18).

[33] David Lesaint, Deepak Mehta, Barry O'Sullivan, and Vincent Vigneron. A Decomposition Approach for Discovering Discriminative Motifs in a Sequence Database. In *21st European Conference on Artificial Intelligence (ECAI)*, Prague, Czech Republic, August 2014. (pages 120, 152 et 153).

[34] David Lesaint, Deepak Mehta, Barry O'Sullivan, and Vincent Vigneron. A Decomposition Approach for Discovering Discriminative Motifs in a Sequence Database. In *IEEE International Conference on Tools with Artificial Intelligence (ICTAI) - Special Track on SAT and CSP technologies*, pages 544–551, Limassol, Cyprus, November 2014. IEEE, IEEE Computer Society. (page 153).

[35] David Lesaint, Deepak Mehta, and Barry O'Sullivan. Soft pattern discovery in preclassified protein families through constraint optimization. In *Proceedings of WCB13 Workshop on Constraint Based Methods for Bioinformatics*, page 47, 2013. (page 153).

[36] Nizar R Mabroukeh and Christie I Ezeife. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys (CSUR)*, 43(1) :3, 2010. (page 153).

[37] Alan K Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8(1) :99–118, 1977. (pages 8, 12 et 14).

[38] Alan K Mackworth. *On reading sketch maps*. Department of Computer Science, University of British Columbia, 1977. (page 12).

[39] Alan K Mackworth and Eugene C Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial intelligence*, 25(1) :65–74, 1985. (page 12).

[40] Kim Marriott, Peter J Stuckey, LD Koninck, and Horst Samulowitz. A minizinc tutorial, 2014. (page 10).

[41] Edward M McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2) :262–272, 1976. (page 37).

[42] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998. (page 102).

[43] Roger Mohr and Thomas C Henderson. Arc and path consistency revisited. *Artificial intelligence*, 28(2) :225–233, 1986. (page 14).

[44] Roger Mohr and Gérald Masini. Good old discrete relaxation. In *8th European Conference on Artificial Intelligence (ECAI'88)*, pages 651–656. Pitmann Publishing, 1988. (page 14).

[45] Ugo Montanari. Networks of constraints : Fundamental properties and applications to picture processing. *Information sciences*, 7 :95–132, 1974. (page 8).

[46] Fabian Mörchen and Dmitriy Fradkin. Robust mining of time intervals with semi-interval partial order patterns. In *Proceedings of the 2010 SIAM International Conference on Data Mining*, pages 315–326. SIAM, 2010. (page 153).

[47] Pablo Moscato and Carlos Cotta. A gentle introduction to memetic algorithms. In *Handbook of metaheuristics*, pages 105–144. Springer, 2003. (page 102).

[48] Pablo Moscato et al. On evolution, search, optimization, genetic algorithms and martial arts : Towards memetic algorithms. *Caltech concurrent computation program, C3P Report*, 826 :1989, 1989. (page 102).

[49] Jean-Philippe Métivier, Patrice Boizumault, Bruno Crémilleux, Mehdi Khiari, and Samir Loudni. A constraint language for declarative pattern discovery. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, page 119–125, New York, NY, USA, 2012. ACM. (page 35).

[50] Jean-Philippe Métivier, Samir Loudni, and Thierry Charnois. A constraint programming approach for mining sequential patterns in a sequence database. (page 35).

[51] Benjamin Negrevergne and Tias Guns. Constraint-based sequence mining using constraint programming. *arXiv preprint arXiv :1501.01178*, 2015. (pages 32 et 69).

[52] Siegfried Nijssen and Tias Guns. Integrating constraint programming and itemset mining. In José Luis Balcázar, Francesco Bonchi, Aristides Gionis, and Michèle Sebag, editors, *Machine Learning and Knowledge Discovery in Databases*, number 6322 in Lecture Notes in Computer Science, pages 467–482. Springer Berlin Heidelberg, January 2010. (page 26).

[53] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Prefixspan : Mining sequential patterns efficiently by prefix-projected pattern growth. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 0215–0215. IEEE Computer Society, 2001. (pages 29 et 69).

[54] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, and Hua Zhu. Mining access patterns efficiently from web logs. In *Knowledge Discovery and Data Mining. Current Issues and New Applications*, pages 396–407. Springer, 2000. (page 30).

[55] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *Principles and Practice of Constraint Programming–CP 2004*, pages 482–495. Springer, 2004. (page 22).

[56] Jean-Francois Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Aaai/Iaai*, pages 359–366, 1998. (page 18).

[57] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006. (page 10).

[58] J David Schaffer and Larry J Eshelman. On crossover as an evolutionarily viable strategy. In *ICGA*, volume 91, pages 61–68, 1991. (page 102).

[59] Vincent Vigneron, David Lesaint, Deepak Mehta, and Barry O'Sullivan. Une Approche par Décomposition pour la découverte de motifs sur données séquentielles. In *10èmes Journées Francophones de la Programmation par Contraintes (JFPC)*, Angers, France, June 2014. (pages 120, 152 et 153).

[60] Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973. (page 36).

[61] Yuanlin Zhang and Roland HC Yap. Making ac-3 an optimal algorithm. In *IJCAI*, volume 1, pages 316–321, 2001. (page 14).

[62] Zheng Zhang, Scott Schwartz, Lukas Wagner, and Webb Miller. A greedy algorithm for aligning dna sequences. *Journal of Computational biology*, 7(1-2) :203–214, 2000. (page 102).

UNIVERSITE BRETAGNE LOIRE

université angers

Thèse de Doctorat

Vincent VIGNERON

Programmation par contraintes et découverte de motifs sur données séquentielles

Constraint programming for sequential pattern mining

Résumé
Des travaux récents ont montré l'intérêt de la programmation par contraintes pour la fouille de données. Dans cette thèse, nous nous intéressons à la recherche de motifs sur séquences, et en particulier à la caractérisation, à l'aide de motifs, de classes de séquences pré-établies. Nous proposons à cet effet un langage de modélisation à base de contraintes qui suppose une représentation matricielle du jeu de séquences. Un motif s'y définit comme un ensemble de caractères (ou de patrons) et pour chacun une localisation dans différentes séquences. Diverses contraintes peuvent alors s'appliquer : validité des localisations, couverture d'une classe de séquences, ordre sur les localisations des caractères commun aux séquences, etc. Nous formulons deux problèmes de caractérisation NP-complets : la caractérisation par motif totalement ordonné (e.g. sous-séquence exclusive à une classe) ou partiellement ordonné. Nous en donnons deux modélisations CSP qui intègrent des contraintes globales pour la preuve d'exclusivité. Nous introduisons ensuite un algorithme mémétique pour l'extraction de motifs partiellement ordonnés qui s'appuie sur la résolution CSP lors des phases d'initialisation et d'intensification. Cette approche hybride se révèle plus performante que l'approche CSP pure sur des séquences biologiques. La mise en forme matricielle de jeux de séquences basée sur une localisation des caractères peut être de taille rédhibitoire. Nous proposons donc de localiser des patrons plutôt que des caractères. Nous présentons deux méthodes ad-hoc, l'une basée sur un parcours de treillis et l'autre sur la programmation dynamique.

Mots clés
Programmation par contraintes, données séquentielles, algorithmes mémétiques, caractérisation.

Abstract
Recent works have shown the relevance of constraint programming to tackle data mining tasks. This thesis follows this approach and addresses motif discovery in sequential data. We focus in particular, in the case of classified sequences, on the search for motifs that best fit each individual class. We propose a language of constraints over matrix domains to model such problems. The language assumes a preprocessing of the data set (e.g., by pre-computing the locations of each character in each sequence) and views a motif as the choice of a sub-matrix (i.e., characters, sequences, and locations). We introduce different matrix constraints (compatibility of locations with the database, class covering, location-based character ordering common to sequences, etc.) and address two NP-complete problems: the search for class-specific totally ordered motifs (e.g., exclusive subsequences) or partially ordered motifs. We provide two CSP models that rely on global constraints to prove exclusivity. We then present a memetic algorithm that uses this CSP model during initialisation and intensification. This hybrid approach proves competitive compared to the pure CSP approach as shown by experiments carried out on protein sequences. Lastly, we investigate data set preprocessing based on patterns rather than characters, in order to reduce the size of the resulting matrix domain. To this end, we present and compare two alternative methods, one based on lattice search, the other on dynamic programming.

Key Words
Constraint Programming, Data Mining, Memetic Algorithms, Artificial Intelligence.