République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université A.MIRA-BEJAIA

Faculté des Sciences Exactes
Département d'Informatique
Laboratoire d'Informatique Médicale LIMED

THÈSE

EN VUE DE L'OBTENTION DU DIPLOME DE DOCTORAT

Domaine : Mathématique et Informatique

Filière : Informatique

Spécialité : Réseaux et Systèmes Distribués

Présentée par

Mme BENNAI SOUFIA

Thème

Combinaison des techniques de Compression et de data-mining pour le Big Data

Soutenue le : 03/06/2023 **Devant le Jury composé de :**

Nom et Prénom	Grade		
Mr TARI Abdelkamel	**Professeur**	Univ de Béjaia	Président
Mr. AMROUN Kamal	**Professeur**	Univ. de Béjaia	Rapporteur
Mr. LOUDNI Samir	**Professeur**	Univ. de Nantes	Co-Rapporteur
Mr. ELMIR Youssef	**MCA**	Ecole ESTIN, Béjaia	Examinateur
Mr. FARAH Zoubeyr	**MCA**	Univ. de Béjaia	Examinateur
Mme. TIGHIDET Soraya	**MCA**	Univ. de Béjaia	Examinatrice

Année Universitaire 2022/2023.

Remerciements

Tout d'abord, je tiens à remercier le directeur de cette thèse Kamal AMROUN d'avoir veillé au bon déroulement de ce travail de recherche et je le remercie également pour la confiance qu'il m'a accordé .

Également, je tiens à remercier très chaleureusement Samir LOUDNI d'avoir accepté la codirection de cette thèse et pour tous ses efforts, son encouragement, et surtout pour ses conseils qui m'ont motivé et qui ont fortement contribué à l'aboutissement de mes travaux de recherche.

Je souhaite aussi remercier le laboratoire GREYC de l'université de Caen Normandie, France, de m'avoir accueilli dans ses locaux. C'est dans ce même sillage que je tiens à je remercier particulièrement Abdelkader OUALI pour son accompagnement durant mon séjour au GREYC, pour ses conseils et le partage de ses idées et son expérience à travers de nombreuses longues discussions.

Je tiens aussi à remercier l'université ABDERRAHMANE MIRA DE BEJAIA et le LIMED d'avoir mis à la disposition des doctorants de bonnes conditions de travail et tous les moyens nécessaires pour le bon déroulement et l'accomplissement de leurs travaux de recherche.

Je remercie mon très cher papa BACHIR, paix à son âme, pour l'éducation qu'il m'a donné, les bonnes valeurs qu'il a semé en moi, tous ses sacrifices, sans lesquels je ne serai pas là aujourd'hui. Il n'est plus de ce monde mais il est toujours présent dans mon cœur et chaque mot de ce travail de recherche je l'ai écrit en pensant fortement à lui.

Je remercie ma très chère MAMAN pour sa présence, son éducation, sa bienveillance et ses encouragements. Je la remercie de m'avoir soutenu, d'avoir cru en moi et surtout de m'avoir supporté dans les moments d'angoisse et de stresse avec un cœur ouvert.

Je remercie mon époux Nadim ELSAKAAN qui m'a toujours épaulé, soutenu, cru en moi, encouragé et surtout m'avoir supporté durant les moments les plus difficiles et qui a beaucoup contribué à l'aboutissement de cette thèse.

Je remercie ma sœur HOUDA qui a toujours un bon exemple à suivre et qui m'a toujours soutenu et encouragé dans mes décisions.

Je tiens à remercier mes grands frères SALIM E et YASSIN qui m'ont toujours offerts un grand amour et qui ont toujours veillé au bien être de leur petite sœur. Je leur suis très reconnaissante. Sans oublier mes petits neveux Alice, Elyan, Elyne qui sont une bonne source de bonheur et de joie.

Je tiens aussi à remercier ma belle-mère, mon beau-père paix à son âme, mes beaux-frères Hani et Ramy pour leurs encouragements et conseils. Enfin, je remercie ma meilleure amie et sœur de cœur Lynda AMALOU qui a su m'encourager et m'aider à me relever à chaque fois que je pense à baisser les bras.

Remerciements aux membres du jury

Je tiens à profiter de l'occasion pour exprimer ma profonde reconnaissance envers les membres du jury qui ont évalué ma thèse de doctorat. Leur expertise, leur engagement et leurs précieux commentaires ont grandement contribué à la réussite de ma soutenance et à l'enrichissement de mes travaux de recherche.

Je tiens tout particulièrement à remercier le professeur **TARI Abdelkamel** d'avoir présidé cette soutenance avec professionnalisme et rigueur. Son expérience et ses conseils m'ont été d'une aide précieuse tout au long de ce processus et ont contribué à façonner ma recherche de manière significative.

Je suis également reconnaissant envers le docteur **ELMIR Youcef**, docteur **TIGHIDET Souraya** et le docteur **FARAH Zoubeyr** pour leur participation active à ma soutenance. leurs remarques pertinentes et leurs questions stimulantes ont mis en évidence des aspects cruciaux de mon travail et m'ont poussé à approfondir ma réflexion. Leur expertise dans leur domaines respectifs a été une source d'inspiration pour moi.

Je tiens également à exprimer ma gratitude envers tous les membres du jury pour le temps et l'attention qu'ils ont consacrés à l'évaluation minutieuse de ma thèse. Je suis consciente que leur rôle va bien au-delà de cette soutenance et que leur engagement envers la recherche et l'avancement des connaissances est inestimable. Leur évaluation et leur validation de mon travail ont une signification particulière pour moi.

Je tiens à exprimer ma reconnaissance envers mon directeur de thèse, le professeur AMROUN Kamal, pour sa guidance, son soutien indéfectible et ses conseils avisés tout au long de cette aventure de recherche. Son expertise et son mentorat ont été essentiels à ma réussite et à mon développement en tant que chercheure.

Enfin, je tiens à exprimer ma profonde reconnaissance envers mon co-directeur de thèse, le professeur **LOUDNI Samir**, pour son soutien inestimable tout au long de ma recherche doctorale. Votre expertise, ses conseils éclairés et sa disponibilité ont été des éléments essentiels de ma réussite académique. Son engagement envers mon projet, son encouragement constant et sa volonté de repousser les limites de la connaissance ont été une source d'inspiration. Je suis extrêmement reconnaissante de pouvoir compter sur sa guidance précieuse tout au long de cette aventure passionnante.

Table des matières

Liste des tableaux

Table des figures

Glossaire

AC Arc Consistency.

BT Backtrak.

CD-BJ Conflict Detected Backjumpin.

CSP Constraint Satisfaction Problem.

CT Compact Table.

FPTCM Frequent Pattern Tree Compression Method.

FP-Tree Frequent Pattern Tree.

GAC Generalize Arc Consistency.

LCM Linear time Closed item set Miner.

MDD Multivalued Decision Diagram.

MFIC Maximal Frequent Itemset Compression method.

PC Path Consistency.

STR Simple Tabular Reduction.

Liste des Publications

1 Soufia BENNAI et al. “An efficient heuristic approach combining maximal itemsets and area measure for compressing voluminous table constraints”. In : *The Journal of Supercomputing* 78 (2023), p. 139-159.

2 Soufia BENNAI, Kamal AMROUN et Samir LOUDNI. “Exploiting Data Mining Techniques for Compressing Table Constraints”. In : *31st IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2019, Portland, OR, USA, November 4-6, 2019*. IEEE, 2019, p. 42-49. DOI : 10.1109/ICTAI.2019.00015. URL : https://doi.org/10.1109/ICTAI.2019.00015.

3 Soufia BENNAI, Kamal AMROUN et Samir LOUDNI. “Exploitation des techniques de fouille de données pour la compression de contraintes table”. In : *Revue des Nouvelles Technologies de l’Information* Extraction et Gestion des Connaissances , RNTI-E-36 (2020), p. 449-456.

4 Soufia BENNAI, Kamal AMROUN et Samir Loudni AND. “ShrFP-Compression method for Big Data,” in : *7ème International Sumposium ISKO-Maghreb*. Bejaia, Algerie, nov. 2018.

Première partie

Introduction

Chapitre 1

Introduction

1.1 Contexte

1.1.1 Big Data

Le phénomène de Big data est devenu omniprésent dans presque toutes les disciplines, de la science à l'ingénierie. Le terme Big Data fait référence au grand volume de données pouvant être dans un format structuré ou non structuré, des données plus variées et complexes avec des difficultés de stockage et d'analyse. Les données du Big Data sont principalement caractérisées par leur :

— Volume qui représente un grand défi, car pour manipuler et analyser un grand volume de données, de grandes capacités de stockage et de calcul peuvent être requises. Le volume fait donc référence à la quantité de données à manipuler et à analyser afin de répondre à des interrogations ou déduire de nouvelles informations.

— Variété : les données peuvent se présenter sous différents formats. Un format structuré dont les données respectent un schéma fixe (une table dont les colonnes représentent les attributs et les lignes représentent les instances). Un format semi-structuré dont les données respectent une certaine structure mais des variations peuvent être autorisées. Parmi les formats semi-structurés, on peut citer le format JSON (JavaScript Object Notation), le format XML (eXtensible Markup Language), format CSV, ...etc. Les données non-structurées quant à elles, elles ne respectent aucune structure et peuvent provenir de différentes sources : Internet des Objets, médias sociaux, le web, ...ect.

— Vélocité : la vélocité dans le Big Data concerne la vitesse par laquelle les données sont générées. Elle peut aussi signifier la durée pendant laquelle les données sont valides.

— Valeur : consiste en l'intérêt des nouvelles informations que les entreprises ou les acteurs du Big Data peuvent tirer des données générées et stockées.

— Variabilité : les données peuvent être moins cohérentes du fait qu'elles peuvent

avoir des significations différentes ou peuvent provenir sous différents formats à partir de différentes sources de données.

— Véracité : la véracité des données signifie le degré de fiablité et la qualité des sources de données à traiter ou à analyser

Dans le contexte de cette thèse on s'est intéressé au volume des données semi-structurées. Afin de réduire la taille de l'espace mémoire requis pour la représentation en mémoire et le temps de traitements des données semi-structurées, nous avons proposé des méthodes de compression basées sur des techniques de fouilles de données (data mining). Pour montrer l'efficacité de la compression, nous avons choisi comme domaine d'application les Problèmes de Satisfactions des Contraintes (CSP) en compressant les contraintes tables composant le CSP ensuite résoudre le CSP compressé.

1.1.2 Compression des contraintes tables volumineuses

La compression des contraintes table volumineuses est un processus qui vise à réduire la tailles de l'espace mémoire requis pour la représentation en mémoire des CSPs mais aussi de réduire la taille de l'espace de recherche de solutions en cherchant des solutions directement à partir du CSP compressé. La compression consiste à éliminer les données redondantes ce qui permet d'éviter de parcourir les mêmes données plusieurs fois durant le processus de recherche de solution.

1.2 Objectifs

1.2.1 Compression des contraintes tables

Le premier objectif de cette thèse est d'exploiter des techniques de fouilles de motifs (itemsets) fréquents afin de proposer de nouvelles approches pour la compression des contraintes tables volumineuses. Les techniques de fouilles de motifs fréquents vont permettre de retrouver les données redondantes dans des contraintes tables. Ensuite de les éliminer afin d'obtenir des contraintes tables sans redondance de données ou au moins minimiser les redondances. Cela permettra de minimiser la taille de l'espace de mémoire requis pour leurs représentation.

1.2.2 Résolution des problèmes de satisfaction de contraintes compressées

Le deuxième objectif de cette thèse est de proposer de nouvelles méthodes de résolution ou adapter des méthodes existantes pour résoudre les CSPs compressés sans passer par une étape de décompression qui peut être très coûteuse en temps et en espace mémoire.

1.3 Contributions

1.3.1 Compression basée sur une structure d'arbre FP

La méthode Sliced-table proposée par Gharbi et al. [1] exploite un arbre de prefixe (FP-tree) et une fonction qui calcul le gain en compression pour extraire des motifs fréquents pertinents pour la compression d'une contrainte table. Bien que cette méthode de compression a montré son efficacité par le fait qu'elle permet de réduire :

- la taille de l'espace mémoire requis pour la représentation en mémoire des contraintes table volumineuses,
- la taille de l'espace de recherche durant le processus de recherche de solution,
- le temps de résolution par le fait que la structure des contraintes table compressées permet d'éliminer à la fois tout un ensemble de tuples ne pouvant pas contenir de solution au lieu d'éliminer les tuples un par un.

Pour qu'un motifs soit considéré comme fréquent, cette méthode fixe un seuil minimal de fréquence (Smin) à deux (2) générant ainsi un très grand nombre de motifs fréquents. Ce qui peut ainsi affecter la qualité de compression. En plus, la fonction de calcul de gain, qui a pour objectif de comparer le nombre d'items avec lequel il est possible de réduire la taille de f tuples en les compressant avec un motif fréquent u par rapport au nombre d'items avec lequel réduire la taille de f' (avec $f > f'$) tuples en les compressant avec un super motif u' de u, n'est pas très efficace. Car la comparaison se fait par rapport à un nombre différent de tuples (f et f') donnant ainsi une approximation du gain en compression moins précise. Pour remédier à ces problèmes, nous avons proposé dans [2] une amélioration de la fonction de calcul de gain afin d'avoir une meilleur approximation du gain en compression. Cela en considérant exactement les mêmes tuples durant la comparaison. Ensuite, nous avons proposé dans [3], de calculer de façon dynamique, pour chaque contrainte table, la valeur du seuil minimal de fréquence *Smin*. Des expérimentations menées sur les mêmes datasets que ceux utilisés dans [1] ont montré que

notre approche est plus efficace en terme de compression mais aussi en terme de temps de résolution.

1.3.2 Compression basée sur les itemsets fréquent maximaux

Dans la première contribution, nous avons travaillé sur l'amélioration de taux de compression des contraintes tables. Ensuite, dans [4] nous avons essayé de comprendre la relation qui existe entre le taux de compression et le temps de résolution (ie. si un meilleur taux compression permet toujours d'améliorer le processus de résolution). Pour cela, nous avons proposé une nouvelle méthode de compression basée sur des motifs fréquents plus longs (contenant un plus grand nombre d'items) et couvrant un maximum de tuples. Nous avons exploité une technique, proposée dans la littérature, pour l'extraction des itemsets fréquents maximaux candidats pour la compression. Ainsi que la notion d'aire d'un itemset pour sélectionner les itemsets les plus pertinents pour la compression. Les résultats expérimentaux, menés sur les mêmes datasets ont montré que les itemsets maximaux compressent moins les contraintes table en comparant avec la méthode proposée dans la première contribution, mais réduit considérablement le temps de résolution des CSPs compressés.

1.3.3 Exploitation de la technologie Spark et du clustering pour une compression parallélisée et distribuée

Les deux méthodes de compression précédentes ont montré leur éfficacité en terme de compression des contraintes tables et en terme de réduction de temps de résolution des CSPs compressés. Pour cela, nous avons proposé dans [5] une version parallélisée, basée sur la technologie Spark et sur le clustering, de la méthode de compression basée sur les itemsets fréquents [3, 2, 6]. Afin de paralléliser la compression, nous avons partitionné les transactions (tuples) à compresser en un ensemble de clusters. Cela en utilisant la méthode de clustering k-means pour regrouper les transactions similaires dans un même cluster. Ensuite, en appliquant les fonctions de Spark, nous avons distribué, sur chaque cluster, la recherche des itemsets fréquents requis pour la compression. Pour étendre le domaine d'application de cette méthode de compression, nous avons effectué une série d'expérimentations sur des datasets plus généralisés de UCI (University of California Ivrine). Les résultats de ces expérimentations ont montré que le partitionnement des transactions en groupes de transactions similaires améliore le taux de compression et

le calcul distribué permet de réduire le temps de compression.

1.4 Plan

Le mémoire est organisé en deux parties principales : la première est consacrée pour les préliminaires et l'état de l'art. La deuxième partie présente nos contributions.

1.4.1 Préliminaires et État de l'art

Dans la première partie, nous présentons et définissons les différents concepts et terminologies associés au domaine des CSPs dans les Big Data et au domaine d'extraction des motifs fréquents, nous étudions aussi les différentes approches et méthodes existantes pour la compression et résolution des CSPs volumineux.

Chapitre 2 présente les problèmes de satisfaction des contraintes table volumineuses, ainsi que les méthodes de consistance et de résolution. Nous commençons d'abord par présenter le concept CSP. Ensuite, nous présentons les approches de résolutions de CSPs par recherche arborescente, les heuristiques de choix de variables et de valeurs durant le processus de résolution, et enfin, introduire le concept de consistance avec ces deux niveaux global et local. Nous détaillons la consistance locale car c'est le niveau auquel on s'intéresse dans le cadre de cette thèse.

Chapitre 3 présente le domaine de fouille des motifs fréquents. Nous commençons par la définition du contexte transactionnel et la définition des concepts liés à l'extraction de motifs fréquents à partir de bases transactionnelles. Ensuite, présenter quelques méthodes de la littérature utilisées dans nos travaux pour l'extraction des motifs fréquents.

Chapitre 4 présente une étude des méthodes proposées dans la littératures pour la compression des problèmes de satisfaction des contraintes tables. Nous commençons par classer ces méthodes en cinq classes principales :

- Méthodes de compression basées sur la structure d'arbre et diagrammes de décision,
- Méthodes de compression basées sur la réduction tabulaire (STR),
- Méthode de compression basée sur une représentation binaire,
- Méthode de compression basée sur segmented tuples,
- Méthodes de compression basée sur l'extraction de motifs.

Pour chaque classe, nous présentons ces principaux avantages et inconvénients. Enfin,

nous comparons ces méthodes dans un tableaux comparatifs en se basant sur des critères qui nous ont conduit vers de nouvelles propositions.

1.4.2 Contributions

Chapitre 5 commence par décrire notre première méthode de compression exploitant une technique de fouille de motifs fréquents basée sur une structure d'arbre de préfixe. Cette méthode permet de représenter une contrainte table sous forme d'un arbre de préfixe, utilise une fonction qui permet extraire des motifs fréquents pertinents pour la compression. Ensuite, créer la contrainte table compressée en se basant sur les motifs énumérés. Une méthode de résolution est aussi utilisée pour résoudre les CSP comprésés.

Les expérimentations réalisées sur des instances CSP, utilisées par la plupart des travaux de l'état de l'art , ont montré l'efficacité de la méthode et son apport en terme de réduction de la taille de l'espace mémoire requis pour la représentation des contraintes table et en terme de temps de recherche de solutions aux CSPs.

Chapitre 6 décrit une autre nouvelle méthode basée sur les itemset (motifs) fréquents maximaux et sur la notion d'aire d'un motif pour la compression des contraintes tables volumineuses. La même méthode de résolution que celle utilisées dans le chapitre précédent est utilisée pour résoudre les CSPs compressés. Les résultats expérimentaux, mené sur les mêmes instances que le chapitre précédent, ont démontré qu'un meilleur taux de compression n'implique pas forcément un meilleur temps de résolution mais qu'il y a d'autres facteurs qui peuvent influencer le temps de résolution tel que la qualité des motifs fréquents utilisés dans la compression.

Chapitre 7 est une déscription d'une version parallélisée et distribuée de la méthode proposée dans le chapitre 5. La technologie Spark et la méthode de clustering Kmeans sont utilisées pour paralléliser et distribuer le processus de compression. Contrairement aux chapitres 5 et 6, les expérimentations sont menées sur des dataset plus générals afin d'étendre le champs d'application des méthodes de compression proposées.

Deuxième partie

Préliminaires

Chapitre 2

Problèmes de satisfaction de contraintes

Dans ce chapitre, nous introduisons certains concepts liés aux problèmes de satisfaction de contraintes (CSP) et leur résolution.

2.1 Définitions d'un Problème de Satisfaction de Contraintes (CSP)

Définition 2.1.1 *(Variable)*
Une variable, généralement notée par v, est définie sur un domaine d spécifiant l'ensemble des valeurs possibles pour cette variable. Le domaine courant d'une variable peut être modifié, des valeurs peuvent être supprimées mais le domaine ne peut en aucun cas être étendu i.e..aucune nouvelle valeur ne peut être ajoutée au domaine d d'une variable v.
Dans le contexte de cette thèse, nous nous intéressons aux variables aux domaines finis.

Définition 2.1.2 *(Relation)*
Une relation est un sous ensemble du produit cartésien d'une séquence de domaines.
Soit d_1, d_2, $\cdots$, d_n une séquence de n domaines. Le produit cartésien de ces domaines est noté par $\prod_{k=1}^{n} d(k)$ et il est défini par $\{(a_1, a_2, \cdots, a_n) | a_1 \in d_1, a_2 \in d_2, \cdots, a_n \in d_n\}$.

Example 2.1.1 *Soit les variables v_1, v_2 et v_3 définies par les domaines $d_1 = d_2 = d_3 = \{0, 1\}$. Le produit cartésien des domaines de ces variables est comme suit :*

$$\prod_{k=1}^{3} d(k) = d_1 \times d_2 \times d_3 = \begin{Bmatrix} (0,0,0) & (1,0,0) \\ (0,0,1) & (1,0,1) \\ (0,1,0) & (1,1,0) \\ (0,1,1) & (1,1,1) \end{Bmatrix}$$

Un exemple de relation R définie sur l'ensemble des domaines d_1, d_2 et d_3 :

$$R = \begin{Bmatrix} (0,0,0) \\ (1,0,1) \\ (1,1,0) \\ (0,1,1) \end{Bmatrix}$$

Définition 2.1.3 *(item) On appel un littéral la paire (v_i, a_i) tel que $v_i \in V$ et $a_i \in d_i$.*

Définition 2.1.4 *(Contrainte)*

Une contrainte $c \in C$ est une paire $(S(c), R(c))$ tel que $S(c) \subseteq V$ est une liste de variables représentant la porté de la contrainte, appelée scope de la contrainte. $R(c) \subseteq \prod_{v_k \in S(c_i)} d_k$ est un sous-ensemble du produit cartésien des domaines $\{d_1, \cdots, d_n\} \in D$ des variables du scope de c, $R(c)$ est l'ensemble de tuples autorisés pour les variables de c.

L'arité d'une contrainte est égale au cardinal de son scope.

En fonction de l'arité, nous pouvons distinguer quatre types de contraintes :

- ***unaire** : on parle de contrainte d'arité égale à 1.*
- ***binaire** : on parle de contrainte d'arité 2.*
- ***ternaire** : on parle de contrainte d'arité égale à 3.*
- ***n-aire** : on parle de contrainte d'arité supérieur à 3.*

Une contrainte c peut être principalement définie soit en :

- ***extension** : en listant un ensemble de tuples autorisés ou non autorisés par la contrainte. i.e.les valeurs possibles pour les variables de $S(c)$.*

  ***Example 2.1.2** Soit la contrainte $v_1 + v_2 \leq v_3 + 1$ qui porte sur les variables $\{v_1, v_2, v_3\} \in V$ dont les domaines respectifs $\{d_1, d_1, d_3\} \in D$, tel que $d_1 = d_1 = d_3 = \{1,2,3\}$. Les tuples autorisés par cette contrainte sont comme suit :*
  $\{(1,1,2),(1,1,3),(1,2,3),(2,1,3)\}$
- ***intention** : en exprimant la sémantique de la contrainte par un prédicat qui peut intégrer les différentes opération arithmétique, relationnels et logiques que les variables doivent satisfaire. l'expression $v_1 \leq v_2 + 2$ est un exemple de contrainte définie en intention.*

Une contrainte globale est une contrainte dont la sémantique peut être appliquée à toutes les variables. Un exemple de contrainte globale est la contrainte alldiffent (toutes différentes) qui veut que toutes les variables doivent avoir des valeurs différentes. On ne peut pas trouver deux variables partageant la même valeur. L'arité d'une contrainte c représente le nombre de variables dans sa portée, c'est à dire $|S(c)|$.

Définition 2.1.5 *(CSP)*

Un problème de satisfaction de contraintes est définit par le triple $P = \langle V, D, C \rangle$ *tel que :*

- $V = \{v_1, \cdots, v_n\}$ *est un ensemble de n variables,*
- $D = \{d_1, \cdots, d_n\}$ *est un ensemble fini de domaines, chaque domaine* d_i *est un ensemble de valeurs possibles pour* v_i*.*
- $C = \{c_1, \cdots, c_m\}$ *un ensemble de m contraintes.*

Example 2.1.3 *Considérons le problème des quatre (4) reines sur un échiquier de 4X4. Aucune reine ne doit pouvoir atteindre l'autre, i.e.deux reines ne peuvent pas être au même temps sur la même colonne ou sur la même ligne ou même sur la même diagonale. La modélisation de ce problème sous forme de CSP est comme suit :*

- *Pour chaque ligne i de l'échiquier on associe une variable* v_i *avec* $i \in \{1,2,3,4\}$*. Les quatre variables portent sur le même domaine* $d = \{1,2,3,4\}$ *tel que chaque valeur de d correspond à une colonne de l'échiquier.*
- *Le problème porte sur les contraintes suivantes :* c_1*,* c_2*,* c_3*,* c_4*,* c_5*,* c_6 *telle que :* $c_1 = \langle (x_1, x_2), R_1 \rangle$*,* $c_2 = \langle (x_1, x_3), R_2 \rangle$*,* $c_3 = \langle (x_1, x_4), R_3 \rangle$*,* $c_4 = \langle (x_2, x_3), R_4 \rangle$*,* $c_5 = \langle (x_2, x_4), R_5 \rangle$*,* $c_6 = \langle (x_3, x_4), R_6 \rangle$*.*
- *Les relations des contraintes sont définies comme suite :*
  - R_1 *= {(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)}*
  - R_2 *= {(1,2),(1,4),(2,1),(2,3),(3,2),(3,4),(4,1),(4,3)}*
  - R_3 *= {(1,2),(1,3),(2,1),(2,3),(2,4),(3,1),(3,2),(3,4),(4,2),(4,3)}*
  - R_4 *= {(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)}*
  - R_5 *= {(1,2),(1,4),(2,1),(2,3),(3,2),(3,4),(4,1),(4,3)}*
  - R_6 *= {(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)}*

Définition 2.1.6 *(Instanciation)*

L'instanciation d'une variable v_i *est l'attribution d'une valeur* a_j *à la variable* v_i*, tel que* $a_i \in d_i$*. L'affectation d'une valeur* a_j *à une variable* v_j *est notée par* (v_i, a_j)*.*
Une instantiation $A = \{(v_1, a_1), ..., (v_n, a_n)\}$ *est dite complète si à chaque variable* $v_i \in V$ *est affectée une valeur de son domaine* d_i*. A est dite partielle, si au moins une variable de V n'est pas instanciée dans A.*

Example 2.1.4 *Soit le CSP de l'exemple 2.1.3 qui porte sur les variables* $\{v_1, v_2, v_3, v_4\}$ *dont les domaines respectifs* $d_1 = d_2 = d_3 = d_4 = \{1,2,3,4\}$*. L'affectation de la valeur* 2 *de* d_1 *à la variable* v_1 *est appelée une instanciation* $(v_1, 1)$*.*

L'ensemble des affectations suivantes : $\{(v_1,),(v_3,1),(v_4,3)\}$ *est une instanciation partielle car il y a au moins une variable de V non instanciée et qui est* v_2*. l'affectation suivante :* $\{(v_1,2),(v_3,1),(v_4,3),(v_2,4)\}$ *est une instanciation complète car elle affecte à chaque variable du réseau de contrainte une valeur de son domaine.*

Définition 2.1.7 *(Tuple valide)*
Un tuple valide est une instanciation complète des variables du scope d'une contrainte $c \in C$ *vérifiant (satisfaisant) la contrainte.*

Définition 2.1.8 *(Contrainte satisfaite)*
Une contrainte $c \in C$ *est satisfaite par une instanciation complète ou partielle, si la restriction de l'instanciation aux variables de* $S(c)$ *est un tuple valide de la contrainte.*

Example 2.1.5 *Soit le CSP de l'exemple 2.1.3, l'instanciation partielle* $\{(v_1,2)(v_4,3)\}$ *satisfait la contrainte* c_2 *car la restriction de cette instanciation aux variables du* $S(c_2)$ *(i.e.*$v_1 = 2$*) est un tuple valide de cette contrainte i.e.le tuple* $(2,1)$ *de* R_2*.*

Définition 2.1.9 *(Support)*
Un support d'une contrainte $c \in C$ *est une instanciation de toutes les variables de* $S(c)$ *satisfaisant la contrainte c. Un support de d'une contrainte c contenant l'instantiation* (v_i,a_i) *est appelé support de* v_i *dans c.*

Example 2.1.6 *Soit l'instantiation partielle* $\{(v_1,2),(v_2,4),(v_4,3)\}$ *des variables du CSP de l'exemple 2.1.3, la restriction de cette instanciation aux variables du scope de* c_1*, i.e.*$\{(v_1,2),(v_2,4)\}$*, est un support de la contrainte* c_1 *car c'est une instanciation de toutes les variables de* $S(c_1)$*.*

Définition 2.1.10 *(Solution) Une solution d'une instance CSP* $P = \langle V,D,C\rangle$ *est une instanciation complète de ses variables qui satisfont toutes les contraintes du CSP.*

Example 2.1.7 *L'instanciation complète suivante* $\{(v_1,2),(v_3,1),(v_4,3),(v_2,4)\}$ *des variables du CSP de l'exemple 2.1.3 est une solution de ce CSP car elle satisfait toutes ces contraintes.*

2.2 Résolution des CSPs par recherche arborescente

La résolution d'un CSP consiste à vérifier s'il admet au moins une solution. C'est un problème NP-Complet. De nombreuses méthodes ont été proposées pour une résolution efficace des problèmes de grande taille. Parmi elles, on trouve celles qui effectuent

un parcours systématique de l'espace de recherche en le représentant sous forme d'un arbre. C'est ce qu'on appelle méthodes complètes. Chaque noeud de l'arbre de recherche consiste en une variable et chaque branche à une instanciation possible de la variable. Les domaines des variables se réduisent tout au long d'une branche jusqu'à l'instanciation de toutes les variables ou jusqu'à la détection d'un échec. Chaque feuille consiste en une instanciation complète résultante de l'ensemble des instanciations faites tout au long de la branche dont elle est feuille. Une instanciation complète est une solution du problème si elle satisfait toutes les contraintes.
Les méthodes de recherche arborescente pour les CSPs se caractérisent par leur complétude, elles assurent un parcours complet de l'espace des solutions possibles. Ces méthodes sont au coeur des systèmes de programmation par contrainte.

2.2.1 Retour arrière chronologique (backtrack)

Le backtrack (BT) [7] est l'algorithme de base le plus répandu pour la résolution des CSPs. C'est un algorithme de recherche en profondeur d'abord qui consiste à énumérer de façon récursive les valeurs possibles des variables. A chaque nouvelle itération, l'instanciation partielle courante $\{v_1 = a_1, \cdots, v_{k-1} = a_{k-1}\}$ est étendue en instanciant une nouvelle variable v_k par une valeur de son domaine $a_k \in d_k$. Si l'instanciation courante n'est pas satisfaite par toutes les contraintes, une nouvelle valeur de d_k est choisie pour la variable v_k. Si aucune instanciation de v_k ne satisfait toutes les contraintes, la variable v_k est désinstanciéé et un backtrack vers la variable précédente v_{k-1}est effectué. L'algorithme 1 illustre le principe du backtrack. Dans le pire des cas, tout l'espace de recherche est exploré. La complexité temporelle de cet algorithme est de $O(d^n)$ avec d est égale à la taille du plus grand domaine des variables et n le nombre de variables.

2.2.2 Retour arrière non chronologique (backjumping)

L'algorithme BT effectue un retour arrière vers une variable v_{k-1} précédant la variable courante v_k lorsqu'un échec est détecté et que toutes la valeurs possibles pour la variable v_k ont été exploré. Un retour arrière vers v_{k-1} ne présente aucune garantie que v_{k-1} est responsable de l'inconsistance. Alors, l'attribution de nouvelles valeurs à v_{k-1} peut conduire aussi à l'échec. Pour cela, un retour arrière non chronologique qui consiste à faire un saut (bachkjuping) vers une variable responsable de l'échec a été proposé. Parmi les algorithmes basés sur le retour arrière non chronologique, on trouve :

Algorithme Backtrack

Data : un CSP $P = \langle V, D, C \rangle$, une instanciation A (initialement $A = $).
Result : une solution à P s'il est consistant

```
1  if A est consistante then
2      if A est une instanciation complète then
3          retourner vrai
4      else
5          choisir une nouvelle variable v_k de V
6          for chaque valeur a_k ∈ d_k avec d_k ∈ D do
7              if Backtrack(P, A ∪ {v_k = a_k}) then
8                  retourner vrai
9              end
10         end
11     end
12 else
13     retourner faux
14 end
```

— **Algorithme de Backjumping [8]** qui fait un saut vers la variable la plus profonde en conflit avec la variable courante.

— **Graph-based backjumping** [9] basé sur le graphe des contraintes, lorsqu'un échec est détecte par une variable v_k, un retour arrière vers la dernière variable instanciée v_j du voisinage de v_k. S'il ne reste aucune valeur à explorer dans le domaine d_j de v_j, un retour arrière vers la variable la plus profonde dans le voisinage de v_j ou dans le voisinage de toute variable instanciée après v_j et qui soit responsable de l'inconsistance.

— **Algorithme Conflict Detected Backjumpin (CD-BJ) [10]** basé sur la sauvegarde d'un ensemble conflit, pour chaque variable v_k, incluant toutes les variables instanciées avant v_k qui sont en conflit avec v_k pour une valeur donnée de son domaine d_k ou qui sont responsables de la non satisfaction de l'extension d'une instanciation contenant v_k par toutes les contraintes. Lorsqu'un échec est détecté par v_k, un retour arrière se fait vers la dernière variable instanciée de l'ensemble conflit de v_k.

2.2.3 Recherche arborescente incomplète

Le temps nécessaire pour trouver une solution à un problème avec les algorithmes complets est souvent exorbitant. Pour cela, des méthodes incomplètes qui considèrent l'espace de recherche en sa totalité mais ne l'explorent pas en entier, ont été proposé afin de donner un résultat acceptable en un temps raisonnable. L'inconvénient avec ces méthodes est qu'elles ne sont pas capables de prouver l'inexistence de solution pour un problème donné. Ce type de méthodes aborde la résolution d'un CSP comme étant un problème d'optimisation combinatoire qui consiste à trouver une affectation satisfaisant le maximum de contraintes, sachant que l'objectif principal est de trouver une affectation satisfaisant toutes les contraintes. Parmi les approches possibles pour une résolution incomplète, on peut citer : Recherche Taboue, Colonie de fourmis, Algorithmes génétiques, etc.

2.3 Heuristiques d'ordre des variables et valeurs

Des heuristiques permettant de guider le choix des variables et des valeurs ont été proposées afin d'améliorer les méthodes de résolution des CSPs.

2.3.1 Heuristiques d'ordre de choix de variables

L'ordre des variables avant ou durant la résolution d'un CSP est une amélioration importante pour les algorithmes de recherche arborescente. Les heuristiques proposées sur l'ordre des choix variables visent à réduire la taille de l'espace de recherche en détectant le plus rapidement les échecs. Cela en commençant par les choix les plus contraints. Il existe deux types d'ordre des variables :
Ordre statique : l'ordre est prédéfini avant la recherche et ne change pas, il est basé sur des caractéristiques structurelles du problème à résoudre, *i.e.*degré (nombre de contraintes portant sur une variable) , taille des domaines des variables et satisfaisabilité des contraintes. Parmi les heuristiques basées sur l'ordre statique, on peut trouver :

- *MaxDeg*[11] : ordonner les variables dans un ordre décroissant de leurs degrés.
- *MinDom*[12] : ordonner les variables dans un ordre croissant selon la taille de leurs domaines.
- *MinCon*[11] : choisir la première variable arbitrairement, la prochaine variables à choisir est celle qui a le plus grand nombre de variables voisines déjà ordonnées.

Ordre dynamique : l'ordre est effectué à chaque noeud de l'arbre au cours de la recherche car les choix dépendent du sous-arbre à explorer.
Dans le cas d'ordre statique, le

— *dom/deg*[13] : la variable à instancier est celle minimisant le ratio entre la taille du domaine courant et le degré.

— *dom+deg*[14] : instancier la variable dont le domaine courant contient le minimum de valeur, dans le cas d'égalité avec d'autres variables, choisir celle avec un degré maximum.

2.3.2 Heuristiques d'ordre de choix de valeurs

Parmi les heuristiques sur l'ordre des valeurs à affecter en priorité aux variables, on peut citer :

— *min-conflict*[14] : ordonner les valeurs d'une variable dans un ordre croissant selon le nombre de conflits, *i.e.*affecter pour une variable courante la valeur ayant un minimum de valeurs incompatibles avec les valeurs des variables non encore instanciées.

— *basée sur les impacts* [15] : ordonner les valeurs d'une variables dans un ordre croissant selon l'impact de l'affectation de chaque valeur sur l'espace de recherche.

2.4 Consistance

Réduire l'espace de recherche à explorer durant le processus de résolution d'un CSP est un défi réel. A cet effet, des méthodes ont été proposées pour le filtrage des domaines des variables en supprimant, au fur et à mesure, des domaines des variables les valeurs (ou combinaisons de valeurs) qui ne peuvent pas apparaître dans une solution. Ceci permet de réduire la taille de l'arbre de recherche et d'accélérer la recherche de solution du fait que les sous-arbres enracinés au niveau des valeurs supprimées ne sont pas explorés. D'autres méthodes ont été développées pour réduire la taille des relations en supprimant des tuples ne pouvant pas participer à une solution. La consistance d'un CSP est vérifiée lorsque aucune valeur ne peut être supprimée des domaines des variables et aucun tuple ne peut être retiré des relation.
Il existe deux niveaux de consistance, une consistance locale et consistance globale. La consistance est dite locale lorsqu'elle s'agit d'un sous ensemble de variables de l'instance

CSP, et elle est dite globale lorsqu'elle s'agit de tout le réseau de contraintes.
Dans le cadre de cette thèse, on s'intéresse à la consistance locale qui est une propriété appliquée pour des sous-ensembles de variables ou de contraintes. On distingue trois types de consistances.

2.4.1 Consistance de Noeud

On dit qu'une instance CSP est consistante de noeud, si pour chaque variable $v_i \in V$ et pour chaque valeur $a \in d_i$, l'instanciation partielle $v_i = a$ satisfait toutes les contraintes unaires du CSP.
Le principe de la consistance de noeud consiste à supprimer des domaines des variables, toutes les valeurs ne satisfaisant pas une contrainte unaire.

2.4.2 Consistance d'arc (AC)

Soit $d_i \in D$ le domaine de la variable $v_i \in V$, si pour toute valeur $a \in d_i$ et pour chaque variable $v_j \in V$ tel que v_i et v_j sont reliés par une contrainte c_{ij}, il existe une valeur $b \in d_j$ telle que (a,b) satisfait la contrainte c_{ij} alors d_i est dit arc consistant. Filtrer un CSP par consistance d'arc consiste à éliminer des domaines des variables les valeurs ne satisfaisant pas la propriété de consistance d'arc.

Example 2.4.1 *Soit le graphe de contraintes P avec trois variables* $V = \{v_1, v_2, v_3\}$*, les domaines des variables sont comme suit :* $d_1 = d_2 = d_3 = \{1,2,3\}$.
Le problème porte sur les contraintes $c_{12} = \langle (v_1, v_2), R_{12} \rangle$, $c_{13} = \langle (v_1, v_3), R_{13} \rangle$, $c_{23} = \langle (v_2, v_3), R_{23} \rangle$.
Les relations des contraintes sont aussi définies comme suit : $R_{12} = \{(1,1),(2,2),(3,3)\}$, $R_{13} = \{(1,2),(2,1),(3,3)\}$, $R_{2,3} = \{(1,2),(3,1)\}$.
Le CSP P n'est pas arc consistant car $v_2 = 2$ $v_3 = 3$ *n'ont pas de support dans* c_{23}.

Pour maintenir la consistance d'arc dans P, on procède au filtrage des domaines et des contrainte. On a la contrainte c_{23} *n'est pas arc-consistante du fait que le* $v_3 = 3$ *n'a pas de support dans cette contraintes, alors on supprime la valeur* 3 *du domaine de la variable* v_3 *i.e..* d_3*. La suppression de cette valeur va engendrer la suppression des support de* $v_3 = 3$ *de toutes les contraintes, on aura* $R_{12} = \{(1,1),(2,2),(3,3)\}$, $R_{13} = \{(1,2),(2,1),$***(3,3)***$\}$, $R_{23} = \{(1,2),(3,1)\}$ *avec le couple en gras est le couple à supprimer. La contrainte* c_{13}*, qui était à l'origine arc-consistante, devient alors non arc-*

consistante car le littéral $v_3 = 1$ *n'est plus arc-consistant. Le littéral* $v_2 = 2$ *aussi n'est pas arc-consistant alors on supprime valeur* 2 *de* d_2 *et on obtient* $R_{12} = \{(1,1),\mathbf{(2,2)},(3,3)\}$, $R_{13} = \{(1,2),(2,1),\mathbf{(3,3)}\}$, $R_{23} = \{(1,2),(3,1)\}$. *Ainsi,* c_{12} *devient non arc-consistante car* $v_1 = 2$ *n'a plus de support dans cette contrainte. Pour maintenir la consistance d'arc dans les deux contraintes* c_{12} *et* c_{13}*, on procède de la même manière. Pour maintenir la consistance d'arc dans* c_{12}*, on supprime du domaine de* v_1 *la valeur* 2*, car* $v_1 = 2$ *n'a pas de support dans* c_{12}*, pour obtenir* $R_{12} = \{(1,1),\mathbf{(2,2)},(3,3)\}$, $R_{13} = \{(1,2),\mathbf{(2,1)},\mathbf{(3,3)}\}$, $R_{23} = \{(1,2),(3,1)\}$. *On remarque que* $v_3 = 1$ *n'a plus de support dans* c_{13}*. Maintenant, pour maintenir la consistance d'arc dans* c_{13}*, on supprime* 1 *de* d_3 *et* 3 *de* d_1 *pour obtenir* $R_{12} = \{(1,1),\mathbf{(2,2)},\mathbf{(3,3)}\}$, $R_{13} = \{(1,2),\mathbf{(2,1)},\mathbf{(3,3)}\}$, $R_{23} = \{(1,2),\mathbf{(3,1)}\}$. *On voit bien que* $(v_2 = 3)$ *n'a pas de support dans* c_{12}*, alors on supprime* 3 *de* d_2 *pour obtenir finalement un réseau de contraintes arc-consistant :* $R_{12} = \{(1,1)\}$, $R_{13} = \{(1,2)\}$, $R_{23} = \{(1,2)\}$.

Mackworth [16] est le premier à proposer un algorithme implémentant la consistance d'arc AC1 (Algorithme 3). L'algorithme utilise une procédure Réviser() (Algorithme 2), qui vérifie pour chaque valeur a d'une variable v_i, l'existence d'un support dans la contrainte c_i portant sur la variable v_i. Si aucun support n'a été trouvé pour $v_i = a$, alors supprimer a du domaine d_i de v_i. Sinon, dans le cas ou un support a été trouvé, continuer avec les valeurs de d_i jusqu'à vérifier toutes les valeurs. La procédure retourne un booléen qui vaut vrai si au moins une valeur a été supprimée du domaine de v_i.

Algorithme 2 Réviser

Data : une contrainte $c_k = (v_i, v_j)$.
Result : $modifier$

```
1 for toute valeur a_i ∈ d_i do
2     if ∄a_j ∈ d_j|(a_i, a_j) ∈ R_k then
3         enlever a_i de d_i
4         modifier ← vrai
5     end
6 end
```

La modification d'un domaine d'une variable même après avoir révisé toutes le contraintes, n'a pas d'influence sur toutes les contraintes. Pour cela, Mackworth a proposé une amélioration dans AC3 [16] qui consiste à appliquer la procédure réviser() uniquement sur les contraintes susceptibles d'être affectées par la modification d'un domaine d'une variable. Pour ce faire, AC3 (Algorithme 4) utilise une file mémorisant les contrainte à

Algorithme 3 AC

```
repeat
    modifier ← faux
    for chaque contrainte c_k do
        modifier ← Reviser(c_k) ∨modifier
    end
until not modifier;
```

re-réviser. L'algorithme AC3 est d'une complexité de $O(|C||d|^3)$ avec $|C|$ est le nombre de contraintes dans P et $|d|$ représente la taille du plus grand domaine des variables de P.

Algorithme 3 AC3

```
Liste ← (v_i, v_j), i ≠ j
while Liste ≠ ∅ do
    Supprimer de Liste un coupe (v_i, v_j)
    if Réviser((v_i, v_j)) then
        Liste ← Liste ∪ {(v_z, v_i z)}/∃ une contrainte reliant les variables v_z et v_i
    end
end
```

D'autres algorithmes de consistance d'arc ont suivi AC3, parmi ces algorithmes on peut citer AC4[17], AC5[18], AC6[19], AC7[20], AC2000[21] et AC2001[21]. Dans AC2001[21], les auteurs ont proposé une amélioration de la procédure Réviser() (appelée Reviser-2001) de manière à accélérer la recherche de support. Ils ont proposé d'utiliser une variable $Last((v_i,a_i),c)$, initialement égale à Nil, pour sauvegarder le dernier support trouvé pour une affectation $v_i = a$ dans une contrainte c. Si par exemple $S(c) = v_i, v_j$, il n'est pas besoin de parcourir toutes les valeurs d_j à la recherche d'un support, il suffit de vérifier si le dernier support sauvegardé existe dans d_j. Dans le cas contraire, commencer la recherche d'un support à partir de la position du dernier support sauvegardé. Si aucun support n'a été trouvé, supprimer la valeur a de d_i car $v_i = a$ devient inconsistant-d'arc sinon, ajouter le nouveau support trouvé à la liste. La procédure Reviser-2001() est détaillée dans Algorithme 5.

2.4.3 Consistance de chemin (PC)

Une instance CSP P est dite consistante de chemin (path consistant) si toute instanciation partielle de deux variables peut être étendue en une instanciation partielle de trois variables. Cette propriété est valable uniquement pour les CSP binaires.
La consistance de chemin peut modifier la structure initiale d'un problème car le filtrage

Algorithme 5 Reviser-2001

Data : une contrainte $c_k = (v_i, v_j)$, une variable v_i.
Result : $modifier$

```
1  modifier ← faux for toute valeur a_i ∈ d_i do
2      if Last[c_k, (v_i, a_i)] ∉ d_y then
3          if ∄a_j ∈ d_j tel que a_j ≥ Last[c_k, (v_i, a_i)] ∧ (a_i, a_j) ∈ R_k then
4              enlever a_i de d_i
5              modifier ← vrai
6          else
7              Last[c_k, (v_i, a_i)] ← a_j
8          end
9      end
10 end
```

par cette consistance supprime des couples de valeurs des relations des contraintes et dans le cas ou la contrainte n'existe pas, elle sera alors ajoutée.

2.4.4 k-consistance

Une instance CSP P est dite **k-consistance**, avec $1 \leq k \leq n$ tel que $n = |vars(P)|$, s'il est possible d'étendre toute instanciation consistante de $k-1$ variables différentes avec une autre variable supplémentaire.
P est **fortement k-consistante** si elle est j-consistante $\forall j \leq k$.
La 1-consistante forte est équivalente à la consistance de noeud et la 2-consistante forte est équivalente à la consistance d'arc.

2.4.5 Arc Consistance Généralisée (GAC)

Une contrainte c_i est GAC (Genaralized Arc Consistent) si toutes les valeurs des domaines des variables du scope de c_i ($S(c_i)$) ont un support dans c_i. Si toutes les contraintes d'un CSP P sont GAC alors P est aussi GAC.
Les deux algorithmes AC3 et AC2001 sont généralisés pour le filtrage des CSPs n-aires (GAC3 et GAC2001). Ils peuvent être appliqués sur des contraintes table. Une contrainte table est définie par une liste de tuples de valeurs allouées (table positive) ou non allouées (table négative) pour un ensemble de variables.
Parmi les techniques proposées pour maintenir la consistance GAC dans les CSPs n-aires, on trouve celle basée sur la réduction tabulaire `STR` (Simple Tabular Reduction)[22] ainsi que ces différentes variantes `STR2 et STR3`.

2.4.6 Simple Tabular Reduction (STR)

Simple Tabular Reduction (STR) est une méthode proposée par Ullmann et al. [22] pour maintenir la consistance d'arc dans les contraintes tables. La méthode STR permet de réduire la taille de l'espace de recherche durant le processus de recherche en gardant que les tuples valides dans chaque contrainte table. Cela veut dire qu'à chaque fois qu'une valeur est supprimée du domaine d'une variable, tout les tuples invalides sont supprimés des contraintes table. C'est une technique qui facilite l'identification de la non satisfaction de la consistance d'arc des valeurs des variables et donc leur suppression. Pour ce faire, la méthode découpe la contrainte table $R(c)$ en deux parties, la première partie, appelée $tuples\ courant$, contient uniquement les tuples valides formant la $table\ coutante$ et la deuxième partie contient tout les tuples supprimés durant le processus de recherche. Pour gérer le déplacement entre tuples d'une contrainte table, la structure de données suivante est utilisée :

— $position[c_i]$: c'est un tableau permettant d'accéder aux différentes positions des tuples. Par exemple, $position[c_i][j]$ permet d'accéder au j^{ieme} tuple de c_i.

— $limite\_courante[c_i]$: c'est l'index du dernier tuple valide de la contrainte table c_i. C'est donc aussi le nombre de tuples dans la table courante.

— $limite\_etape[c_i]$: est représenté par un tableau, tel que $limite\_etape[c_i][p]$ consiste à la position du premier tuple invalide supprimé à l'étape p du processus de recherche. Si à une étape p donnée, aucun tuple n'a été supprimé, la variable $limite\_etape[c_i][p]$ prendra la valeur -1.

— $gacValues[c_i]$ [] est un tableau de taille égale au nombre de variables du scope de la contrainte c_i. Pour une variable v_j, $gacValues[c_i][v_j]$ contient l'ensemble des valeurs de d_j ayant un support dans c_i.

Example 2.4.2 *Dans cet exemple, nous allons montrer comment cette structure est utilisée pour maintenir la consistance d'arc dans une contrainte table. Nous décrivons les étapes de filtrage de la contrainte c_1 défini sur les variables $\{x_1, x_2, x_3\}$ dont les domaines respectifs $d_1 = d_2 = d_3 = \{0, 1, 2\}$. Les tuples autorisés pour cette contrainte sont représentés dans la figure 5.4. Cette figure est aussi une initialisation de la structure utilisée par STR.*

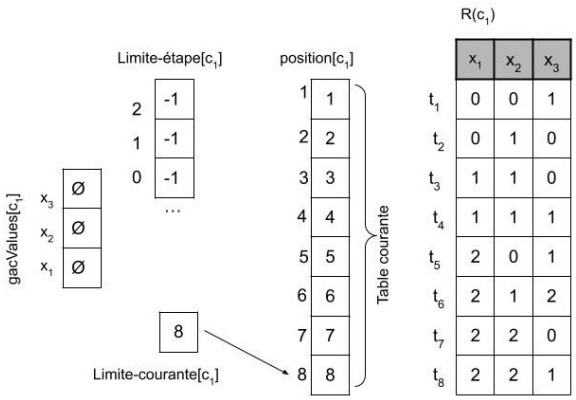


FIGURE 2.1 – Initialisation de la structure de données de STR.

Supposons que l'on supprime $x_2 = 1$, *donc* $d_2 = \{0,2\}$, *cela déclenche le processus de filtrage pour supprimer les tuples invalides. Le tuple* t_2 *est invalide il doit donc être supprimé de la contrainte table en permutant sa position avec celle du tuple correspondant à limite_courante. i.e.permuter* $position[c_1][2]$ *et* $position[c_1][8]$, *la variable limite_tape prendra la position du premier tuple invalide, qui est le tuple à la position* 8, *la valeur de limite_courante prendra la position du dernier tuple valide, i.e.*$position[c_1][8]$. *Figure 2.2(a). En passant au tuple* t_3, *ce tuple n'est pas supporté donc il doit aussi être supprimé. De la même manière que le tuple* t_2, *la structure doit être mise à jour, voir la figure 2.2(b). La figure 2.2(c) représente la structure mise-à-jour après la suppression de* t_4 *et enfin dans la FIgure 2.2(d), le dernier tuple invalide de cette étape (*t_6*) est supprimé, la valeur* $x_1 = 1$ *n'a plus de support donc elle est supprimée de* d_1, *la même chose aussi pour* $x_3 = 2$. *Cette mise à jour des domaines est représentée dans* $gacValues[c_1]$. *La table courant contient uniquement* 4 *tuples valides.*

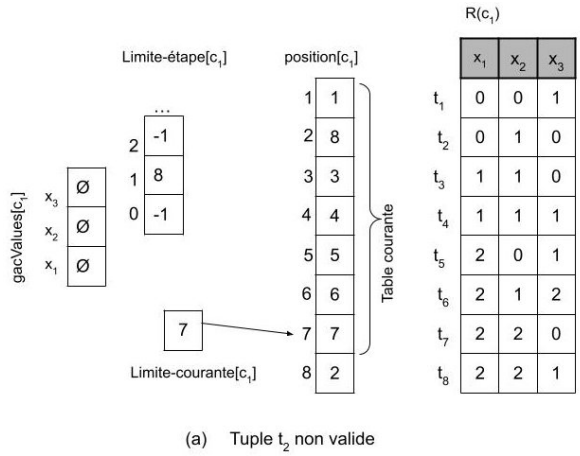


FIGURE 2.2 – Maintenir la consistance d'arc dans la contrainte table c_i après la suppression de $x_2 = 1$ à l'étape 1.

Dans la deuxième étape, $x_1 = 0$ est supprimé ce qui nécessite aussi la mise à jour de la structure. Le tuple t_1 n'est plus valide donc on le supprime en permuttant sa position avec le dernier tuple valide de l'étape précédente i.e. $position[c_1][1]$ et $position[c_1][4]$, de ce fait la valeur de $limite\_courante[c_1]$ prend la valeur 3. La Figure 2.3 représente l'état de la structure utilisée par STR à cette étape 2.

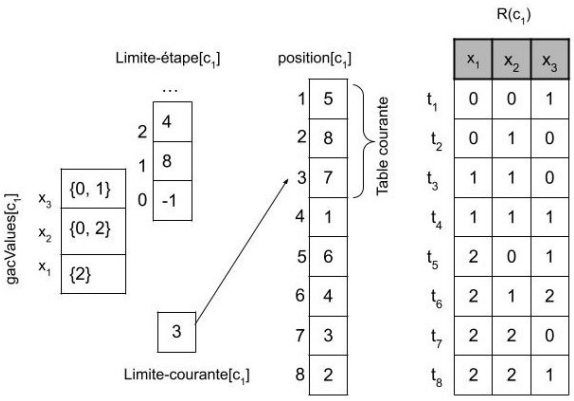


FIGURE 2.3 – Maintenir la consistance d'arc dans la contrainte table c_i après la suppression de $x_1 = 0$ à l'étape 2.

La compléxité temporelle au pire des cas de GACstr est de $O(r'd + rt')$ avec r représente le nombre total de variables, r' le nombre de variables non encore instanciées, t le nombre total de tuples et t' la taille de la table courante. La complexité spatiale dans le pire des cas est de $O(n + rt)$.

2.4.7 STR2

Lecoutre a proposé dans [23] deux optimisations pour la méthode STR[22]. La première optimisation consiste à arrêter la recherche de support pour les valeurs d'un domaine après avoir vérifié que celles-ci sont GAC-consistantes. Pour ce faire, l'auteur a utilisé une variable S^{sup} qui contient la liste des variables non encore instanciées et dont le domaine contient au moins une valeur pour laquelle aucun support n'a encore été trouvé. Ensuite, à chaque fois qu'un support est trouvé pour toutes les valeurs du domaine d'une variable, celle-ci est supprimé de S^{sup}. La deuxième optimisation consiste en l'introduction d'une autre variable S^{val} dont le contenu est l'ensemble des variables non encore instanciées et dont les domaines ont été modifiés lors d'un dernier appel de GACstr ainsi que la dernière variable instanciée si elle appartient au scope de c_i . Cela permet de réduire le nombre d'opérations à effectuer pour vérifier la validité des valeurs des domaines des variables, cela parce que si aucun retour arrière n'a été effectué et que

le domaine d'une variable x_i n'est pas modifié lors du dernier appel de GACstr, il est évident que les valeurs d_i restent toujours valides au prochain appel de GACstr.
Dans le cadre de cette thèse, c'est la version améliorée de STR2 qui est utilisée pour résoudre les CSPs compressés. Alors, avant de détailler la version améliorée, nous détaillons les étapes de STR2 dans Algorithme 6.

2.4.8 STR3

Le principe de la méthode STR3 est le même que STR1 et STR2 dans le fait qu'elle vise à réduire la taille de la table au fur et à mesure qu'on avance dans le processus de recherche. Cela, en supprimant les tuples invalides, mais elle utilise une représentation différente de la contrainte table. Tout comme GAC4, STR3 utilise un index pour chaque contrainte table qui permet un accès direct à un tuple supportant une valeur sans parcourir tout les tuples. Pour chaque contrainte table, la méthode utilise une table qui maintient pour chaque valeur du domaine d'une variable, les index des tuples supportés. Contrairement à STR1 et STR2, la structure utilisée par STR3 permet à ce qu'un tuple de la contrainte table soit visité une et une seule6 fois tout le long d'un chemin de l'arbre de recherche, allant de la racine jusqu'à la feuille. Réduisant ainsi la complexité temporelle.

Algorithme 6 STR2

Data : Réseau de contrainte P, une contrainte table c_i.
Result : consistance

```
consistance ← Vrai
S^sup ← ∅
if derVarAss ∈ scp(c_i) then
    S^val ← {derVarAss}
else
    S^val ← ∅
end
for chaque variable non instanciée x_j ∈ X do
    gacValeurs[x_j] ← ∅
    S^sup ← S^sup ∪ {x_j}
    if | d_j | ≠ taille_av[c_i][x_j] then
        ▷ taille_av[c_i][x_j] est la taille de d_j avant la dernière invocation de STR2
        S^val ← S^val ∪ {x_j}
        taille_av[c_i][x_j] ←| d_j |
    end
end
k ← 1
while k ≤ limite_courante[c_i] do
    t ← R(c_i)[position[c_i][k]
    if ValideT(c_i, S^val, t) then
        for chaque variable x_j ∈ S(c_i) | x_j ∈ S^sup do
            if t[x_j] ∉ gacValeurs[c_i][x_j] then
                gacValeurs[c_i][x_i] ← gacValeurs[c_i][x_j] ∪ {t[x_j]}
                if | d_j |=| gacValeurs[c_i][x_j] then
                    S^sup ← S^sup \ {x_j}
                end
            end
        end
    else
        supprimer t de la table courante et mettre à jour la valeur de
        limite_courante
    end
end
X' ← ∅
while (x_j ∈ S^sup)&(consistance = Vrai) do
    d_j ← gacValeurs[c_i][x_j]
    if gacValeurs[c_i][x_j] = ∅ then
        consistance ← Faux
    else
        X' ← X' ∪ {x_j}
        taille_av[c_i][x_j] ←| d_j |
    end
end
```

Algorithm 7 ValideT()

Data : contrainte table : c_i, liste de variables : S^{val}, tuple : t.
Result : $valide$

```
1 valide ← true j ← 1 while (j ≤| S^sup |)&(valide = true) do
2 |   if t[x_j] ∉ d_j then
3 |   |   valid ← false
4 |   end
5 end
```

Chapitre 3

Énumération des ensembles de motifs (itemsets)

Découvrir de nouvelles connaissances à partir d'une source de données est un domaine de recherche auquel s'intéresse une grande communauté de chercheurs pour son importance dans de nombreuses applications.
L'objectif principal de la fouille de données, à base de motifs à partir d'une base de données, est la découverte des nouvelles informations cachées [24]. Nous présentons dans ce chapitre les notions de bases sur les ensembles de motifs et les différentes méthodes de fouille de données à base de motifs à partir des bases de données transactionnelles.

3.1 Contexte transactionnel

Définition 3.1.1 *(base de données transactionnelle)*
Une base de données transactionnelle $\mathscr{TD}$ est définie par un triple $(\mathscr{I},\mathscr{R},\mathscr{T})$ ou $\mathscr{I} = \{i_1,\cdots,i_n\}$ est un ensemble d'attributs appelés items, $\mathscr{T} = \{t_1,\cdots,t_m\}$ un ensemble de de transactions tel que t_i est un sous-ensemble de $\mathscr{I}$ et $\mathscr{R}$ représente une relation $\mathscr{T} \times \mathscr{I}$ qui permet d'indiquer si un items i est présent dans une transaction t.

Example 3.1.1 *La table 5.1.1 est une base transactionnelle contenant dix (10) transactions, étiquetées $t_1,t_2,\cdots,t_10$, décrites par l'ensemble des items A,B,C,D,E.*

ID	Items				
t_1	A	B	C	D	E
t_2	A	B	C		e
t_3	A	B		D	E
t_4	A	B		D	
t_5		B	C	D	E
t_6		B	C	D	
t_7		B	C		E
t_8		B		D	E
t_9			C	D	E
t_{10}			C		E

TABLE 3.1 – Base de données transactionnelle $\mathscr{TD}$.

Définition 3.1.2 *(itemset, taille)*

Un itemset u est un sous-ensemble de $\mathscr{I}$. La taille de u est égale au nombre d'items composant u.

$$\forall u \subset \mathscr{I}, taille(u) = \mid u \mid \tag{3.1}$$

Example 3.1.2 *Soit l'ensemble $\mathscr{I}$ des items de la base transactionnelle de l'exemple 3.1.1, $u = abd$ est un itemset et sa taille est égale à trois : $taille(u) = 3$.*

Définition 3.1.3 *(Couverture d'un item/itemsets)*

Étant donné un item/itemset u et un ensemble de transactions $\mathscr{T}$, la couverture de u, notée $cover(u)$, est l'ensemble des transactions supportant u.

On dit qu'une transaction t supporte un item (itemset) u si $\forall i \in u, t\mathscr{R}i$ (ou $u \subset t$).

$$cover(u) = t \in \mathscr{T} \mid u \subset \mathscr{I} \tag{3.2}$$

Example 3.1.3 *Soit la base transactionnelle $\mathscr{TD}$ de l'exemple 3.1.1 et soit $u = abd$ un itemset de cette base. Le support de u = $\{t_1, t_3, t_4\}$.*

Définition 3.1.4 *(Fréquence) Étant donné un itemset u, ou I est l'ensemble des items le constituant et T l'ensemble des transactions le couvrant, la fréquence de u est égale à la cordialité de sa couverture.*

$$freq(u) = \mid cover(u) \mid . \tag{3.3}$$

Définition 3.1.5 *(L'aire d'un itemset) L'aire d'un itemset i est définie comme étant le produit de sa taille $\mid u \mid$ et sa fréquence $freq(u)$.*

$$Aire(u) = \mid u \mid \times freq(u). \tag{3.4}$$

Définition 3.1.6 *L'espace de recherche des itemset est défini par le langage $\mathscr{L}_{\mathscr{I}}$ constitué de tous les sous-ensembles non vides de $\mathscr{I}$.*

$$\mathscr{L}_{\mathscr{I}} = 2^{\mathscr{I}} \setminus \emptyset. \tag{3.5}$$

3.2 Contraintes locales pour énumérer des itemsets

Le nombre d'itemsets que peut contenir une base de données transactionnelle peut être trop important, ce qui peut rendre leur processus d'extraction ainsi que d'exploitation très coûteux. Pour améliorer la qualité des itemsets à énumérer et réduire le coût de

leur exploitation, l'extraction des itemsets sous contrainte est généralement considérée comme une solution. Une contrainte locale est une contrainte définie sur des itemsets individuels.

Les itemsets fréquents est un exemple de contrainte locale qui permet d'énumérer uniquement les itemsets dont la fréquence est supérieure ou égale à un seuil définie au préalable. Ce seuil est appelé *seuil minimum* S_{min}. Un autre exemple de contrainte locale est celui portant sur l'*aire* des itemsets, ou il faut énumérer les itemsets dont l'aire est supérieure à une valeur donnée.

Example 3.2.1 *En considérant la table transactionnelle $\mathscr{TD}$ de l'exemple 3.1.1 et un seuil minimum $S_{min} = 3$, les itemsets fréquents de cette table sont : AB $\langle 4 \rangle$, AD $\langle 3 \rangle$, AE $\langle 3 \rangle$, BC $\langle 5 \rangle$, BD$\langle 6 \rangle$, BE$\langle 6 \rangle$, CD $\langle 4 \rangle$, CE $\langle 6 \rangle$, DE $\langle 5 \rangle$.*

Parmi ces itemsets, ceux dont l'aire est supérieure ou égale à 10 sont : BC $\langle 5 \rangle$, BD$\langle 6 \rangle$, BE$\langle 6 \rangle$, CE $\langle 6 \rangle$, DE $\langle 5 \rangle$.

3.3 Élagage des espaces de recherche et représentation des itemsets sous une forme condensée

Pour un ensemble $\mathscr{I}$ d'items, la taille de l'espace de recherche des itemsets est de $2^n - 1$ avec $n = \mid \mathscr{I} \mid$. La recherche des itemsets dans de tel espaces de recherche de très grande taille est d'un point de vu algorithmique très difficile. Généralement, il est même impossible de parcourir tout l'espace de recherche. Pour cela *Mitchell et al.* [25] ont proposé une relation de spécialisation pour une structuration du langage $\mathscr{L}_{\mathscr{I}}$ et pour faciliter la recherche des itemsets.

Définition 3.3.1 *(Spécialisation des itemsets)*

La spécialisation, notée $\preceq$, est une relation d'ordre partiel définie sur des itemsets du langage $\mathscr{L}_{\mathscr{I}}$. On dit qu'un itemset u_1 est une spécialisation (ou il est plus spécifique) d'un autre itemset u_2 si $u_2 \preceq u_1$. Inversement, on parle d'une généralisation.

Un exemple courant de relation de spécialisation entre deux itemsets est la relation d'inclusion. Si un itemset u_1 est inclus dans un itemset u_2, alors on dit que u_2 est une spécification de u_1.

Pour un élagage de l'espace de recherche dans le but de faciliter la recherche d'itemsets, des propriétés de monotonie et d'anti-monotonie de certaines contraintes sont exploitées.

Définition 3.3.2 *(Anti-monotonie) Une contrainte $c(u_i, \mathscr{T})$ est dite anti-monotone, si et seulement si :*

$$\forall u_i, u_j \in \mathscr{L}_{\mathscr{I}}, u_i \subseteq u_j \Rightarrow (c(u_j, \mathscr{T}) \Rightarrow c(u_i, \mathscr{T})). \tag{3.6}$$

La contrainte du seuil de fréquence minimum S_{min} est un exemple de contrainte anti-monotone. Si un itemset u satisfait la contrainte c alors tout sous-itemsets de u satisfont aussi cette contrainte.

3.3.1 Frontière d'un ensemble d'itemsets

Selon [25] [26], une contrainte anti-monotone peut découper un espace de recherche en deux sous-espaces. Étant donné un itemset u respectant la contrainte anti-monotone, toutes les spécialisation de cet itemset satisfont la contrainte (premier sous-espace) et toutes les généralisations de u ne satisfont pas la contrainte c (deuxième sous-espace).

Définition 3.3.3 *(frontière positive- itemsets maximaux)*
Soit $\mathscr{S}$ un ensemble d'itemsets de l'espace de recherche $\mathscr{L}_{\mathscr{I}}$ satisfaisant une contrainte anti-monotone c, on note par $\mathscr{B}d^+$ la frontière positive de $\mathscr{S}$ qui est un ensemble d'itemsets tels que :

$$\mathscr{B}d^+(\mathscr{S}) = \{u_i \in \mathscr{S} \mid \nexists u_j (u_j \in \mathscr{S} \wedge u_i \prec u_j)\} \tag{3.7}$$

L'ensemble des itemsets dans la frontière positive sont donc les itemsets maximaux de $\mathscr{S}$.

Définition 3.3.4 *(frontière négative-itemsets minimaux)*
Soit $\mathscr{S}$ un ensemble d'itemsets de l'espace de recherche $\mathscr{L}_{\mathscr{I}}$ ne satisfaisant une contrainte anti-monotone c, on note par $\mathscr{B}d^-$ la frontière négative de $\mathscr{S}$ qui est un ensemble d'itemsets tels que :

$$\mathscr{B}d^-(\mathscr{S}) = \{u_i \notin \mathscr{S} \mid \nexists u_j (u_j \notin \mathscr{S} \wedge u_i \prec u_j)\} \tag{3.8}$$

L'ensemble des itemsets dans la frontière négative sont donc les itemsets minimaux de $\mathscr{S}$.

3.3.2 Représentation condensée d'un ensemble des itemsets

La définition d'une contrainte locale est une solution proposée pour améliorer la qualité des itemsets à énumérer, mais dans certaines situations, la définition d'une contrainte

locale n'est pas suffisante. Par exemple lorsque le nombre d'itemset qui satisfont la contrainte est très grand. Cela rend le processus de recherche des itemsets très coûteux en temps et en espace de stockage. Pour cela, Mannilla et toivonen [26] ont introduit la représentation condensée des itemsets pour faciliter le processus d'extraction et réduire la taille de l'ensemble des itemsets obtenus.

Il existe deux types de representations condensées des ensembles d'itemsets :

— représentation exacte : dans ce type de représentation, il est possible d'avoir pour chaque itemsets sa valeur de fréquence exacte.

— représentation approximative : dans ce type de représentation, on obtient juste une approximation de la valeur de fréquence des itemsets.

Définition 3.3.5 *(représentation condensée)*
Étant donné un ensemble $\mathscr{S}$ d'itemsets selon une contrainte locale c, une représentation condensée de $\mathscr{S}$ est sous ensemble $\mathscr{S}_{con}$ de $\mathscr{S}$ ayant une cardinalité inférieure à celle de $\mathscr{S}$, tel que $\mathscr{S}_{con} \subseteq \mathscr{S}$ et le reste des éléments de $\mathscr{S} \setminus \mathscr{S}_{con}$ peuvent être générés sans accéder à la base de données mais directement à partir de l'ensemble $\mathscr{S}_{con}$.

Parmi les représentations condensées proposées pour les itemsets fréquents, on peut trouver des représentations par des itemsets fermés (clos) [27] [28] [29], représentations par les itemsets maximaux [30] [31] [32] et représentations par itemsets libres [33].

Représentation condensée par itemsets maximaux

La représentation condensée par les itemsets maximaux consiste en l'ensemble des itemsets maximaux fréquents formant la frontière positive de l'espace de recherche.

Définition 3.3.6 *(itemset maximal fréquent)*
Étant donné un seuil minimal de fréquence S_{min}, un itemset u est dit itemset maximal fréquent, noté $maximal_{\mathscr{T}}(u)$, si et seulement si, sa fréquence est supérieure ou égale à S_{min} et que tous ses sur-ensembles sont moins fréquents :

$$maximal_{\mathscr{T}}(u) \Longleftrightarrow freq(u) \geq S_{min} \wedge \forall i \in \mathscr{I} \setminus u : freq(u \cup \{i\}) \leq S_{min}. \tag{3.9}$$

La représentation condensée par itemsets maximaux est approximative car elle permet d'avoir uniquement une borne inférieur de fréquence des itemset fréquents non maximaux et non pas de dériver la fréquence exacte de chaque itemsets non maximal.

Example 3.3.1 *Considérons la base de données transactionnelle $\mathscr{T}\mathscr{D}$ représentée dans la table 5.1.1, et soit une valeur de seuil minimal de fréquence $S_{min} = 3$.*
Les itemsets fréquents de $\mathscr{T}\mathscr{D}$ sont : $\{A\}:4$, $\{B\}:8$, $\{C\}:7$, $\{D\}:7$, $\{E\}:8$, $\{AB\}:4$, $\{AD\}:3$, $\{AE\}:3$, $\{BC\}:5$, $\{BD\}:6$, $\{BE\}:6$, $\{CD\}:4$, $\{CE\}:6$, $\{DE\}:5$, $\{ABD\}:3$, $\{ABE\}:3$, $\{BCD\}:3$, $\{BCE\}:4$, $\{BDE\}:4$, $\{CDE\}:3$.
La représentation condensée par itemsets fréquents maximaux $\mathscr{F}_{con}(\mathscr{T}\mathscr{D},3)$ est : $\{ABD\}:3$, $\{ABE\}:3$, $\{BCD\}:3$, $\{BCE\}:4$, $\{BDE\}:4$, $\{CDE\}:3$.
A partir de $\mathscr{F}_{con}(\mathscr{T}\mathscr{D},3)$, il n'est pas possible de générer les valeurs de fréquences exactes de tous les itemsets fréquents. Par exemple, à partir de l'itemset maximal fréquent $\{ABE\}$, on ne peut pas générer les fréquences des sous-itemsets fréquent de cet itemset. Mais en se basant sur la propriété d'anti-monotonie, on peut déduire que par exemple le sous itemset $\{AE\} \subseteq \{ABE\}$ est fréquent : $freq(\{AE\}) \geq freq(\{ABE\})$.

Représentation condensée par itemsets fermés (clos)

La représentation condensée par les itemsets clos, notée $\mathscr{F}_{clos}$, consiste en l'ensemble des itemsets clos formant la frontière positive de l'espace de recherche.

Définition 3.3.7 *(itemset clos)*
Étant donné un seuil minimal de fréquence S_{min}, un itemset u est dit itemset clos fréquent, noté $clos_{\mathscr{T}}(u)$, si et seulement si, sa fréquence est supérieure ou égale à S_{min} et que tous ses sur-ensembles ont une fréquence strictement inférieure à celle de u :

$$clos_{\mathscr{T}}(u) \Longleftrightarrow freq(u) \geq S_{min} \wedge \forall i \in \mathscr{I} \setminus u : freq(u \cup \{i\}) \leq freq_u. \tag{3.10}$$

Selon [30], la représentation condensée par itemsets clos est une représentation exacte, du fait qu'elle permet de générer de façon exacte l'ensemble des itemsets fréquents ainsi que leur fréquences respectives à partir de l'ensemble des itemsets clos.
Soit un itemsets u dont on souhaite connaître sa fréquence :

- Si aucun itemset de l'ensemble des clos n'est un spécialisation de u, alors u n'est pas fréquent,
- si deux itemsets u' et u'' sont deux spécialisations de u dans l'ensemble des clos, alors la fréquence de u est celle de la spécialisation la moins fréquente.

De nos jour, la méthode la plus efficace et la plus utilisée pour le calcul de la représentation condensée par itemsets clos est celle proposée dans [34].

Example 3.3.2 *Soit Table 3.2 représentant un ensemble d'itemsets fréquents clos énumérés à partir une base de données transactionnelle en fixant la valeur de seuil minimal de fréquence à* $S_{min} = 2$.

À partir de l'ensemble de ces itemsets fréquents clos, il possible d'avoir la fréquence d'autres itemsets fréquents non clos.

Par exemple, si on souhaite connaître la fréquence de l'itemets $\{AE\}$ *qui ne figure pas dans la table 3.2, et qui n'est donc un clos, on cherche s'il possède des spécialisation dans cette table. L'itemset fréquent clos* $u_{22} = \{ABE\}$ *est une spécialisation de* $\{AE\}$ *est il est d'une valeur de fréquence de* 2*, on déduit donc que* $freq(\{AE\}) = 2$ *et il est couvert par les tuples* $\{t_5, t_6\}$

ID_{clos}	Itemsets fréquents clos	Couvertures
u_0	$\{A\}$	$\{t_2,t_3,t_4,t_5,t_6,t_7,t_8,t_9,t_{10},t_{12}\}$
u_1	$\{B\}$	$\{t_3,t_4,t_5,t_6,t_7,t_8\}$
u_2	$\{C\}$	$\{t_7,t_8,t_9,t_{10},t_{11}\}$
u_3	$\{D\}$	$\{t_1,t_2,t_3,t_4,t_{12}\}$
u_4	$\{E\}$	$\{t_1,t_5,t_6\}$
u_5	$\{F\}$	$\{t_2,t_{11},t_{12}\}$
u_6	$\{G\}$	$\{t_1,t_3,t_6,t_7,t_9,t_{11},t_{12}\}$
u_7	$\{H\}$	$\{t_1,t_2,t_5,t_7,t_8,t_{10},t_{11}\}$
u_8	$\{AB\}$	$\{t_3,t_4,t_5,t_6,t_7,t_8\}$
u_9	$\{AC\}$	$\{t_7,t_8,t_9,t_{10}\}$
u_{10}	$\{AD\}$	$\{t_2,t_3,t_4,t_12\}$
u_{11}	$\{AFH\}$	$\{t_2,t_{11}\}$
u_{12}	$\{AG\}$	$\{t_3,t_6,t_7,t_9\}$
u_{13}	$\{AH\}$	$\{t_2,t_5,t_7,t_8,t_{10}\}$
u_{14}	$\{CH\}$	$\{t_7,t_8,t_{10},t_{11}\}$
u_{16}	$\{DH\}$	$\{t_1,t_2\}$
u_{17}	$\{EH\}$	$\{t_1,t_5\}$
u_{18}	$\{FH\}$	$\{t_2,t_{11}\}$
u_{19}	$\{GH\}$	$\{t_1,t_7,t_{11}\}$
u_{20}	$\{ABC\}$	$\{t_7,t_8\}$
u_{21}	$\{ABD\}$	$\{t_3,t_4\}$
u_{22}	$\{ABE\}$	$\{t_5,t_6\}$
u_{23}	$\{CG\}$	$\{t_7,t_9,t_{11}\}$
u_{22}	$\{ACG\}$	$\{t_7,t_9\}$
u_{23}	$\{ABG\}$	$\{t_3,t_6,t_7\}$

TABLE 3.2 – Liste d'itemsets fréquents clos avec leur couvertures extraits d'une base transactionnelle avec une fréquence minimal $S_{min} = 2$.

Représentation condensée par itemsets libres

selon [35], l'ensemble des itemsets fréquents libres et leur bordure négative forment une représentation condensée exacte des itemsets fréquents. Étant donné un seuil minimal de fréquence S_{min}, un ensemble d'itemsets libres fréquents $\mathscr{F}_{libre}(T, S_{min})$) ainsi que leur frontière négative $\mathscr{B}d^{-}(\mathscr{F}_{libre}(T, S_{min}))$, la fréquence exacte d'un itemsets u peut être déduite comme suit :

— u n'est pas fréquent si $\exists\, u' \in \mathscr{B}d^{-}(\mathscr{F}_{libre}(T, S_{min}))$

— la fréquence de u' est égale à la plus petite fréquence des itemsets libres fréquents dans u.

Example 3.3.3 *Considérons la table 5.1.1 et un Smin* $= 3$. *Les itemsets fréquents libres de cette table sont :* $\mathscr{F}_{libre}(\mathscr{TD}, 3) = \{\{A\} : 4, \{B\} : 8, \{C\} : 7, \{D\} : 7, \{E\} : 8, \{AB\} : 4, \{ABE\} : 3, \{ABD\} : 3, \{BCD\} : 3, \{BCE\} : 3, \{BDE\} : 3, \{CDE\} : 3, \{CD\} : 4, \{DE\} : 5, \{BC\} : 5, \{AD\} : 3, \{AE\} : 3, \{BD\} : 6, \{BE\} : 6, \{CE\} : 6\}$.

Dans la suite de nos travaux, nous nous intéressons aux différentes représentations condensées basées sur les trois types de motifs.

3.3.3 FP-Tree : Structure d'arbre pour la recherche de motifs fréquents

Frequent Patter Tree (TP-Tree) est une structure qui permet de représenter un ensemble de transactions sous forme d'un arbre de préfixes. Tel que, chaque arrête reliant deux noeuds de l'arbre représente un item et chaque noeud représente la fréquence de l'item. Une branche est un motif fréquent tel que les items le composant sont ordonnés dans un ordre décroissant de leur fréquence. Cette représentation sous forme d'arbre permet aux préfixes partagés par plusieurs transactions d'être stockés une seule fois sous forme d'une seule branche. La structure FP-Tree a été proposé pour dans le but d'extraire des motifs fréquents sans générer des motifs fréquents candidats (FP-Growth).

Étant donné un ensemble de transactions avec un ensemble d'items et un seuil de fréquence minimal S_{min}, une table d'en-tête contenant les items triés dans un ordre décroissant de leur fréquences est d'abord créée. Ensuite, chaque transaction est ordonnée dans un ordre décroissant des items. Les items fréquents de chaque transaction sont ensuite insérés dans le FP-Tree sous forme d'une branche. Si un motif partage le même préfixe avec un motif déjà inséré dans l'arbre, cette partie de l'arbre sera partagée tel que la fréquence de chaque noeud sera incrémentée de 1. Table 3.3 représente la table d'en-

tête correspondante à la table 5.1.1. Dans Table 3.4, les transaction de la table 5.1.1 sont

item	fréquence
B	8
E	8
C	7
D	7
A	4

TABLE 3.3 – Table d'en-tête correspondante à $\mathscr{TD}$.

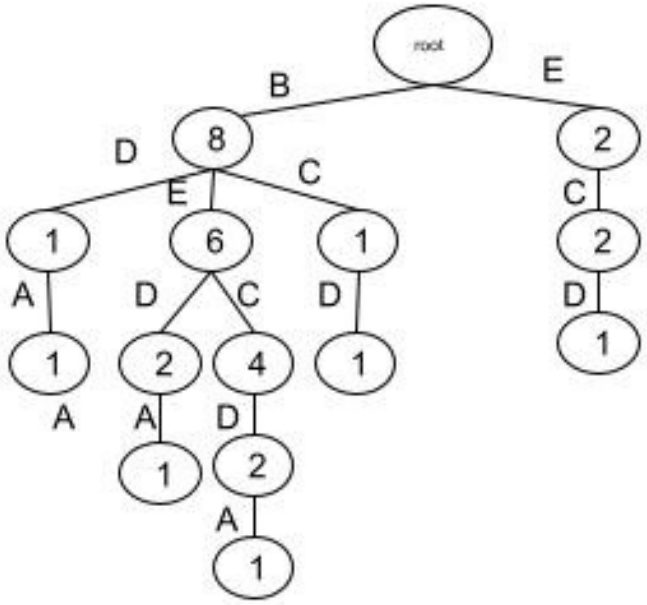
triées dans un ordre décroissant des fréquences des items.

ID			Items		
t_1	B	E	C	D	A
t_2	B	E	C	A	
t_3	B	E	D	A	
t_4	B	D	A		
t_5	B	E	C	D	
t_6	B	C	D		
t_7	B	E	C		
t_8	B	E	D		
t_9	E	C	D		
t_{10}	E	C			

TABLE 3.4 – Table $\mathscr{TD}$ triée l'ordre croissant des fréquences des items.

Figure 7.4 est un FP-tree créé pour la table transactions $\mathscr{TD}$.

FIGURE 3.1 – FP-tree correspondant à la table transactionnelle $\mathscr{TD}$



3.3.4 LCM : Énumération des motifs fréquents fermés et maximaux

Linear time Closed item set Miner (LCM) est un algorithme proposé par [36] pour énumérer des motifs fréquents fermés en un temps linéaire. Une relation parent-enfant est établie entre les motifs fréquents fermés créant ainsi un arbre a explorer avec des chemins transversaux composés de tout les motifs fréquents fermés. Cet algorithme est obtenu à partir des algorithmes d'énumération de cliques maximales biparties [37] [38]. Lors du processus d'exploration, LCM utilise une technique appelé *occurrence deliver* pour accélérer la mise à jour de la fréquence des motifs fermés. Cela en calculant de façon parallèle la couverture de tout les successeurs d'un motif courant durant une seule itération de balayage sur l'ensemble actuel des couvertures. Une autre technique appelée *diffsets* [39] est utilisée pour minimiser l'utilisation de la mémoire des calculs intermédiaires.

A partir de LCM, les auteurs ont dérivé un algorithme appelé LCMmax pour le calcul des motifs fréquents maximaux. Comme un motif fréquent maximal est aussi un motif fermé, lors du processus de recherche, LCMmax recherche tout les motifs fréquents fermés et retourne uniquement les maximaux. Cela en se basant sur le principe de backtracking pour effectuer des tests de maximalité des motifs fermés trouvés. L'algorithme utilise une technique d'élagage pour couper les branches non nécessaires pour la récursivité. Cette technique est basée sur la réorganisation des indices des motifs de taille 1 à chaque itération.

Chapitre 4

Méthodes de l'état de l'art

Dans notre étude des méthodes de l'état sur la compression des Big Data, on s'est intéressé au domaine des CSPs (Problèmes de satisfaction de contraintes tables). Plus précisément à la compression des contraintes tables.

Les principaux objectifs derrière la compression des contraintes tables sont :

— réduire la taille de l'espace mémoire requis pour la représentation des contraintes tables,

— Réduire la taille de l'espace de recherche de solution en évitant l'exploration d'un même espace de recherche plusieurs fois.

— Réduire le temps de résolution des CSPs (temps de recherche de solutions).

Plusieurs méthodes ont été proposées dans la littérature pour la compression des contraintes tables. On peut principalement trouver des méthodes basées sur :

— la structure d'arbre et diagramme de décision [40], [**mdd**], [41].

— la réduction tabulaire (STR) ,

— Tuples segmentés ;

— Extraction de motifs fréquents.

Dans ce chapitre nous présentons quelques travaux proposés dans la littérature sur la compression contraintes tables.

4.1 Méthodes de compression basées sur la structure d'arbre et diagramme de décision

Katsirelos and Walsh [40] sont les premiers à proposer une méthode de compression pour les contraintes tables. Ils représentent la contrainte table à compresser à l'aide d'un arbre de décision sous forme d'une disjonction de tuples. L'arbre de décision construit est ensuite exploité afin d'extraire des tuples compressés, appelés *c-tuple*, qui seront utilisés pour définir une nouvelle représentons de la contrainte table, sous forme d'une conjonction de *c-tuple*. Cette forme de compression permet de réduire la taille de la contrainte

table en mémoire. Une autre forme de compression de contraintes tables a été proposée par **Cheng et *al.***[**mdd**]. Cette méthode est basée sur l'utilisation de MDD (Multivalued Decision Diagram) qui est une forme de "trie" où les redondances de préfixes sont supprimées. La représentation d'une contrainte table sous forme d'un arbre occupe moins d'espace mémoire par rapport à une table. C'est pourquoi les auteurs ont proposé d'exploiter cette structure pour la vérification et recherche de support. En plus de la représentation sous-forme d'arbre, les auteurs ont proposé de fusionner les sous-arbres identiques dans l'arbre de décision donnant ainsi un graphe orienté acyclique (DAG) appelé MDD. `MDDC` [41] et `MDD4R` [42] sont les deux principaux algorithmes utilisant les MDD comme structure de données. Durant la propagation, la méthode `MDDC` [41] parcours le diagramme de décision en profondeur d'abord afin de vérifier la consistance sans modifier le MDD. Contrairement à la méthode `MDD4R` [42] qui maintient dynamiquement le MDD en supprimant les nœuds et les arêtes qui n'appartiennent pas à une solution.

4.1.1 Avantages

La compression des contraintes tables à base d'arbre et diagramme de décision permet de réduire la taille de l'espace mémoire en supprimant les redondances de préfixes. Cette forme de compression permet aussi d'accélérer le processus de recherche de solution du fait que les redondances sont supprimées durant le processus de compression et donc garantir que les espaces redondants ne soient pas explorés.

4.2 Méthodes de compression basées sur STR (Simple Tabular Reduction)

La méthode STR [22], en plus d'être une méthode de filtrage, elle est aussi considérée comme une méthode de compression par le fait qu'elle permet de réduire la taille de l'espace de recherche durant le processus de recherche de solution. Cela, en supprimant de cet espace les tuples non valides (qui ne peuvent pas contenir de solution). Plusieurs variantes de STR ont été proposées, principalement STR2 [23] et STR3 [43]. (voir 2.4.6 pour plus de détails sur ces méthodes).

4.3 Méthode de compression basée sur une représentation binaire

Wang et al. [44] ont proposé une représentation des contrainte table sous forme de table à bits. Cela, en créant une table dual (dual table) où pour chaque item sont sauvegardés les indices de tuples qui le vérifient (les supports). Ensuite, la table de contraintes est découpée en sous-tables, chaque sous-table est composée de w tuples (uniquement indices des tuples). Puis, pour chaque item et pour chaque sous-table va correspondre un vecteur à n bits (n : le nombre de tuples dans la sous-table) qui indique si un tuple est un support (1) ou non (0) d'un item. On parle alors de bit-support. Compact-table (CT) [45] est une autre approche qui combine la représentation des supports sous forme de vecteurs de bits, support de résidus (residue support) dont le principe est de sauvegarder à chaque fois les supports d'une solution partielle et de continuer le processus de recherche en considérant les supports de résidus afin d'éviter des vérifications inutiles.

4.3.1 Avantages

Cette forme de compression permet de réduire la taille de l'espace de recherche durant le processus de recherche de solution, en garantissant d'éliminer les tuples ne pouvant pas faire partie de la solution. En plus de ça, le fait d'utiliser des vecteurs de bits pour la représentation des supports, permet d'accélérer le processus de recherche en évitant de parcourir à chaque fois la table de contrainte.

4.3.2 Inconvénients

Bien que la représentation binaire possède des avantages, mais elle a aussi des inconvénients. Le principal inconvénient de cette forme de compression est que, en plus de la table de contrainte, d'autre tables (dont la table dual et la table à bit) seront créées, ce qui peut nécessiter plus d'espace mémoire pour les stocker.

4.4 Méthode de compression basée sur segmented tuples

Audemard et al. [46] ont introduit la notion de table segmentée (segmented table) qui est considérée comme une généralisation de la compression. Une table segmentée est composée d'un ensemble de tuples segmentés. Ou chaque tuple segmenté peut être présenté sous forme d'une valeur universelle (*), d'une valeur ordinaire ou une sous table.

Les auteurs ont aussi proposé une méthode pour le filtrage des tables segmentées.

4.5 Méthodes de compression basée sur l'extraction de motifs

Jabbour et al. [47] ont proposé une méthode basée sur les motifs fréquents fermés pour la compression des contraintes tables. Ils ont proposé deux nouvelles règles de réécriture du réseau de contrainte ainsi que la taille des contraintes table tout en préservant la structure de la contrainte table d'origine. Pour cela, ils ont utilisé les itemssets fréquents fermés. La première règle de réécriture de contraintes consiste à représenter le réseau de contraintes sous forme d'une table transactionnelle, où les attributs correspondent au scope des contraintes et les valeurs aux valeurs des variables de scope. Les itemsets fréquents fermés correspondent aux variables les plus partagées par les scopes des contraintes et non pas aux paire (variable, valeur). Ensuite, une deuxième règle de réécriture est appliquée pour extraire de chaque contrainte table une séquence de tuples. Les motifs fréquents fermés sont ensuite utilisés pour écrire les contraintes tables sous format compressé. Gharbi et al. [1] quant à eux, ils ont proposé une méthode de compression, appelée Sliced-table, est basée sur l'extraction de motifs fréquents à partir d'une contrainte table en exploitant un arbre de préfixe (FP-Tree) et une fonction de gain. La méthode prend en entrée la contrainte table à compresser et retourne une contrainte table compressée constituée d'un ensemble de fragments et une table par défaut contenant les tuples non compressés. Pour se faire, la méthode commence par représenter la contrainte table sous forme d'un FP-Tree. Ensuite, afin de décider si un motifs fréquent u est pertinent pour la compression, les auteurs ont proposé d'appliquer une fonction qui calcul le gain en nombre d'items qu'on peut obtenir en compressant n tuples avec u aux gain que l'on peut obtenir en compressant m tuples (avec m <=n) en utilisant un super motif de u. La fonction de calcul de gain est donnée comme suit : $|u| * (freq(u) - 1)$ avec u le motif fréquent et $freq(u)$ sa valeur de fréquence. Une fois les motifs fréquents énumérés, construire pour chaque motif fréquent un fragment lui correspond, ou chaque fragment est composé d'un motif fréquent et une sous table contenant les parties des tuples de la couverture du motifs après la suppression du motif. Les auteurs ont aussi proposé une méthode, appelé STR-Slice basée sur STR2 pour la résolution des CSPs compressés.

4.5.1 Avantages

Les deux méthodes ont l'avantage de réduire la taille de l'espace mémoire requis pour la représentation des contraintes tables. En plus de ça, la méthode proposée par Gharbi et al. [1] permet de réduire l'espace de recherche durant le processus de résolution et accélère le temps résolution en éliminant les fragments, les motifs et les sous-tuples non valides.

4.5.2 Inconvénients

Jabour et al. [47] n'ont pas proposé une méthode de résolution des CSPs compressés donc on ne connaît l'impact de la compression sur le processus de résolution qui reste un point très important vue que l'un des objectifs de compression des CSPs étant d'améliorer le processus de résolution. Quant à la méthode Sliced-table, elle calcul le nombre d'items avec lequel réduire la taille de f tuples en les compressant avec un motif fréquent u et le compare au nombre d'items avec lequel réduire la taille de f' tuples (avec $f > f'$) en les compressant avec un super motif u' de u. Le gain en nombre d'items est calculé par rapport à un nombre différent de tuples, conduisant ainsi à une approximation moins précise du gain de compression.

4.6 Synthèse

Dans cette section, on a définit un ensemble de critères pour comparer les travaux de l'état de l'art. Les critères de comparaison sont comme suit :

- Espace mémoire : si la méthode de compression permet de réduire la taille de l'espace mémoire requis pour la représentation des contraintes tables ;
- Espace de recherche : si la méthode de compression proposée permet de réduire la taille de l'espace de recherche au fur et à mesure qu'on avance dans le processus de résolution ;
- Temps de résolution : si la méthode de compression proposée permet d'accélérer le processus de résolution ;
- Niveau de compression : la compression des contraintes table se fait avant le début ou durant du processus de résolution.

La table 4.1 représente un tableau comparatif des différents méthodes de l'état de l'art selon les critères définits ci-dessus. Les lignes de la table represente les travaux de l'etat

de l'art et les colonnes représentent les critères. Si la méthode de l'état de l'art répond à un critère on utilise X sinon / .

TABLE 4.1 – Tableau comparatif des méthodes de l'etat de l'art

Methode	Reduction éspace mémoire	Réduction espace de recherche	Réduction de temps de résolution	Niveau de compression
c-tuples [40]	X	/	/	avant
mddc et mdd4r [41]	X	X	X	avant
STR [22]	/	X	X	durant
STR2 [23]	/	X	X	durant
STR3 [43]	/	X	X	durant
segmented table. [46]	X	X	X	avant
Jabbour et al. [47]	X	/	/	avant
Sliced-table. [1]	X	X	X	Avant et durant

Comme on peut le remarquer dans ce tableau, les méthodes de l'état de l'art compressent les contraintes table avant ou durant le processus de résolution, sauf la méthode de Gharbi et al. [1] qui possède deux niveaux de compression. Sliced-table [1] commence par compresser les contraintes table en réduisant ainsi la taille l'espace mémoire requis pour la représentation en mémoire des contraintes table. Puis, la résolution du CSP compressé se fait avec la méthode STR-Slice (une adaptation de STR2 [23] pour les CSP compressés). STR-Slice est aussi considérée comme une méthode de compression du fait qu'elle permet de réduire la taille l'espace de recherche durant le processus de résolution en éliminant les fragments invalides.

Dans le cadre des nos travaux de recherche, nous nous sommes intéréssés à proposer des méthodes de compression à deux niveax (avant et durant la résolution) en exploitant des technique d'extraction de motifs fréquents et qui répondent aux différents critères présentés dans Table 4.1.

4.7 Conclusion

Dans ce chapitre, nous avons présenté les principales méthodes proposées dans la littérature pour la compression des contraintes table. Nous avons classées ces méthodes dans quatre catégories principales. Nous avons des méthodes de compression basées sur la l'arbre et diagramme de décision, des méthodes basées sur la réduction tabulaire, sur la représentation binaire, sur les tuples segmentés et enfin sur l'extraction de motifs fréquents. Comme nous avons cité les avantages et inconvénients des méthodes de chaque classe. Enfin, on a comparé ces méthodes dans un tableau comparatif en se basant sur des critères que nous avons nous même définit. Dans la partie suivante, on vas présenter les travaux réalisés dans le cadre de cette thèse.

Troisième partie

Contributions

Depuis l'arrivée de l'aire de Big Data, nous nous retrouvons de plus en plus face à des contraintes tables très volumineuses. Dans ce chapitre, nous présentons nos méthodes de compression des contraintes tables volumineuses basées sur des téchniques de fouille de motifs fréquents. Pour valider chacune des propositions, nous menons un ensemble d'expérimentations dans lesquelles nous étudions la qualité de compression et son impact sur le processus de résolution des problèmes de statisfaction de contraintes (CSP).
Ce chapitre est organisé comme suit :

— dans chapitre 5, nous introduisons des concepts et définitions relatifs à la compression à base de motifs (itemsets) fréquents.

— dans chapitre 6, nous introduisons notre première méthode de compression `FPTCM` [2] [6] basée sur l'énumération des motifs fréquents et sur une fonction de calcul de taux de compression ;

— dans chapitre 7, nous introduisons la méthode `MFIC` [4] qui est une méthode de compression des contraintes table à base de motifs fréquents maximaux et la mesure d'aire d'un motif ;

— dans chapitre 8 nous terminons avec une conclusion et perspéctives.

Chapitre 5

Méthode de compression basée sur une structure d'arbre FP (FPTCM)

5.1 Définitions

Dans cette section, nous montrons comment représenter une contrainte table c par une table transactionnelle $\mathscr{TD}$ et comment compresser $\mathscr{TD}$ en exploitant des techniques d'extraction de motifs fréquents.

Définition 5.1.1 *(Table transactionnelle)*
Soit un CSP $P = \langle X, D, C \rangle$ *et* $\mathscr{R}(c)$ *la relation associée à la contrainte* $c \in C$*. La table transactionnelle* $\mathscr{TD}_c$ *est définie comme suit :*

- *(i) l'union des valeurs des variables de scope* $S(c)$ *de la contrainte c représente l'ensemble des items* $\mathscr{I}$*.*
- *(ii) l'ensemble des valeurs invoquées dans un tuple* $t \in \mathscr{R}(c)$ *représente une transaction (tuple) de* $\mathscr{TD}$*.*

Example 5.1.1 *Soit le CSP définie en extension :*
$X = \{x_0, \ldots, x_3\}$, $D = \{d_0, \ldots, d_3\}$ *tel que,* $d_0 = d_1 = d_2 = d_3 = \{0,1,2,3\}$.
$C = \{c\}$ *tel que* $c = \{(x_0, x_1, x_2, x_3), R(c)\}$ *et*
$R(c) = \{(0\ 1\ 0\ 0), (0\ 1\ 0\ 1), (0\ 1\ 0\ 2), (0\ 2\ 0\ 0), (0\ 2\ 1\ 0), (0\ 2\ 0\ 1)\}$.
La table transactionnelle $\mathscr{TD}(c)$ *associée à la contrainte table* $\mathscr{R}(c)$ *est donnée par Table 5.1. Dans la première colonne, un indice* t_i *est associé à chaque tuple (transaction) de la table.*

TABLE 5.1 – Table transactionnelle $\mathscr{TD}_c$

TID	x_0	x_1	x_2	x_3
t_0	0	1	0	0
t_1	0	1	0	1
t_2	0	1	0	2
t_3	0	2	0	0
t_4	0	2	1	0
t_5	0	2	0	1

Définition 5.1.2 *(motif) Dans le contexte de ces travaux, un motif correspond à l'affectation de certaines valeurs de $S(c)$.*

Example 5.1.2 *Soit la table transactionnelle $\mathscr{TD}_c$ dans Table 5.1. $u = \{x_0 = 0, x_1 = 1, x_2 = 0\}$ est un motif qui apparaît dans les tuples t_0, t_1, t_2.*

Définition 5.1.3 *(sous-table) Une sous-table St associée à un motif u correspond aux parties restantes des tuples de sa couverture ($cover(u)$) après la suppression de u.*

Définition 5.1.4 *(fragment) Un fragment (ou entry) est la paire (u, St) telque u est un motif et St sa sous-table correspondante.*

Example 5.1.3 *Soit le motif $u = \{x_0 = 1, x_1 = 1, x_2 = 0\}$ de Table 5.1 qui couvre les tuples t_0, t_1 et t_2. Le fragment correspondant à ce motif est donné par Table 5.2*

TABLE 5.2 – exemple de fragment

x_0	x_1	x_2	x_3
			0
0	1	0	1
			2

Définition 5.1.5 *La taille d'une base transactionnelle $taille(\mathscr{TD})$ est égale au résultat du produit de son arité et son nombre de tuples.*

$$taille(\mathscr{TD}) = arite * nombre_{tuples} \tag{5.1}$$

Définition 5.1.6 *La taille d'un fragment (u, St) est égale à la taille de le motif u le constituant plus la taille de sa sous-tale St correspondante.*

$$taille(e) = taille(u) + taille(St). \tag{5.2}$$

Définition 5.1.7 *La taille d'une base transactionnelle compressée $\mathscr{TD}^c$ est égale à la somme des tailles de ses fragments.*

$$taille(\mathscr{TD}^c) = \sum_{i=0}^{i=n} taille(e_i) \tag{5.3}$$

Définition 5.1.8 *(taux de compression) Le taux de compression Rate d'une base transactionnelle est donné par :*

$$Rate = 1 - \frac{taille(\mathscr{T}\mathscr{D}^c)}{taille(\mathscr{T}\mathscr{D})} \tag{5.4}$$

5.2 Présentation de la méthode FPTCM

Notre première méthode de compression, appelée FPTCM (Fréquente Pattern Tree based Compression Method), est basée sur la structure d'arbre FP-Tree [48] pour énumérer, à partir d'une contrainte table, les motifs fréquents nécessaires à sa compression. Cette méthode présente des similitudes avec la méthode Sliced-tabe [1], présentée dans l'état de l'art 4.5, du fait que les deux méthodes exploitent l'arbre FP-Tree pour énumérer les motifs fréquents requis pour la compression et utilisent une même structure de la contrainte table compressée. La différence entre les deux méthodes réside dans la façon dont l'arbre FP est exploité et les motifs fréquent résultants.
La méthode Sliced-table a été proposé par **Gharbi et *al.*** [1] afin de réduire l'espace requis pour la présentation d'une contrainte table en mémoire. Bien que les taux de compression qu'obtient cette méthode sont intéressants mais ils ne sont pas optimals. La méthode présente plusieurs faiblesses :

— La recherche de motifs fréquents se base sur un seuil minimal de fréquence S_{min} permettant de déterminer si un motif est fréquent ou non fréquent. La méthode Sliced-table[1] fixe la valeur de S_{min} à deux (2), générant ainsi un grand nombre de motifs fréquents avec des valeurs fréquences trop petites par rapport au nombre de tuples dans la table à compresser.

— Pour énumérer les motifs fréquents requis pour la compression d'une contrainte table, la méthode Sliced-table [1] utilise une fonction de gain qui calcule le nombre d'items avec lequel la taille de f tuples peut être réduite en les compressant avec un motif fréquent u, et le compare aux nombre d'items avec lequel la taille de f' tuples (avec $f < f'$) peut être réduite en les compressant avec un motif parent u' de u. *i.e.* $u' \in u$. Si le motif u permet de réduire les f tuples avec un plus grand nombre d'items par rapport à celui que nous pouvons obtenir si nous compressons les f' tuples avec u', alors u est considéré comme motif fréquent nécessaire à la compression. Cela ne donne pas une approximation exacte sur le

gain en compression car la comparaison se fait par rapport à un nombre différent de tuples ;

— Pour calculer la couverture d'une motif fréquent, la méthode `Sliced-table` [1] parcours toute la contrainte table pour vérifier et déterminer les tuples contenant le motif en question. Cela implique plusieurs parcours de la contrainte table.

Pour remédier à ces faiblesses, nous avons proposé dans ce travail une nouvelle méthode `FPTCM` qui permet de : (i) calculer de façon dynamique, pour chaque contrainte table, la valeur de S_{min} à utiliser pour énumérer les motifs fréquents. (ii) calculer et comparer le taux de compression d'un même ensemble de tuples par des motifs différents au lieu d'utiliser la fonction de gain pour sélectionner les motifs fréquents. (iii) remplacer, dans l'arbre FP, la fréquence d'un motif par sa couverture afin de connaître la couverture de chaque motif fréquent sans parcourir la contrainte table. Les étapes de la méthode `FPTCM` sont illustrées dans Algorithme 5.1 et détaillées dans les sous-sections suivantes.

5.3 Calcul de seuil minimal de fréquence S_{min}

Trouver une meilleure valeur de seuil minimal de fréquence S_{min} à utiliser pour énumérer les motifs fréquents nécessaires à la compression d'une contrainte table est un défi majeur. Si la valeur de S_{min} est trop petite, un grand nombre de motifs couvrant très peu de tuples peut être énuméré et utilisé pour la compression de la contrainte table. Si sa valeur est trop grande, des motifs fréquents permettant de mieux compresser la contrainte table peuvent être négligés ou m ème obtenir un ensemble vide de motifs vu la grandeur de la valeur de S_{min}. Pour cela, nous avons proposé de calculer de façon dynamique, pour chaque contrainte table, la valeur de seuil minimal de fréquence. Cela en énumérant les top-k motifs fréquents ayant une valeur de fréquence supérieure ou égale à 2, ensuite affecter la dernière valeur de fréquence trouvé à S_{min}. Algorithme 8 illustre la méthode de calcul de la valeur de S_{min}. En procédant ainsi, la méthode garantie un ensemble non vide de motifs fréquents avec des fréquences pas trop petites par rapport à ta taille de la contrainte table à compresser.

Example 5.3.1 *Soit la contrainte table représentée par Table 5.3, on souhaite calculer la valeur de S_{min} à utiliser pour énumérer les motifs fréquent nécessaires à la compression. Si on fixe la valeur de k au nombre de tuples de la table, la plus petites fréquence des*

Algorithme 8 CalculS_{min}

Data : $\mathscr{TD}$: contrainte table, k : nombre de meilleurs motifs à énumérer
Result : S_{min} : seuil minimal de fréquence
1 $\mathscr{F}^k \leftarrow \texttt{topK}(\mathscr{TD}, \texttt{k})$
2 $S_{min} \leftarrow \min_{f \in \mathscr{F}^k} freq(f)$
3 retourner S_{min}

top-20 *motifs fréquents fermés de Table 5.3 est égale à* 3. *On affecte cette valeur à* S_{min}. *Alors dans les exemples des prochaines étapes nous considérerons* S_{min}=3.

TABLE 5.3 – Contrainte table c

TID	x_0	x_1	x_2	x_3	x_4	x_5
t_0	1	2	1	1	0	0
t_1	1	2	1	1	0	1
t_2	1	2	1	1	1	1
t_3	1	2	1	2	0	0
t_4	1	2	1	2	1	1
t_5	1	2	1	2	1	2
t_6	1	2	1	2	2	2
t_7	1	2	2	0	0	0
t_8	1	2	2	1	0	0
t_9	1	2	2	1	0	2
t_{10}	1	2	2	1	1	0
t_{11}	1	2	2	1	1	2
t_{12}	1	2	3	1	0	1
t_{13}	1	2	3	1	1	1
t_{14}	1	2	3	1	1	2
t_{15}	1	2	3	2	0	0
t_{16}	1	2	3	2	2	2
t_{17}	1	3	1	0	0	0
t_{18}	2	0	1	0	0	0
t_{19}	2	1	1	0	0	0
t_{20}	2	1	1	2	0	1
t_{21}	2	1	1	4	0	1
t_{22}	3	3	1	2	0	0

5.4 Construction l'arbre FP

La construction de l'abre FP se fait en trois étapes :

— Pour connaître la couverture d'un motif fréquent sans scanner à chaque fois la contrainte table et pour construire le FP-tree, nous utilisons une table, appelée table d'en-tête, dans laquelle on sauvegarde pour chaque motif de taille 1 (1-

Algorithm 9 getCovers

Data : $\mathscr{TD}$: contrainte table, *items* : les items dans $\mathscr{TD}$
Result : *couvertures* : tableau de l'ensemble des couvertures des items de $\mathscr{TD}$

```
for each item α ∈ items do
    couverture[α] ← ∅
end
for each tuple 𝒯 ∈ 𝒯𝒟 do
    for each item α ∈ 𝒯 do
        couverture[α] = couverture[α] ∪ index_𝒯
    end
end
retourner couverture
```

itemset) les indices des tuples appartenant à couverture. Algorithm 9 représente les étape de création de la table d'en-tête.

— ordonner les items de chaque tuple de la contrainte table dans un ordre décroissant selon de la taille de leurs couvertures tout en supprimant les items dont la fréquence est inférieure à S_{min}, sachant que la fréquence d'un item α ($freq(\alpha)$) est égale à la taille de sa couverture $|cover(\alpha)|$. Soit $ordonner_{tuple}(t, couvertures)$ la fonction qui permet d'ordonner les items d'un tuple et supprimer ceux ayant une fréquence inférieur à S_{min} ;

— construire l'arbre FP en insérant les tuples ordonnés un par un dans l'arbre : si une partie d'un tuple correspond à un chemin déja existant dans l'arbre, ce dernier sera partagé. À chaque noeud de l'arbre, au lieu de sauvegarder uniquement la fréquence, on sauvegarde les idenfiants des tuples de la couverture du motif réprésenté par le chemin allant du noeud racine au noeud en question. Soit $inserer_{tuple}(t, FP\text{-}tree)$ la fonction qui permet d'ajouter un tuple t dans l'arbre FP $FP\text{-}tree$.

— supprimer de l'arbre tout les noeuds ayant une taille de couverture inférieur à S_{min}, car ces derniers ne peuvent pas appartenir à des motifs fréquents de la contrainte table.

Example 5.4.1 *Considérons la contrainte table c représentée par Table 5.3. On crée une table d'en-tete dont les colonnes resprésentent les variables du scope de la contrainte et les lignes représentent les differenetes valeurs possibles pour les variables. Chaque case* (i, j) *correspond à l'ensemble des indices des tuples couverant l'item* $(x_i = a_j)$*. Par exemple, l'item* $(x_3 = 0)$ *apparait dans les tuples* $\{t_7, t_{17}, t_{18}, t_{19}\}$*. Table 5.4 représente la table d'en-tete créée pour la contrainte table c.*

TABLE 5.4 – Table d'entete associée à la contrainte table 5.3

item	x_0	x_1	x_2	x_3	x_4	x_5
0	$\emptyset$	$\{18\}$	$\emptyset$	$\{7,17,18,19\}$	$\{0,1,3,7,8,9,12,15,17,18,19,20,21,22\}$	$\{0,3,7,8,10,15,17,18,19,22\}$
1	$\{0,\cdots,17\}$	$\{19,20,21\}$	$\{0,\cdots,6,17,\cdots,22\}$	$\{0,1,2,8,\cdots,14\}$	$\{2,4,5,10,11,13,14\}$	$\{1,2,4,12,13,20,21\}$
2	$\{18,19,20,21\}$	$\{0,\cdots,16\}$	$\{7,\cdots,11\}$	$\{3,4,5,6,15,16,20,22\}$	$\{6,16\}$	$\{5,6,9,11,14,16\}$
3	$\{22\}$	$\{17,22\}$	$\{12,\cdots,16\}$	$\emptyset$	$\emptyset$	$\emptyset$
4	$\emptyset$	$\emptyset$	$\emptyset$	$\{21\}$	$\emptyset$	$\emptyset$

Ensuite, on ordonne chaque tuple t de c dans un ordre décroissant des fréquences des items (taille de leur couverture). La contrainte table ordonnée est donnée par Table 5.5. Maintenant, ajouter chaque tuple de Table 5.5 à l'arbre FP. Pour un tuple donné, on ajoute dans l'arbre que les items ayant une fréquence supérieure ou égale à S_{min}. Par exemple, les items $(x_1 = 3)$ et $(x_0 = 3)$ du tuple t_{22} ne peuvent pas être ajoutés à l'arbre FP car leur fréquence est inférieure à S_{min}. Chaque branche reliant deux noeuds est labellée par le nom de l'item. Un noeud contient la couverture de l'item en question. L'abre FP correspondant à Table 5.5 est donné par Figure **??**. *c Les noeuds en gras ont une fréquence inférieure à S_{min}, ils sont à supprimer de l'arbre car ils ne peuvent pas contribuer à la compression des tuples de la contrainte table. L'arbre FP après la supréssion des noeuds non fréquents est donné par la figure Figure 7.4.*

5.5 Extraction des motifs fréquents nécessaires à la compression

Une fois que l'arbre FP est construit et est réduit, la méthode énumère les motifs fréquents pertinents pour la compression en utilisant une fonction qui calcul et compare le taux de compression d'un ensemble de tuples avec un motif fréquent u avec le taux de compression du meme ensemble de tuples mais avec les super-motifs de u. Les étapes

TABLE 5.5 – Contrainte table *c* ordonnée selon la taille des couvertures des items.

t_0	x_0=1 :18	x_1=2 :17	x_4=0 :14	x_2=1 :13	x_3=1 :10	x_5=0 :10
t_1	x_0=1 :18	x_1=2 :17	x_4=0 :14	x_2=1 :13	x_3=1 :10	x_5=1 :7
t_2	x_0=1 :18	x_1=2 :17	x_2=1 :13	x_3=1 :10	x_4=1 :7	x_5=1 :7
t_3	x_0=1 18	x_1=2 :17	x_4=0 :14	x_2=1 :13	x_5=0 :10	x_3=2 :8
t_4	x_0=1 :18	x_1=2 :17	x_2=1 :13	x_3=2 :8	x_4=1 :7	x_5=1 :7
t_5	x_0=1 :18	x_1=2 :17	x_2=1 :13	x_3=2 :8	x_4=1 :7	x_5=2 :6
t_6	x_0=1 :18	x_1=2 :17	x_2=1 :13	x_3=2 :8	x_5=2 :6	**x_4=2 :2**
t_7	x_0=1 :18	x_1=2 :17	x_4=0 :14	x_5=0 :10	x_2=2 :5	x_3=0 :4
t_8	x_0=1 :18	x_1=2 :17	x_4=0 :14	x_3=1 :10	x_5=0 :10	x_2=2 :5
t_9	x_0=1 :18	x_1=2 :17	x_4=0 :14	x_3=1 :10	x_5=2 :6	x_2=2 :5
t_{10}	x_0=1 :18	x_1=2 :17	x_3=1 :10	x_5=0 :10	x_4=1 :7	x_2=2 :5
t_{11}	x_0=1 :18	x_1=2 :17	x_3=1 :10	x_4=1 :7	x_5=2 :6	x_2=2 :5
t_{12}	x_0=1 :18	x_1=2 :17	x_4=0 :14	x_3=1 :10	x_5=1 :7	x_2=3 :5
t_{13}	x_0=1 :18	x_1=2 :17	x_3=1 :10	x_4=1 :7	x_5=1 :7	x_2=3 :5
t_{14}	x_0=1 :18	x_1=2 :17	x_3=1 :10	x_4=1 :7	x_5=2 :6	x_2=3 :5
t_{15}	x_0=1 :18	x_1=2 :17	x_4=0 :14	x_5=0 :10	x_3=2 :8	x_2=3 :5
t_{16}	x_0=1 :18	x_1=2 :17	x_3=2 :8	x_5=2 :6	x_2=3 :5	**x_4=2 :2**
t_{17}	x_0=1 :18	x_4=0 :14	x_2=1 :13	x_5=0 :10	x_3=0 :4	**x_1=3 :2**
t_{18}	x_4=0 :14	x_2=1 :13	x_5=0 :10	x_3=0 :4	x_0=2 :4	**x_1=0 :1**
t_{19}	x_4=0 :14	x_2=1 :13	x_5=0 :10	x_0=2 :4	x_3=0 :4	x_1=1 :3
t_{20}	x_4=0 :14	x_2=1 :13	x_5=1 :7	x_3=2 :8	x_0=2 :4	x_1=1 :3
t_{21}	x_4=0 :14	x_2=1 :13	x_5=1 :7	x_0=2 :4	x_1=1 :3	**x_3=4 :1**
t_{22}	x_4=0 :14	x_2=1 :13	x_5=0 :10	x_3=2 :8	**x_1=3 :2**	**x_0=3 :1**
t_{23}	x_4=0 :15	x_2=1 :14	x_5=0 :10	x_3=2 :9	**x_1=3 :2**	**x_0=3 :1**

d'éxtraction des motifs fréquent pertinents à la compression sont comme suit :

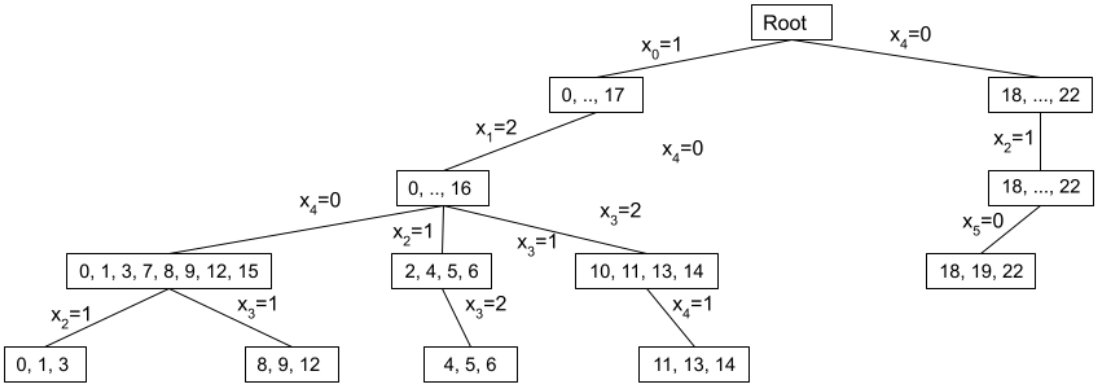
— parcourir l'arbre FP à partir de la racine. Au niveau de chaque noeud *node* qui n'est pas un noeud feuille, calculer pour le motif *u* représenté par le chemin allant du noeud racine au noeud *node*, le taux de compression des tuples appartenant à sa couverture $cover(u)$ (les tuples dont les indices sont sauvegardé dans le noeud *node*) en les compressant avec ce motif fréquent. La fonction qui calcul le taux de compression *Rate* avec motif fréquent *u* est donné par :

$$Rate = 1 - \frac{freq(u) + (arite - size(u)) * freq(u)}{arite * freq(u)} \tag{5.5}$$

Avec arite représente la taille du scope de la contraite table à compresser

Ensuite, calculer taux de compression *ChRate* que l'on peut obtenir en compressant les meme tuples avec les super-motifs u' de *u*. Ces super-motifs sont représentés par les chemins allant du noeud root aux différents noeuds fils de *node*. Pour le calcul de taux de compression des super-motifs, nous utilisons une liste $chlid_{node}$ (soit nbr_{ch} la taille de cette liste.) dans laquelle nous sauvegardons les noeuds fils du noeud *node* ayant une valeur de fréquence supérieure ou égale à S_{min}. Le taux de compression que l'on peut obtenir en compressant les f tuples

FIGURE 5.1 – FP-tree réduit.



appartenant à $cover(u)$ avec les super-motifs de u est donné par :

— soit $freq'$ la somme des fréquences des super-motifs de u

$$freq' = \sum_{u' \in child_{node}} freq(u') \tag{5.6}$$

— soit $taille_{cmp}$ la taille des tuples couverts par les super-motifs freéquens de u après leur compression avec ces super-motifs :

$$taille_{cmp} = |nbr_{ch}| * (size(u')) + (arite - size(u')) * freq' \tag{5.7}$$

— soit $taille_{ncmp}$ la taille des tuples couvert par u et non couverts par les super-motifs de u

$$taille_{ncmp} = (freq(u) - freq') * arite \tag{5.8}$$

Algorithm 10 MFItemsets

Data : n : noeud, S_{min} : seuil minimal de fréquence, $arite$: arité de la contrainte table.
Result : $L_{itemsets}$: liste de motifs

```
1  if n est un noeud feuille then
2      u ← n.getItemset(n)                                   ▷ Algorithme 11
3      L_itemsets.add(u)
4  else
5      Rate = n.getRate()          ▷ calcul de taux de compression avec u (équation 5.5)
6      for chaque neoud fils n' de n do
7                                  ▷ créer la liste des noeuds fils de u ayant une fréquence ≥ S_min
8          if freq(n') ≥S_min then
9              child_n.add(n')
10         end
11     end
12     Chrate = n.getChRate(child_n)          ▷ calcul de taux de compression ChRate
         (équation 5.9)
13     if Chrate ≤ Rate then
14         u ← n.getItemset(n)                               ▷ Algorithme 11
15         L_itemsets.add(u)
16     else
17         for chaque n' ∈ child_n do
18             MFItemsets(n', S_min, arite)
19         end
20         f' = Σ_{n'∈child_n} freq(n')
21         if f' ≤ freq(u) then
22             for n ∈ child_n do
23                 cover(u) ← cover(u) \ cover(n)
24             end
25             L_itemsets.add(u)
26         end
27     end
28 end
```

— le taux de compression *Chrate* est donc calculé par la formule suivante :

$$Chrate = \frac{taille_{ncmp} + taille_{cmp}}{f * arite} \tag{5.9}$$

Si la valeur de *Chrate* est inférieure à la valeur de *Rate* alors le motif u est considéré comme pertinents à la compression sinon explorer les sous-motifs de u (le sous-arbre dont la racine est le noeud *node*) en répétant le même calcul. Algorithme 10 résume les étapes de parcours de l'arbre FP et d'énumération des motifs fréquents nécéssaires à la compression.

Example 5.5.1 *Considérons l'arbre FP de Figure 7.4 crée dans l'étape précedante. Pour compresser la contrainte table on considère uniquement les motifs fréquents de taille su-*

Algorithm 11 getItemset

Data : n : noeud
Result : u : itemset fréquent

```
u ← ∅
while n ≠ root do
    u ← u + n
    n ← n.getParent()
end
return u
```

périeure à 2. Pour cela, le parcours doit commencer à partir du niveau 2 de l'arbre FP. Le noeud $(x_0 = 1)$*, est un fils du noeud root alors on passe directement à l'exploration de ses noeuds fils. Ce noeud possède un seul noeud fils* $(x_1 = 2)$ *ayant une fréquence supérieur ou égale à* S_{min}*. On calcul le taux de compression Rate que peut offrir le motif fréquent* $u = \{x_1 = 2, x_0 = 1\}$ *(chemin allant de noeud* $x_1 = 2$ *au noeud root). La couverture de u est égale à la couverture du noeud* $(x_1 = 2)$*, donc* $freq(u) = 17$*. L'arité de la contrainte table est égale à 6. Selon la formule 5.5,* $Rate = 0.3$*. Maintenant, calculons le taux de compression Chrate des fils de u. Le motif fréquent u possède trois noeuds fils, alors* $child_n = \{x_4 = 0, x_2 = 1, x_3 = 1\}$*. La somme* $freq'$ *des fréquences des noeuds de* $child_n$ *est égale à 16. Alors selon la formule 5.9,* $Chrate = 0.38$*. On voit bien que* $Chrate > Rate$*, alors u n'est pas retenu pour la compression, il faut explorer ces noeuds fils. On explore le noeud* $(x_4 = 0)$ *(premier noeud fils de* $(x_1 = 0)$*. Le motif correspondant au chemin allons de ce noeud au noeud root est donné par* $u' = \{(x_0 = 1), (x_1 = 2), (x_4 = 0)\}$ *et sa fréquence* $freq(u') = 8$*. Le motif* u' *est composé de deux noeuds fils* $child_n = \{(x_2 = 1), (x_3 = 1)\}$ *dont la somme des fréquences* $freq' = 6$*. Pour le motif* u'*,* $Rate = 0.4$*, et pour les super-motifs de* u'*,* $Chrate = 0.33$*. Le motif* u' *offre un meilleur taux de compression par rapport à ses super-motifs. On arrête le parcours la branche et on ajoute* u' *ainsi que sa couverture à l'ensemble S des motifs fréquents nécéssaires à la compression (*$S = \{(\{(x_0 = 1), (x_1 = 2), (x_4 = 0)\} : \{0, 1, 3, 7, 8, 9, 12, 15\})\}$*). De la même manière on explore le reste des noeuds et on obtient l'ensemble S suivant :*

- $\{(x_0 = 1), (x_1 = 2), (x_4 = 0)\} : \{0, 1, 3, 7, 8, 9, 12, 15\}$
- $\{(x_0 = 1), (x_1 = 2), (x_2 = 1)\} : \{2, 4, 5, 6\}$
- $\{(x_0 = 1), (x_1 = 2), (x_3 = 1)\} : \{10, 11, 13, 14\}$
- $\{(x_4 = 0), (x_2 = 1)\} : \{18, \cdots, 22\}$

Algorithm 12 TableCompressée

Data : $\mathscr{S}$: Liste d'itemsets.
Result : c_c : contrainte table compressée

```
for each u_i ∈ M* do
    sub_t ← ∅;
    for each t_j ∈ cover(u_i) do
        sub_t ← sub_t ∪ (t_j \ u_i);
    end
    R^c ← R^c ∪ (u_i, sub_t);
end
return R^c
```

5.6 Construction de la contrainte table compressée

Une fois que les motifs fréquents nécessaires à la compression sont énumérés, la méthode FPTCM crée la contrainte table compressée en utilisant la même structure de la méthode STR-Slice [1]. Une contrainte table compressée consiste en un ensemble de fragments (*entries*) et une table appellée *table-par-defaut* qui contient les tuples non compressés. Chaque fragment *entry* correspond à un motif fréquent u et une sous-table St contenant les parties non compréssées de chaque tuple de $cover(u)$. Contrairement à la méthode STR-Slice qui parcours à chaque fois la contrainte table pour retrouver les tuples couvert par un motif fréquent u, la méthode FPTCM connait déjà les couvertures des motifs. Alors pour créer un fragment correspondant à un motif fréquent donné, la méthode accède directement aux tuples dont les indices sont donnés par la couverture du motif. Algorithme 12 illustre les création de la contrainte table compressée.

Example 5.6.1 *Tout les tuples de t_0 à t_15 et de t_18 à t_22 sont couverts par les itemsets fréquents énumérés. Uniquement deux tuples t_{16} et t_{17} ne sont pas couverts. Les fragments correspondants aux motifs fréquents de S et la table par défaut contenant les tuples non compressés sont représentés par Table 6.4.*

5.7 Résultats éxperimentaux

Pour montrer l'éffecacité de notre méthode en terme de compression et son impact sur la résolution des CSPs, nous avons implémenté la méthode FPTCM dans le solveur Oscar [1] développé en Scala. Nous avons menés des expérimentations en utilisant une machine Intel (R) Core(TM), $i5-7200$ CPU, 2.5 GHz avec une RAM de 4 GB, avec un distribution

1. Solveur disponible sur https://bitbucket.org/oscarlib/oscar/src/dev/

TABLE 5.6 – Contrainte table compressée obtenue avec S_{min}= 3.

Premier fragment e_1.

x_0	x_1	x_4	x_2	x_3	x_5
			1	1	0
			1	1	1
			1	2	0
1	2	0	1	2	2
			2	0	0
			2	1	0
			2	1	2
			3	1	2

(b) Deuxième fragment e_2.

x_0	x_1	x_2	x_3	x_4	x_5
			1	1	1
1	2	1	2	1	0
			2	1	1
			2	1	2

(c) Troisième fragment e_3.

x_0	x_1	x_3	x_2	x_4	x_5
			2	0	2
1	2	1	2	1	0
			3	0	1
			3	1	1

(d) Quatrième fragment e_4.

x_2	x_4	x_5	x_0	x_1	x_3
			1	3	0
1	0	0	2	0	0
			2	1	4

(e) Fifth entry e_5.

x_0	x_1	x_2	x_3	x_4	x_5
1	0	1	2	1	0
			2	1	2

(f) Table par defaut DF.

x_0	x_1	x_2	x_3	x_4	x_5
1	2	3	2	0	0
1	2	3	2	2	2

Ubuntu sur 64 bits. Ces expérimentations ont été effectuées sur des benchmarks [2] utilisés par [].

Déscription des benchmarks

Les benchmarks utilisés sont décrit comme suit :

— ModifiedRenault : qui est un problème obtenu à partir de Renault Megane. Il invoque de grandes contraintes tables avec une grande arité. C'est un problème qui contient 50 instances structurées avec de grandes contraintes table pouvant contenir jusqu'à 48 721 tuples.

— bdd : contient deux series, bddLarge et bddSmall, quasi-aléatoires dont les instances sont générées à partir de diagrammes de décision binaires. Chaque serie contient 35 instances dont les contraintes tables peuvent contenir jusqu'à ... tuples.

— crossword : c'est un problème qui consiste à remplir une grille vide par des mots d'un dictionnaire. Crossword contient plusieurs series dont on trouve wordsVg, LexVg, UkVg, ogdVg ; Ou Uk fait référence à une grille vide et words, lex, uk et ogd sont des dictionnaires. Parmis ces séries nous avons utilisé deux d'entre elles LexVg et wordsVg.

2. Data sets are available at https://bitbucket.org/pschaus/xp-table/src/master/instances/

Algorithme FPTCM

Data : c : contrainte table
Result : c' : contrainte table compressée

$S_{min} \leftarrow CalculS_{min}()$
$Coverages \leftarrow getCovers(c)$
for *chaque tuple* $t \in c$ **do**
    sort(t, coverages, S_{min})
    insert(t, FP-tree)
end
Réduire la l'abre $FP-tree$ en supprimant les noeuds ayant une fréquence inférieure au S_{min}
for *chaque fils n du noeud root* **do**
    $L \leftarrow MFItemsets(n, S_{min}$, arite)
    $L_{itemsets}$.addAll(L)
end
for *chaque itemset* $u \in L_{itemsets}$ **do**
    $fragment \leftarrow creerFragment(u, cover(u))$
    $table\_comp$.addEntry(fragment)
    C.remove(couver(u))
end
$table\_comp.addDfTable(C)$

— randsJC :

Une déscription détaillée des différents benchmarks est donnée par le tableau Table 5.7. Pour chaque benchmark on donne le nombre d'instances qu'il contient $Inst_{nbr}$, le nombre maximal de variable dans une instance X, la taille du plus grand domaine de variable $|D|$, le nombre de relations dans une instance R_{nbr}, la taille de la plus grande relation R_{max}, l'arité de la plus grande relation arité et le nombre maximal de contraintes dans une instance c_{nbr} .

TABLE 5.7 – Caratéristiques des benchmarks utilisés.

Benchmark	$Inst_{nbr}$	X	$\|D\|$	R_{nbr}	R_{max}	$arity$	C_{nbr}
bddLarge	35	21	2	1	57971	18	133
bddSmall	35	21	2	1	6945	15	2713
randsJC2500	10	40	8	40	2500	7	40
randsJC5000	10	40	8	40	5000	7	40
randsJC7500	10	40	8	40	7500	7	40
randsJC10000	10	40	8	40	10000	7	40
Crossword-Lex-Vg	63	288	26	2	3607	18	34
Crossword-Words-Vg	65	320	25	2	68064	20	36
Modified-Renault	50	111	42	142	48721	10	159

5.7.1 Mesures de performances

Notre contribution est une amélioration de la méthode `Sliced-table` [1] de l'etat de l'art. Alors, pour la valider, nous avons comparé entre ces deux méthodes en considérons les paramètres suivants : le taux de compression, nombre de motifs fréquents par instance, fréquence moyenne d'un motif fréquent, la taille moyenne d'un motif fréquent (nombre d'items) et enfin le temps d'exécution CPU. Un temps limite de 1800 secondes est fixé, si une instance n'est pas résolue en un temps inférieur ou égale à ce temps limite, on pose un TO (Time Out).

Comparaison en terme de compression

Pour comparer les deux méthodes `FPTCM` et `Sliced-table` [1] en terme de qualité de compression, nous avons rapporté dans Table 5.8 les résulats de compression de chaque benchmark avec chacune des méthodes. Pour chaque benchmark, nous avons donné en pourcentage son taux de compression $comp_{rate}$, le nombre total de motifs fréquents énumérés Nbr_{motifs}, la fréquence moyenne d'un motif fréquent $freq_{moy}$ ainsi que la taille (longueur) moyenne d'un motif fréquent $taille_{motif}$. Comme nous pouvons le constater, notre méthode `FPTCM` offre un meilleur taux de compression des benchmarks en utilisant moins de motifs fréquents avec de plus grandes valeurs de fréquence par rapport à la méthode `Slied-table`. Le nombre de motifs fréquents utilisés par `FPTCM` pour la compression du benchmark ModifiedRenault est plus grand par rapport à celui de `Slied-table`. Cela est expliqué par le fait que l'écart en taux de compression entre les deux méthodes est très grand. Le benchmark est compressé à 78% avec `FPTCM` alors qu'il est compressé à 37% seulement avec la méthode `Slied-table`.

TABLE 5.8 – Résultats de compression obtenus pour les différents benchmarks

Mesure	Méthode	benchmarks							
		bddLarge	bddSmall	wordsVg	LexVg	modRenault	RandsJC2500	RandsJC5000	RandsJC7500	RandsJC100000
$comp_{rate}(\%)$	sliced-table	58		18.8	29.4	37.06	28.4	34.8	38.4	40.9
	FPTCM	63.4		21.3	36.1	78.8	32.1	36.7	40	41.2
$Nbr_{itemsets}$	Sliced-table	3467214		73870	306015	120207	188304	416258	577619	870843
	FPTCM	873882		112413	232190	390816	91005	155881	234348	265533
$freq_{moy}$	Sliced-table	246		3	3	8	3	3	3	3
	FPTCM	1262		3	8	12	13	25	36	46
$taille_{itemsets}$	Sliced-table	3		4	3	3	3	3	3	3
	FPTCM	2		4	3	3	2	2	2	2

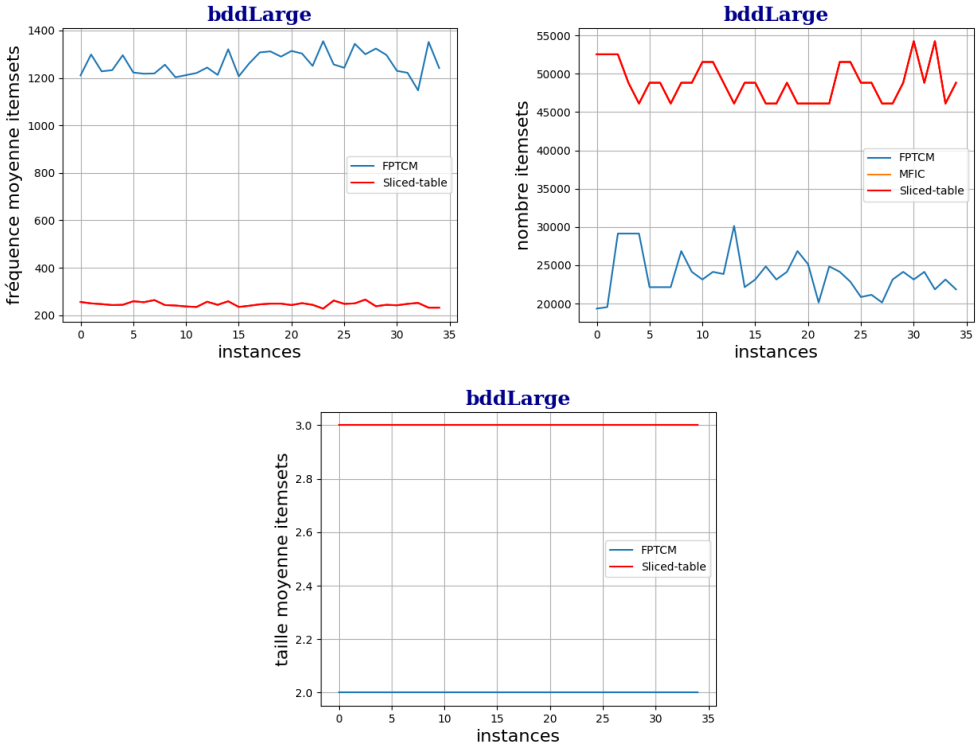


FIGURE 5.2 – Courbes obtenues pour le benchmark BddLarge.

Les Figures 5.2, 5.3 et 5.4 montrent des courbes detaillant les resultats obtenus pour les différentes instances des trois benchmarks bddLarge, crossword-lexVg et randsJC10000. Pour chaque benchmark nous donnons une courbe représentant le nombre de motifs fréquents utilisés par chaque instance pour la compression, taille des motifs fréquents ainsi que leur valeurs moyenne de fréquence. L'axe des x représente les instances et l'axe des y représente les différentes mesures. Comme nous pouvons le constater, la méthode `FPTCM` compresse les benchmarks avec des motifs fréquents de tailles inférieures mais avec des fréquences largement supérieures à celles des motifs fréquents utilisés par la méthode `Slied-table`. Le nombre total de motifs fréquents utilisés par `FPTCM` pour la compression est beaucoup plus petit comparant au nombre utilisé par la méthode `Slied-table`. Alors la méthode `FPTCM` compresse plus de tuples en utilisant moins de motifs fréquents par rapport à la méthode `STR-Slice`.

5.7.2 Comparaison en terme de résolution

La méthode `FPTCM` utilise la meme structure de contrainte table compressée que celle utilisée par la méthode `Sliced-table`. Pour cela, pour résoudre les CSPs compressés avec la méthode `FPTCM` on a utilisé la méthode `STR-slice` [1] detaillée dans la section État de l'art **??**. Dans ce qui suit, on note par `STR-FPTCM` l'application de `STR-slice`

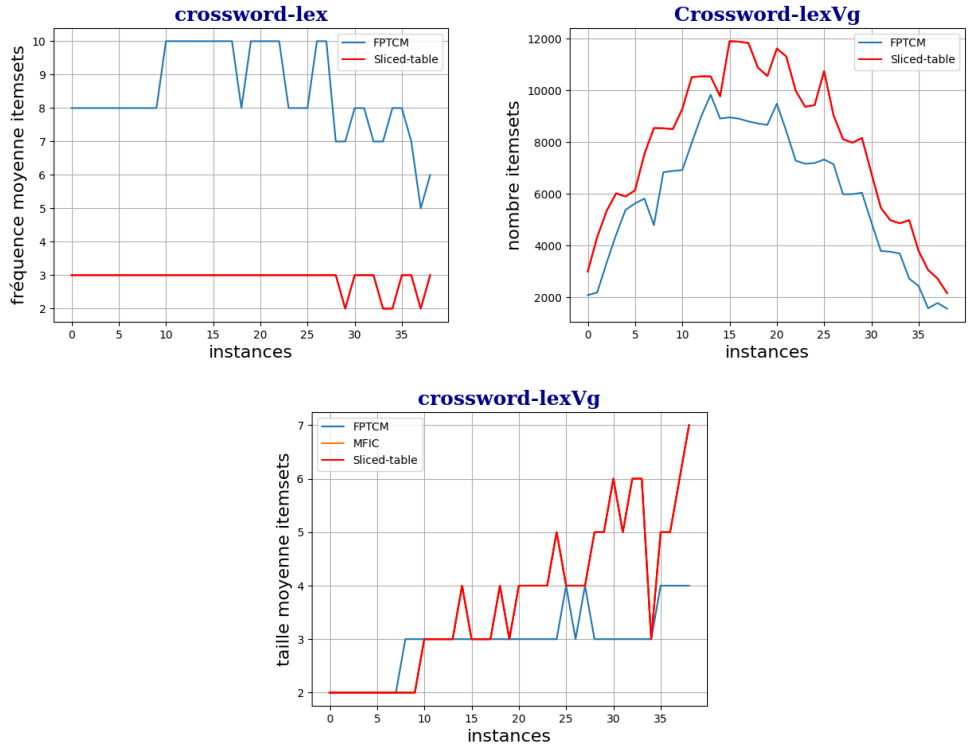


FIGURE 5.3 – Courbes obtenues pour le benchmark Crossword-LexVg.

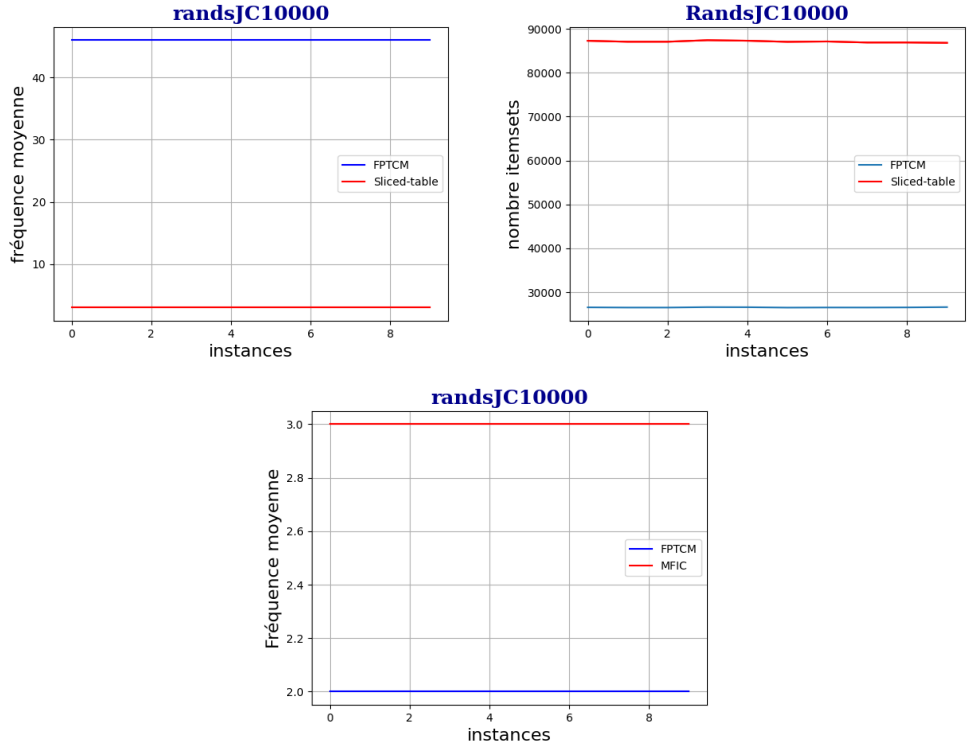


FIGURE 5.4 – Courbes obtenues pour le benchmark randsJC10000.

aux tables compressées avec FPTCM . La méthode STR-slice est une version optimisée de la méthode STR2 pour les contraintes tables compressées. Pour cela, nous allons aussi comparer notre méthode avec STR2 pour connaitre l'impact de la compression sur la réso-

lution. Pour comparer les méthodes STR-FPTCM , STR-Slice et STR2 en terme de résolution, nous avons rapporté dans Table 6.6, pour chaque méthode et pour chaque benchmark, le nombre d'instances résolues $inst_s$, le temps CPU moyen de résolution CPU_t ainsi que le taux de compression cmp_{rate} obtenus par les deux méthodes STR-FPTCM et STR-Slice. Le temps CPU moyen est calculé en prenant en considération que les instances resolues par toutes les méthodes. En comparant les trois méthodes en terme de nombre d'instances resolues, nous pouvons remarquer que pour le benchmark *crossword-lexVg* notre méthode résoud 11 instances de plus par rapport à STR-Slice et 3 instances de plus par rapport à STR2. Pour le benchmark *randsJC10000*, la méthode STR-Slice n'a résolue aucune instance tandis que STR-FPTCM à résolu 5 et STR2 a résolue 3 instances de plus par rapport à STR-FPTCM. Pour le reste des benchmarks, les trois méthodes ont résolus le meme nombre d'instances. En comparant ces méthodes en terme de temps moyen CPU requis pour la résolution d'une instance, nous pouvons remarquer que STR-FPTCM resoud les instances de chaque benchmark en un temps moyen CPU largement inferieur à celui requit par STR-Slice tandis que les deux méthodes STR-FPTCM et STR2 sont compétitives, tel qu'on peut bien voir que pour certains benchmarks (crossword-lexVg, crossword-wordsVG, modifiedRenault, randsJC7500) la méthode STR-FPTCM resouds les instances en un temps CPU moyen plus petit par rapport a celui obtenu par STR2 et pour le reste des benchmarks le temps CPU moyen obtenu par STR2 est inférieur à celui obtenu par STR-FPTCM.

TABLE 5.9 – Comparaison de la méthode STR-FPTCM avec les deux méthodes STR-Slice et STR2 en terme de taux de compression, nombre d'instances résolues en un temps limite de 1 800 s ainsi que le temps CPU moyen de résolution.

benchmark	STR-FPTCM			STR-Slice			STR2	
	$inst_s$	$CPU_t(s)$	cmp_{rate}	$inst_s$	$CPU_t(s)$	cmp_{rate}	$inst_s$	$CPU_t(s)$
bddLarge	35	75.2	63.4	35	382	58	35	65.7
bddSmall	35	37.8		35	195		35	37.5
crossword-lexVg	37	163	36.1	28	325	29.4	34	352
crossword-words	23	35	21.3	23	157	18.8	23	46.5
modifiedRenault	39	85.7	78.8	39	82.2	37.06	39	150
randsJC2500	10	23.1	32.1	10	35.2	28.4	10	12.3
randsJC5000	10	311	36.7	10	553	34.8	10	130
randsJC7500	9	320	40	9	1373	38.4	9	563
randsJC10000	5	1125	41.2	0	TO	40.9	8	750

5.8 Conclusion

Dans cette section nous avons présenté notre méthode de compression FPTCM qui est une amélioration de la méthode Sliced-table. La méthode FPTCM utilise l'algorithme des topK pour fixer, pour chaque contrainte table, la valeur du seuil minimal de fréquence S_{min} et la fonction taux de compression pour énumerer les itemsets fréquents pertinents à la compression. Les résultats expérimentaux ont montré l'efficatité de notre méthode par rapport à la méthode Sliced-table en terme de taux de compression et temps de résolution des CSPs compressés.

Chapitre 6

Méthode de Compréssion basée sur les Itemsets Frequent Maximaux (MFIC)

6.1 Présentation de la méthode MFIC

Dans la section précedente, nous avons proposé une nouvelle méthode, appellée FPTCM, permettant de maximiser le taux de compression d'une contrainte table. Les expérimentations ont montré l'éfficacité de la méthode en terme de taux de compression et son impact positif sur l'accélération du processus de résolution des CSPs. Dans cette deuxième contribution, nous nous sommes posé la question suivante : *Maximiser le taux de compression des contraintes table est-il suffisant pour accélérer le processus de résolution de CSPs ?*. Pour répondre à cette question, nous nous somme intéressés à la compression des contraintes table en exploitant des motifs plus long *i.e.*couvrant un nombre maximal de variables de scope des contraintes table et couvrant le plus grand nombre de tuples. La compression de contraintes tables par motifs fréquents nous permet d'éviter le parcours des parties redondantes durant le processus de recherche de solution. Pour éliminer un maximaum de redonances et mieux compresser une contrainte table, le mieux sera d'extraire les motifs les plus long et les plus fréquents. Ces deux caractéristiques sont celles des motifs fréquents maximaux (MFI)(définition 6.2). Mais la compression avec MFI représente plusieurs défis : (i) plus la taille d'un MFI est plus grande moins est sa fréquence. (ii) un tuple d'une contrainte table peut etre compressé par un et seul MFI mais la plupart des méthodes de fouille des MFI proposées dans la litérature ne prennent pas en considération la propriété de non chevauchement des couvertures lors de la recherche des MFI. Chercher à maximimiser la taille et la fréquence d'un MFI revient à maximiser son aire qui une combinaison de la fréquence et la taille d'un MFI.

Pour cela, nous avons proposé une heuristique appelée MFIC pour la compression des contraintes table basée sur la notion d'aire pour selectionner les MFI permettant de mieux de réduire la taille des tuples et dont les couvertures ne se chevauchent pas. Cela permet de garantir qu'un tuple ne soit comperessé qu'avec un et un seul MFI. Les principales étapes de notre heuristique MFICsont illustrées par Algorithm 14 : (i) fixer pour

chaque contrainte table, une valeur de S_{min} et énumerer tout les MFI ayant une fréquence supérieure ou égale à S_{min} . (ii) filtrer l'ensemble des MFI pour garder uniquement ceux dont les couvertures ne se chevauchent pas et qui offrent une meilleure compression de la contrainte table. (iv) construire la contrainte table compressée. Ces étape sont détaillées dans les sous séctions suivantes.

Algorithm 14 MFIC

Data : $\mathscr{T}\mathscr{D}$: contrainte table c à compression, k : nombre de CFI pour fixer S_{min}.
Result : R^c : contrainte table compressée.

$\mathscr{F}^k \leftarrow \texttt{topK}(\mathscr{T}\mathscr{D}, \texttt{k})$;
▷ seuil minimal de fréquence S_{min};
$S_{min} \leftarrow \min_{F \in \mathscr{F}^k} freq(F)$;
$\mathscr{M} \leftarrow \texttt{LCMmax}(\mathscr{T}\mathscr{D}, \texttt{S}_{\texttt{min}})$;
$\mathscr{M}^* \leftarrow \texttt{Select}(\mathscr{M})$;
$R^c \leftarrow \texttt{CompressedTable}(\mathscr{M}^*, \mathscr{T}\mathscr{D})$;
Retourner R^c;

Trouver les motifs fréquents maximaux (MFI) les plus pertinents pour la compression d'une contrainte table est un réel défi :

— Un tuple d'une contrainte table peut etre compressée par un et un seul MFI. La plupart des algorithms de fouille de MFI ne prennent pas en considération la contrainte de non-chevauchement des couvertures des MFI. Pour cela, nous devons procéder au trie de l'ensemble des MFI candidats pour garder uniquement ceux dont les couvertures ne se chevauchent pas.

Avec la méthode MFIC, nous avons essayé de résoudre ces défis en suivant les étapes illustrées dans Algorithm 14 : (i) fixer pour chaque contrainte table, une valeur de S_{min} en exploitant l'algorithme des topK motifs fréquents fermés (CFI). (ii) énumerer tout les MFI ayant une fréquence supérieure ou égale à S_{min} . (iii) filtrer l'ensemble des MFI pour garder uniquement ceux dont les couvertures ne se chevauchent pas et qui offrent une meilleure compression de la contrainte table. (iv) construire la contrainte table compressée. Ces étape sont détaillées dans les sous séctions suivantes.

6.2 Choix de seuil minimal de fréquence S_{min}

Une contrainte table peut contenir un trés grand nombre de MFI qu'il devient difficil de les parcourir en un temps raisonable. Il est donc nécéssaire de choisir une valeur de seuil minimal de fréquence S_{min} qui nous permettera d'énumérer uniquement les meilleurs MFI

candidats pour la compression. Avec une petite valeur de S_{min}, le nombre de MFI peut etre exorbitant qu'il peut etre très difficile d'identifier les plus pertinents pour la compression. Mais aussi, avec une très grande valeur de S_{min} on peut ne pas du tout énumerer des MFI candidats. Le choix de fixer de façon dynamique la valeur de S_{min} pour chaque contrainte table en exploitant l'algorithme des topK CFI est motivé par le fait que les MFI représentent un sous-ensemble des CFI respectant la propriété de maximalité. Le meme algorithme est utilisé dans la méthode FPTCM de la séction précédante, mais avec MFIC, pour une meilleure approximation de S_{min} nous varions la valeur de k pour le calcul des topK ensuite nous calculons la moyenne des S_{min} retournés par la méthode des topK.

Example 6.2.1 *Considérons la contrainte table représentée dans Table 5.3. En appliquant la méthode des* topK *sur cette table et en variant la valeur de k, soit* $k \in \{5, 10, 15, 20, 30, 35\}$. *Les plus petites valeurs de fréquence retournées par la méthode des* topK CFI *sont données par Table 6.1. La valeur de* S_{min} *correspond à la moyennes des valeurs de fréquences trouvées i.e.*S_{min} *=5.*

TABLE 6.1 – Caption

	k=5	$k=10$	$k=15$	$k=20$	$k=25$	$k=30$	$k=35$
fréquence	9	7	6	5	4	4	3

6.3 Calcul des MFI candidats

La seconde étape de la méthode MFIC consiste à énumérer les motifs fréquents maximaux ayant une fréquence supérieure ou égale à S_{min}. L'utilisation des maximaux pour la compréssion assure que les tuples soient compréssés par les motifs les plus long et plus fréquents. L'extraction des MFI se fait par l'algorithme LCMmax [36] qui est une extension de l'algorithme Linear Time Closed frequent itemsets Mining (LCM) [36] pour la recherche des MFI. L'avantage de LCMmax est qu'elle a une compléxité linéaire. LCMmax énumère tout les CFI en exploitant le backtracking, ensuite, selectionne ceux qui sont maximaux. C'est un algorithme basé sur l'élagage et la verification de la maximality des itemsets afin d'accélérer le temps de clacul et eviter la sauvegarde en mémoire des MFI retournés auparavant.

Example 6.3.1 *En considérant la valeur de* S_{min} *calculée à l'étape précedente et en appliquant* LCMmax *sur la table Table* **??**, *les motifs fréquents maximaux ayant une fréquence*

supérieure ou égale à S_{min} sont donnés dans Table 6.2. La première colonne correspond aux `MFI`*, la deuxième colonne correspond aux couvertures des* `MFI` *et la troisième colonne correspond aux valeurs de fréquence des* `MFI`*. La méthode* `LCMmax` *a retourné dix*

TABLE 6.2 – `MFI` énumérés avec S_{min}=5

Id	`MFI`	couverture	fréquence
1	$\{x_0{=}1, x_1{=}2, x_2{=}1\}$	$\{0,1,2,3,4,5,6\}$	7
2	$\{x_1{=}2, x_4{=}0, x_5{=}0\}$	$\{0,3,7,8,15\}$	5
3	$\{x_2{=}1, x_4{=}0, x_5{=}0\}$	$\{0,3,17,18,19,19,22\}$	7
4	$\{x_1{=}2, x_{\cdot}{=}1, x_4{=}0\}$	$\{0,2,8,9,12\}$	5
5	$\{x_0{=}1, x_1{=}2, x_3{=}2\}$	$\{3,4,5,6,15,16\}$	6
6	$\{x_2{=}1, x_3{=}1\}$	$\{3,4,5,6,20,22\}$	6
7	$\{x_0{=}1, x_1{=}2, x_3{=}1, x_4{=}1\}$	$\{2,10,11,13,14\}$	5
8	$\{x_0{=}1, x_1{=}2, x_5{=}2\}$	$\{5,6,9,11,14,16\}$	6
9	$\{x_0{=}1, x_1{=}2, x_2{=}2\}$	$\{7,8,9,10,11\}$	5
10	$\{x_0{=}1, x_1{=}2, x_2 = 3\}$	$\{12,13,14,15,16\}$	5

`MFI` *candidats pour la compression.*

6.4 Filtrage des `MFI` candidats

Les couvertures des `MFI` énumérés dans l'étape précedente se chevauchent entre elles, alors qu'un tuple peut etre compressé avec un et un seul `MFI`. Il faut donc filtrer l'ensemble des `MFI` pour garder uniquement ceux dont les couvertures ne se chevauchent pas et qui compressent au mieux les tuples de la contrainte table. Pour cela, nous avons proposé une heuristique basée sur la notion d'aire d'un `MFI` (définition 3.1.5). Les `MFI` énuméres sont comparés deux à deux. Si les couvertures de deux `MFI` se chevauchent, seul l'`MFI` ayant une plus grande valeur de l'aire sera retnue. Pour s'assurer que seul les `MFI` ayant une plus grande valeur de l'aire seront seléctionné, notre heuristique calcul l'aire de chaque `MFI` ensuite ordonne l'ensemble des `MFI` dans un ordre décroissant selon les valeurs de l'aire trouvées. Cela permet de favoriser les `MFI` permettant de mieux réduire la taille des tuples de leurs couvertures. Algorithme 15 résume les étapes de séléction des `MFI` pertinents pour la compression. Nous avons utilisé pour cela, une nouvelle structure E composée de : un `MFI` p, l'aire *aire* de p ainsi qu'une fonction, appellée *Comparator* qui permet de comparer l'aire des `MFI`. L'algorithme commence par calculer pour chauque `MFI` m valeur d'aire $area_m$ et ajoutee le couple $(m, area_m)$ dans une liste L de type E. Ordonner les élements de L dans un ordre décroissants selon l'aire des `MFI`. Ensuite, séléctionner le premier `MFI` m de la liste L (le `MFI` ayant la plus grande valeur de l'aire),

l'ajouter à la liste S des MFI pertinents pour la compression, supprimer de L le MFI m ainsi que tout les MFI dont les couvertures se chevauchent avec m. Cette derniere étape est répétée jusqu'à ce qu'il n'y a plus de MFI à parcourir dans L.

Algorithme 15 `Select()`

```
   Data : 𝒜 : Liste de MFI.
   Result : 𝒮 : Liste de MFI pertinents.
 1 struct {
 2 |   p : ITEMSET;
 3 |   aire : RÉEL;
 4 |   Function Comparateur :
 5 |   |   if aire > autres.aire then
 6 |   |   |   Retourner vrai ;
 7 |   |   else
 8 |   |   |   Retourner faux;
 9 |   |   end
10 |   end
11 } E;
12 L ← nouvelle liste de E;
13 for m ∈ 𝒜 do
14 |   aire ← taille(m) ∗ cover(m);
15 |   L.inserer(new E(m, aire));
16 end
17     ▷ éléments de L ordonnés dans un ordre décroissant selon l'attribut aire.;
18 L.ordonner();
19 while L ≠ ∅ do
20 |   m ← L.premier();
21 |   mathcalS.ajouter(m);
22 |           ▷ supprimer de L l'itemset séléctionné m. L.supprimer(m);
23 |   i ← 0;
24 |   while i < L.taille() do
25 |   |   if cover(m) ∩ cover(L.lire(i).p) then
26 |   |   |   ▷ supprimer l'itemset avec l'index i dont la couverture se chevauche
   |   |   |   avec m.;
27 |   |   |   L.supprimer(i) ;
28 |   |   else
29 |   |   |   i ← i + 1 ;
30 |   |   end
31 |   end
32 end
33 Retourner S;
```

Example 6.4.1 *Dans cet exemple nous allons filtrer les* MFI *de Table 6.2 pour garder uniquement ceux qui sont pertinents à la compression de la contrainte table donnée par Table 5.3. Nous commençons par calculer l'aire de chaque* MFI *ensuite ordonner l'ensemble des* MFI *dans un ordre décroissant des valeurs de l'aire. La table ordonnée est*

donnée dans Table 6.3. La prmiere colonne correpond aux indices des MFI *et la deuxième colonne à l'aire des* MFI.

TABLE 6.3 – MFI ordonnés dans un ordre décroissant de leur aire.

id	aire
1	21
2	20
4	20
7	20
3	18
5	18
8	18
9	15
10	15
6	12

On selectionne le premier MFI *de Table 6.3 qui correspond au* MFI *avec id=1 dont la valeur de l'aire est la plus grande. On ajoute cet* MFI *à la liste S des* MFI *pertinents pour la compression et on le supprime de Table 6.3.* $S = \{(\{x_0{=}1, x_1{=}2, x_2{=}1\}, \{0,1,2,3,4,5,6\})\}$. *On remarque que les couvertures de* MFI *dont avec les indices 2, 3, 4, 5, 6, 7, 8 se chevauchent avec l'*MFI *avec l'indice 1, alors on les supprime de la table car ils ne peuvent pas contribuer à la compression. La table mise à jour est donnée par Table* **??**. *Il reste que deux* MFI *d'indices 9 et 10. On selectionne le* MFI *avec indice 9, on l'ajoute à S (*$S = \{(\{x_0{=}1, x_1{=}2, x_2{=}1\}, \{0,1,2,3,4,5,6\})\}, (\{x_0{=}1, x_1{=}2, x_2{=}2\}, \{7,8,9,10,11\})$*) et on le supprime de Table* **??**. *le* MFI *d'indice 10 ne se chevauve pas avec* MFI *d'indice 9 alors il est lui aussi seléctrionné pour la compression et donc ajouté à S. Finalement la liste des* MFI *pertinents à la compression sont :*

— $\{x_0{=}1, x_1{=}2, x_2{=}1\}, \{0,1,2,3,4,5,6\}$

— $\{x_0{=}1, x_1{=}2, x_2{=}2\}, \{7,8,9,10,11\}$

— $\{x_0{=}1, x_1{=}2, x_2 = 3\}, \{12,13,14,15,16\}$

6.4.1 Création de la contrainte table compressée

La création de la contrainte table compressée se fait de la meme manière que la méthode FPTCM. Les couvertures des MFI pertinents pour la compression sont déja connue, alors, la méthode crée un fragment pour chaque MFI en compressant directement les tuples dont les indices sont donnés par les couvertures des itemsets (Algorithme 12).

TABLE 6.4 – Contrainte table compressée obtenue avec S_{min}= 3.

Premier fragment e_1.

x_0	x_1	x_2	x_3	x_4	x_5
			1	0	0
			1	0	1
			1	1	1
1	2	1	2	0	0
			2	1	1
			2	1	2
			2	2	2

(b) Deuxième fragment e_2.

x_0	x_1	x_2	x_3	x_4	x_5
			0	0	0
			1	0	0
1	2	2	1	0	2
			1	1	0
			1	1	2

(c) Troisième fragment e_3.

x_0	x_1	x_2	x_3	x_4	x_5
			1	0	1
			1	1	1
1	2	3	1	1	2
			2	0	0
			2	2	2

(d) Table par defaut DF.

x_0	x_1	x_2	x_3	x_4	x_5
1	3	1	0	0	0
2	0	1	0	0	0
2	1	1	0	0	0
2	1	1	2	0	1
2	1	1	4	0	1
3	3	1	2	0	0

6.5 Résultats éxpérimentaux

Dans cette section, nous présentons les différentes expérimentations menées ainsi que les resultats obtenus afin de valider notre proposition . La section est organisé comme suit, dans un premier temps, nous présentons le protocol expérimental et les mesures de performances. Ensuite, rapporter les résultats relatifs au calcul du seuil minimal de fréquence S_{min}. Comparer notre heuristique `MFIC` à la méthode `STR-Slice` ainsi que la méthode STR2. Enfin comparer notre méthode `MFIC` à notre méthode précedante `FPTCM`ainsi qu'à un ensemble de méthodes de l'état de l'art basées sur GAC.

Protocol éxpérimental et mesures de performances

Notre méthode est implémentée dans le solveur Oscar [1] développé en Scala. Nous avons menés des expérimentations en utilisant une machine Intel (R) Core(TM), $i5-7200$ CPU, 2.5 GHz avec une RAM de 4 GB, avec un distribution Ubuntu sur 64 bits. Ces expérimentations ont été effectuées sur les memes benchmarks utilisés dans la partie expérimentale du chapitre précedent 5.7 . Afin de mesurer les performances de notre pro-

1. Solveur disponible sur https://bitbucket.org/oscarlib/oscar/src/dev/

position et terme de compression et de temps de résolution des CSPs, nous nous sommes basés sur les critères suivants : nombre (en pourcentage) de tuples compressés (tup_{cmp}), taille moyenne des MFI utilisés pour la compression $taille_{moy}$, fréquence moyenne d'un MFI $freq_{moy}$, nombre d'MFI par instance $nbr_{itemsets}$ et enfin le temps de résolution CPU_t donné en secondes. Un temps limite de 1800 secondes est fixé, dépassant cette limite, un time out (TO) sera posé.

Choix du seuil minimal de fréquence S_{min}

Afin de fixer la valeur du seuil minimal de fréquence S_{min} à utiliser pour énumerer les MFI candidats pour la compression, nous utilisons l'algorithme des topK sur les CFI. La valeur de k doit etre fixé par l'utilisateur. Pour cela, nous avons mené un ensemble d'expérimentations afin d'étudier l'impact de k sur les resultats de compression et sur le temps de résolution et enfin aboutir à un meilleur choix de la valeur de k. Dans ce qui suit, nous avons choisi arbitrairement 16 instances à partir des différents benchmarks et nous avons varié la valeur de k entre 20 et 60 pourcent (%) du nombre de tuples dans une contrainte table pour le calcul de S_{min}. Nous avons ensuite rapporté dans Table 6.5 le taux moyen de compression $rate_c$ ainsi que le temps moyen de résolution CPU_t obtenus pour ces instances avec les différentes valeurs de k. On peut remarquer que plus la valeur de k est plus grande plus le taux de compression est plus grand. Cela s'explique par le fait que plus k est plus grand, moins est la valeur de S_{min}. Donc plus d'MFI énumérés et utilisés pour la compression d'une contrainte tables. Mais comme on peut bien le voir, un meilleur taux de compression ne permet de toujours d'accélerer le temps de résolution du CSP compressé.

Il n'y a pas une valeur de k pour laquelle toutes les instances sont résolues en un temps meilleurs, pour cela, dans la suite des expérimentations, le S_{min}prend comme valeur la moyenne des valeurs de fréquence retournées par l'algorithme des $topk$ en variant la valeur de k entre 20 et 60 pourcent de tuples dans une contrainte table à compresser.

6.5.1 Comparaison de STR-MFIC *avec* STR-Slice *et* STR2

Les deux méthodes STR-MFIC et STR-Slice [1] exploitent les motifs fréquents pour la compression et utilisent la meme structure de contrainte table compréssées. Il existe deux différences majeures entre ces deux méthodes : (i) la méthode STR-MFIC fixe de façon dynamique la valeur de S_{min} pour chaque contrainte table alors que la méthode

TABLE 6.5 – Taux de compression $Rate_c$ et temps de résolution CPU_t obtenus pour 16 instances choisies arbitrairements en variant la valeur de k entre 20% et 60% du nombre de tuples des contraintes à compresser.

Instance	$k = 20\%$		$k = 30\%$		$k = 40\%$		$k = 50\%$		$k = 60\%$	
	$rate_c$	CPU_t	$rate_c$	CPU_t	$rate_c$	CPU_t	$rate_c$	CPU_t	$rate_c$	CPU_t
BddLargre31	11	**33.6**	11.6	34.7	12	35	13	36.2	14	**33.9**
BddLargre24	10	84	11	94	12	**78.5**	12.4	82.1	13	84
BddLargre17	11	**63**	12	**63**	12.8	65.4	13	65.9	14	64.4
randsJC2500-4	14	6.8	15	6.5	16	**5.1**	17	7.9	18	8
randsJC2500-9	15	6.7	15.3	7.4	15.9	5.3	17	**4.9**	18	**4.9**
randsJC5000-0	14	65	14.6	70	16.6	70	15	**62**	17.5	80
randsJC5000-9	13	**38**	14	**38**	14.5	**38**	16	52	17.6	46
LexVg4-8	20	**49**	23	52	25	54	26	60	29	73
LexVg5-6	20	**18**	23	19	24	26	27	21	28	23
LexVG7-11	17.5	240	18	250	18	250	19	234	20	**223**
LexVg8-11	18	183	18.5	**168**	19	176	19.4	208	20.4	284
LexVg8-12	18	**9.3**	18.4	9.5	19	13	19.3	13	21	14
WordsVg9-12	19	38	20	37	22	33	26	11	29	**16**
WordsVg10-13	18.5	19.4	19.4	20	20.5	22	21.3	14	22.3	**12**
WordsVg5-7	21.5	124	24	128	25	134	26	129	27	**120**
WordsVg11-12	12.5	18.6	14	19.6	16	**17**	19	17.5	20	18.5

`STR-Slice` fixe S_{min} à 2 pour toutes les contrainte table. (ii) La méthode `STR-MFIC` compresse les contrainte tables en utilisant les motifs fréquents maximaux alors que `STR-Slice` utilise les motifs fréquents. Les deux méthodes sont basées sur `STR-Slice` pour la résolution des CSPs compressés. La méthode `STR-Slice` est une version optimisée de `STR2` pour les CSPs compressés. D'ou l'intéret de comparer ces trois méthodes.

Table 6.6 rapporte pour chacune des méthode `STR-MFIC`, `STR-Slice` et `STR2` et pour chaque benchmark le nombre d'instances $inst_s$ résolues en un temps limite de 1800s et le temps moyen CPU requis pour la résolu d'une instance.

— *Nombre d'instances résolues* : le nombre d'instances résolues est le meme pour les trois méthodes à l'exception du benchmark *randsJC1000* pour lequel la méthode `STR-Slice` ne résoud aucune instance et le benchmark *crossword-lexVg* pour lequel les deux méthodes `STR-MFIC` et `STR2` arrivent à résoudre 7 instances de plus par rapport à `STR-Slice`.

— *Temps CPU moyen* : Pour la plupart des benchmarks, le temps CPU moyen que prend `STR-MFIC` pour résoudre une instance est inférieur au temps CPU moyen de `STR-Slice` et `STR2`. Sauf pour le benchmark *crossword-lexVg* pour lequel le meilleur temps CPU moyen requis pour la resolution d'une instance est obtenu

avec STR-Slice et pour les deux benckmark *randsJC5000* et *randsJC7500* dont le meilleur temps CPU moyen est obtenu avec la méthode STR2.

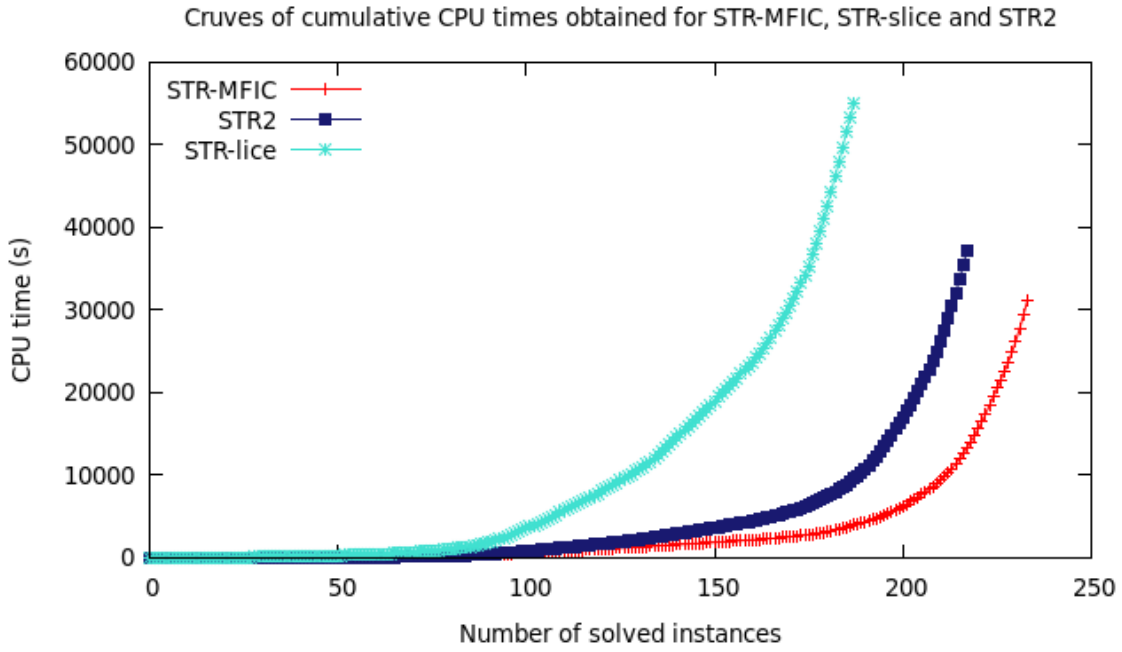
TABLE 6.6 – Nombre d'instances résolues $inst_s$ et temps CPU moyen CPU_t de résolution d'une instance des différents benchmarks avec STR-MFIC, STR-Slice et STR2.

benchmark	STR-MFIC		STR-Slice		STR2	
	$inst_s$	$CPU_t(s)$	$inst_s$	$CPU_t(s)$	$inst_s$	$CPU_t(s)$
bddLarge	*35*	**60**	*35*	382	*35*	65
bddSmall	*35*	**28**	*35*	195	*35*	37
crossword-lexVg	*35*	338	*28*	**325**	*35*	352
crossword-words	*23*	**33**	*23*	157	*23*	46
modifiedRenault	*39*	**72**	*39*	88	*39*	150
randsJC2500	*10*	**11**	*10*	35	*10*	12
randsJC5000	*10*	154	*10*	553	*10*	**130**
randsJC7500	*10*	636	*10*	1373	*10*	**563**
randsJC10000	*10*	**703**	*0*	TO	*10*	750

Figure 6.1 représente les courbes des temps CPU cumulés obtenus pour STR-MFIC, STR-Slice et STR2 après la résolution de 250 instances de l'ensemble des benchmarks. On remarque que pour les 60 premières instances, les courbes des trois méthodes sont presque identiques. Ensuite, on voit bien que la courbe de STR-Slice est dominée par celles de STR2 et STR-MFIC sur le reste des instances. Les deux courbes de STR2 et STR-MFIC restent très rapprochées pour les premières 100 instances résolues puis la courbe de STR-MFIC domine completement celle de STR2 pour le reste des instances. On remarque aussi que la méthode STR-MFIC a résolu plus d'instances par rapport à STR-Slice et STR2.

Bien que les deux méthodes STR-MFIC et STR-Slice compressent les contraintes table avant la résolution, pour la plupart des instances, STR-MFIC prend moins de temps pour la résolution. Afin d'expliquer cette différence, nous avons rapporté dans Table 6.7 quelques détails sur compression des différents benchmarks. Pour chaque benchmark et pour chacune des méthodes nous avons rapporté le nombre en pourcentage (%) de tuples compressés *c-tup*, le taux de compression des benchmarks *c-rate*, le nombre moyen d'itemsets utlisés pour la compression $|M|$, la taille moyenne d'un itemset $|u|$ et la fréquence moyenne d'un itemset $freq(u)$. La table montre que STR-MFIC compresse moins de tuples et obtient un taux de compression inférieur à celui obtenu avec STR-Slice. Mais contrairement à STR-Slice, STR-MFIC compresse les contraintes table avec beaucoup moins d'itemsets ayant de grandes valeurs de fréquence. Par exemple, STR-MFIC compresse en moyenne environ 20% de tuples du benchmark *randsJC2500* en utilisant

FIGURE 6.1 – Courbe des temps de résolution cumulés obtenus par les méthodes STR-MFIC, STR-ST et STR2 pour les instances des différents benchmarks. L'axe des X représente les instances et l'axe des Y représente le temps cumulé.



seulement 86 MFI ayant une fréquence moyenne de 16. La méthode STR-Slice compresse environ 20% de tuples en plus par rapport à STR-MFIC en utilisant 470 itemsets d'une fréquence moyenne de 3. La méthode STR-Slice compresse les contraintes avec un très grand nombre d'itemsets couvrant un petit nombre de tuples, ce qui peut ralentir le processus de résolution vu le grand nombre de petites entrées à parcourir.

6.5.2 *Comparaison de* STR-MFIC *avec des méthodes de l'état de l'art basées sur les algorithmes* GAC

Dans cette section nous allons comparer notre méthode STR-MFIC à un ensemble de méthodes, basée sur GAC, proposées dans la litérature. Principalement, nous allons nous comparer à certaines versions plus récentes de STR : STR3 [43], shortSTR2 [49] et STRBit [44]. Afin de diversifier notre comparaison, nous allons aussi comparer STR-MFIC à la méthode MDD4R [42] basée sur MDD, GAC4 [50], GAC4R [42] ainsi que CT [45] basée sur STR et la représentation en bitset. Toutes ces méthodes sont déjà implémentées dans le solveur Oscar dans lequel nous avons implémenté notre méthode STR-MFIC.

Pour comparer la méthode STR-MFIC aux différentes méthodes citées, nous avons utilisés les memes benchmarks que les éxpérimentations précédantes. Figure 6.2 représente les courbes des temps de résolutions cumulés obtenue pour STR-MFIC (courbe en

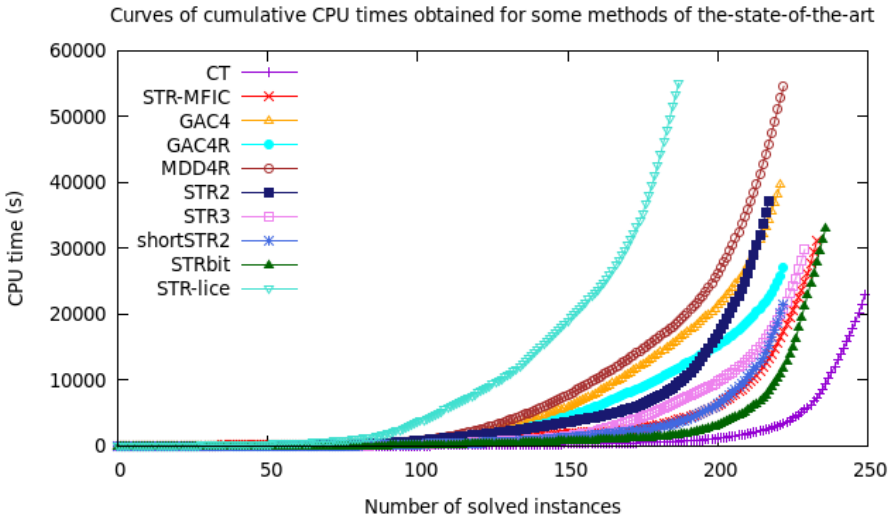
TABLE 6.7 – Comparer STR-MFIC et STR-Slice en nombre de tuples compressés *c-tup*, taux de compression *rate*, nombre moyen d'itemsets dans une instance $|M|$, taille moyenne d'un itemsets $|u|$ et la fréquence moyenne d'un itemset $freq(u)$.

benchmark	method	c-tup(%)	c-rate(%)	$\|M\|$	$\|u\|$	$freq(u)$
Crossword-LexVg	STR-MFIC	44.3	22.1	88	4	18
	STR-Slice	57.13	29.45	667	3	2
Crossword-WordsVg	STR-MFIC	52.04	29.04	95	4	30
	STR-Slice	69.4	18.8	376	4	3
randsJC2500	STR-MFIC	39.2	17.9	86	3	16
	STR-Slice	59.27	28.04	470	3	3
randsJC5000	STR-MFIC	38.14	15.6	98	3	24
	STR-Slice	71.26	34.87	1039	3	3
randsJC7500	STR-MFIC	36.17	15.8	91	3	28
	STR-Slice	76.44	38.4	1604	3	3
randsJC10000	STR-MFIC	35.5	16.1	91	3	32
	STR-Slice	79.54	40.91	2177	3	3
bddLarge	STR-MFIC	24.5	13.4	25	7	82
	STR-Slice	91	58	1655	6	3

rouge) et chacune des méthodes séléctionées sur l'ensemble des instances des différents benchmarks. On voit bien que CT et STRbit basées sur la représentation bitset dominent toutes les méthodes y compris STR-MFIC. Bien que la courbe de STRbit est en dessous de celle de STR-MFIC, les deux courbes se rapprochent de plus en plus jusqu'à ce qu'elle soient presque collées sur les quelques dernières instances. Les deux méthodes shortSTR2 et STR-MFIC sont compétitives, tel que leur courbes sont presque quasi identiques sur les 200 prmières instances ensuite la courbe de STR-MFIC domine celle de STRshort. Comme il est bien clair que STR-MFIC résoud plus d'instances. Pour le reste des méthodes, il est bien clair que STR-MFIC domine et résoud plus d'instances.

Pour comparer les temps de résolution obtenus par les différentes méthodes pour chacun des benchmarks indépendemanent, nous avons donné dans Figure 6.3 les coubes des temps cumulés obtenus par ces méthodes pour trois benchmarks *Crossword-words-vg*, *Crossword-lex-vg* et *randsJC10000* choisis de façon arbitraire. On remarque que pour les trois benchmarks, STR-MFIC domine les méthodes STR-Slice, STR3 [43], shortSTR2 [49], MDD4R [42], GAC4 [50] et GAC4R [42] et résolud plus d'instances. Concerant les deux mé-

FIGURE 6.2 – Courbes des temps CPU (s) cumulés obtenues pour les différentes méthodes séléctionnées pour notre comparaison. L'axe des x représente les instances résolues et l'axe des y représente les temps cumulés.



thodes STRbit [44] et CT [45] :

— STRbit [44] : la courbe obtenue pour STR-MFIC sur le benchmark *Crossword-words-vg* domine celle obtenue pour STRbit [44] mais cette dernière résoud plus d'instances. Pour le benchmark *Crossword-lex-vg*, les deux méthodes STR-MFIC and STRbit [44] arrivent à résoudre le meme nombre d'instances et leur courbes sont compétitives. Enfin, pour le benchmark *randsJC10000*, la méthode STRbit [44] domine STR-MFIC en terme de temps de résolution et nombre d'instances résolues.

— CT [45] : pour les deux benchmarks *Crossword-words-vg* et *Crossword-lex-vg*, les coubes de temps cumulés obtenues pour STR-MFIC et CT [45] sont presque idenditques pour les 28 premières instances de chacun des benchmarks, puis les deux coubes de CT [45] dominent celle de STR-MFIC.

6.6 Conclusion

Dans cette section, nous avons présenté une nouvelle approche de compression des contraintes tables, appelée MFIC, basée les motifs fréquents maximaux et la notion d'air d'un motif. Afin de remédier au problème de seuil minimal de fréquence S_{min}, nous avons proposé d'exploiter l'approche des topK motifs fréquents afin de fixer de façon dynamique la valeur de S_{min} pour chaque contrainte table. Afin de mieux compresser les

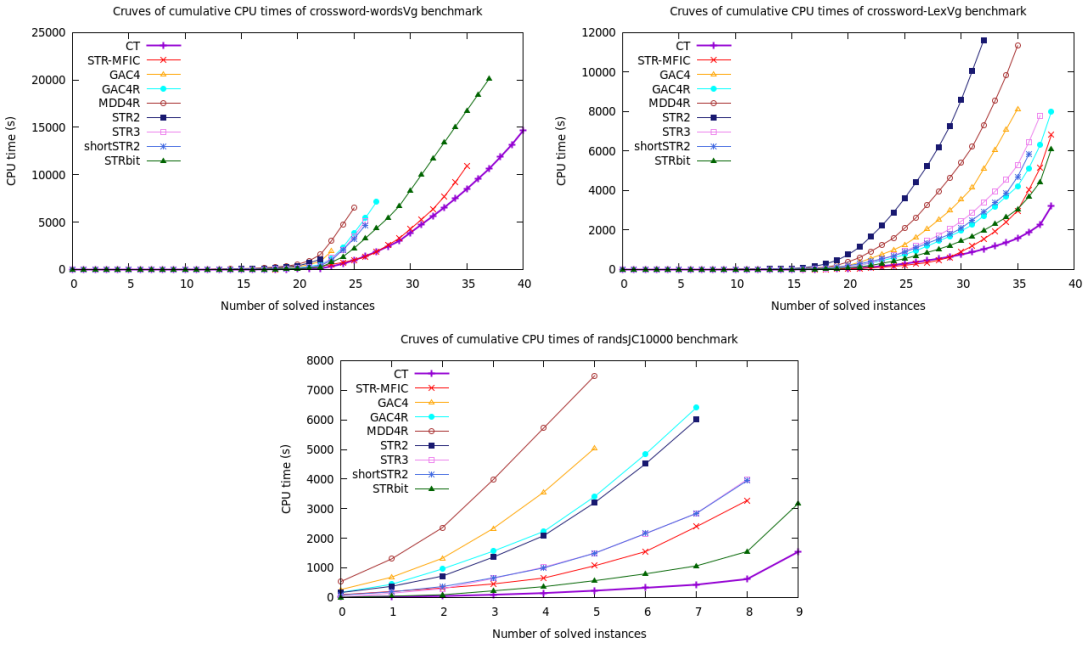


FIGURE 6.3 – Courbes des temps de résolutions cumulés obtenues pour `STR-MFIC` et les différentes méthodes de notre comparaison sur les trois benchmarks *crossswords-words-vg*, *crossswords-lex-vg* and *randJC10000*.

tuples d'une contrainte table, l'idéal serait d'utiliser les motifs les plus long et plus fréquents. Pour cela nous avons utilisé la notion d'aire qui est une combinaison de la taille d'un motifs fréquent et de la taille de sa couverture afin de sélectionner les `MFI` les plus pertinents pour la compression. Afin d'éviter le non chevauchement des `MFI` nous avons choisi les motifs ayant la plus grande valeur de l'aire et dont les couvertures ne se chevauchent pas. Pour résoudre les CSPs compressés, nous avons utilisé
STR-slice [1] qui est une optimisation de la méthode `STR2` [23] pour CSP compressés. Nous avons mené un ensemble d'expérimentations durant lequelles nous avons comparé notre méthode `MFIC`à un ensemble de méthodes de l'état de l'art, principalement les variantes de STR (STR2 [23], STR3 [43], STR-Slice [1], shortSTR2 [49] and STRbit [44]), GAC4 [50], GAC4R [42], MDD4R [42] et CT [45]. Les résultats nous ont permis de montrer que la compression des contraintes tables avec la méthode `MFIC`permet de résoudre les CSPs compressés en un temps inférieur par rapport à la plus part des méthodes de l'état de l'art et reste compétitives avec la méthode CT [45] et STR-bit [44].

Chapitre 7

Exploitation de la technologie Spark et de Clustering pour distribuer la compression des données volumineuses à base de motifs fréquents maximaux

Dans les deux chapitres précédents, nous avons présentés deux méthodes itératives pour la compression de contraintes tables volumineuses. Ces deux méthodes ont montrés leur efficacité en terme de qualité de compression ainsi qu'en terme de temps CPU de résolution des problèmes de satisfaction de contraintes (CSP). Dans ce chapitre, nous présentons une nouvelle méthode distribuée pour la compression de données voluminous. Cette dernière est basée sur une technique de clustering pour la répartition de données en un ensemble de clusters, sur une méthode d'énumération de motifs fréquents ainsi que sur les fonctions de Spark pour la distribution du calcul de motifs fréquents et de la compression des données. Pour valider notre proposition, nous menons un ensemble d'expérimentations sur un ensemble de datasets de UCI (University of California Irvine) [1].

Ce chapitre est organisé comme suit :

— dans la première section, nous introduisons des concepts relatifs au clustering.

— dans la deuxième section, nous présentons notre méthode de compression basée sur les fonctions de Spark et sur la méthode de clustering kmeans.

— dans la troisième section, nous discutons les résultats des expérimentations,

— Enfin, nous concluons.

7.1 Définitions

Dans cette section nous définissons le concept de clustering et son interet pour la fouille de motifs fréquents. Ensuite, nous introduisons la méthode de clustering k-means qui est utilisée dans notre proposition.

Définition 7.1.1 *(Clustering)*

Le clustering est le processus qui permet de regrouper des données présentant des similitudes dans un meme groupe. C'est un concept très utilisé dans le domaine de fouille

1. les datasets peuvent etre obtenus à partir de ce lien : http://fimi.uantwerpen.be/data/

de données. Dans le cadre de ce travail, le clustering est utilisé pour partitionner un ensemble de transactions en des clusters relativement homogènes et sans chevauchement.

Définition 7.1.2 *(k-means clustering)*

Le k-means est un algorithme de clustering non-supervisée qui permet de partitionner les données en k clusters disjoints avec le nombre k de clusters fixé à l'avance. Il permet de regrouper les données avec un grand degré de similarité dans un meme cluster. Le degré de similarité entre les données peut etre obtenu par le calcul de la distance de similiraté, telque les données fortement similaires ont une distance de similarité réduite. Il existe deux types de distances : distance euclidienne qui est la distance géometrique et la distance de Manhattan.

7.2 `FPTCM` (Frquent Pattern Tree Compression Method) parallélisée

Dans cette section nous présentons la version parallélisée de la méthode `FPTCM` présentée dans la section 5.1. Lorsque la taille des données à compresser est très grande, la consommation en temps et en mémoire peut être très élevée. Pour cela, la parallélisation des étapes de compression et la distribution de calcul sur plusieurs noeuds de calcul permet d'accélérer le temps de calcul mais aussi résoudre le problème de mémoire requise. `FPTCM` parallélisée permet de compresser des données à plus grande échelle tout en minimisant le temps de compression. Cette méthode est basée sur l'algorithme de clustering k-means qui permet la répartition des données à compresser en un ensemble de clusters disjoints, tels que les données présentant une forte similarité sont regroupées dans un même cluster. La répartition de données en k clusters disjoint permet de distribuer le processus de compression sur des machines différentes mais aussi d'améliorer la qualité de compression car plus le degré de similarité entre les données est grand plus la probabilité de trouver des itemsets plus long (couvrant un nombre maximum d'items) et plus fréquents est grande. Afin de paralléliser les étapes itératives de la méthode `FPTCM`, nous avons exploité Spark qui est une plate-forme Big Data basée sur les RDD (Resilient Distributed Data) qui permettent un partitionnement efficace des données et le maintien en mémoire des résultats intermédiaires de traitement[51]. Les étapes principales de la parallélisation de la méthode `FPTCM` sont : (i) écriture des données à compresser dans une représentation binaire, (ii) partitionner les données en K clusters disjoints, (iii) calculer sur chaque cluster la fréquence des items et les ordonner dans un ordre décroissant de

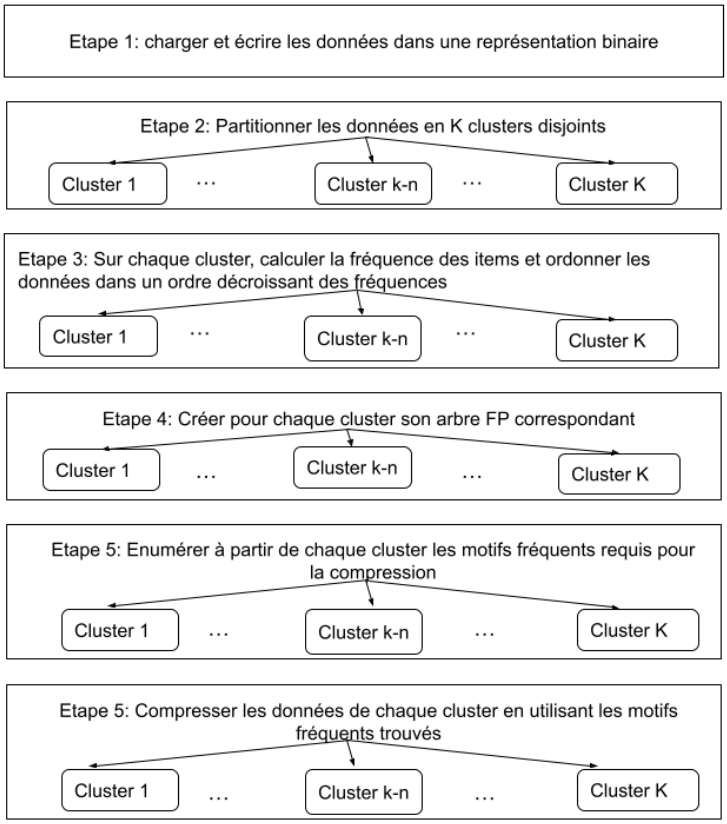


FIGURE 7.1 – Principales étapes de la méthode FPTCM parallélisée.

fréquences obtenues, (iv) créer dans chaque cluster un FP-Tree à partir des données du cluster, (v) extraire, à partir de chaque cluster, les motifs fréquents requis pour la compression des données du cluster, (vi) enfin, compresser les données de chaque cluster en utilisant les motifs fréquents retrouvés. La figure 7.1 résume les étapes de la parallélisation de la méthode FPTCM . Ces étapes sont détaillées dans ce qui suit :

Étape 1 : Chargement et représentation binaire des données

La première étape de la parallélisation de la méthode FPTCM consiste à charger les données à compresser dans des RDDs sur plusieurs machines. Tel que chaque transaction est chargée sous forme d'un vecteur binaire dense (equivalent d'un tableau à une à une dimension) pour pouvoir ensuite appliquer une méthode de clustering. L'écriture d'un vecteur dense dans une représentation binaire consiste à indiquer pour chaque item sa présence (1) ou son absence (0) dans le vecteur. Une fois que les données sont chargées dans RDDs, chaque vecteur est écrits dans une représentation binaire. En Spark, cette transformation est effectuée en appliquant la fonction *map()* sur les RDDs.

Example 7.2.1 *Soit Table 7.1 à compresser, la représentation binaire du vecteur de la transaction* t_0 *est donnée comme suit :* t'_2 = {1 1 1 0 1}. *La représentation binaires de Table 7.1 est donnée par Table 7.2*

ID	Items				
t_0	0	3	5	8	10
t_1	0	3	5	9	10
t_2	1	4	7	9	11
t_3	1	3	6	9	10
t_4	1	4	6	9	11
t_5	2	3	6	8	10
t_6	2	3	6	9	11
t_7	2	4	7	8	10
t_8	2	3	7	9	10
t_9	2	4	7	9	11
t_{10}	2	4	7	9	11

TABLE 7.1 – Table transactionnelle à compresser TD_{cmp}.

ID	Items											
t_0	1	0	0	1	0	1	0	0	1	0	1	0
t_1	1	0	0	1	0	1	0	0	0	1	1	0
t_2	0	1	0	0	1	0	0	1	0	1	0	1
t_3	0	1	0	1	0	0	1	0	0	1	1	0
t_4	0	1	0	0	1	0	1	0	0	1	0	1
t_5	0	0	1	1	0	0	1	0	1	0	1	0
t_6	0	0	1	1	0	0	1	0	0	1	0	1
t_7	0	0	1	0	1	0	0	1	1	0	1	0
t_8	0	0	1	1	0	0	0	1	0	1	1	0
t_9	0	0	1	0	1	0	0	1	0	1	0	1
t_{10}	0	0	1	0	1	0	0	1	0	1	0	1

TABLE 7.2 – Table transactionnelle à compresser TD'_{cmp}.

Étape 2 : Partitionnement de données en K clusters

Dans cette étape, la méthode partitionne les données binarisées en K clusters disjoints. Les transactions avec un grand degré de similarité sont regroupées dans un même cluster pour ensuite être compressées indépendemment des autres clusters. Une transaction peut appartenir à un et un seul cluster à la fois, Cela veut dire que les clusters doivent être sans chevauchement. Le regroupement des données avec une forte similarité dans un même cluster peut améliorer la compression. Cela, car plus les données sont similaires plus la présence de motifs fréquents requis pour la compression est garantie. La recherche de motifs fréquents à partir d'un cluster se fait indépendemment des autres clusters, c'est-à-dire, il suffit qu'un motif soit fréquent sur un cluster pour etre considéré comme motif fréquent, on a pas besoin de vérifier s'il est fréquent dans le reste des clusters. Pour partitionner les données en k clusters nous avons choisi l'algorithme de clustering k-

means parallélisé [52] qui est basé sur la technologie Spark pour la paralléliser les étapes de calcul de similarité et de mise à jour des centroïde des clusters. Il existe plusieurs algorithmes de clustering dans la littérature, notre choix du clustering avec k-means est motivé par le fait qu'il est un algorithme de clustering incrémental adapté aux données à grande échelle.

Les données binarisées seront donc partitionnées en K clusters disjoints et chaque cluster sera représenté par un data-frame qui est une structure de données adaptée pour le stockage de données volumineuses. Un data-frame est l'équivalent d'une matrice de données avec des colonnes et des lignes. Dans notre contexte, les colonnes représentent l'ensemble des items et les lignes correspondent aux vecteurs binarisés d'un cluster. L'application du clustering k-means sur Table 7.2 de l'exemple 7.2.1 donne les deux partitions représentées par Table 7.3. La colonne cl_{id} donne le numéro de la partition (cluster) tandis que la colonne $Tids$ donne les identifiant des vecteurs binarisés de chaque cluster.

cl_{id}	$Tids$					
1	t_0	t_1	t_5	t_7	t_8	
2	t_2	t_3	t_4	t_6	t_9	t_{10}

TABLE 7.3 – Les partitions obtenues après application du clustering k-means binarisé sur les données de Table 7.2.

Étape 3 : Calcul des fréquences des items

La répartition des données en k clusters a pour but de paralléliser le processus de compression, tel que chaque cluster sera traité et compressé de façon indépendante du reste des clusters. Notre méthode de compression est basée sur l'exploitation d'un arbre FP sur chaque cluster pour l'extraction des motifs fréquents requis pour sa compression. La première étape pour la construction d'un arbre FP consiste à calcules les fréquences des items. Comme chaque cluster sera compressé indépendemment des autres, la fréquences des items sera calculé sur chaque cluster. Comme chaque transaction est écrite en binaire et chaque cluster est représenté par un data-frame dont les colonnes représentent les items de $\mathscr{I}$ et les lignes représentent les colonnes, pour calculer la fréquence d'un item, il suffit de calculer le nombre des 1 dans la colonne correspondante à l'item en question. En Spark, méthode *agg()* permet d'obtenir le nombre d'apparitions d'une valeur dans une colonne d'un data-frame. Une fois que les fréquences des items sont calculées sur chaque cluster, on réordonne les lignes de chaque data-frame par ordre décroissant des valeurs de fréquence obtenues. En Spark, il suffit de réordonner les colonnes du data-frame dans un ordre décroissant des valeurs de fréquence des items. Dans les étape de construction

d'un arbre FP, une fois que les fréquences des items sont calculées, les items ayant une fréquence inférieur au seuil minimal de fréquence seront supprimés des transactions ordonnées. Dans notre cas, on ne supprime pas des lignes des data-frames les items avec une valeur de fréquence inférieur au seuil minimal de fréquence car les mêmes data-frames seront utilisés pour la création des tables compressées.

Étape4 : Création de l'arbre FP de chaque cluster

Cette étape consiste à créer pour chaque cluster son FP-tree correspondant à de son data-frame. Si on considère n comme étant le nombre d'items fréquents dans $\mathscr{I}$, la première étape de la construction de l'arbre FP consiste à sauvegarder les n items fréquents de chaque ligne du data-frame dans l'arbre de préfixe. Il est important de rappeler qu'une ligne du data-frame est un vecteur binaire dont chaque entrée indique la présence (1) ou l'absence (0) d'un item de $\mathscr{I}$ dans la transaction correspondant dans le dataset original. Alors, pour chaque ligne du data-frame et pour chaque i inférieur ou égale à n, si la valeur de la i^{eme} colonne est égale à 1, l'item correspondant à cette colonne dans la transaction d'origine sera ajouté à l'arbre de préfixe sinon l'ignorer. Au lieu de sauvegarder uniquement la valeur de fréquence, chaque noeud de l'arbre de préfixe enregistre les identifiant des transactions contenant le motif correspondant au chemin allant du noeud root au noeud courant. Cela permettra de faciliter la création du dataset compressé car les couverture des motifs fréquents sont déjà connues et pour chaque motifs fréquent, la méthode accède directement aux transactions dont les identifiant apparaissent dans la couverture du motif pour les compresser et n'a pas besoin de scanner le dataset de base pour vérifier si une transaction contient un motif fréquent. Figure 7.2 et Figure 7.3 représentent les FP-tree crées pour chacun des deux clusters obtenus à l'étape 2 avec la valeur de Smin =2.

Étape 5 : cette étape consiste à parcourir chaque FP-tree créé pour chaque cluster afin d'énumérer les motifs fréquents requis pour la compression. Pour se faire, une fonction de calcul de taux de compression [] d'un motif est utilisée. Le parcours d'un FP-Tree se fait comme suit : pour chaque itemset u allant du noeud racine "root" au noeud courant nod, comparer le taux de compression $Rate$ que offrir u au taux de compression que peuvent offrir ses super-itemsets $ChRate$. Si la valeur de $ChRate$ est plus petite que celle de $Rate$, alors u est considéré comme étant un motif fréquent requis pour la compression sinon si les super-motifs de u offrent un meilleur taux de compression, dans ce cas u sera ignoré et la méthode continue à explorer les super-motifs de u. Tel que le même traitement est

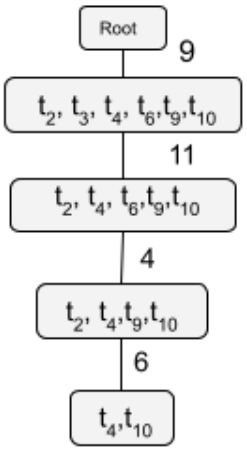


FIGURE 7.2 – FP-Tree correspondant au cluster $cl_{id} = 2$.

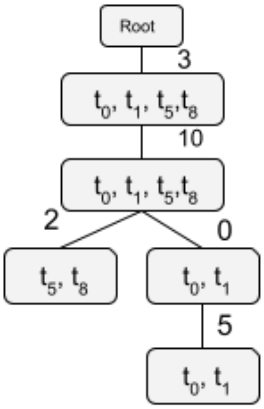


FIGURE 7.3 – FP-Tree correspondant au cluster $cl_{id} = 1$.

répété jusqu'à parcourir tout l'arbre FP. Dans le cas ou un noeud feuille est atteint, le motif correspondant au chemin allant de ce noeud au noeud root est considéré comme requis pour la compression.

L'intérêt d'utiliser la fonction taux de compression réside dans le fait qu'elle permet une comparaison précise du gain en compression qu'on peut obtenir en compressant f transactions avec u par rapport au gain qu'on peut obtenir en compressant les même transactions avec les super-motifs de u.

Les motifs fréquents obtenus en parcourant les deux FP-Tree obtenus dans l'étape 4 pour les deux clusters $cl_{id} = 1$ et $cl_{id} = 2$ sont comme suit :

— Cluster $cl_{id} = 1$:
  — {0, 3, 5, 10} couvrant les transaction $\{t_0, t_1\}$;
  — {2,3, 10} couvrent les transaction $\{t_5, t_8\}$;
— Cluster $cl_{id} = 2$:
  — {9, 9, 11} couvrent les transaction $\{t_2, t_4, t_9, t_{10}\}$;

Etape 6 : cette étape consiste à créer sur chaque cluster la table compressée lui correspondant en utilisant les motifs fréquents obtenus. La couverture de chaque motif fréquent est déjà connue, alors il ne reste qu'à créer pour chaque motif fréquent u un fragment (u, St) tel que St est la sous-table correspondante à u et elle peut être obtenu en supprimant u de chaque transaction t_i de $cover(u)$ ensuite l'ajouter à St. Une fois que les tables compressées de tout les motifs fréquents sont obtenues sur chaque cluster, regrouper les tables compressées obtenus pour obtenir la table compressée globale du dataset.

Table 7.4 correspond au dataset compressé globale obtenue en compressant le dataset 7.1 avec les motifs fréquents obtenus à l'étape 5. La taille du dataset compressé est égale à 39 items. Le taux de compression obtenu en partitionnant les données du dataset sur deux clusters est de 29.1% alors qu'il est d'environ 18.18% sans partitionnement des données. On constate que le partitionnement des données du dataset en deux cluster à amélioré son taux de compression.

0 3 5 10 8
9

(a) premier fragment e'_1.

2 3 10 6 8
7 9

(b) deuxième fragment e'_2.

1 7
4 9 11 1 6
2 7
2 6

(c) troisième fragment e'_3.

1 3 6 9 10
2 3 6 9 11
2 4 7 8 10

(d) quatrième fragment e'_4.

TABLE 7.4 – Dataet TD'_c compressé $S_{min} = 2$ en utilisant le partitionnement en deux clusters.

7.3 Résultats expérimentaux

Dans cette section, nous présentons les différentes expérimentations menées ainsi que les résultats obtenus afin de montrer l'apport de la parallélisation de la méthode `FPTCM`en terme de temps CPU et d'espace mémoire. La section est organisé comme suit, dans un premier temps, nous présentons le protocole expérimental. Ensuite, rapporter les résultats obtenus avant et après la parallélisation de la méthode `FPTCM`et les comparer en terme de temps CPU puis en terme de la taille d'espace mémoire réduit.

Protocole expérimental et mesures de performances

La méthode **parallèle** `FPTCM` est implémentée et développée en langage Python utilisant le framework Pyspark pour la parallélisation et la distribution de calcul. L'ensemble des expérimentations ont été réalisés sur une machine Intel (R) Core(TM), $i5-7200$ CPU, 2.5 GHz avec une RAM de 4 GB, avec un distribution Ubuntu sur 64 bits.

Pour mener les expérimentations, nous avons utilisés huit (8) datasets de UCI (Uni-

versity of California Irvine)[2]. Table 7.5 donne une description de l'ensemble des datasets utilisés. La colonne *dataset* donne le nom de chaque dataset, la colonne *transactions* représente le nombre transaction dans chaque dataset, max_{lgt} (resp. min_{lgt} donne le nombre maximal (resp. minimal)d'items dans une transaction, avg_{lgt} est le nombre moyen d'items dans une transaction, la colonne *items* rapporte la taille de l'ensemble d'items dans chaque dataset et enfin, la colonne *taille* donne la taille, en mégabit, de chaque dataset.

TABLE 7.5 – dataset transactionnel $\mathscr{TD}$.

Dataset	*transactions*	*max-lg*$_t$	*min-lg*$_t$	*avg-lg*$_t$	*items*	*size*
Chess	3,196	37	37	37	76	0.34MB
Mushroom	8,124	23	23	23	120	0.57MB
Connect	67,557	43	43	43	130	9.3MB
Accidents	340,138				468	35.5MB
Pumsb	49,046	74	74	74	7117	16.7MB
Pumsb-star	49,046				7116	11.3MB
T40I10D100K	100,000	77	4	40	1000	15.5MB
T10I4D100K	100,000	29	1	10	1000	4MB

Le temps CPU et la taille des datasets compressés sont les principaux paramètres utilisés dans notre comparaison.Nous avons rapporté pour chaque dataset son temps CPU de compression et sa taille, en méga-octets et en nombre d'items, après la compression.

Pour une meilleure visualisation et interprétation des résultats obtenus, nous avons les avons rapportés sous forme d'histogrammes.

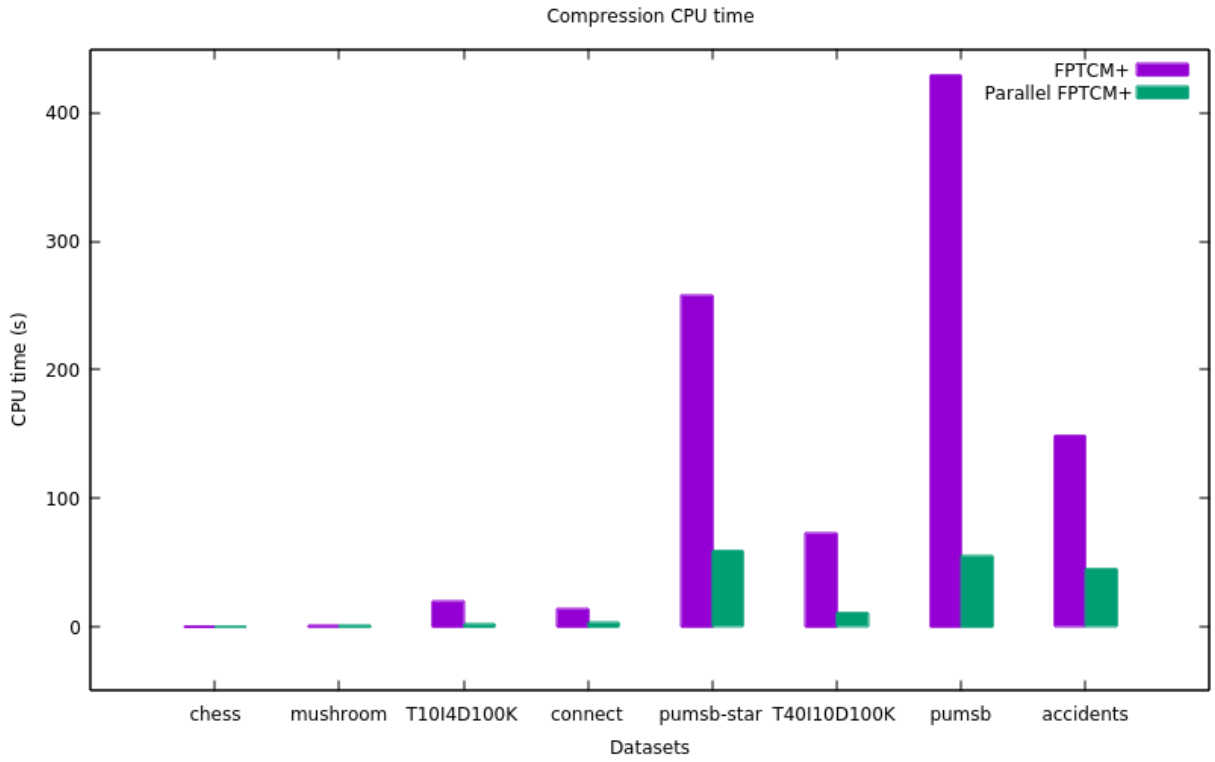
7.3.1 *Comparaison des deux méthodes* `FPTCM` *vs Parallel* `FPTCM` *?? en terme de temps CPU*

La méthode Parallèle `FPTCM` est reposée sur le principe du clustering, tel qu'elle partitionne les données d'un dataset en un ensemble de clusters regroupant des transactions avec un grand degré de similarité. Dans le cadre de nos expérimentations, nous avons fixé le nombre de clusters à quatre (4). Les données de chaque dataset sont partitionnées

2. les datasets peuvent etre obtenus à partir de ce lien : http://fimi.uantwerpen.be/data/

en clusters différents (aucun chevauchement entres clusters). Le temps CPU de compression d'un dataset est donc égal au temps de représentation du dataset dans un format binarisé plus le temps CPU moyen passé dans un cluster pour énumérer les FI (itemset fréquent) requis pour la compression plus le temps CPU moyen de compression des transactions d'un cluster en utilisant les FI énumérés. L'histogramme de Figure 7.4 rapporte le temps CPU de compression requis par chacune des deux méthodes `FPTCM` **??** et **Parallèle** `FPTCM` pour compresser chacun des datasets sélectionnés. Nous pouvons bien voir que la méthode **Parallel** `FPTCM` (barres en vert) compresse les datasets en un temps inférieur à celui requis par la méthode `FPTCM`pour compresser les mêmes datasets. Cette différence en temps CPU de compression confirme l'importance de partitionner les transactions d'un datasets en un ensemble de clusters mais aussi l'importance d'utilisation du framework PySpark qui distribue le calcul (calcul de fréquences des items par exemple) sur différents noeuds de calcul en utilisant le concept de Map and Reduce.

FIGURE 7.4 – `FPTCM` [2] vs **Paralle** `FPTCM` : comparaison en terme de temps de compression.



7.3.2 Comparaison des deux méthodes FPTCM vs Parallel FPTCM en terme de taille des datasets compressés

Après avoir comparé les deux méthodes en terme de temps CPU, nous allons dans cette sous section les comparer en terme de qualité de compression. Figure 7.5 consiste en un histogramme dans lequel est rapportée la taille, en MegaBits, des datasets avant leur compression et après leur compression en utilisant les deux méthodes FPTCMet **Parallel** FPTCM . Nous pouvons bien voir que les datasets sont mieux compressés en utilisant la méthode **Parallel** FPTCM comparant à la méthode FPTCM. Cela implique que le partitionnement des données et l'utilisation de PySpark n'améliore pas seulement le temps de calcul mais aussi peut avoir un impact sur la qualité de compression. Cette différence en terme de taille de datasets compressés peut s'expliquer par le fait que le partitionnement des données en clusters regroupant des transactions avec un grand degrés de similarité peut améliorer la qualité des motifs fréquents énumérés. Plus les transactions sont similaires, plus les motifs énumérés peuvent etre plus fréquents et plus longs.

FIGURE 7.5 – FPTCM [2] vs **Paralle** FPTCM : comparaison en terme de taille, en MégaBits, des datasets compressés.

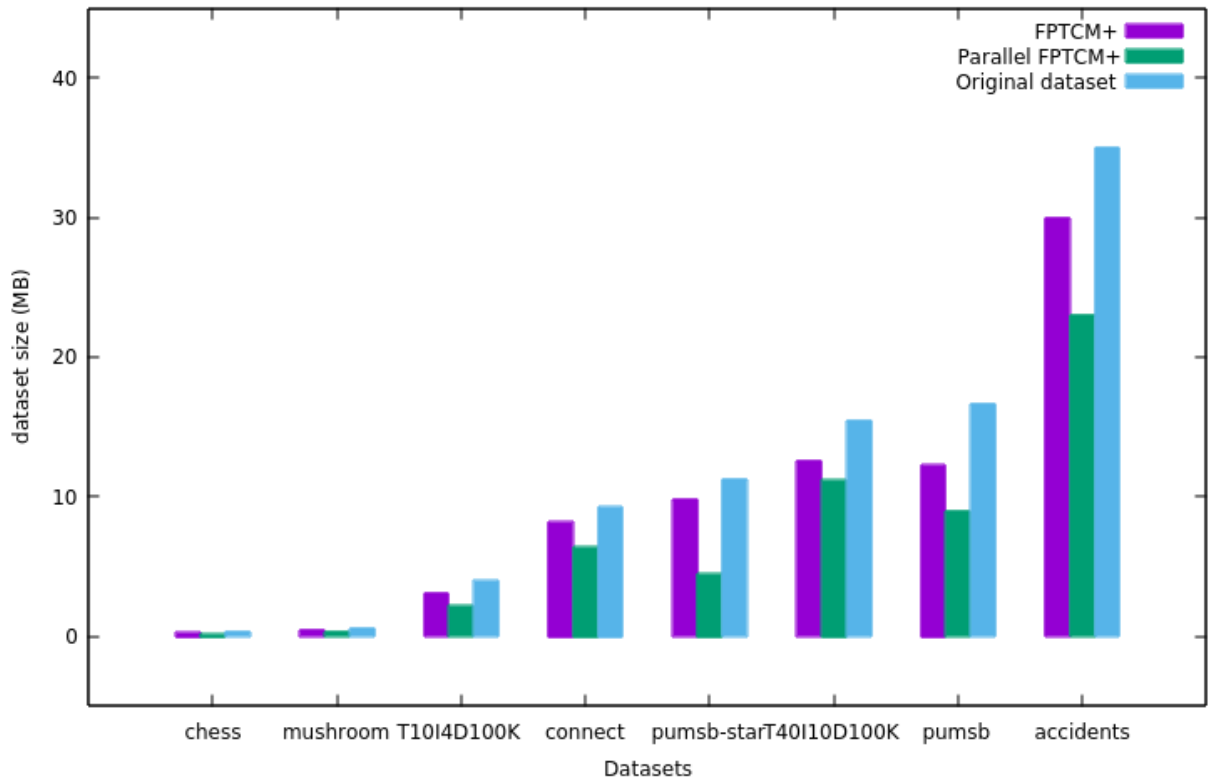


Figure 7.6 est un héstogramme qui représente en nombre d'items la taille des datasets avant et après leurs compressions avec les deux méthodes. Puisque la méthode **Parallel**

FPTCM compresse mieux les datasets nous remarquons que les datasets compressés avec **Parallel** FPTCM contiennent moins d'items comparant au même datasets compressés avec la méthode FPTCM.

FIGURE 7.6 – FPTCM vs **Paralle** FPTCM : comparaison en terme de nombre d'items dans les datasets compressés.

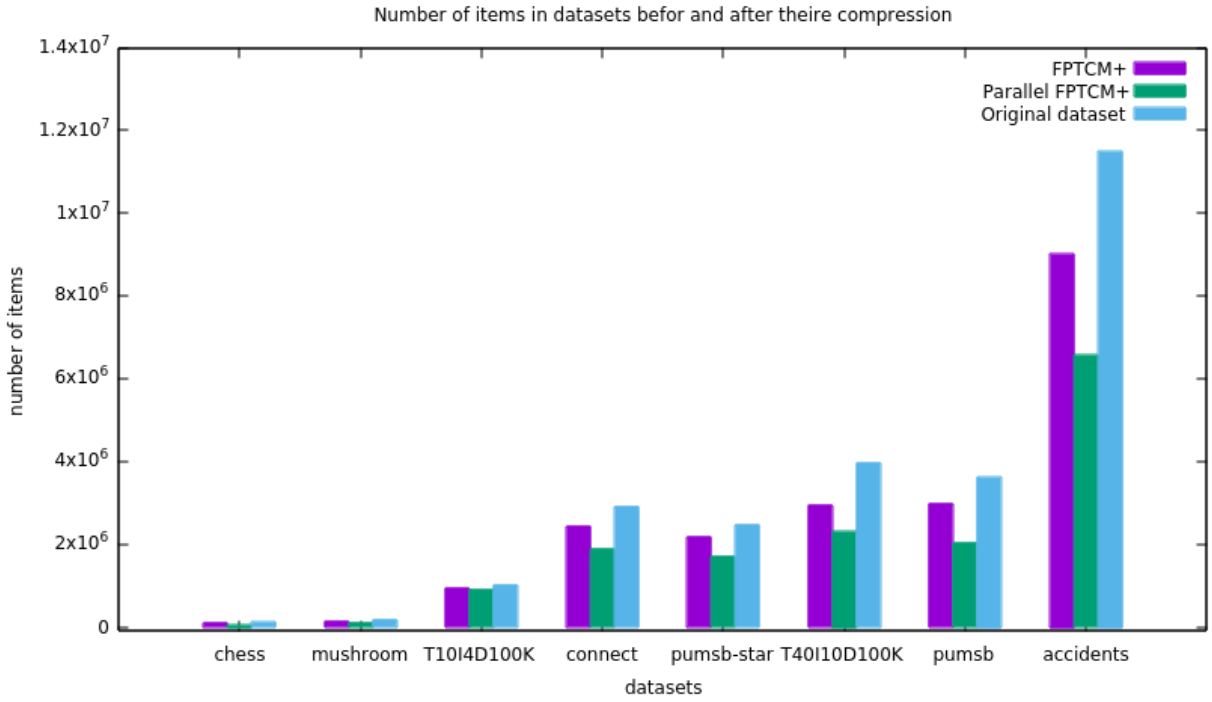
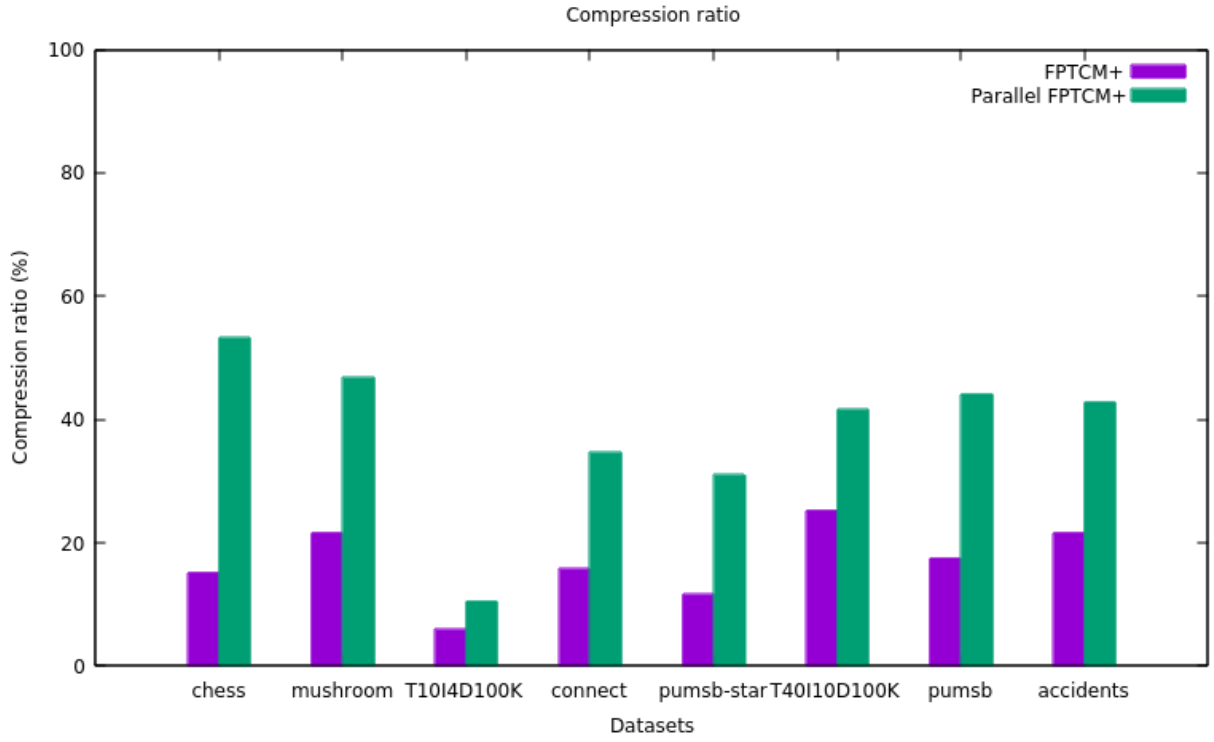


Figure 7.7 montre le taux de compression des datasets avec chacune des deux méthodes **Paralle** FPTCM et FPTCM. Les taux de compression obtenus en compressant les datasets avec la méthode **Paralle** FPTCM sont plus important par rapport à ceux obtenus en compressant les memes datasets avec la méthode FPTCM. La méthode **Parralel** FPTCMaméliore le taux de compression du dataset *chess* de plus de 40%, la compression des datasets *mushroom*, *pumsb*, *accidents* et *connect* de plus de 20%.

FIGURE 7.7 – FPTCMvs **Paralle** FPTCM : comparaison en terme de taux de compression.



7.4 Conclusion

Dans cette partie nous avons combiné une technique de clustering, la puissance de Spark ainsi que la recherche de motifs fréquents pour proposer une nouvelle méthode compression. Nous avons dans un premier temps proposé de représenter les transactions d'un dataset dans un format binarisé. Ensuite, pour paralléliser la compression d'un dataset, nous avons proposé de partitionner ces transactions en un ensemble de clusters regroupant les transactions les plus similaires en appliquant la méthode de clustering K-means. Enfin, en utilisant les fonctions de Spark nous avons proposé de distribuer, sur chaque cluster, la recherche de motifs fréquents requis pour la compression ainsi que création des clusters compressés.

Chapitre 8

Conclusion et Perspectives

8.1 Conclusion

Le phénomène du Big Data est omniprésent dans presque tout les domaines et présente plusieurs défis dont la variété et le grand volume des données à stocker et à analyser. Dans cette thèse on s'est intéressé au problème du volume de contraintes table dont l'utilisation est répandu dans le domaine industriel. Plusieurs travaux ont été proposé dans la littérature pour réduire la taille de stockage des contraintes table en proposant des méthodes de compression permettant de représenter les contraintes table en utilisant de nouvelle structures de données permettant de résoudre le réseau de contraintes tables sans avoir besoin de décompresser les contraintes table. Nous avons proposé, dans le cadre de cette thèse, de nouvelles méthodes de compression pour les contraintes table à deux niveaux : compresser les contraintes table en exploitant des techniques de fouille de motifs fréquents, ensuite, réduire la taille de l'espace de recherche de solutions au CSP en éliminant les données ne pouvant pas contenir de solution au fur et à mesure qu'on avance dans le processus de recherche. On s'est ensuite intéressé à la technologie Spark de Big Data ainsi qu'au techniques de clustering pour distribuer et paralléliser la compression.

8.1.1 Exploitation d'un arbre de préfixes (FP-Tree) pour la compression des contraintes tables

Nous avons proposé une nouvelle méthode de compression pour les contraintes tables basé sur la suppression des redondances. Elle est une amélioration de la méthode proposée par Gharbi et *al.* [1]. Nous avons proposé trois principales amélioration qui peuvent être résumé comme suit :

— remplacer la fréquence des items par leur couverture dans les noeuds de l'arbre afin de déduire directement la couverture d'un itemset fréquent sans avoir a parcourir une autre fois la contrainte table.

— fixer dynamiquement la valeur du seuil minimal de fréquence pour chaque contrainte table afin d'extraire uniquement les itemsets les plus fréquents pouvant compres-

ser un maximum de transactions. Cela, dans le but d'éviter une explosion du nombre de fragments dans la contrainte table compressée.

— utiliser une nouvelle fonction de calcul de gain en compression appliqué durant le parcours de l'arbre de préfixe pour en extraire les itemsets fréquents pertinents pour la compression. Cette fonction a pour but de calculer le nombre d'items avec lequel réduire la taille d'un même nombre de transactions en utilisant des itemsets différents. Cela afin de d'obtenir une approximation exacte sur le gain en compression.

Nous avons ensuite, appliqué la méthode STR-slice [1] pour résoudre le contraintes compressées. Nous avons validé notre proposition par une série d'expérimentations qui ont prouvé l'efficacité de la méthode en terme de compression et surtout en terme de temps de résolution des contraintes tables compresées.

publications associées : [2, 6, 3]

8.1.2 Compression des contraintes table à base d'itemsets fréquents maximaux

Nous avons proposé une autre méthode de compression, appelée MFIC, basée sur l'extraction d'itemsets fréquents maximaux (MFI), cette méthode exploite les MFI afin d'assurer la compression des contraintes table avec les itemsets les plus longs et plus fréquents. Pour extraire, à partir d'une contrainte table les MFI, nous calculons dynamiquement le seuil minimal de fréquence. Ensuite, en faisant appel à l'algorithme LCM [36], nous énumérons les itemsets fréquents maximaux dont la fréquence est supérieure ou égale à seuil fixé. Une transaction d'une contrainte table peut être compressée au plus par un seul itemset. Or, les MFIs que LCM énumèrent peuvent se chevaucher entre eux et donc ne peuvent pas tous être exploités pour la compression. Pour cela, nous avons utilisé la notion d'aire d'un itemset pour choisir les MFIs les plus pertinents pour la compression. Nous avons ensuite, appliqué la méthode STR-slice [1] pour résoudre les CSPs compressés. Nous avons validé notre proposition par une série d'expérimentations qui ont prouvé l'efficacité de la méthode en terme de compression et en terme de temps de résolution des contraintes tables compressées. Nous sommes aussi arrivé à un résultat important qui est : un meilleur taux de compression n'améliore pas forcement le temps de résolution. Mais que la qualité des itemsets utilisés pour la compression affecte aussi la résolution.

Publication concernée : [4]

8.1.3 Parralélisation et distribution de la méthode de compression FPTCM en exploitant la technologie Spark et le clustering K-means

Nous avons exploité la méthode de clustering k-means afin de partitionner les donner à compresser en un K groupes de transactions similaires. Ensuite, en utilisant la technologie de Spark basée sur les RDD, nous proposé une version distribuée de la méthode FPCM. Pour valider notre proposition, nous effectué une serie d'experimentations sur un ensemble de datasets connus dans le domaine de data mining.

Publication concernée : [5]

8.2 Perspectives

Comme perspectives pour ce travail, nous proposons les voies suivantes :

- Combiner la fouille de motifs fréquents et le clustering conceptuel afin de proposer de nouvelles méthodes de compression de contraintes tables.
- Proposer une méthode de compression du réseau de contraintes au lieu de compresser les contraintes table une par une.
- Améliorer la résolution du réseau de contraintes tables compressées en proposant une représentation en bit support des contraintes table compressées.
- proposer une nouvelle approche plus efficace pour fixer la valeur de seuil minimale de fréquence utilisée pour le choix des itemsets fréquents pertinements pour la compression.
- Etudier de plus prêt la relation entre les caractéristiques des itemsets fréquents (type, longueur, fréquence) ainsi que l'impact de ces trois caractéristiques sur la qualité de compression et sur le temps résolution du réseau de contrainte compressé.
- Prpoposer une parallélisation de la méthode de compression basée sur les itemsets fréquents maximaux.

References

[1] Nebras GHARBI et al. “Sliced table constraints : Combining compression and tabular reduction”. In : *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer. 2014, p. 120-135.

[2] Soufia BENNAI, Kamal AMROUN et Samir LOUDNI. “Exploiting Data Mining Techniques for Compressing Table Constraints”. In : *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE. 2019, p. 42-49.

[3] Soufia BENNAI, Kamal AMROUN et Samir LOUDNI. “New Methods for Compressing Table Constraints”. In : *AKDM-10 : Advances in Knowledge Discovery and Management Vol. 10*. 2023.

[4] Soufia BENNAI et al. “An efficient heuristic approach combining maximal itemsets and area measure for compressing voluminous table constraints”. In : *The journal of Supercomputing* 78.11 (2022).

[5] Soufia BENNAI et Kamal AMROUN. “Parallel FPTCM+ : Parallel FP-Tree based Compression Method for large datasets”. In : *Future of Information and Communication Conference (FICC) 2023*. Mars 2023.

[6] Soufia BENNAI, Kamal AMROUN et Samir LOUDNI. “Exploitation des techniques de fouille de données pour la compression de contraintes tables”. In : *20èmes Journées Francophones Extraction et Gestion des Connaissances, EGC 2020*. 2020.

[7] Solomon W GOLOMB et Leonard D BAUMERT. “Backtrack programming”. In : *Journal of the ACM (JACM)* 12.4 (1965), p. 516-524.

[8] John Gary GASCHNIG. *Performance measurement and analysis of certain search algorithms*. Carnegie Mellon University, 1979.

[9] Rina DECHTER. “Enhancement schemes for constraint processing : Backjumping, learning, and cutset decomposition”. In : *Artificial Intelligence* 41.3 (1990), p. 273-312.

[10] Patrick PROSSER. “Hybrid algorithms for the constraint satisfaction problem”. In : *Computational intelligence* 9.3 (1993), p. 268-299.

[11] Rina DECHTER et Itay MEIRI. “Experimental evaluation of preprocessing algorithms for constraint satisfaction problems”. In : *Artificial Intelligence* 68.2 (1994), p. 211-241.

[12] Robert M HARALICK et Gordon L ELLIOTT. “Increasing tree search efficiency for constraint satisfaction problems”. In : *Artificial intelligence* 14.3 (1980), p. 263-313.

[13] Christian BESSIERE et Jean-Charles RÉGIN. “MAC and combined heuristics : Two reasons to forsake FC (and CBJ ?) on hard problems”. In : *International Conference on Principles and Practice of Constraint Programming*. Springer. 1996, p. 61-75.

[14] Daniel FROST, Rina DECHTER et al. “Look-ahead value ordering for constraint satisfaction problems”. In : *IJCAI (1)*. Citeseer. 1995, p. 572-578.

[15] Philippe REFALO. “Impact-based search strategies for constraint programming”. In : *International Conference on Principles and Practice of Constraint Programming*. Springer. 2004, p. 557-571.

[16] Alan K MACKWORTH. “Consistency in networks of relations”. In : *Artificial intelligence* 8.1 (1977), p. 99-118.

[17] Roger MOHR et Thomas C HENDERSON. “Arc and path consistency revisited”. In : *Artificial intelligence* 28.2 (1986), p. 225-233.

[18] Pascal VAN HENTENRYCK, Yves DEVILLE et Choh-Man TENG. “A generic arc-consistency algorithm and its specializations”. In : *Artificial intelligence* 57.2-3 (1992), p. 291-321.

[19] Christian BESSIERE. “Arc-consistency and arc-consistency again”. In : *Artificial intelligence* 65.1 (1994), p. 179-190.

[20] Christian BESSIERE, Assef CHMEISS et Lakhdar SAIS. “Neighborhood-based variable ordering heuristics for the constraint satisfaction problem”. In : *International Conference on Principles and Practice of Constraint Programming*. Springer. 2001, p. 565-569.

[21] Christian BESSIERE et Jean-Charles RÉGIN. “Refining the Basic Constraint Propagation Algorithm.” In : *IJCAI*. T. 1. 2001, p. 309-315.

[22] Julian R ULLMANN. “Partition search for non-binary constraint satisfaction”. In : *Information Sciences* 177.18 (2007), p. 3639-3678.

[23] Christophe LECOUTRE. “Optimization of simple tabular reduction for table constraints”. In : *International conference on principles and practice of constraint programming*. Springer. 2008, p. 128-143.

[24] Rakesh AGRAWAL, Tomasz IMIELIŃSKI et Arun SWAMI. “Mining association rules between sets of items in large databases”. In : *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*. 1993, p. 207-216.

[25] Tom M MITCHELL. “Generalization as search”. In : *Artificial intelligence* 18.2 (1982), p. 203-226.

[26] Heikki MANNILA et Hannu TOIVONEN. “Multiple uses of frequent sets and condensed representations : Extended abstract”. In : *Proc. of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD’96)*. 1996, p. 189-194.

[27] Gerd STUMME et al. “Fast computation of concept lattices using data mining techniques”. In : (2000).

[28] Nicolas PASQUIER et al. “Discovering frequent closed itemsets for association rules”. In : *International Conference on Database Theory*. Springer. 1999, p. 398-416.

[29] Mohammed J ZAKI et Ching-Jui HSIAO. “CHARM : An efficient algorithm for closed itemset mining”. In : *Proceedings of the 2002 SIAM international conference on data mining*. SIAM. 2002, p. 457-473.

[30] Roberto J BAYARDO JR. “Efficiently mining long patterns from databases”. In : *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. 1998, p. 85-93.

[31] Douglas BURDICK, Manuel CALIMLIM et Johannes GEHRKE. “Mafia : A maximal frequent itemset algorithm for transactional databases”. In : *Proceedings 17th international conference on data engineering*. IEEE. 2001, p. 443-452.

[32] Mohammed Javeed ZAKI et al. “New algorithms for fast discovery of association rules.” In : *KDD*. T. 97. 1997, p. 283-286.

[33] Jean-François BOULICAUT, Artur BYKOWSKI et Christophe RIGOTTI. “Free-sets : a condensed representation of boolean data for the approximation of frequency queries”. In : *Data Mining and Knowledge Discovery* 7.1 (2003), p. 5-22.

[34] Takeaki UNO, Masashi KIYOMI, Hiroki ARIMURA et al. “LCM ver. 2 : Efficient mining algorithms for frequent/closed/maximal itemsets”. In : *Fimi*. T. 126. 2004.

[35] Jean-François BOULICAUT, Artur BYKOWSKI et Christophe RIGOTTI. “Free-sets : a condensed representation of boolean data for the approximation of frequency queries”. In : *Data Mining and Knowledge Discovery* 7.1 (2003), p. 5-22.

[36] Takeaki UNO, Masashi KIYOMI, Hiroki ARIMURA et al. “LCM ver. 2 : Efficient mining algorithms for frequent/closed/maximal itemsets”. In : *Fimi*. T. 126. 2004.

[37] Takeaki UNO. “A practical fast algorithm for enumerating cliques in huge bipartite graphs and its implementation”. In : *89th Special Interest Group of Algorithms* (2003).

[38] Takeaki UNO. “A practical fast algorithm for enumerating cliques in huge bipartite graphs and its implementation”. In : *89th Special Interest Group of Algorithms* (2003).

[39] Mohammed J ZAKI et Ching-Jui HSIAO. “CHARM : An efficient algorithm for closed itemset mining”. In : *Proceedings of the 2002 SIAM international conference on data mining*. SIAM. 2002, p. 457-473.

[40] George KATSIRELOS et Toby WALSH. “A compression algorithm for large arity extensional constraints”. In : *International conference on principles and practice of constraint programming*. Springer. 2007, p. 379-393.

[41] Kenil CK CHENG et Roland HC YAP. “An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints”. In : *Constraints* 15.2 (2010), p. 265.

[42] Guillaume PEREZ et Jean-Charles RÉGIN. “Improving GAC-4 for table and MDD constraints”. In : *International Conference on Principles and Practice of Constraint Programming*. Springer. 2014, p. 606-621.

[43] Christophe LECOUTRE, Chavalit LIKITVIVATANAVONG et Roland HC YAP. “STR3 : A path-optimal filtering algorithm for table constraints”. In : *Artificial Intelligence* 220 (2015), p. 1-27.

[44] Ruiwei WANG et al. “Optimizing Simple Tabular Reduction with a Bitwise Representation.” In : *IJCAI*. 2016, p. 787-795.

[45] Hélene VERHAEGHE, Christophe LECOUTRE et Pierre SCHAUS. “Extending compact-table to negative and short tables”. In : *Thirty-First AAAI Conference on Artificial Intelligence*. 2017.

[46] Gilles AUDEMARD, Christophe LECOUTRE et Mehdi MAAMAR. “Segmented tables : An efficient modeling tool for constraint reasoning”. In : *ECAI 2020*. IOS Press, 2020, p. 315-322.

[47] Said JABBOUR et al. “Mining to Compress Table Constraints”. In : *2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE. 2015, p. 405-412.

[48] Jiawei HAN, Jian PEI et Yiwen YIN. “Mining frequent patterns without candidate generation”. In : *ACM sigmod record* 29.2 (2000), p. 1-12.

[49] Peter NIGHTINGALE et al. “Short and long supports for constraint propagation”. In : *Journal of Artificial Intelligence Research* 46 (2013), p. 1-45.

[50] Roger MOHR et Gérald MASINI. “Good old discrete relaxation”. In : *Proceedings of the 8th European conference on artificial intelligence*. 1988, p. 651-656.

[51] Matei ZAHARIA et al. “Resilient distributed datasets : A {Fault-Tolerant} abstraction for {In-Memory} cluster computing”. In : *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 2012, p. 15-28.

[52] Bowen WANG et al. “Parallelizing k-means-based clustering on spark”. In : *2016 International Conference on Advanced Cloud and Big Data (CBD)*. IEEE. 2016, p. 31-36.

Bibliography

MJ ZAKI et CJ HSIAO. “An efficient algorithm for closed itemset mining”. In : *0-Porc. SIAM Int. Conf. Data Mining, Arlington, VA*. 2000.

Wei XIA et Roland HC YAP. “Optimizing STR algorithms with tuple compression”. In : *International Conference on Principles and Practice of Constraint Programming*. Springer. 2013, p. 724-732.

Debra A LELEWER et Daniel S HIRSCHBERG. “Data compression”. In : *ACM Computing Surveys (CSUR)* 19.3 (1987), p. 261-296.

B. ANGLOIS. *This is a paper*. 2005.

F. OVALIE. *Robert*. 2005.

C. CUTHOR. *Charlie*. 2003.

E. ABSOL. *Danger*. 2005.

S. ANDRE. *Orange*. 2003.

W. KOR. *Poule*. 2001.

Soufia BENNAI et al. “An efficient heuristic approach combining maximal itemsets and area measure for compressing voluminous table constraints”. In : *The Journal of Supercomputing* 78 (2023), p. 139-159.

Soufia BENNAI, Kamal AMROUN et Samir LOUDNI. “Exploiting Data Mining Techniques for Compressing Table Constraints”. In : *31st IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2019, Portland, OR, USA, November 4-6, 2019*. IEEE, 2019, p. 42-49. DOI : 10.1109/ICTAI.2019.00015. URL : https://doi.org/10.1109/ICTAI.2019.00015.

Soufia BENNAI, Kamal AMROUN et Samir LOUDNI. “Exploitation des techniques de fouille de données pour la compression de contraintes table”. In : *Revue des Nouvelles Technologies de l’Information* Extraction et Gestion des Connaissances , RNTI-E-36 (2020), p. 449-456.

Soufia BENNAI, Kamal AMROUN et Samir Loudni AND. “ShrFP-Compression method for Big Data,” in : *7ème International Sumposium ISKO-Maghreb*. Bejaia, Algerie, nov. 2018.

Abstract

With the arrival of Big data the volume of data to be stored and analyzed exponentially increase. This why the volume is an important challenge in all domains. As part of this thesis we focused on the field of CSPs (Constraint Satisfaction Problems). We have proposed new compression techniques for table constraints based on datamining techniques in order to reduce the size of the memory space required for their representation in memory. But also in order to reduce the resolution times of CSPs. This by applying solving techniques on compressed CSPs. Our first contribution is an improvement of a literature method based on an FP-Tree. Then, we proposed a new compression method based on maximal frequent itemsets. Finally, we proposed a distributed version of the method based on a FP-Tree by exploiting the K-means clustering method and Spark technology. The obtained results for the three contributions have been published in a journal and at internationally renowned conferences.

Key words: big data, compression, data mining, frequent itemsets.

Résumé

Le volume des données à stocker est à analyser connait une augmentation exponentielle depuis l'aire du big data. Le volume est donc un des plus grand défis dans tous les domaines. Dans le cadre de cette thèse on s'est intéressé au domaine des CSPs (Problèmes de Satisfaction des Contraintes). Nous avons proposé de nouvelles techniques de compression pour les contraintes table, basées sur des techniques de datamining, dans le but de réduire la taille de l'espace mémoire requis pour leur représentation en mémoire. Mais aussi dans le but de réduire les temps de résolution des CSPs. Cela en appliquant des techniques de résolutions sur les CSPs compressés. Notre première contribution est une amélioration d'une méthode déjà existence basée sur un FP-Tree. La deuxième contribution est une nouvelle méthode de compression basée sur les itemsets fréquents maximaux. La troisième proposition est une version distribuée de la méthode basé sur un FP-Tree en exploitant la méthode de clustering K-means et la technologie Spark. Les résultats obtenus pour les trois contributions ont fait objets de publications dans un journal et dans des conférences de renommé internationale.

Mots clé: big data, motifs fréquents, datamining, itemsets fréquents

ملخص

يشهد حجم البيانات المراد تخزينها وتحليلها زيادة هائلة منذ عصر البيانات الضخمة. وبالتالي فإن الحجم هو أحد أكبر التحديات في جميع المجالات. كجزء من هذه الأطروحة ، ركزنا على مجال مسائل شرط **الرضا**. لقد اقترحنا تقنيات ضغط جديدة لجدول القيود ، بناءً على تقنيات معالجة البيانات، من أجل تقليل حجم الذاكرة المطلوبة لتمثيلها وأيضا من أجل التقليل من زمن حل مسائل شرط **الرضا**. هذا عن طريق تطبيق تقنيات الحل مباشرة على جداول القيود المضغوطة. مساهمتنا الأولى تتمثل في تحسين طريقة موجودة في الأدب. ثم اقترحنا طريقة ضغط جديدة تعتمد على شجرة FP من أجل استخراج العناصر المتكررة. أخيرًا ، اقترحنا نسخة موزعة من الطريقة التي تعتمد على شجرة FP من خلال استغلال طريقةالتقسيم الى مجموعات وتقنية Spark. تم نشر النتائج التي تم الحصول عليها في مجلة وفي مؤتمرات ذات شهرة عالمية.

الكلمات الدالة :مسائل شرط **الرضا** , البيانات الضخمة, تقنيات ضغط, تقنيات معالجة البيانات, العناصر المتكررة