République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université A/Mira de Béjaia
Faculté des Sciences Exactes
Département d'Informatique

MÉMOIRE DE MASTER RECHERCHE

En
Informatique

Option
Réseaux et Systèmes Distribués

Thème

Mise en œuvre parallèle d'un algorithme de Consistance d'Arc avec des tables compressées

Présenté par : Mll. Maouche Karima & M. Mokrani Bilal

Soutenu le 04 Juillet 2016 devant le jury composé de :

Président	Touazi Djoudi	Maître de assist. A	U. A/Mira Béjaia.
Rapporteur	Amroun Kamal	Maître de conf. A	U. A/Mira Béjaia.
Examinateur	Farah Zoubeyr	Maître de conf. B	U. A/Mira Béjaia.
Examinatrice	Benmerbi Samah	Doct."LMD"	U. A/Mira Béjaia.

Nous tenons à remercier notre promoteur Monsieur AMROUN Kamal, et lui exprimer notre profonde gratitude pour sa disponibilité, ses orientations et ses conseils.

Nous remercions tout particulièrement les membres du jury qui ont accepté d'examiner notre travail, ainsi que tous les enseignants qui ont contribué à notre formation.

Enfin, Nous remercions aussi tous nos amis et collègues qui nous ont soutenu et tous ceux qui ont contribué de près ou de loin à la réalisation de ce travail.

*M. **Merci à tous***

⁕ *Dédicaces* ⁕

Je dédie ce travail :
Aux êtres les plus chères à mon cœur
À mes très chères parents pour leurs soutiens et conseils, que Dieu tout puissant vous préserve et vous procure santé et longue vie à la fin que je puisse à mon tour vous combler.
A mes très chères sœurs Yasmina, Zineb et soria d'avoir étaient toujours là pour moi.
A mon cher frère : Nourdine pour son soutien.
A ma future femme : Hakima.
A tous mes cousins et cousines chacun à son nom.
A tous mes oncles et mes tantes.
A tous mes amis et tous les étudiants de la promotion 2015-2016.

Bilal

⁂ *Dédicaces* ⁂

Je dédie ce travail

A ma très chère famille et à mes amis
qui m'ont soutenu
tout au long de ce projet

Karima

Table des matières

Table des figures

Liste des abréviations

AC Arc Consistency.
BE Bucket Elimination.
BJ Backjumping.
BICOMP Biconnected Component.
BM Backmarking.
BT Backtracking.
BTD Backtracking on Tree-Decomposition.
CBJ Backmarking and Backjumping.
SCDnew Spread CutNEW.
CCM CycleCluster Methode.
CHD Component Hypertree.
CSP Constraint Satisfaction Problems.
DAC Directional arc consistency.
DBE Dual Bucket Elimination.
ECHD Extend Component Hypertree.
FC Forward Checking.
FHD Fractional Hypertree Décomposition.
GHD Generalized Hypertree Decomposition.
HD Hypertree Décomposition.
Hinge Hinge Decomposition.
MAC Maintaining Arc-Consistency.
MIMD Multiple Instruction, Multiple Data.
MISD Multiple Instruction, Single Data.

MPI Message Passing Interface.
MRV Minimum Remaining Value.
NUMA Non Uniform Memory Access.
PVM Parallel Virtual Machine.
RMA Remote Memory Access.
TD Tree Decomposition.
SCD Spread Cuts.
SIMD Single Instruction, Multiple Data.
SISD Single Instruction, Single Data.
TCLUSTER Tree Clustering.
UMA Uniform Memory Access.

Introduction générale

Les contraintes tables, qui sont des contraintes en extension énumérant les tuples de valeurs autorisées ou interdites pour un ensemble de variables et Le filtrage des domaines de résolution des problèmes de satisfaction des contraintes est une méthode qui consiste à éliminer toutes les valeurs des variables non consistante dans leurs domaines. Ses deux méthodes sont largement étudiées en programmation par contraintes (CP), En effet, elles sont présentes dans de nombreux champs d'applications du monde réel : problème d'ordonnancement de tâches, problèmes d'emplois du temps, dans des domaines tels que : conception, la configuration, les bases de données etc.

Dans certains cas, représenter celles-ci n'est malheureusement pas possible car l'espace mémoire requis est trop important. Rappelons que la complexité spatiale pour représenter tous les tuples croît de manière exponentielle avec l'arité des contraintes. En vue de réduire la complexité spatiale, différentes approches ont été proposées dans la littérature. Certaines introduisent des structures de données compactes telles que les tries (arbres de préfixes), les MDDs, les tables compressées ou encore les automates déterministes finis.

A ce jour, les algorithmes de filtrage qui ont réussie à prouver leurs efficacités pour les contraintes tables sont ceux basés sur l'utilisation des MDDs, et ceux basés sur la technique la consistance d'arc AC En particulier, les variantes AC4 et AC6 de l'algorithme AC original ont été montrées particulièrement compétitives sur de nombreuses classes de problèmes.

La motivation principale de ce document est de combiné l'approche STR à une

technique de compression simple différente de celles proposées dans [2]. Le principe serait d'identifier tous les motifs récurrents de la contrainte table et d'effectué une sélection de ceux qui seront utiles pour la compression. Le résultat de cette compression serait une table fragmentée, il faudra alors apporter des modifications à l'algorithme de filtrage STR afin qu'il puisse prendre en charge ce type de représentation et qu'il puisse être efficace.
Ce document est devisé en cinq chapitres. Le premier chapitre est une introduction générale au problème de satisfaction de contrainte (CSP). On y définira quelques notions qui seront utilisées tout au long de ce document.

Le troisième chapitre traitera des différentes méthodes de compression qui ont été étudié pour aboutir à la réalisation de ce mémoire. On y expliquera brièvement chaque méthode qu'on classera par la suite selon les critères suivants : statiques, dynamiques, hybrides et smart table.
Le quatrième chapitre servira à présenter notre contribution dans la compression des contraintes tables. On détaillera les différentes étapes de notre approche de compression, qu'on appliquera sur un exemple explicatif. Ensuite nous comparerons notre méthode de compression à celles proposée dans [2]. A défaut de résultats expérimentaux nous proposons un déroulement des deux méthodes sur un exemple.
Le cinquième chapitre servira à présenter notre contribution dans le filtrage parallèle. On détaillera les différentes étapes de notre approche et on présentera les algorithmes de parallélisassions de filtrage qu'on a développés.
. On clôturera ce mémoire par une conclusion générale, qui résume tout ce qui a été dit et qui a été fait tout au long de ce travail, ainsi que nos perspectives d'amélioration de notre proposition.

Chapitre 1

généralité sur les problèmes de satisfaction de contraintes

1.1 Introduction

Dans ce chapitre nous présenterons les problèmes de satisfactions contraintes. Nous introduisons un certain nombre de concepts, notations et définition utilisés dans la suite du document. Ensuite nous donnerons un exemple de modélisation d'un CSP : la coloration des graphes, et pour finir, la présentation d'algorithmes de résolution.

1.2 Le formalisme CSP

1.2.1 Le problème CSP

Un problème de satisfaction de contraintes ou CSP (Constraint Satisfaction Problem) est un formalisme très puissant pour la résolution des problèmes combinatoires, fondé sur des contraintes posées sur des variables, elles prennent leurs valeurs dans un ensemble fini (domaine de la variable). Chaque contrainte restreint les combinaisons des valeurs possibles prises par les variables. La résolution d'un problème de satisfaction de contraintes consiste à trouver un ensemble d'instanciations qui satisfasse l'ensemble des contraintes.

1.2.2 Definition

Définition 1.2.1. Un problème de satisfaction de contraintes peut être représenté par un triplet (X, D, C), où :

- $X = X_1, X_2, \ldots, X_n$ est l'ensemble des variables du problème.
- $D = D_1, D_2, \ldots, D_n$ est l'ensemble des n domaines finis associés aux variables.
- $C = C_1, C_2, \ldots, C_k$ est l'ensemble des k contraintes, chaque contrainte C_i étant définie par un couple (v_i, r_i) ou :
  - v_i est une liste de variable $X_{i1}, ..., X_{ini}$
  - r_i est une relation définie par un sous-ensemble du produit cartésien $D_{i1} \times ... \times D_{ini}$.

Définition 1.2.2. (contrainte) Une contrainte est une relation logique entre différentes variables, chacune prenant ses valeurs dans un ensemble donné, appelé domaine. Une contrainte peut être :

- Contrainte en intension : elle est représentée lorsqu'elle exprime la relation entre plusieurs variables en utilisant les opérateurs arithmétiques ou logiques (par exemple : $x \Leftarrow y, A \wedge B \Rightarrow$ Ψ
-

Définition 1.2.3. (Arité) C'est le nombre de variables sur lequel elle est définie. On la note scope(C). Une contrainte est dite unaire (respectivement binaire) si elle est d'arité un (respectivement deux). Dans les autres cas, on la qualifie de n-aire, où n est l'arité de la contrainte.

- Un CSP dont toutes les contraintes sont unaires ou binaires, est appelé un CSP binaire.
- Un CSP est n-aire s'il comporte des contraintes d'arité supérieure à deux.

Définition 1.2.4. (Réseau de contrainte) Un réseau de contraintes est défini par un triplet (X, D, C).

- L'ensemble $X = \{1, ..., n\}$ est un ensemble de n variables.
- Chaque variable $x \in X$ est associée à un domaine de valeurs $d_i \in D$ et peut donc recevoir toute valeur $a \in d_i$ (cette valeur sera aussi notée (x, a)).

— L'ensemble C est un ensemble de contraintes. Chaque contrainte $c \in C$ est définie sur un ensemble de variables $X_c \subset X$ par une relation, également notée c, sous-ensemble du produit cartésien $\Pi_{i \in x} d_i$ qui spécifie les combinaisons de valeurs (ou tuples) autorisées pour les variables de X_c.[

Définition 1.2.5. (Instanciation) Etant donné un CSP $P = (X, D, C)$, on appelle instanciation A de $Y \subset X$ une application qui associe à chaque variable y de Y une valeur $D(y)$ (D est le domaine de la variable y). Une instanciation A de Y satisfait la contrainte $c_i = (v_i, r_i)$ de C si et seulement si $v_i \subset$ Y et $A(v_i) \in r_i$. A l'opposé, on dira qu'une instanciation A de Y viole la contrainte c_i si et seulement si $v_i \subset Y$ et $A(v_i) \notin r_i$.

Définition 1.2.6. (Instanciation consistante) Une instanciation est consistante si elle ne viole aucune contrainte. De manière plus formelle, étant donné un CSP, $P = (X, D, C)$, et soit $Y \subset X$ une instanciation des variables de Y est dite consistante si et seulement si elle satisfait toutes les contraintes portant sur ses variables.

Définition 1.2.7. (Solution d'un CSP) Une solution S d'un CSP, $P = (X, D, C, R)$ est une instanciation consistante de toutes les variables X sur D

1.2.3 Notion de consistance

1.2.3.1 Consistance de noeud

Pour chaque contrainte unaire C_i (ne concernant qu'une seule variable) on retire du domaine de v_i toutes les valeurs qui ne satisfont pas C_i. On emploie souvent la consistance de nœud pour tester l'existence.

Définition 1.2.8. Un CSP est nœud consistant si pour chaque variable $x \in X$, et pour toute valeur v de D, l'affectation partielle (x, v) satisfait toutes les contraintes unaires de C.

1.2.3.2 Consistance d'arc

Il s'agit de la plus ancienne et de la plus utilisée des consistances locales. L'idée est simple : si une valeur v du domaine de X_i n'apparait dans aucun des n-uplets d'une contrainte portant sur X_i, il est certain que v ne participera pas à une solution.

Définition 1.2.9. Un CSP est arc consistant si et seulement si aucun domaine $D_i \in D$ n'est vide et si toutes les valeurs de tous les domaines sont viables.

1.2.3.3 Consistence de chemin

Un CSP vérifie la consistance de chemin, si et seulement si, pour tout couple de variables (X, Y) et pour toute troisième variable Z, si (x, y) est une instanciation consistante de X et de Y, alors il existe z dans D_z tel que (x, z) et (z, y) soient consistants. Un arc (i, j) est consistant si le chemin (i, j, i) est chemin-consistant. Il existe donc un CSP arc-consistant mais non chemin-consistant.

1.2.3.4 K-consistance

Un CSP est dit $k - consistant$ si pour tout n-uplet de k variables $(X1, ..., Xk)$, pour toute instanciation A consistante des $(k-1)$ variables $(X1, ..., Xk-1)$, il existe une valeur $v \in D_k$ telle que l'instanciation $A \cup X_k = v$ soit consistante. La consistance de nœud correspond à la $1 - consistance$. La consistance d'arc, pour les CSP binaires, équivaut à la $2 - consistance$. La $3 - consistance$ équivaut à la consistance de chemin.

1.3 Extension des CSP

Pour que le formalisme CSP apporte des réponses satisfaisantes même pour la modélisation des problèmes, des extensions ont été proposées :

— Max-CSP : dans la pratique, les CSP peuvent être sur-contraints de sorte qu'ils n'admettent pas de solution. Dans ce cas, on peut chercher une affectation

totale maximisant le nombre de contraintes satisfaites. Cette catégorie de problème est appelée MaxCSP.

— VCSP : un poids est associé à chaque contrainte et on recherche l'affectation totale qui minimise la somme des poids des contraintes violées.

— CSOP : (Constraint Satisfaction Optimisation Problem). Il consiste à déterminer l'affectation des variables qui maximise une fonction d'optimisation, la forme basique d'un CSOP est définie de telle façon de nous permettre d'éliminer de solutions mais en préservant toujours l'ensemble de solutions optimales.

1.4 Exemple de modélisation à l'aide de CSP : Coloration de graphe

Un problème utilisé très fréquemment dans la littérature pour expliquer les concepts et les algorithmes de résolution de CSPs est le problème de coloriage de graphes. Ce problème joue aussi un rôle important en recherche opérationnelle car de nombreux problèmes d'emploi du temps ou d'ordonnancement peuvent être exprimés comme des problèmes de coloriage de graphes. Étant donné un graphe et un certain nombre de couleurs, il s'agit d'affecter une couleur à chaque nœud tel que deux nœuds adjacents n'aient pas la même couleur. Une instance de ce problème est le problème de coloriage de cartes. Dans ce cas, le problème consiste à affecter une couleur à chaque zone d'une carte en utilisant un nombre limité de couleurs et en respectant la contrainte que deux zones

Ici notre carte est divisée en 4 régions, supposant que chaque zone peut être colorée avec les couleurs ; rouge (r), vert (v), ou azur (a), sa formulation comme un CSP est la suivante : $x = \{w, x, y, z\}$ $D_w = D_x = D_y = D_z = \{r, v, a\}$ $C = \{w \neq x, w \neq y, x \neq y, x \neq z, y \neq z\}$

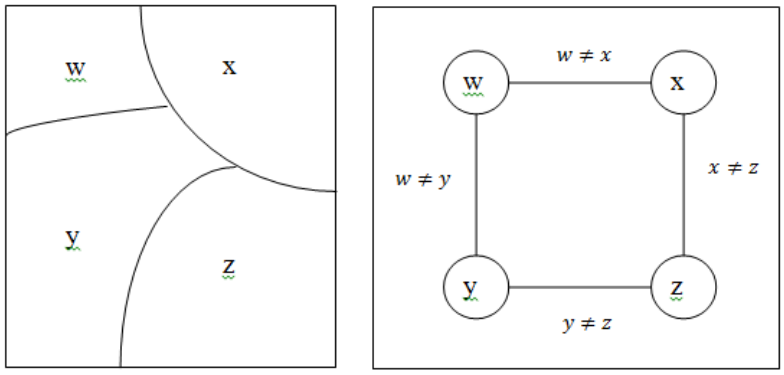


FIGURE 1.1 – exemple de problème de coloration de graphe

1.5 Algorithmes de résolution

Les algorithmes qui sont mis en œuvre lors de la résolution d'un CSP se divisent en trois groupes :

— algorithmes de consistance : ce sont des algorithmes de prétraitement pour améliorer le travail des algorithmes de recherche.
— Les algorithmes de recherche : L'algorithme de recherche de base le plus souvent utilisé est l'algorithme de retour arrière ou Backtrack. Cet algorithme met en place une stratégie de parcours de l'arbre de recherche en profondeur d'abord avec un mécanisme de retour arrière lorsque l'affectation partielle courante n'est pas consistante.
— Les algorithmes exploitant les propriétés topologiques des structures pour guider la recherche.

1.5.1 Generate and test

L'algorithme de Generate and Test provient de l'approche mathématique pour résoudre des problèmes combinatoires. Tout d'abord, l'algorithme GT devine la solution et ensuite, on teste si cette solution est correcte, à savoir que la solution satisfait les contraintes d'origine. Dans ce paradigme, chaque combinaison possible des affec-

tations de variables est systématiquement générée et testé pour voir si elle satisfait toutes les contraintes. La première combinaison qui satisfait toutes les contraintes est la solution. Le nombre de combinaisons envisagées par cette méthode est la taille du produit cartésien de l'ensemble des domaines variables.

1.5.2 Backtrack

L'algorithme Backtrack constitue la méthode de base pour la résolution des CSPs. Il correspond à une recherche aveugle de la solution en essayant successivement les ensembles des affectations de variables jusqu'à trouver une solution. Cet algorithme est fondé sur les points de retour. Chaque fois qu'un choix est effectué, un point de retour est mentionné dans l'algorithme. L'ordre de variable à tester est choisi d'une façon aléatoire ainsi que l'ordre de valeurs choisies dans le domaine de chaque variable pour construire les affectations. Le chemin en cours de test est éliminé de l'ensemble de solutions si un domaine donné devient vide. La procédure choisir revient à l'itération précédente et essaie une nouvelle valeur pour la variable en question, si aucune valeur n'est possible on revient au point précédent. Si par contre, nous sommes au premier point alors on sort de l'algorithme en échec (autrement dit avec impossibilité de trouver une solution).

1.6 Conclusion

Nous avons vu dans ce chapitre que les CSPs permettaient de modéliser de problèmes réels, ainsi que les différents outils utilisés dans la résolution leurs résolutions. Le chapitre suivant servira d'initiation au parallélisme nous y présenterons les différents notions qui y sont reliées.

Chapitre 2

Généralité sur les architecture parallèle

2.1 Introduction

L'architecture des ordinateurs, qu'il s'agisse de microprocesseurs ou de supercalculateur, est fortement influencée par l'exploitation d'une propriété fondamentale des applications : le parallélisme. Un grand nombre d'architectures présentes dans les sites informatique sont parallèles. Ce type d'architecture touche une large gamme de machine depuis les PC biprocesseurs jusqu'aux supercalculateurs. La suite de se chapitre est un itroduction aux systèmes parallèles

Définition 2.1.1. (ordinateur parallèles) sont des machines qui comportent une architecture parallèle, constituée de plusieurs processeurs identique, ou non, qui concourent au traitement d'une application. La performance d'une architecture parallèle est la combinaison des performances de ses ressources et de leur agencement.

Définition 2.1.2. (programme séquentiel) se caractérise par l'exécution de plusieurs taches l'une après l'autre avec un ordre prédéfini.

Définition 2.1.3. (programme parallèle se caractérise) par l'exécution de plusieurs taches distinctes ou non en même temps.

Définition 2.1.4. (système distribué (ou repartie)) est un système de plusieurs processeurs impliqués dans la résolution d'un ou plusieurs problèmes.

Définition 2.1.5. (programmation parallèle) est une programmation dans un langage permettant d'exprimer le parallélisme dans une application réelle. Différents niveaux d'abstraction possibles. La parallélisassions automatique serait la solution idéale, mais difficile à mettre en œuvre. La façon de programmer n'est pas indépendante de la machine utilisée.

Définition 2.1.6. (programmation distribué) chaque composant s'exécute sur un matériel interconnecté par un réseau local ou global. Les processus envoient entre eux des messages.

2.2 Classification des architectures parallèle

2.2.1 Machine SISD

Une machine SISD (Single Instruction Single Data) est ce que l'on appelle d'habitude une machine de Von Neuman. Une seule instruction est exécutée et une seule donnée (simple, non-structurée) est traitée à tout instant.

2.2.2 Machine SIMD

Une machine SIMD (Single Instruction Multiple Data) peut être de plusieurs types, parallèle ou systolique. En général l'exécution en parallèle de la même instruction se fait en même temps sur des processeurs différents (parallélisme de donnée synchrone).

2.2.3 Machine MISD

Une machine MISD (Multiple Instruction Single Data) peut exécuter plusieurs instructions en même temps sur la même donnée. Cela peut paraître paradoxal mais cela recouvre en fait un type très ordinaire de micro-parallélisme dans les microprocesseurs modernes : les processeurs vectoriels et les architectures pipelines.

2.2.4 Machine MIMD

Désigne les machines multiprocesseurs où chaque processeur exécute son code de manière asynchrone et indépendante. Pour assurer la cohérence des données, il est souvent nécessaire de synchroniser les processeurs entre eux, les techniques de synchronisation dépendent de l'organisation de la mémoire. Autrement dit, selon le modèle de programmation il existe deux types de parallélisme : le parallélisme de données et le parallélisme de programme ou parallélisme.

2.3 Types de parallèlisme

2.3.1 Parallèlisme de données

Dans ce modèle de parallélisme, les données sont partitionnées en des sous ensembles de fragments, le nombre de fragments appelé degré de partitionnement. Les différents fragments sont alloués aux différents processeurs. Le partitionnement peut être statique, selon un modèle fixe adapté, ou dynamique dans lequel les données sont partitionnées dynamiquement.

2.3.2 Parallèlisme de programme

Appelé généralement, parallélisme pipeline, il se base sur la décomposition du traitement en des sous traitements qui seront exécutés en parallèle. Il est difficile à implémenter dans des architectures à mémoire distribuée (le nombre de taches et le temps d'exécution varient dynamiquement). Comme le placement de données dans le data parallelism des sous traitements aux processeurs a un impact sur les performances pour le control parallelism.

2.4 Les architectures parallèles

Un ordinateur parallèle est une machine qui comportent une architecture parallèle, constituée de plusieurs processeurs identiques, ou non, qui concourent au traitement d'une application. En se basant sur le type d'interconnexion des différents composants (les processeurs, les mémoires principales et les disques). Les machines parallèles ont été classifiées en trois principales catégories :

2.4.1 Architecture à mémoire partagée (SM)

Les machines parallèles à mémoire partagée (Shared Memory (SM)), dont la structure est illustrée à la Figure4 (a), possèdent une mémoire commune accessible par tous les processeurs. La communication entre processeurs se fait par des lectures et écriture successives dans la mémoire de sorte qu'une écriture faite par un processeur peut ultérieurement être lue par un autre. Le lien entre les processeurs et la mémoire se fait par un réseau d'interconnexion ou un bus. Cette organisation entraîne toutefois un problème de gestion des conflits d'accès à la mémoire et qui augmente par l'augmentation du nombre de processeurs. Celui-ci peut être minimisé, entre autres, par la division de la mémoire en bancs accessibles par un seul processeur à un instant donné.

2.4.2 Architecture à disque réparti (SN)

Dans la classe des machines parallèles à disque réparti (shared-nothing SN), dont un schéma est donné à la Figure4(c), chaque processeur possède sa propre mémoire locale et ses propres disques et en a l'accès exclusif. La communication entre processeurs ne peut donc plus se faire par le biais de la mémoire. Elle se fait plutôt par l'envoi de messages entre processeurs à travers un réseau d'interconnexion qui relie les processeurs entre eux plutôt que de les relier à la mémoire comme dans le cas des machines SM.

2.4.3 Architecture à disque partagé (SD)

Dans une machine à disque partagé, chaque processeur dispose de sa mémoire locale alors que les disques sont partagés par tous les processeurs (voir la Figure4 (b)). La communication se fait via un réseau d'interconnexion, par passage de message. Comme la machine partagée, ce type de machine permet une flexibilité dans le choix du degré du parallélisme lié au partage des données (disques) sans pour autant souffrir du problème d'extensibilité et un potentiel d'équilibrage de charge dû au fait que les résultats intermédiaires sont accessibles vers tous les processeurs via les disques. Comme inconvénient, on peut citer le surcoût de gestion de la cohérence des données dans les mémoires des processeurs, et l'embouteillage lié à l'accès concurrent aux disques.

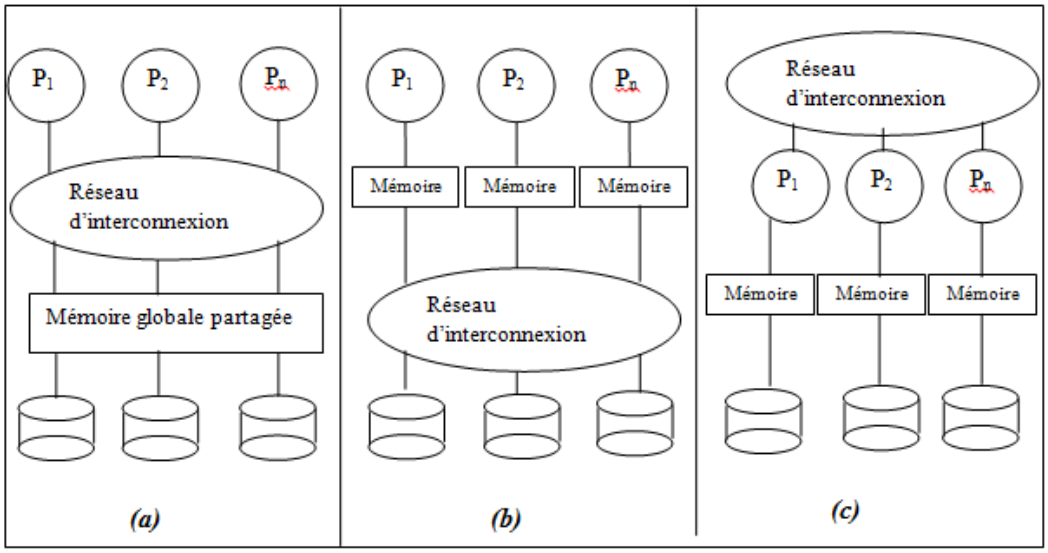


FIGURE 2.1 – Les architectures parallels (a) Shared Memory, (b) Shared Disks, (c) Shared Nothing.

2.5 Téchnique de résolution parallèle de CSP

On peut distinguer deux grandes catégories d'approches pour la résolution de contraintes parallèles :

2.5.1 Recherche concurrente

Chaque processeur exécute le même algorithme de recherche avec un ordre différent sur les variables ou carrément des algorithmes différents. L'intérêt est que les exécutions sont totalement indépendantes, quand un processus termine sa recherche c'est la fin globale de l'algorithme. La méthode n'est pas trop efficace (quantité de travail est augmentée pour le même calcul).

2.5.2 Résolution parallèle au niveau du nœud

La recherche se déroule de façon séquentielle, au niveau d'un nœud les contraintes sont vérifiées en parallèle mais les contraintes concernées ne sont pas les mêmes, ce qui pose problème (allocation statique ou dynamique de la charge). Certains auteurs ont préféré l'allocation statique, les processeurs libres peuvent exécuter par exemple un algorithme de filtrage.

2.5.3 Distribution de l'arbre de recherche

La distribution de l'arbre de recherche est connue sous le nom d'OÙ-parallèle en programmation logique. L'exploitation en parallèle des branches conduit à une solution si une branche au moins aboutit à une solution. Les différentes parties séquentielles s'exécutent indépendamment, la difficulté réside dans la technique de distribution et de l'équilibrage de charge.

2.6 Mesure de performance d'un algorithme parallèle

II existe plusieurs critères permettant d'évaluer la performance d'un algorithme parallèle. Cette section en présente quelques-uns parmi les plus utilisés, comme l'accélération, l'efficacité et le scale-up.

Définition 2.6.1. (Granularité du travail) On appelle granularité du travail la quantité du travail réalisé par chaque processeurs c'est à dire le rapport n_t/n_p ou nt représente le nombre de taches à réaliser et npreprésente le nombre de processeurs.

Définition 2.6.2. (Complexité en temps séquentiel) La complexité en temps séquentiel $T \times (n)$ correspond au temps du meilleur algorithme connu.

Définition 2.6.3. (Complexité en temps parallèle) La complexité en temps d'une application parallèle $T_p(n)$ est égale au temps d'exécution maximal d'un algorithme pour une exécution quelconque.

Définition 2.6.4. (Complexité en communication) La complexité en communication d'une application parallèle est égale au nombre maximal de messages échangés par les processeurs au cours d'une exécution quelconque d'algorithme.

2.7 L'environnement de développement utilisé dans la programmation parallèle

2.7.1 Langage de programmation

Pour développer une application parallèle sur le langage C++, il existe plusieurs méthodes, citons par exemple les MPI.

2.7.2 Message Passing Interface (MPI)

MPI (Message Passing Interface) est une spécification pour les développeurs et les utilisateurs des bibliothèques de passage de messages. MPI porte principalement

sur le modèle de programmation parallèle passage de messages : les données sont déplacées de l'espace d'adressage d'un processus visant à celle d'un autre processus à travers des opérations de coopération sur chaque processus. Les fonctionnalités que MPI :

- Environnement d'exécution.
- Types de données dérivés.
- Communicateurs et topologies.
- Gestion dynamique de processus.
- Entrées-sorties parallèles.
- Interface de profilage.

2.7.2.1 Les raisons d'utilisation de MPI

- Normalisation : MPI est la seule bibliothèque de passage de message qui peut être considéré comme une norme. Il est soutenu sur pratiquement toutes les plateformes HPC.
- Portabilité : Il y a peu ou pas besoin de modifier le code source lorsque vous portez votre application à une autre plate-forme que les supports le standard MPI.
- Occasions de rendement : implémentations de fournisseurs devraient être en mesure d'exploiter les fonctionnalités matérielles indigènes pour optimiser les performances.
- Fonctionnalité : Il y a plus de 440 routines définies dans MPI - 3, qui comprend la majorité des personnes dans MPI - 2 et MPI - 1.

2.7.2.2 Type de communication de MPI

Communication point à point Est la communication de base de MPI et les opérations effectuées sont un VHQG et un UHFHLYH. Une communication point à point a lieu entre deux processus, l'un est appelé émetteur, l'autre récepteur, identifiés par leur rang. De manière générale, les paramètres nécessaires à la construction d'une communication sont : les rangs des processus émetteur et récepteur ; l'étiquette du

message ; le nom du communicateur qui définit le contexte de la communication, Le type des données échangées, les données. Communications collectives Les communications collectives permettent de réaliser des communications impliquant plusieurs processus. Elles peuvent toujours être simulées par un ensemble de communications point à point avec éventuellement des opérations de réduction. Une communication collective implique l'ensemble des processus du communicateur utilisé. Jusqu'à MPI 2.2, il s'agissait d'appels bloquants. La norme MPI 3.0 introduit les appels non bloquants pour les communications collectives. communications mémoire à mémoire Les communications mémoire à mémoire (ou RMA pour Remote Memory Access ou one sided communications) consistent à accéder en écriture ou en lecture à la mémoire d'un processus distant sans que ce dernier doive gérer cet accès explicitement.

2.7.2.3 Avantages de MPI

- Permet de mettre en place plus efficacement certains algorithmes.
- Possibilité pour l'implémentation de regrouper plusieurs opérations.
- Plus performant que la communication point à point sur certaines machines (utilisation de matériels spécialisés tels que coprocesseur, mémoire spécialisée...).

2.7.2.4 Inconvénients de MPI

- La gestion des synchronisations est délicate.
- Pour les synchronisations cible passive, obligation d'allouer la mémoire avec MPI_Alloc_mem qui ne respecte pas la norme Fortran (utilisation de pointeurs Cray non supportés par certains compilateurs).
- Complexité et risques d'erreurs élevés.
- Moins performant que la communication point à point sur certaines machines.

2.8 Conclusion

Dans ce chapitre nous avons présentés différentes notions liées au parallélisme en essayant d'expliquer aux mieux chaque points. Dans le chapitre suivant nous allons présenter différents méthodes utilisées dans la compression des contraintes tables proposées dans la littérature.

Chapitre 3

Compression des constraints tables

3.1 Introduction

Les problèmes de satisfaction de contraintes forment une grande classe de problèmes combinatoires. L'objectif des CSP est de trouver une instanciation possible pour chaque variable qui satisfait toutes les contraintes. Pour cela, de nombreuses méthodes ont été proposées pour résoudre les CSPs. Dans ce chapitre, nous allons présenter quelques méthodes utilisées dans la littérature pour la compression des contraintes tables ainsi que le filtrage séquentiel et les méthodes parallèle et séquentiel de résolutions.

3.2 Méthodes de Compression

3.2.1 Méthode Statique

Les Multi-Valued Decsion Diagram (MDD) [9] est l'une des méthodes qui offrent une représentation compacte de la contrainte table pour facilite le filtrage. C'est une extension multi-valeur des BDDs . Un MDD, est un Directed Acyclic Graph (DAG) avec une racine, qui est utilisé pour représenter des fonctions $f : 0...d-1r \rightarrow true, false$, basées sur un entier donné d. Pour r variables données, la représentation

par un DAG est faite pour contenir r niveaux, tel que chaque variable soit représentée par un niveau spécifique du graphe. Chaque nœud d'un niveau spécifique possède au plus d arcs sortants vers les nœuds du niveau suivant du graphe. Chaque arc possède une étiquette correspondant à sa valeur. Le niveau final est représenté par le noeud terminal true (le nœud terminal false est souvent omis). Il y a une équivalence entre $f(v_1, ..., v_r) = true$ et l'existence d'un chemin du noeud racine au noeud terminal true et dont les arcs sont étiquetés par $v_1, ..., v_r$.
Les nœuds sans aucun arc sortant ou arc entrant sont supprimés. Dans une contrainte MDD, le MDD modélise l'ensemble des tuples qui satisfont la contrainte, tel que chaque chemin du nœud racine au nœud terminal true correspond à un tuple autorisé. Chaque variable du MDD correspond à une variable de la contrainte. Un arc associé à une variable du MDD correspond à une valeur de la variable correspondante de la contrainte. Par commodité, dans cet article les auteurs notent d le nombre maximum de valeur dans le domaine d'une variable et un arc sortant de x vers y étiqueté par v sera noté (x, v, y), et aussi par $\omega+(n)$ l'ensemble des arcs sortants de n. Un exemple de MDD est donné en . Ce MDD représente tous les tuples possibles pour les valeurs 0, 1, 2, 3. Pour chaque tuple, il y a un chemin du nœud racine (0) vers le nœud terminal (tt) dont les arcs sont étiquetés par les valeurs du tuple.

3.2.2 Méthodes dynamique

Les méthodes dynamique permettent une compression de la contrainte table lors du filtrage, en supprimant les tuples invalides. Les algorithmes de réduction tabulaires simple STR1 [10], STR2 [7], STR3 [6] proposent ce genre de compression. L'uniformité d'arc (arc consistency AC) est une propriété qui peut être employé pour identifier et enlever quelques valeurs contradictoires, c.-à-d. les valeurs qui ne peuvent pas mener à aucune solution. C'est un composant essentiel de l'algorithme de maintien d'uniformité d'arc (MAC), qui est utilisé généralement pour résoudre des exemples binaires de CSP. Pour les contraintes non-binaires, l'uniformité d'arc généralisé (generalized arc consistency GAC) remplace AC. Les différentes variantes de STR, sont des algorithmes basés sur la réduction tabulaire simple pour imposer GAC sur des contraintes positives de table. Les algorithmes de GAC suivent normalement

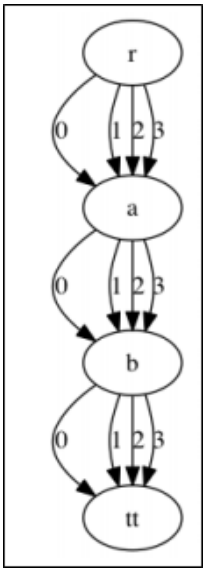


FIGURE 3.1 – Représentation d'un MDD autorisant tous les tuples pour le domaine 0,1,2,3.

le même modèle : une valeur d'un domaine est avérée être conforme (uniforme) en produisant un tuple valide contenant cette valeur (dans le cas des contraintes positives de table) ou en produisant des preuves de structures auxiliaires, par exemple, un chemin en cas de BDDs (diagrammes de décision booléens), de MDDs (diagrammes de décision à valeurs multiples). Ceci est habituellement fait en parcourant ces structures et en exécutant des tests sur chaque sous-structure STR1, est un algorithme de GAC qui met à jour dynamiquement des tables pendant la recherche. Tandis que d'autres algorithmes de GAC traitent des tables comme des structures fixes ou recourent à aborder les structures comparables créées de ces tables, STR1 montre que la manipulation des tables directement pendant la recherche de backtracking n'est pas aussi couteuse qu'on a pu par le passé le pensé. STR1 fonctionne en principe en scannant chaque tuple un à un. Si un tuple est invalide, il est enlevé, et la table est en conséquent réduite d'une rangée. Autrement, le tuple est valide et ses composants (valeurs) sont donc des valeurs de domaine qui se sont avéré avoir un support. STR1 reconsidère alors le tuple encore une fois et sauvegarde ses valeurs (dans un ensemble appelés les $gacValues(X)$ qui est défini pour chaque variable X dans la portée de la

contrainte). Quand STR1 a fini son parcourt de la table entière, n'importe quelle valeur n'apparaissant pas dans ces ensembles ($gacValues$) n'a de ce fait aucun support et sera supprimée du domaine où elle apparait. STR2 apporte deux améliorations à STR1. Quand un tuple t est inspecté pour assurer la validité, il n'y a aucun besoin de vérifier si $\tau[X]? \in D(X)$ s'il n'y a eu aucun changement au domaine de X depuis la dernière fois que STR2 a été appliqué à cette contrainte. En outre, dans le cas où le tuple τ est valide, quand l'algorithme passe par chaque composant de τ, $\tau[X]$ est ignoré, si $gacValues(X) = D(X)$ est déjà connue (c.-à-d., chacune des valeurs simple de $D(X)$ possède déjà un support). La première amélioration cependant implique les structures de données additionnelles qui doivent être maintenues afin de fournir le maximum d'avantage d'optimisation. Comme STR1 et STR2, STR3 n'examine un tuple qu'une fois qu'on l'a identifié comme étant invalide. À la différence de STR1 et de STR2, STR3 n'écarte pas immédiatement les tuples invalides des tables. En effet, STR3 ne traite pas des tables directement, mais utilise à la place des index permettant l'identification rapide de tous les tuples contenant une valeur donnée d'une variable donnée. STR3 conserve une structure de données séparées qui permet à des contrôles de validité d'être faits dans un temps constant, plutôt que la révision des indexes dynamiquement pendant la recherche.

3.2.3 Méthodes Hybrides

La dénotation hybride vient du fait qu'on cherche à combiner compression statique et compression dynamique en échangeant leurs avantages respective. En effet dans [2,3] les auteurs reviennent à la simple idée de compression produit cartésien et table d'indexation et examinent si les approches de compression statiques et dynamiques peuvent être efficacement combinées sur les contraintes de table. Ils étendent l'algorithme STR pour gérer les tuples compressés grâce à leurs propres méthodes de compression présenté ci-dessous Dans [3] les auteurs proposent une méthode de compression qui est basée sur la réduction des motifs fréquents. Les motifs ont été définis comme une séquence de valeurs consécutives qui peuvent ne pas être affectés aux mêmes variables c'est-à-dire une même suite d'affectation à une séquence de variables consécutives (qui peut différer d'un motif à l'autre). Afin de réduire la complexité spa-

tiale de la représentation des tuples de chaque contrainte table, ils commencent par repérer les motifs les plus fréquents et remplacer chaque occurrence de motif par un symbole unique. De ce fait, la taille utilisée pour représenter la contrainte table sera d'autant plus faible que la longueur des motifs sera grande et que leur nombre d'occurrences sera important. Pour identifier les motifs pertinents, dans un premier temps on commence par créer une forêt d'arbres de préfixes à partir des différents tuples d'une contrainte table donnée. Un arbre enregistre toutes les séquences existantes de valeurs de longueur donnée et leur nombre d'occurrences. Dans un second temps, il faut identifier les motifs les plus efficaces pour le processus de compression. Pour cela ils introduisent la notion de score d'un motif $\mu quiest$:score$(\mu) = \|\mu\| \times nbOcc(\mu)$. Un seuil de sélection est fixé, seuls les motifs dont le score est supérieur au seuil de sélection sont retenus dans l'algorithme de compression et stockés dans la table des motifs. Pour des raisons d'efficacité, le nombre total de motifs retenus est borné par un second paramètre afin de contrôler le temps de compression. Un parcours de la table est effectué pour détecter la présence des motifs retenus et établir une référence vers la table de motifs. Si dans un tuple, plusieurs motifs se recouvrent, l'algorithme de compression choisit en priorité le motif ayant le meilleur score. Après compression, la table contient des tuples de longueurs différentes composés de valeurs et de références vers la table des motifs.

Dans [2] les auteurs proposent aussi leur propre méthode de compression associée à STR. Cet article est une sorte de suite à [3] Tel que les motifs sont cette fois ci sont définit comme étant une séquence de valeurs qui sont affectés aux mêmes variables qui ne sont pas obligatoirement consécutives c'est-à-dire une même suite d'affectation à une séquence de variables non-consécutives. Afin d'identifier les motifs qui seront utile à la compression la construction d'un FP-Tree (Free –Pattern-tree) qui est la première étape de l'algorithme FP-Growth dont les détails sont donnés dans [4,5] s'avère nécessaire. L'algorithme prend comme paramètre minSupport qui est le nombre minimal d'occurrence d'un motif pour qu'il soit considéré comme fréquent. Dans une première étape on collecte le nombre d'occurrence de chaque valeurs pour chaque variables, noté fréquence. Dans un second temps on triera les valeurs des tuples par ordre décroissant de fréquence (les valeurs qui ont une fréquence en dessous de minSupport sont supprimées). Chaque tuple est ensuite inséré dans le FP-Tree

qui est essentiellement un arbre de préfixe où chaque branche représente la partie fréquente d'un tuple et chaque nœud contient le nombre de branches qui partagent ce nœud. Chaque arête liant un parent à son enfant est étiquetée avec une valeur. Le nœud racine n'est pas étiqueté. On insert le premier tuple dans l'arbre qui est le début de $\tau 1$. Cela crée la branche la plus à gauche de l'arbre. Chaque nœud de cette branche a initialement une fréquence de 1. On insert ensuite les autres tuples en ne créant de nouvelles branches que si cela est nécessaire, lorsque deux tuples partagent la même arrête la fréquence est incrémenter. Chaque nœud de l'arbre correspond à un motif fréquent μ qui peut être lu sur le chemin menant de la racine au nœud lui-même. La fréquence f de ce motif est donnée par le nœud lui-même. Les gains qui peuvent être obtenus par lafactorisation de ce motif fréquent est $\|\mu\| \times (f-1)$ valeurs (on peut supprimer toutes les occurrences du motif, sauf une). On réduit l'arbre en supprimant les nœuds qui offrent un gain moindre que leur père. Les feuilles de l'arbre obtenu représentent les motifs fréquents utilisés dans la compression. Pour terminer, on crée un fragment pour chaque motif fréquent que nous avons identifié.

3.2.4 Smart Table

Les auteurs dans [8] propose une toute autre façon de représenter les contraintes tables, des contraintes arithmétiques simples. Ces tuples sont appelés smart tuples et les contraintes table sur des smart tuples, des contraintes smart table. Un ensemble de smart tuples est appelé smart table et est représenté par $table(sc)$ pour une contrainte smart table sc. Un smart tuple σ est lui-même un ensemble de contraintes de tuple. Une contrainte tuple peut prendre quatre formes différentes :

— $var\ op\ a$
— $var\ \in Souvar\ \notin\in S$
— $var\ op\ var$
— $var\ op\ var\ b$

où var est une variable de la portée de la contrainte smart table, a et b, des constantes, S, un ensemble de constantes et op un opérateur choisi dans $, \leq, , \neq, \geq,$. La sémantique d'une contrainte smart table est simple et naturelle : un tuple classique t est accepté par une contrainte smart table sc ssi il existe $\sigma \in table(sc)$

tel que t satisfait s. Une variable de la contrainte qui n'est pas contrainte par une contrainte de tuple peut prendre n'importe quelle valeur de son domaine dans le smart tuple. Les contraintes smart table peuvent être utilisées pour modéliser efficacement de nombreuses contraintes, y compris certaines contraintes globales bien connues. Nous donnons ci-dessous un exemple de contrainte smart table encodant la contrainte de l'ensemble d'éléments suivant (1,2,1),(1,3,1), (2,2,2), (2,3,2), (3,2,3), (3,3,3). Les contraintes de tuple sont écrites directement dans la table de la contrainte dans la colonne correspondant à la première variable de la contrainte de tuple.

X1	X2	X3
= X3	≥ 2	*

FIGURE 3.2 – exemple de smart table.

Voici un résumé de notre classification des méthodes de compression sous forme de figure récapitulatif :

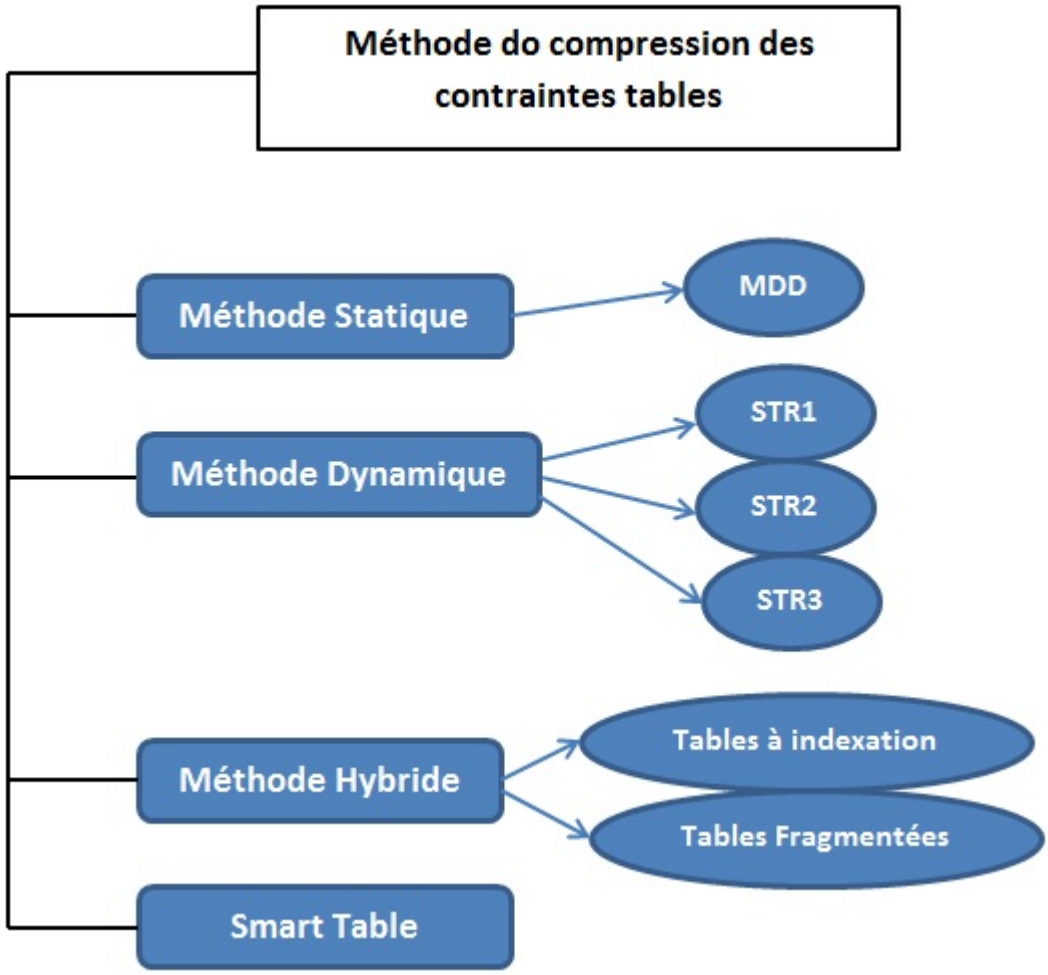


FIGURE 3.3 – Classification des méthodes de compressions.

3.3 Conclusion

Dans ce chapitre, nous avons présenté les principales techniques de compressions des contraintes tables et les méthodes filtrage et de résolution des problèmes de satisfaction de contrainte. Dans le chapitre suivant nous allons présenter notre contribution pour ces deux axes qui sont la compression et le filtrage des CSP

Chapitre 4

Proposition d'une méthode de compression

4.1 Introduction

Après avoir étudié les différentes approches citées dans le chapitre précédant, nous nous sommes intéressés aux méthodes de compressions hybrides. En effet, l'avantage d'une double compression (la première se faisant à l'aide d'un algorithme de compression et la seconde pendant le filtrage en supprimant les tuples invalides) est intéressante, cela additionné à des résultats concurrentiels par rapport aux algorithmes de filtrage STR2 et STR3 dont ils ont fait preuve a suscité notre intérêt, notamment [2]. Nous nous sommes donc penchés sur de possibles améliorations à apporter à la méthode de compression que les auteurs ont proposée dans [2]. Après avoir appliqué l'algorithme de compression sur différentes tables, on s'est rendues compte que lors de la compression, les tuples qui n'ont pas été compressés, pouvait subir une seconde compression. On s'est donc demandé pour quoi et nous sommes arrivé aux conclusions suivantes : lors du ré-ordonnancement des valeurs des tuples on privilégie celles qui ont une fréquence plus grande, ceux qui veut dire, les valeurs qui sont les plus récurent et on néglige ceux qui ont une fréquence moindre et qui peuvent quand même être utile. Lors de la construction du FP-Tree on remarque aussi que certaines valeurs qui pourraient faire partie des motifs à testés se retrouvent parmi les feuilles et sont supprimées car leurs fréquences dans l'arbre est inférieure au minSupport, et cela est due aussi au ré-ordonnancement des valeurs

car après avoir privilégié les valeurs qui ont une plus grande fréquence, celles qui ont une fréquence équivalente ne sont pas réordonnées et gardes leurs hiérarchie selon l'ordre d'apparition des variables, alors que les valeurs qui ont une même fréquence n'ont pas forcément la même utilité lors de la compression. Ceci dit le FP-tree peut aussi ne pas être adapté pour aider à choisir les motifs mais on ne s'est pas attardés sur cette question car dans notre approche nous n'utilisons pas d'arbres. Enfin C'est pour toutes ces raisons que nous avons décidé de reprendre quelques grandes lignes de cette compression et de suivre une tout autre philosophie. Notre contribution qui est présenté dans ce chapitre consistes en quelques mots à trouver tous les motifs pouvant être contenue dans la contraintes table et effectue un trie dynamique par la suite afin de choisir les motifs qui seront retenus. En second lieu nous allons présenter notre contribution aux méthodes de filtrages, qui consiste à paralléliser les deux algorithmes AC4 et AC6 qui sont connus pour leurs performances en matière de temps d'exécution. Dans l'optique de diminuer encore plus le temps d'exécution de ces deux algorithmes nous nous sommes donc intéresser à l'idée d'exploiter les avantage de l'approche parallèle, nous présenterons notre idée de décomposition de AC4 et AC6 et les différentes procédures utilisées pour se faire.

4.2 Problématique

Les contraintes tables sont très étudiées dans la programmation par contrainte, elles sont un support de sauvegarde des solutions d'une contrainte ou d'un CSP. Mais leur taille peut croitre exponentiellement avec leurs arités, ce qui les rend difficile à manipuler. Afin de palier à cette difficulté nous proposons dans ce document un algorithme de compression afin de réduire l'espace nécessaire pour les représenter.

4.3 La compression

4.3.1 La construction de la table des motifs :

Afin de trouver tous les motifs qui peuvent exister dans la table de contrainte, nous explorerons tous les cas possibles en procédant de manière ordonnée c'est-à-dire en parcourant colonne par colonne de la première jusqu'à l'avant dernière pour trouver les motifs qui commencent par la variable de chaque colonne en court de traitement. Pour ce faire lors du traitement d'une colonne on cherchera les valeurs dont la fréquence d'apparition est supérieure ou égale à deux (apparaissant au moins dans deux tuples), on identifiera les tuples dans lesquels cette valeur apparait et on refait presque le même traitement mais cette fois sur les tuples identifié seulement et pour les autres colonnes à chaque fois qu'on trouve une valeurs récurrentes on sauvegarde les tuples, la variable sur laquelle la valeur apparait et la valeur en question, ajouté à la combinaison (variable, valeur) en court de traitement ce qui nous fait un motif de longueur 2, on réitère le traitement jusqu'à avoir parcouru toutes les colonnes. On refait le traitement pour les autres variables de la colonne en court de traitement. Ensuite on passe aux autres colonnes ainsi de suite. A chaque fois qu'on construira un tableau de motif de longueur 2 on fusionnera les motifs qui apparaissent sur les mêmes tuples, ce qui nous donnera au final la liste des motifs commençant par la variable traitée. On sauvegardera bien évidemment le résultat dans un tableau final ayant pour colonnes, motifs, tuples et gain où le gain est : $gain = \|\mu\| \times (f - 1)$.

4.3.2 La sélection

Dans la table où seront enregistrés les motifs on effectuera le traitement suivant afin de sélectionner ceux qui seront jugés utiles pour la compression. Pour avoir une meilleure compression possible il nous faudra retenir dans un premier temps les motifs ayant le plus grand gain puis on passera à ceux ayant un gain en dessous en procédant par élimination tel que à chaque fois qu'un tuple sera ajouté à la tablé compressé, les motifs dans lesquels il apparait ne seront plus pris en considération.

Dans le cas où plusieurs motifs ont le même gain, on envisagera plusieurs cas de figures :

- Les motifs ont le même gain mais apparaissent sur des tuples différent des uns des autres, dans ce cas tous les motifs seront sélectionnés.
- Les motifs ont des tuples en commun, dans ce cas on devra faire un choix en privilégiant les motifs qui apparaissent sur le plus de motifs possibles puis viendra la longueur des motifs (dans le cas où le nombre de tuples est identique par exemple).

4.4 Exemple d'application de la méthode de compression

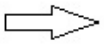
Soit la contrainte table suivante définie sur C : tel que $Scope(C) = \{x_1, x_2, x3, x4, x5\}$. $D(x_1) = D(x_2) = D(x_3) = D(x_4) = D(x_5) = \{a, b, c\}$.

X1	X2	X3	X4	X5
c	b	c	a	c
a	a	b	b	a
a	c	b	c	a
b	a	c	b	c
b	a	a	b	b
a	c	b	c	b
a	c	a	c	a

FIGURE 4.1 – Contrainte table.

4.4.1 Le résultat de la première

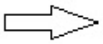
x1=a, x2=c	6, 7
x1=a, x3=b	2, 3, 7
x1=a, x4=c	3, 6, 7
x1=a, x5=a	2, 3, 7



x1=a, x2=c, x4=c,	6, 7
x1=a, x3=b, x5=a	2, 3, 7
x1=a, x4=c	3, 6, 7

FIGURE 4.2 – résultat de la première étape de compression pour x1=a.

x1=b, x2=b	2, 5
x1=b, x4=b	2, 5



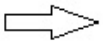
X1=b, x2=a, x4=b	2, 5

FIGURE 4.3 – résultat de la première étape de compression pour x1=b.

x2=a, x4=b	2, 4, 5

FIGURE 4.4 – résultat de la première étape de compression pour x2=a.

x2=c, x3=b	3, 6
x2=c, x4=c	3, 6, 7
x2=c, x5=a	3, 7



x2=c, x3=b, x4=c	3, 6
x2=c, x4=c	3, 6, 7
x2=c, x4=c, x5=a	3, 7

FIGURE 4.5 – résultat de la première étape de compression pour x2=c

x3=c, x5=c	1, 4

FIGURE 4.6 – résultat de la première étape de compression pour x3=c.

x3=b, x4=c	3, 6
x3=b, x5=a	2, 3

FIGURE 4.7 – résultat de la première étape de compression pour x3=b.

x4=c, x5=a	3, 7

FIGURE 4.8 – résultat de la première étape de compression pour x4=c.

4.4.2 La table des motifs

On obtient le tableau des motifs suivant :

Motifs	Tuples	Résultat du produit
x1=a, x2=c, x4=c	6, 7	3
x1=a, x3=b, x5=c	2, 3, 7	6
x1=a, x2=c, x4=c	3, 6, 7	6
x1=b, x2=a, x4=b	2, 5	3
x2=a, x4=b	2, 4, 5	4
x2=c, x3=b, x4=c	3, 6	3
x3=c, x5=c	1, 4	2
x3=b, x5=a	2, 3	2
x4=c, x5=a	3, 7	2

FIGURE 4.9 – Table des motifs.

4.4.3 La table compressée

X1	X2	X4
A	c	c

X3	X5	
b	a	T3
b	b	T6
a	a	T7

X2	X4
a	b

X1	X3	X5	
a	b	a	T2
b	c	c	T4
b	a	b	T5

X1	X2	X3	X4	X5	
C	b	C	a	c	T1

FIGURE 4.10 – résultat de la compression.

4.5 Expérimentation

Nous allons maintenant appliquer les deux algorithmes à une contrainte table. Soit la contrainte table suivante, définie sur C tel que : $Scope(C) = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$. $D = D(x_1) = D(x_2) = D(x_3) = D(x_4) = D(x_5) = D(x_6) = D(x_7) = \{a, b, c\}$.

X1	X2	X3	X4	X5	X6	X7	
a	b	b	a	c	c	a	T1
b	a	b	c	c	a	a	T2
c	c	a	a	b	b	c	T3
a	b	a	c	b	c	b	T4
b	c	a	b	a	a	c	T5
b	a	c	a	c	c	a	T6
c	b	a	b	c	b	a	T7
c	a	b	c	c	a	b	T8
b	b	a	c	a	c	b	T9
a	b	c	a	c	b	b	T10

FIGURE 4.11 – La contrainte table de l'expérimentation.

4.5.1 Application de la méthode dans [2]

Avant de commencer résumant les différentes étapes de compression de cette méthode :

- Construction de la table des fréquences : qui consiste à rassemblé toutes les fréquences de toutes les valeurs de chaque domaine pour chaque variable .
- Ré-ordonnancement des tuples : qui consiste à réordonner chaque tuples selon la fréquence des valeurs affectées aux variables .
- Construction du FP-tree : qui consiste à insérer les tuples dans un arbres de préfixes où chaque branche représente la partie fréquente du tuple et le nœud une valeur qui est incrémenté à chaque insertion .

	X1	X2	X3	X4	X5	X6	X7
a	3	3	5	4	2	3	4
b	4	5	3	2	2	3	4
c	3	2	2	4	6	4	2

FIGURE 4.12 – Table des fréquences.

X5= c	X2= b	X4=a	X6=c	X7=a	X1=a	X3=b
X5=c	X1=b	X4=c	X7=a	X2=a	X3=b	X6=a
X3=a	X4=a	X1=c	X6=b	X2=c	X5=b	X7=c
X2=b	X3=c	X4=c	X6=c	X7=b	X1=a	X5=b
X3=a	X1=b	X6=a	X2=c	X4=b	X5=a	X7=c
X5=c	X1=b	X4=a	X6=c	X7=a	X2=a	X3=c
X5=c	X2=b	X3=c	X7=a	X1=c	X6=b	X4=b
X5=c	X4=c	X7=c	X1=c	X2=a	X3=b	X6=a
X2=b	X3=a	X1=b	X4=c	X6=c	X7=b	X5=a
X5=c	X2=b	X4=a	X7=b	X1=a	X6=b	X3=c

FIGURE 4.13 – Ré-ordonnancement des tuples..

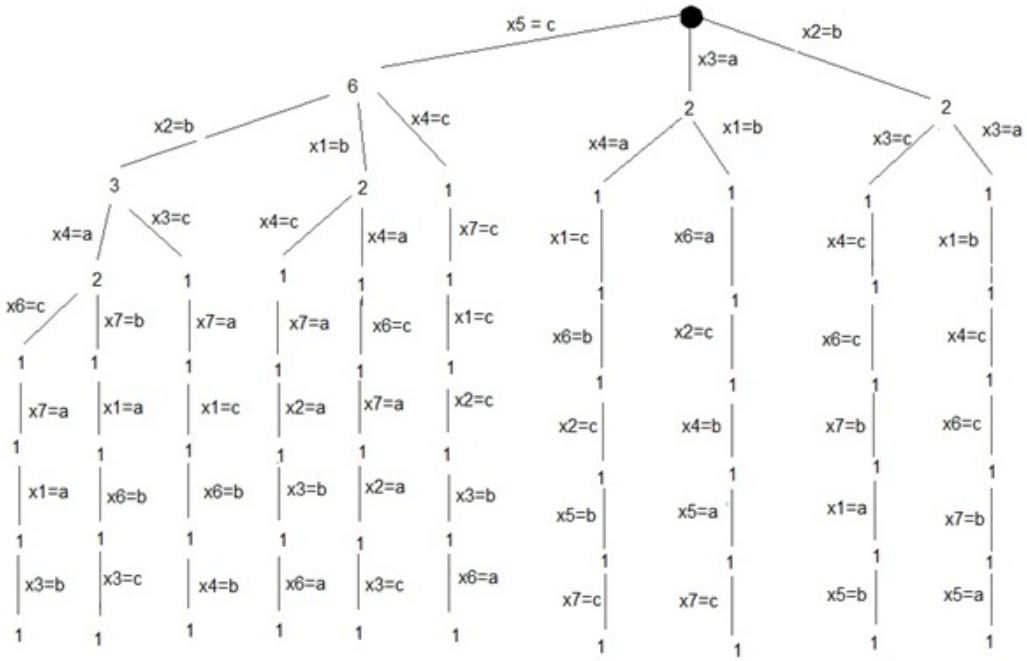


FIGURE 4.14 – Le FP-tree.

X2	X5
b	c

X1	X3	X4	X6	X7
a	b	a	c	a
c	a	b	b	a
a	c	a	b	b

X1	X5
b	c

X2	X3	X4	X6	X7
a	b	c	a	a
a	c	a	c	a

X1	X2	X3	X4	X5	X6	X7	
c	c	a	a	b	b	c	T3
a	b	a	c	b	c	b	T4
b	c	a	b	a	a	c	T5
c	a	b	c	c	a	b	T8
b	b	a	c	a	c	b	T9

FIGURE 4.15 – Résultat de la compression avec la première méthode.

4.5.2 Application de la méthode proposée

Les différentes étapes de notre méthode de compression sont les suivantes :

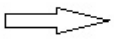
— Pour chaque valeur de chaque récurrente de chaque colonne trouvé d'autres valeurs récurrentes dans les autres colonnes afin de construire des motifs de longueur 2.

— Fusionner les motifs de longueur 2 qui apparaissent sur les même tuples pour chaque résultat de chaque colonne.

x1=a, x2=b	1, 4, 10
x1=a, x4=a	1, 10
x1=a, x5=c	1, 10
x1=a, x6=c	1, 4
x1=a, x7=b	4, 10

x1=b, x2=a	2, 6
x1=b, x3=a	5, 9
x1=b, x4=c	2, 9
x1=b, x5=c	2, 9
x1=b, x5=a	5, 9
x1=b, x6=a	2, 5
x1=b, x6=c	6, 9
x1=b, x7=a	2, 6

x1=c, x3=a	3, 7
x1=c, x5=c	7, 8
x1=c, x6=b	3, 7



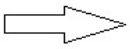
x1=a, x2=b	1, 4, 10
x1=a, x2=b, x4=a, x5=c	1, 10
x1=a, x2=b, x6=c	1, 4
x1=a, x2=b, x7=b	4, 10
x1=b, x2=a, x5=c, x7=a	2, 6
x1=b, x3=a, x5=a	5, 9
x1=b, x4=c	2, 9
x1=b, x6=a	2, 5
x1=b, x6=c	6, 9
x1=c, x3=a, x6=b	3, 7
x1=c, x5=c	7,8

FIGURE 4.16 – résultat de la première étape de la compression sur x1

x2=b, x3=a	7, 9
x2=b, x4=a	1, 10
x2=b, x5=c	1, 7, 10
x2=b, x6=c	1, 9
x2=b, x6=b	7, 10
x2=b, x7=a	1, 7
x2=b, x7=b	9, 10

x2=a, x3=b	2, 8
x2=a, x4=c	2,8
x2=a, x5=c	2, 6, 8
x2=a, x6=a	2, 8
x2=a, x7=a	2, 6

x2=c, x3=a	3, 5
x2=c, x7=c	3, 5



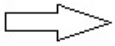
x2=b, x3=a	7, 9
x2=b, x4=a, x5=c	1, 10
x2=b, x2=b, x6=c	1, 4
x2=b, x5=c	1, 7, 10
x2=b, x6=c	1, 9
x2=b, x5=c, x6=b	7, 10
x2=b, x5=c, x7=b	1, 7
x2=b, x7=b	9, 10
x2=a, x3=b, x4=c, x5=c, x6=a	2, 8
x2=a, x5=c	2, 6, 8
x2=a, x5=c, x7=a	2, 6
x2=c, x3=a, x7=c	3, 5

FIGURE 4.17 – résultat de la première étape de la compression sur x2

x3=b, x4=c	2, 8
x3=b, x5=c	1, 2, 8
x3=b, x6=a	2, 8
x3=b, x7=a	1, 2

x3=a, x4=b	5, 7
x3=a, x5=b	3, 4
x3=a, x6=b	3, 7
x3=a, x7=c	3, 5

x3=c, x4=a	6, 10
x3=c, x5=c	6, 10

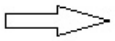


x3=b, x4=c, x5=c, x6=a	2, 8
x3=b, x5=c	1, 2,8
x3=b, x5=c, x7=a	1, 2
x3=a, x4=b	5, 7
x3=a, x5=b	3, 4
x3=a, x6=b	3, 7
x3=a, x7=c	3, 5
x3=c, x4=a, x5=c	6, 10

FIGURE 4.18 – résultat de la première étape de la compression sur x3

x4=a, x5=c	1, 6, 10
x4=a, x6=c	1, 6
x4=a, x6=b	3, 10
x4=a, x7=a	1, 6

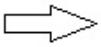
x4=c, x5=c	2, 8
x4=c, x6=a	2, 8
x4=c, x6=c	4, 9
x4=c, x7=b	4, 8, 9



x4=a, x5=c	1, 6, 10
x4=a, x5=c, x6=c, x7=a	1, 6
x4=a, x6=b	3, 10
x4=c, x5=c, x6=a	2, 8
x4=c, x6=c, x7=b	4, 9
x4=c, x7=b	4, 8, 9

FIGURE 4.19 – résultat de la première étape de la compression sur x4

x5=c, x6=c	1, 6
x5=c, x6=a	2, 8
x5=c, x6=b	7, 10
x5=c, x7=a	1, 2, 6, 7
x5=c, x7=b	8, 10



x5=c, x6=c, x7=a	1, 6
x5=c, x6=a	2, 8
x5=c, x6=b	7, 10
x5=c, x7=a	1, 2, 6, 7
x5=c, x7=b	8, 10

FIGURE 4.20 – résultat de la première étape de la compression sur x5

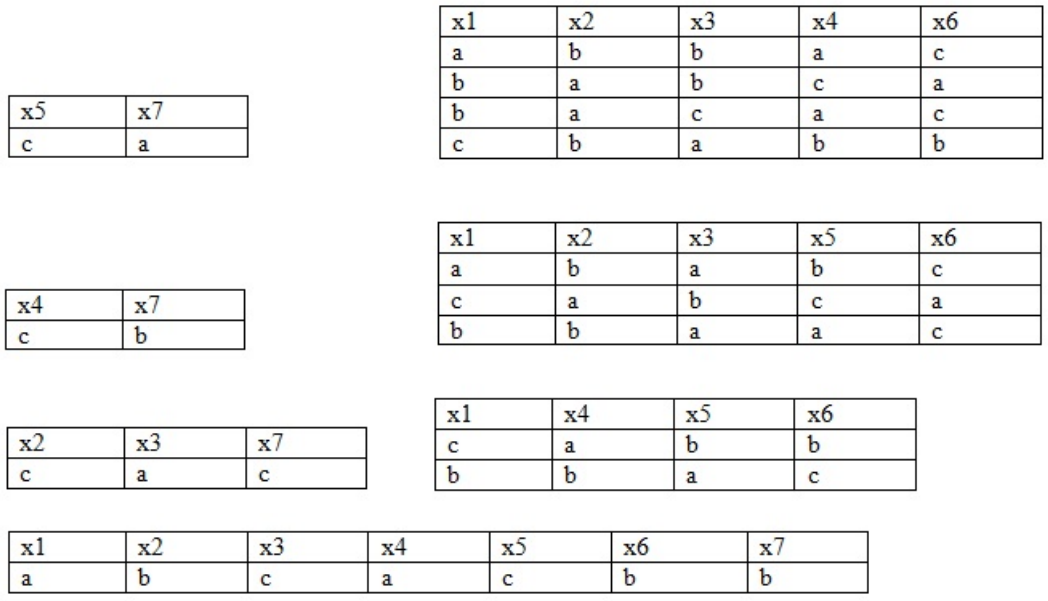
x6=c, x7=a	1, 6
x6=c, x7=b	4, 9

FIGURE 4.21 – résultat de la première étape de la compression sur x6

Motifs	Tuples	Gain
x1=a, x2=b	1, 4, 10	4
x1=a, x2=b, x4=a, x5=c	1, 10	4
x1=a, x2=b, x6=c	1, 4	3
x1=a, x2=b, x7=b	4, 10	3
x1=b, x2=a, x5=c, x7=a	2, 6	4
x1=b, x3=a, x5=a	5, 9	3
x1=b, x4=c	2, 9	2
x1=b, x6=a	2, 5	2
x1=b, x6=c	6, 9	2
x1=c, x3=a, x6=b	3, 7	3
x1=c, x5=c	7, 8	2
x2=b, x3=a	7, 9	2
x2=b, x5=c	1, 7, 10	4
x2=b, x6=c	1, 9	2
x2=b, x5=c, x6=b	7,10	3
x2=b, x5=c, x7=a	1, 7	3
x2=b, x7=b	9, 10	2

Motifs	Tuples	Gain
x2=a, x3=b, x4=c, x5=c, x6=a	2, 8	5
x2=a, x5=c	2, 6, 8	4
x2=a, x5=c, x7=a	2, 6	3
x2=c, x3=a, x7=c	3, 5	3
x3=b, x5=c	1, 2, 8	4
x3=b, x5=c, x7=a	1, 2	3
x3=a, x4=b	5, 7	2
x3=a, x5=b	3, 4	2
x3=c, x4=a, x5=c	6, 10	3
x4=a, x5=c	1, 6, 10	4
x4=a, x5=c, x6=c, x7=a	1, 6	4
x4=a, x6=b	3, 10	2
x4=c, x6=c, x7=b	4, 9	3
x4=c, x7=b	4, 8, 9	4
x5=c, x7=a	1, 2, 6, 7	6
x5=c, x7=b	8, 10	2

FIGURE 4.22 – Table des motifs.



x5	x7
c	a

x1	x2	x3	x4	x6
a	b	b	a	c
b	a	b	c	a
b	a	c	a	c
c	b	a	b	b

x4	x7
c	b

x1	x2	x3	x5	x6
a	b	a	b	c
c	a	b	c	a
b	b	a	a	c

x2	x3	x7
c	a	c

x1	x4	x5	x6
c	a	b	b
b	b	a	c

x1	x2	x3	x4	x5	x6	x7
a	b	c	a	c	b	b

FIGURE 4.23 – Résultat de la compression.

4.5.3 Discussion

Nous pouvons pu voir clairement la différence entre les deux résultats. Le résultat de notre compression étant plus compact, notre approche a donc tenue ses promesses, nous avons réussie à avoir un seul tuple non compressé, contrairement à l'approche de [2] pour laquelle on a cinq tuples non compressé qui peuvent subir une compression par exemple pour les tuples 5 et 9 on peut les compressé grâce au motif (x1, x2, x5) ou aussi les tuples 3 et 4 grâce au motif (x3, x5). A défaut de temps nous n'avons malheureusement pas implémenté un algorithme de filtrage pour notre approche, ce qui lui aurait apporté une seconde compression (en supprimant les tuples invalides) et nous aurions eu un résultat encore plus compact. Cela aussi nous aurait aussi permis de tester notre approche sur des exemples réel de contraintes tables (dont la taille dépasse de très loin celle de la contrainte table présentée) et on aurait pu comparer les méthodes selon d'autres critères qui sont, la taille (en Megaoctets), le pourcentage de compression, et le temps CPU pour l'exécution de l'algorithme.

Nous comptons quand même améliorer cette méthode de compression grâce à un algorithme de filtrage très prochainement, en tenant compte des résultats du travail mon collègue M.Mokrani qui s'est penché sur la question de la parallélisassions des algorithmes de filtrage.

4.6 Conclusion

Nous avons vu dans ce chapitre le déroulement des deux méthodes qui sont [2] et celle proposée, on a pu ainsi comparer les résultats des approches et on a pu voir clairement que celle que nous proposons est plus compacte. Dans le prochain chapitre nous présenterons un algorithme parallèle de consistence d'arc.

Chapitre 5

Proposition d'une méthode de filtrage

5.1 Introduction

dans ce chapitre nous allons présenter au début quelque méthodes de filtrage des CSPs basé sur la notion de la consistance d'arc ainsi qu'une méthode de résolution parallèle aussi basé sur la consistance d'arc, en suit nous proposerons des méthodes de filtrage parallèle basé sur les algorithmes AC-4 et AC-6.

5.2 Algorithms de filtrage

5.2.1 Making AC-3 an Optimal Algorithm

Dans cet article [Yuanlin Zhang et Roland H.C. Yap] disent que la conception traditionnelle de l'AC-3 avec la pire complexité temporelle de pire des cas de (ed3) repose sur une implémentation naïve de la ligne 1 de la Figure 1.2, qui y est toujours recherché à partir de zéro. La nouvelle approche AC-3 dans le document de [Yuanlin Zhang et Roland H.C. Yap],est t'appelé AC-3.1,et elle est basée sur l'observation qui est dans la ligne 1 de la Figure1.1 qui ne doit pas être recherché à partir de zéro même si le même arc (i, j) peut entrer dans la liste Q plusieurs fois, la recherche est tout simplement repris à partir du point où il est arrêté dans la révision précédente

de (i, j). Cette idée est mise en œuvre par la procédure EXISTy ((i, x), j) sur la Figure 1.3. Ils supposent que chaque domaine Di est associé à un ordre total. Le Point Résume ((i, x), j) se souvient de la première valeur y appartenant à Dj tels que cij (x, y) occupe dans la précédente révision de (i, j). La fonction succ(y, D_j^0)), où D_j^0 désigne le domaine de j avant l'exécution de la consistance d'arc, retourne le successeur de y dans l'ordre des D_j^0 ou NIL si aucun élément existe.

Algorithm 1 AC-3

begin
1.$Q \leftarrow \{(i,j) \| c_{ij} \in C$ or $c_{ij} \in C,\ i \neq j\}$
whileQ not empty **do**
select and delete any arc (k, m) from Q ;
ifREVISE$((k, m))$ **then**
$Q \leftarrow Q \cup \{(i,k) \| (i,k) \in C, i \neq k, i \neq m$
endwhile end

Algorithm 2 REVISE

begin
$DELETE \leftarrow$ **false**
if there is no $y \in D_j$ such that $c_{ij}(x, y)$ **then**
delete x from D_i ;
DELETE $\leftarrow$ **true**
endif
returnDELTE
end

Algorithm 3 EXIST

begin
$y \leftarrow ResunePoint((i,x),j)$;
1.**if** $y \in D_j$ **then**
return true;
2.**while**$((y \leftarrow succ(y, D_j^0)\text{and}(y \neq NILL))$
if$(y \in D_j$ and $c_{ij}(x,y)$ **then**
ResumePoint$((i,x),j) \leftarrow y)$;
return true
endif;
return false
endif
end

5.2.2 Arc-Consistency and Arc-Consistency Again

Comme Mohr et Henderson ont souligné dans [Moh-Hen86], l'arc-consistance est basée sur la notion du support. Tant qu'une valeur a pour une variable i supportant les valeurs de chacune des autres variables j liées à i dans le graphe de contrainte. a est considéré comme étant une valeur viable pour i. Mais dès qu'il existe une variable dans laquelle aucune valeur restante ne satisfait la relation avec (i,a), alors a est éliminé de D_i. L'algorithme proposé dans l'article de [Moh-Hen86] a rendu ce support explicite en assignant un compteur $counter[(i,j),a]$ à chaque paire arc-valeur incluant l'arc (i,j) et la valeur a dans la variable i, ce compteur enregistre le nombre des supports de (i,a) dans D_j. Pour chaque valeur (j,b), l'ensemble Sjb est construit, où $S_{jb} = (i,a)/(j,b)supporte(i,a)$. Si (j,b) est éliminé de D_j, $counter[(i,j),a]$ doit être décrémenté pour chaque (i,a) dans S_{jb}. Cette structure de données est à l'origine de l'optimalité de l'AC- 4 en termes de temps. Mais le calcul du nombre de supports pour chaque valeur (i,a) implique un coût spatiale de complexité de $O(ed^2)$. Le but de l'AC-6 est alors d'éviter la vérification coûteuse des relations pour trouver tous les supports pour toutes les valeurs. AC-6 conserve le même principe que l'algorithme AC-4, mais au lieu de vérifier tous les supports pour une valeur, il vérifie uniquement un support (la première valeur) pour chaque valeur (i,a) sur chaque contrainte R_{ij},

La seule condition requise dans l'utilisation de l'AC-6 est d'avoir un ordre total dans tous les domaines D_j.

5.2.2.1 Algorithme AC6

L'algorithme proposé ici fonctionne avec les structures suivantes :

- Une table M de booléen garde la trace des valeurs du domaine initial qui sont dans le domaine courant ou non.
- Les fonctions temporelles constantes suivantes pour gérer les ensembles Di qui sont considérés comme des listes :
  - first(D_i) retourne la plus petite valeur dans D_i.
  - last (D_i) renvoie la plus grande valeur dans D_i.
  - next (a, D_i) renvoie la valeur a' dans Di telle que chaque valeur a'' plus grand que a et plus petit que a' est hors de $D_i(a < a'' < a')$
- $S_{jb} = (i, a)/(j, b)$ est la plus petite valeur dans D_j supportant (i, a) sur R_{ij}.
- AC-6 utilise la procédure nextsupport() pour trouver la plus petite valeur dans D_j pas plus petit que b et soutenir (i, a) sur R_{ij}.

Algorithm 4 AC6

{initialization} **for**$(i, a) \in D$ **do** $S_{ia} \leftarrow \varnothing$;M(i,a)$\leftarrow$ **true**
for(i,j)$\in$ $arcs(G)$ **do**
begin
if$D_j = \varnothing$
then*emptysupport* $\leftarrow$ **true**
else$b \leftarrow (D_j)$;
$nextsupport(i, j, a, b, emptysupport)$;
if*emptysupport*
thenD_{jj} a ;
$M(i, a) \leftarrow$ **false** ;
Append(List,(i,a))
; **else**Append$(S_{jb}, (i, a))$**end**

Pour chaque paire arc-valeur $[(i, j), a]$, chaque valeur D_j sera vérifié au plus une fois. Il existe des paires arc-valeur (ed), donc $O(ed^2)$ est la complexité dans le pire des cas pour AC-6, La matrice M a une taille proportionnelle $O(nd)$, la taille totale

{propagation} **while***list* $\neq \varnothing$**do**
begin
$choose(j,b) fromListandremove(j,b) fromList$;
for(i,a) $\in S_{jb}$**do**$\{beforeitsdeletion(j,b)wasthe$
begin$smallestsupportin D_j for(i,a) on R_j\}$
$remove(i,a) from \mathrm{S}_{jb}$;
ifM(i,a)**then**
begin
c; nextsupport(i, j, a, c, emptysupport) ;
if emptysupport
then $D_j \leftarrow D_i\ a$; $M(i,a) \leftarrow$ **false** ;
append(List, (i,a))
else Append($S_{jc,(i,a)}$)
end end end

des ensembles de S_{jb} est au plus égal au nombre de paires d'arc valeurs : $O(ed)$. Par conséquent, la complexité spatiale des AC-6 dans le cas le plus défavorable est $O(ed)$. Le problème de la complexité spatiale d'AC-4 est élaboré.

5.2.2.2 Using Parallel Singleton Arc Consistency to enhance Constraint Solving

Dans cet article l'auteur a divisé les méthodes de résolution parallèle des problèmes de satisfaction de contraintes (CSP) comme suit :

- La méthode de fractionnement qui consiste à répartir le travail (le problème) sur le nombre de processeurs disponible.
- La méthode de portefeuilles qui se base sur une concurrence dans la résolution du problème (CSP) et la communication par envoi de message.

Ensuite, ils proposent une nouvelle approche qui se base sur l'architecture maitre esclave et la communication par envoi de messages. Le plus important est que l'objectif est de minimiser le nombre de message de synchronisation entre les nœuds. Dans cette approche chaque nœud est doté de quelques astuces de données.

Le maitre contient deux structures de données qui sont la pile d'affectation et

la file d'attente des messages et de l'ensemble de décision déjà prise par les autres solveurs (esclave) :

— La pile d'affectation : emmagasine toutes les décisions positives ou négatives et chaque décision est accompagnée par un niveau temporaire au qu'elle appartient et qui est donné par une horloge globale. Cette horloge est initialisée à '0' et incrémenté à chaque prise d'une décision.

— ? La file d'attente des messages et l'ensemble des décisions : cette file contient les décisions déjà prise par les autres solveurs accompagnée du niveau est la valeur $\bar{\text{S}}$AC et une horloge et enfin la variable et la valeur instanciée. Si le message est d'actualité il sera déplacé à la file d'affectation.

Le coté esclave il contient 3 étapes de structure de données qui sont définies comme suit :

— L'esclave copie une instance du problème initiale qui est généré par le maitre.

— La liste de décision prise par le maitre.

— Une horloge pour pouvoir se synchroniser avec le maitre pour éviter de travailler sur une partie du (CSP) dépassé. Si la partie envoyée est dépassé l'esclave demande une réémission.

La file des littéraux contient des valeurs SAC pour les réutilisé plus tard, si la valeur est $\bar{\text{S}}$AC elle est mise dans la file de message pour l'envoyer aux autres pour l'éviter par la suite

5.3 Proposition d'un algorithme de filtrage parallèle

5.3.1 L'architecture utilisée

Dans cette proposition on suppose que le réseau est fiable et tous les nœuds sons fiable. On a opté pour une architecture maître-esclaves pour paralléliser les deux algorithmes de filtrage de CSP AC4 et AC6. La communication entre le maitre et les esclaves se fait via échange de messages.

5.3.2 L'algorithme parallèle AC4

L'algorithme AC4 est divisé en deux étapes dans son exécution, la première étape est l'initialisation et la seconde étape est la propagation. Avant de commencer la résolution par les solveurs. Le nœud maitre commence par une étape de répartition du problème initiale en sous-problèmes. Ensuite il envoi à tous les esclaves un sous problème.
Nous allons maintenant présenter les procédures ajoutées poru parallèliser AC-4

5.3.2.1 Le côté maitre

La procédure *Devise_CSP*() : un problème CSP est représenté par le triplet (X, D, C). L'objectif de cette procédure est de diviser ces trois domaines pour obtenir des sous-problèmes qui vont être répartie sur les processus esclave.

Algorithm 5 procedure *Devise_CSP*()

Nbr$_P$*artition* = $\|Nbr_V ariable/Nbr_p rocesseur\|$;
intk; **pour tout**x$_i \in X, c_{ij} \in C, v_i \in D(x_i)\{$
pour(k 1 ; k $\leqslant Nbr_P artition)Ajouter(touteslesvaleursde$x$_i)D'(x)$;
$Ajouter(x_i)X'$; **Pout tout**x$_j \in scop(c_i, c_{ij} \in C\{$
Ajouter(x_j)X';
$Ajouter(touteslesvaleursde$x$_j)D'(x)$;
$Ajouter(scop($c$_i$, Rel(c_i))C';
}
$K++$;
}
$Retourner(D', C', X')$;
$Envoyer(D', C', X')$;
}
}

La procédure *Reconstruire_Liste*() : le rôle de cette procédure est de reconstruire les lister Q, S et couter qui sont les structures de données les plus importante pour l'exécution de AC4

Algorithm 6 *Reconstruire_Liste*()

pour tous $Q'_i, S'_i, counter'_i$\{
Ajouter(Q'_i)à Q ;
Ajouter(S'_i)à S ;
Ajouter($counter'_i$)à counter ;
}
}

La procédure $Divise\_CSP2()$: est conçue dans l'objectif de décomposer les structures Q, S et couter de la première phase (phase d'initialisation) ensuite les envoyer à tous les esclaves pour faire la phase propagation.

$telque(\mathrm{x}_j, v_j) \in Q'$ {
Ajouter $(Sx_j, v_j])S'$;
pour $(x_i, vj \in S[x_j, v_j]$\{
Ajouter (counter$[x_i, v_i, xj]$) acounter ;
}
}
Envoyer2 (Q', S', Counte') à P
}
}

La procédure $Reconstruire\_Liste\_D()$: cette procédure a pour but de reconstruire les domaines $D(x_i)$ des variables des CSP à partir des résultats reçues via les nœuds esclaves.

Algorithm 8 *Reconstruire_Liste_D*()

pour tout D'_i {
ajouter (D'_i) à D ;
}
}

5.3.2.2 Le coté esclave

Dans le coté esclave on ajoute seulement les procédures de réception et d'émission. Les esclaves attendent les messages de leur maitre pour pouvoir commencer l'exécution.

5.3.2.3 Les messages utilisés

pour transmettre l'information entre les nœuds on utilise quatre types de messages.

- Le message 1 envoyé par le mètre et qui transporte les informations suivantes :
  - Le sous-ensemble X' : qui est un sous-ensemble de l'ensemble des variables X du problème principal (CSP principale).
  - Le sous ensemble C' : qui est le sous-ensemble des contraintes supporté par les variables $x'_i \in X'$ ce dernier est inclue dans l'ensemble C qui est l'ensemble des contraintes CSP.
  - Le sous-ensemble D'(x) : est un sous ensemble de D(x) qui est l'ensemble des valeurs des variables $x'_i \in X'$.
- Le message 2 envoyé par les esclaves et qui transporte les informations suivantes :
  - La liste Q'_i qui contient l'ensemble des (xi,vi) qui on été supprimer des domaines $D(x_i)$;
  - La liste $Counter'_i$ qui contient l'ensemble compteurs $Counter'_i[x_i, v_i, x_j]$;
  - La liste S'_i qui contient l'ensemble liste $S'_i[x_i, v_i]$.
- c) Le message 3 envoyé par le mètre et qui transporte les informations suivantes :
  - La liste Q'_i qui contient l'ensemble des (x_i, v_i) qui on été supprimer des domaines $D(x_i)$;
  - La liste $Counter'_i$ qui contient l'ensemble compteurs $Counter'_i[x_i, v_i, x_j]$;
  - La liste S'i qui contient l'ensemble liste S'i[xi,vi].
- d) Le message 4 envoyé par les esclaves et qui transporte les informations de la liste $D(x_i)$.

5.3.3 Lalgorithme parallèle AC6

Les procédures ajoutées à Ac6 pour le rendre parallèle

5.3.3.1 Le coté maitre

La procédure *Devise_CSP* : l'objectif de cette procédure est de diviser les trois domaines pour répartir les sous-problèmes sur les processeurs esclave.

Algorithm 9 procedure *Devise_CSP*()

Nbr$_P$*artition* = $\|Nbr_V ariable/Nbr_p rocesseur\|$;
intk; **pour tout**x$_i \in X, c_{ij} \in C, v_i \in D(x_i)\{$
pour(k 1 ; k $\leqslant Nbr_P artition)Ajouter(touteslesvaleursde$x$_i)D'(x)$;
$Ajouter(x_i)X'$; **Pout tout**x$_j \in scop(c_i, c_{ij} \in C\{$
Ajouter(x_j)X';
$Ajouter(touteslesvaleursde$x$_j)D'(x)$;
$Ajouter(scop($c$_i$, Rel(c_i))C';
}
$K++$;
}
$Retourner(D', C', X')$;
$Envoyer(D', C', X')$;
}
}

La procédure *Reconstruire_Liste* : son rôle est de reconstruire les listes Q, S, et counter qui sont les structures de données les plus importantes pour l'exécution

Algorithm 10 *Reconstruire_Liste*()

pour tous $Q'_i, S'_i, \{$
Ajouter(Q'_i)à Q ;
Ajouter(S'_i)à S ;

}
}

La procédure *Divise_CSP2* : qui permet de décomposer les structure Q, S et counter et les envoyer à tous les esclaves pour faire la phase de propagation.

telque(x$_j, v_j$) $\in Q'$ {
Ajouter $(Sx_j, v_j])S'$;
}
Envoyer2 (Q', S') à P
}
}

La procédure *Reconstruire_Liste_D* : qui permet de reconstruire les domaines D(xi) des variables des CSP à partir des résultats reçus via les nœuds esclaves.

Algorithm 12 *Reconstruire_Liste_D*()

pour tout D'_i {
ajouter (D'_i) à D ;
}
}

Le coté esclave [**?**] Dans le coté esclave on ajoute seulement les procédures de réception et d'émission. Les esclaves attendent les messages de leur maitre pour pouvoir s'exécuter. Ac6 est répartie en deux grandes parties ; la première est la partie initialisation et la seconde est la partie propagation.

5.3.4 Conclusion

Dans ce chapitre nous avons présenté quelques méthodes de filtrages existantes dans la littérature basé sur la consistance d'arc ainsi que une méthode parallèle de résolution des CSPs en suite nous avons présentés une proposition de parallélisassions des algorithmes de filtrage AC4 et AC6.

Conclusion et perspectives

Nos approches était principalement basé sur celle proposé dans [2] et [11], C'est-à-dire combiné une technique de compression qui utilise les tables fragmentées à une amélioration d'un algorithme de filtrage afin qu'il puisse gérer ces dernières.
Nous avons détaillé notre méthode de compression qui consiste en la recherche de motifs fréquents, sachant que nous proposons de recenser tous les motifs existant, on a donc développé un algorithme qui réussit à accomplir cette tâche, on a d'ailleurs pu le voir à l'œuvre sur deux exemples et les résultats sont concluant. Nous avons par la suite proposé une suite à cet algorithme pour qu'il puisse détecter les motifs qui seront utilisés lors de la compression, pour cela, et après de nombreuses expériences, on est arrivé à un certain nombre de critères qui aident dans la sélection des motifs, de sorte que, nous privilégions les motifs qui permettent une plus grande réduction grâce notamment à la notion de gains, et éviter la redondance de tuplses compressés.
Nous avons proposé un algorithme de filtrage parallèle qui consiste à réduire les tables obtenue dans la compression mais malheureusement nous n'avons pas pu terminer l'implémentation de ce dernier qui aurait pu améliorer la compression afin de proposé une approche complète qui allie compression et filtrage parallèle.
Nous n'avons malheureusement pas proposé d'algorithme de filtrage. Mais nous envisageons en tant que perspective de concentrer nos recherches là-dessus, afin de proposé une approche complète qui allie compression et filtrage.

Bibliographie

[1] M. Blaise. Algorithmes évolutionnaires et résolution de problèmes de satisfaction de contraintes en domaines finis. Automatique / Robotique. Université Nice Sophia Antipolis, 2002.

[2] N. Gharbi, F. Hemery, C. Lecoutre, and O. Roussel. Les contraintes table fragmentées : Combiner la compression et la réduction tabulaire. 10e Journées Francophones de Programmation par Contraintes (2014).

[3] N. Gharbi, F. Hemery, C. Lecoutre, and O. Roussel. Optimizing STR algorithms with tuple compression. In Proceedings of JFPC'13,(2013) 143–146.

[4] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In Proceedings of SIGMOD'00, (2000) 1–12.

[5] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation : A frequent-pattern tree approach. Data Mining and Knowledge Discovery, 8(1), (2004), 53–87.

[6] C. Lecoutre , C. Likitvivatanavong , Roland H.C. Yap. STR3 : A path-optimal filtering algorithm for table constraints. Artificial Intelligence 220 (2015) 1–27.

[7] C. Lecoutre. STR2 : optimized simple tabular reduction for table constraints. Constraints 16 (4) (2011) 341–371.

[8] J.B. Mairy1, Y. Deville1, C. Lecoutre. The Smart Table Constraint. 11e Journées Francophones de Programmation par Contraintes (2015).

[9] G. Perez, J.C. Régin, Relations entre MDDs et Tuples et Modifications dy-

namique de contraintes MDDs, 11e Journées Francophones de Programmation par Contraintes (2015).

[10] J.R. Ullmann. Partition search for non-binary constraint satisfaction. Inf. Sci. 177 (18) (2007) 3639–3678.