# On Compressing and Parallelizing Constraint Satisfaction Problems

By

Nebras GHARBI

A THESIS

submitted to

UNIVERSITY OF ARTOIS

in fulfilment of the requirements for the degree of

**Doctor of Philosophy**

of Artois University

**Department of Computer Science**

**Thesis committee:**

| | |
|---|---|
| Jean-Charles RÉGIN (University of Nice-Sophia Antipolis) | *Referee* |
| Xavier LORCA (École des Mines de Nantes) | *Referee* |
| Bertrand LE CUN (University of Paris-Ouest-Nanterre-La défense) | *Examiner* |
| Gilles DEQUEN (University of Picardie Jules Verne) | *Examiner* |
| Gilles AUDEMARD (University of Artois) | *Examiner* |
| Christophe LECOUTRE (University of Artois) | *Supervisor* |
| Fred HEMERY (University of Artois) | *Co-Supervisor* |
| Olivier ROUSSEL (University of Artois) | *Co-Supervisor* |

CENTRE DE RECHERCHE EN INFORMATIQUE DE LENS (CRIL CNRS UMR 8188)
**Université d'Artois, rue Jean Souvraz, S.P. 18 F-62307, Lens Cedex**
**France**

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Résumé Détaillé de la Thèse

## Introduction

La programmation par contraintes (CP) est un cadre puissant utilisé pour modéliser et résoudre des problèmes combinatoires, employant des techniques d'intelligence artificielle, de la recherche opérationnelle, de théorie des graphes, ...,etc. L'idée de base de la programmation par contraintes est que l'utilisateur exprime ses contraintes et qu'un solveur de contraintes cherche une ou plusieurs solutions.

Les problèmes de satisfaction de contraintes (CSP) [Montanari, 1974], sont au coeur de la programmation par contraintes. Ce sont des problèmes de décision où nous recherchons des états ou des objets satisfaisant un certain nombre de contraintes ou de critères. Ces problèmes de décision revoient *vrai*, si le problème admet une solution, *faux*, sinon. Les problèmes de satisfaction de contraintes sont le sujet de recherche intense tant en recherche opérationnelle qu'en intelligence artificielle. Beaucoup de CSPs exigent la combinaison d'heuristiques et de méthode d'inférences combinatoires pour les résoudre dans un temps raisonnable.

La résolution des CSPs peut rapidement devenir difficile quand le problème représente un volume important de données. Les solveurs sont composés de deux mécanismes importants : la "recherche" et "l'inférence". En effet, le mécanisme "d'inférence" se réfère à l'élaboration des déductions, tandis que, le mécanisme de "recherche" consiste à l'élaboration d'une solution en étendant itérativement une solution partielle. Pour former une solution, ou prouver qu'aucune solution n'existe, un solveur alterne entre la recherche et l'inférence. Il prend une nouvelle décision pour étendre une solution partielle et fait ensuite autant de déductions que possible. La déduction pourrait, par exemple, enlever toutes les valeurs qui ne peuvent plus apparaître dans une solution. Quand il n'y a plus de déduction qui pourrait être déduite de la dernière décision, le solveur prend une nouvelle décision. En fait, la décision est faite sur la base d'une heuristique donnée, c'est-à-dire, le solveur attribue une valeur à une variable choisie par l'heuristique qui supposent que ce choix puisse accélérer la recherche ou mener à une solution. Ce n'est pas le cas toujours, c'est pourquoi le solveur pourrait effectuer un retour en arrière jusqu'aux décisions précédentes, concluant que la dernière décision mène à un échec et, ainsi, aucune solution ne pourrait être trouvée à partir de cet endroit.

Avec l'amélioration des ordinateurs, la résolution de plus grands problèmes devient plus facile. Bien qu'il y ait plus de capacités offertes par la nouvelle génération de machines, les problèmes industriels deviennent de plus en plus grand ce qui implique un espace énorme pour les stocker et aussi plus de temps pour les résoudre. Durant ces dernières décennies, plusieurs travaux ont été proposés pour traiter ces problèmes.

D'une part, plusieurs approches ont été proposé pour réduire l'espace mémoire requis pour représenter les contraintes et, particulièrement les contraintes table qui sont des contraintes exprimés en extension. De différentes approches utilisent des structures de données compactes pour représenter des contraintes table comme des Tries [Gent et al., 2007], des Diagrammes de Décision multi-valués (MDDs) [Cheng and Yap, 2010] et des Automates Finis Déterministes (DFA) [Pesant, 2004]. D'autres ont proposé des nouvelles représentations compactes comme les tuples compressés [Hubbe and Freuder, 1992], des short supports [Nightingale et al., 2011, Nightingale et al., 2013], les smart tuples [Mairy et al., 2015] et des approches basées sur la fouille de données [Jabbour et al., 2013a, Jabbour et al., 2013b].

D'autre part, plusieurs travaux ont proposé des approches différentes pour l'accélération du temps de résolution à travers l'utilisation des architectures parallèles. Nous distinguons des techniques différentes comme le work sharing [Schulte, 2000, Régin et al., 2013, Régin et al., 2014], le work stealing [Sleep, 1981, Kotthoff and Moore, 2010, Kotthoff and Moore, 2010, Chu et al., 2009], la Recherche Multi-agent [Rao and Kumar, 1988, Bordeaux et al., 2009] et l'utilisation des Portfolios [Gomes and Selman, 2001, O'Mahony et al., 2008, Amadini et al., 2015, Dasygenis and Stergiou, 2014].

Dans cette thèse, nous nous intéressons aux problèmes de satisfaction de contraintes et en particulier les différentes techniques utilisées dans la réduction de l'espace mémoire requis pour la représentation des contraintes et aussi le temps de résolution. Dans le **premier chapitre**, nous introduisons les notions de bases de l'état de l'art utilisés dans ce manuscrit. nous détaillons aussi les différentes techniques utilisées dans la recherche et l'inférence. Le **deuxième chapitre** se compose principalement de deux parties. Dans la première partie, nous décrivons les différentes méthodes de l'état de l'art utilisées dans la compression des contrainte table. Tandis que la deuxième partie, décrit l'état de l'art de la programmation parallèle ainsi que les différents travaux utilisant une architecture parallèle dans la résolution des CSPs. Dans le **troisième chapitre**, nous présentons deux contributions autour la compression des contraintes table. Pour chacune des contributions, nous décrivons la méthode de compression ainsi que l'algorithme de filtrage. Dans le **quatrième chapitre**, nous présentons une contribution qui consiste à résoudre une instance CSP en établissant des cohérences en parallèle. En effet, nous explorons une nouvelle manière d'utilisation d'une architecture parallèle dans laquelle un solveur principale est aidé par différent esclaves qui établissent des cohérences fortes qui sont couteuse pour être maintenue durant la recherche. Finalement, nous concluons et nous présentons les perspectives de nos travaux dans le **dernier chapitre**.

## Préliminaires

Un *réseau de contraintes* (discret) (CN) $N$ est un ensemble fini de $n$ variables reliées par un ensemble fini de $e$ contraintes. Chaque *variable* $x$ a un *domaine* qui représente l'ensemble fini de valeurs qui peuvent être assignées à $x$. Le domaine *initial* d'une variable $x$ est noté $dom^{init}(x)$ tandis que le domaine *courant* de $x$ est noté $dom(x)$. Nous avons toujours $dom(x) \subseteq dom^{init}(x)$. Chaque *contrainte* $c$ porte sur un ensemble ordonné de variables, appelé *portée (scope)* de $c$ et noté $scp(c)$, et est sémantiquement défini par une *relation*, notée $rel(c)$, qui contient l'ensemble des tuples autorisés pour les variables impliquées dans $c$. Une *contrainte table* (positive) $c$ est une contrainte telle que $rel(c)$ est définie explicitement en énumérant les tuples qui sont autorisés par $c$ (voir exemple ci-dessous). L'*arité* d'une contrainte $c$ est la taille de $scp(c)$. L'arité maximale du réseau sera notée $r$.

**Exemple 1** *Soit $c$ une contrainte table positive portant sur les variables $x_1, x_2, x_3, x_4, x_5$ telles que $dom(x_1) = dom(x_2) = dom(x_3) = dom(x_4) = dom(x_5) = \{a, b, c\}$. La table 1 liste les 7 tuples autorisés par la contrainte $c$.*

| | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|---|
| $\tau_1$ | (c, | b, | c, | a, | c) |
| $\tau_2$ | (a, | a, | b, | c, | a) |
| $\tau_3$ | (a, | c, | b, | c, | a) |
| $\tau_4$ | (b, | a, | c, | b, | c) |
| $\tau_5$ | (b, | a, | a, | b, | b) |
| $\tau_6$ | (c, | c, | b, | c, | a) |
| $\tau_7$ | (a, | c, | a, | c, | a) |

Table 1: Une contrainte table $c$ portant sur $x_1, x_2, x_3, x_4, x_5$.

Soit $X = \{x_1, \ldots, x_r\}$ un ensemble ordonné de variables. Une *instanciation* $I$ de $X$ est un ensemble $\{(x_1, a_1), \ldots, (x_r, a_r)\}$ qui est également noté $\{x_1 = a_1, \ldots, x_r = a_r\}$ tel que $\forall i \in 1..r, a_i \in dom^{init}(x_i)$. $X$ est notée $vars(I)$ et chaque $a_i$ est notée $I[x_i]$. Un *littéral* est une paire $(x, a_i)$ où $x \in \mathscr{X}$ et $a_i \in dom(x)$. Une instanciation $I$ est *valide ssi* $\forall (x, a) \in I, a \in dom(x)$. Un *$r$-tuple* $\tau$ sur $X$ est une suite de valeurs $(a_1, \ldots, a_r)$ telle que $\forall i \in 1..r, a_i \in dom^{init}(x_i)$; la valeur $a_i$ sera notée $\tau[x_i]$. Un tuple défini sur un ensemble $X$ peut être vu comme une instanciation de $X$ et inversement. De ce fait, un $r$-tuple $\tau$ sur $scp(c)$ est *valide ssi* l'instanciation sous-jacente est valide. Un $r$-tuple $\tau$ sur $scp(c)$ est un *support* sur la contrainte $r$-aire $c$ *ssi* $\tau$ est un tuple valide qui est autorisé par $c$. Si $\tau$ est un support sur une contrainte $c$ impliquant une variable $x$ et tel que $\tau[x] = a$, on dit que $\tau$ est un *support pour* $(x, a)$ sur $c$. GAC est une cohérence de domaine définie comme suit :

**Définition 1** *Une contrainte $c$ satisfait la* cohérence d'arc généralisée *(GAC) ssi $\forall x \in scp(c), \forall a \in dom(x)$, il existe au moins un support pour $(x, a)$ sur $c$. Un CN $N$ est GAC ssi chaque contrainte de $N$ est GAC.*

L'application de GAC implique la suppression de toutes les valeurs qui n'ont pas de support sur une contrainte. De nombreux algorithmes ont été mis au point pour établir GAC selon la nature des contraintes. STR [Ullmann, 2007] est l'un de ces algorithmes pour les contraintes table : il supprime les tuples invalides lors de la recherche de supports en utilisant une structure de données qui sépare les tuples valides des tuples invalides. Cette méthode de recherche des supports améliore le temps de recherche en évitant les tests redondants sur des tuples invalides qui ont déjà été détectés comme invalides lors des précédents application de GAC. STR2 [Lecoutre, 2011], une optimisation de STR, évite certaines opérations de base concernant la validité de tuples et l'identification des supports, par l'introduction de deux ensembles importants appelés $S^{sup}$ et $S^{val}$ (décrits ultérieurement). Dans le meilleur des cas, STR2 est $r$ fois plus rapide que STR.

## La compression des contraintes tables

### STR$^c$

La première approche, appelée (STR$^c$), consiste à combiner l'algorithme STR avec un algorithme de compression basé sur les "tries", qui diffère des approches décrites dans [Katsirelos and Walsh, 2007] et [Xia and Yap, 2013] où une représentation sur la base des produits cartésiens est utilisée pour la compression. L'idée de base est d'identifier les motifs récurrents, appelés sous tuples, dans le tuples de chaque contrainte et remplacer leurs occurrences par des références vers une table de motifs. Le processus de filtrage est une adaptation de l'algorithme STR qui prend en considération les motifs apparaissant dans des positions différentes.

**Méthode de compression** Un **motif** $\mu$ d'une contrainte table $c$ est une séquence de valeurs consécutives (sous-tuples) dans un tuple $\tau$ d'une table. On note $|\mu|$ la longueur d'un motif $\mu$, et $nbOcc(\mu)$ le nombre d'occurrences du motif repérées dans l'ensemble des tuples d'une contrainte donnée.

Afin de réduire la complexité spatiale de la représentation des tuples de chaque contrainte, nous allons 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 table sera d'autant plus faible que la longueur des motifs sera grande et que leur nombre d'occurrences sera important.

Il est important de noter que nous considérons que les motifs extraits dans le cadre de notre approche sont indépendants de leur position initiale dans le tuple. En conséquence, un motif ne correspond pas obligatoirement à l'affectation des mêmes valeurs aux mêmes variables mais plutôt la même suite d'affectation à une séquence de variables consécutives. Ce choix a été fait dans l'espoir d'obtenir des motifs les plus fréquents possibles, et donc une meilleure compression.

Pour identifier les motifs pertinents, nous allons dans un premier temps 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. Pour garantir un certain niveau d'efficacité de compression, la longueur minimale des séquences est fixée dans notre approche à 3, et la longueur maximale à l'arité de la contrainte moins 1.

Dans un second temps, il nous faut identifier les motifs les plus efficaces pour le processus de compression. Pour cela nous allons introduire la notion de **score** d'un motif $\mu$ comme suit :

$$\text{score}(\mu) = \text{—}\mu\text{—} \times \text{nbOcc}(\mu)$$

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.

Le processus de compression utilise donc les motifs dont le score est supérieur au seuil de sélection. 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.

La figure 3.4 illustre la structure de données compressée. En fait, une fois qu'un motif est trouvé, il est remplacé par une référence vers une table de motifs où les motifs fréquents sont stockés. La référence est codée par un entier négatif (*-patternid*). En fait, le tuple ne contient pas les valeurs composant le motif, mais plutôt une référence vers le motif.

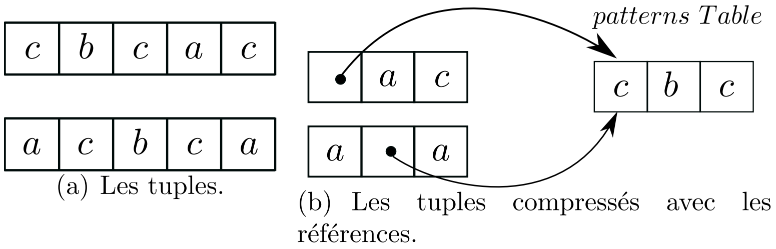


Figure 1: Compression des tuples.

L'algorithme 24 décrit la méthode de compression. D'abord, pour chaque tuple de la contrainte table nous cherchons le meilleur motif contenu dans *FP-Queue*. En cherchant le meilleur motif, nous respectons un ordre décroissant des scores. Si un motif est trouvé, nous l'ajoutons à la table de motifs et remplaçons ensuite son occurrence dans le tuple par une référence vers celui-ci dans la table des motifs. Ce processus continue jusqu'à il n'y a plus de motif possible qui pourrait être remplacé dans le tuple. Nous notons qu'en cherchant le meilleur motif contenu dans un tuple, nous prenons en considération les remplacements fait auparavant. En conséquence, les motifs ne se chevauchent pas puisque nous choisissons de remplacer toujours l'occurrence du meilleur. Une fois que nous avons itéré sur toutes les valeurs d'un tuple nous vérifions si nous avons déjà remplacé au moins un motif. Si c'est le cas, le tuple $\tau_i$ de la contrainte est remplacé par sa version compressée $\tau_i^c$. La table 3.1 présente une contrainte table positive d'arité 5, impliquant les variables $x_1, x_2, x_3, x_4$ et $x_5$. nous notons que plusieurs motifs sont répétés sur l'ensemble des tuples tels que *cbc*, *aab* et *abb* comme des motifs de taille 3 et *aabb*, *acbc* et *cbca* comme des motifs de taille 4. En itérant sur l'ensemble des tuples de notre contrainte, on peut construire le trie décrit dans la figure 3.5, où le nombre des occurrences est donné au niveau de chaque noeud. Chaque noeud représente un chemin $\mu$ (allant de la racine à une feuille) associé avec le compteur $nbOcc(\mu)$.

Le mécanisme de compression engendre une réduction importante de l'espace mémoire occupé par la contrainte table. La vue physique décrite dans la figure 3.8 illustre mieux ce phénomène. Les tuples compressés font référence à leur motifs correspondant stockés dans la tables des motifs. La contrainte table est donc composée de tuples de différentes longueurs puisqu'ils contiennent des motifs de différentes tailles.

**Algorithm 1:** compress-STR$^c$($c$: Constraint,$FP$-$Queue$: Queue)

```
1  foreach $\tau_i \in table(c)$ do
2      repeat
           // searching for the best frequent pattern contained in $\tau_i$
3          $bestPattern \leftarrow \prec\succ$
4          foreach $\mu_i \in FP\text{-}Queue$ do
5              if $contains(\mu_i,\tau_i)$ then
6                  $bestPattern \leftarrow \mu_i$
7                  $bestPattern.position \leftarrow$ index($\mu_i$,$\tau_i$)
8                  break
9          if $bestPattern \neq nil$ then
               // Compressing each tuple $\tau_i$ of $c$: $\tau_i^c$ is the
                  prospective compressed tuple which is encoded as
                  an array of maximal length $|scp(c)|$. Its length is
                  denoted $length^c$
10             $position \leftarrow 0$
11             $length^c \leftarrow 0$
12             while $position < \tau_i.length$ do
13                 if $position{=}bestPattern.position$ then
14                     $\tau_i^c[length^c] \leftarrow$ -$bestPattern.id$ // a reference
                       towards the pattern
15                     $position \leftarrow position + bestPattern.length$
16                 else
17                     $\tau_i^c[length^c] \leftarrow \tau_i[position]$
18                     $position \leftarrow position + 1$
19                 $length^c \leftarrow length^c + 1$
20     until $bestPattern = nil$
21     if $length^c < \tau_i.length$ then
22         $\tau_i \leftarrow \tau_i^c$
```

**Algorithme de filtrage** Lors du filtrage d'une contrainte table, il est nécessaire de vérifier la validité des tuples, ce qui implique de vérifier la validité des motifs. Quand un motif apparaît plusieurs fois dans la table à partir de la même position, nous souhaitons n'effectuer le test de validité du motif qu'une seule fois, ce qui permet de traduire la compression spatiale en une réduction du temps de filtrage.

Pour ce faire, nous utilisons un compteur *time* qui est incrémenté à chaque filtrage de la contrainte table et un tableau $stamps[\mu_i, j]$ associé à la contrainte.

|  | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|---|
| $\Rightarrow \tau_1$ | (c, | b, | c, | a, | c) |
| $\tau_2$ | (a, | a, | b, | b, | a) |
| $\tau_3$ | (a, | c, | b, | c, | a) |
| $\tau_4$ | (b, | a, | c, | b, | c) |
| $\tau_5$ | (b, | a, | a, | b, | b) |
| $\tau_6$ | (a, | c, | b, | c, | b) |
| $\tau_7$ | (a, | c, | a, | c, | a) |

Table 2: La contrainte table $C_{x_1,x_2,x_3,x_4,x_5}$.

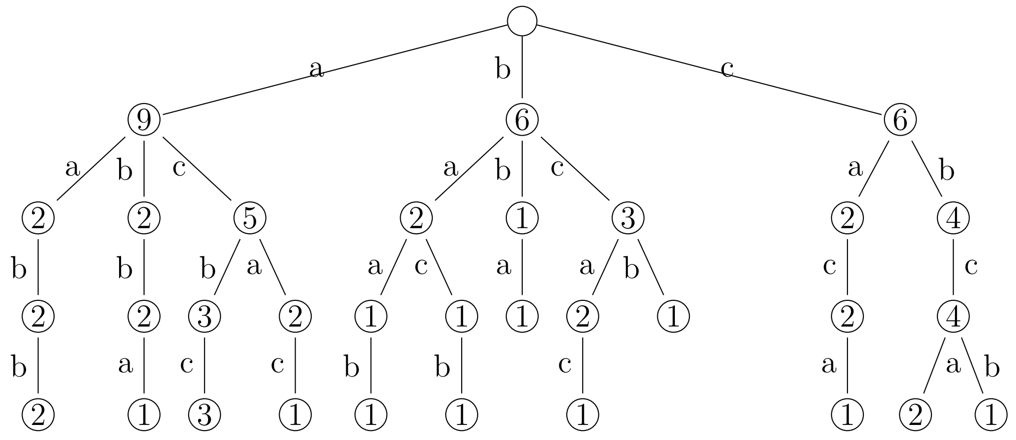
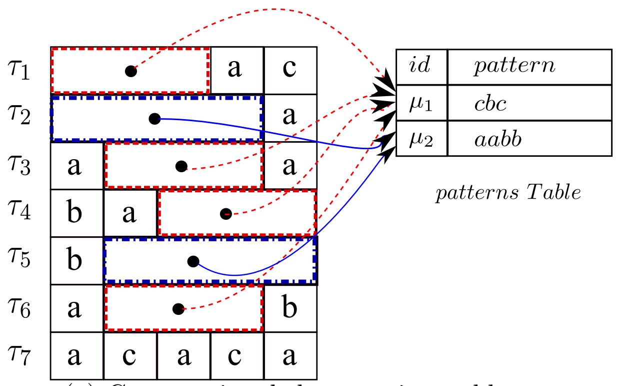


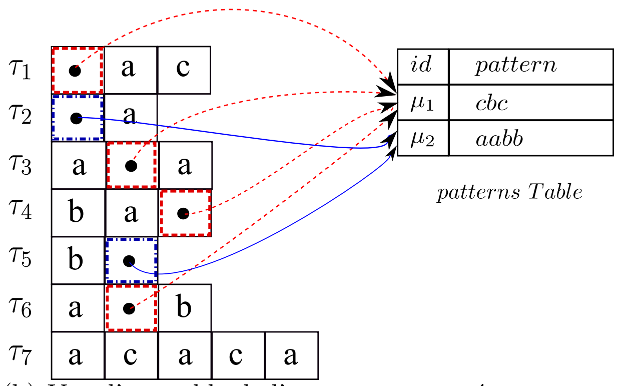
Figure 2: Le trie des motifs construits à partir de la contrainte décrite dans la table 3.1.

Pour un motif $\mu_i$ qui s'applique à partir d'une position $j$, $stamps[\mu_i, j]$ donne le résultat du dernier test de validité de $(\mu_i, j)$ (champ $stamps[\mu_i, j].valid$) ainsi que la valeur $stamps[\mu_i, j].time$ du compteur $time$ lors de ce test. Chaque fois que la validité de $(\mu_i, j)$ doit être testée, nous vérifions d'abord si $stamps[\mu_i, j].time$ est égal à la valeur courante de $time$. Si c'est le cas, la validité a déjà été testée dans l'opération de filtrage courante et donc $stamps[\mu_i, j].valid$ fournit directement la réponse, ce qui évite des calculs inutiles. Sinon, il faut effectivement tester la validité de $(\mu_i, j)$ et sauvegarder le résultat dans $stamps[\mu_i, j]$.

La figure 3.10 illustre l'évolution de la structure `stamps`. D'abord, tous les éléments sont initialisés à `currentTime`=0 et les champs `valid` reste vide (voir figure 3.10(a)). Durant le premier filtrage de la table, le compteur global est mis à 1. Par exemple, pour déterminer si le tuples $\tau_3$ est valid, il faut s'assurer que $\mu_1$ à la position $j = 2$ est valide. Comme la valeur `time` du champs `stamps`[$\mu_1$,2] n'est pas égal à la valeur courante du `currentTime`, nous constatons que le test



(a) Compression de la contrainte table en utilisant des références vers la table des motifs.



(b) Vue d'ensemble de l'espace compressée.

Figure 3: La vue physique de la table compressée.

| *id* \ *position* | 1 | 2 | 3 |
|---|---|---|---|
| $\mu_1$ | $time = 0$ $valid = ?$ | $time = 0$ $valid = ?$ | $time = 0$ $valid = ?$ |
| $\mu_2$ | $time = 0$ $valid = ?$ | $time = 0$ $valid = ?$ | $time = 0$ $valid = ?$ |

(a) Initialisation (`currentTime`=0).

| *id* \ *position* | 1 | 2 | 3 |
|---|---|---|---|
| $\mu_1$ | $time = 1$ $valid = F$ | $time = 1$ $valid = T$ | $time = 1$ $valid = F$ |
| $\mu_2$ | $time = 1$ $valid = T$ | $time = 1$ $valid = T$ | $time = 0$ $valid = ?$ |

(b) À la fin de l'itération `currentTime`=1.

Figure 4: Évolution de la structure `stamps`.

n'est pas encore fait dans le filtrage courant. Par conséquent, nous vérifions si $c$, $b$ et $c$ sont encore présents dans les domaines respectifs des variables $x_2$, $x_3$ et $x_4$. Nous supposons que c'est le cas et donc les champs `time` et `valid` de `stamps`[$\mu_1$,2] prennent les valeurs 1 et *true*. Plus tard, quand nous testons la validité du tuple $\tau_6$, nous nous apercevons immédiatement que la validité du motif $\mu_1$ a été déjà vérifié dans le filtrage courant et ça suffit donc d'utiliser le résultat enregistré dans le champs `valid` de `stamps`[$\mu_1$,2]. Supposons que $\mu_1$ est invalide à la position 1 et 3, valide à la position 2 et que $\mu_2$ est valide à la position 1 et 2, nous obtenons à la fin du filtrage le résultat illustré dans la figure 3.10(b).

**Résultats expérimentaux** Pour montrer le potentiel de notre approche (STR$^c$), nous avons comparé le comportement des algorithmes MDD, STR1, STR2, STR3 et STR$^c$ lorsqu'ils sont intégrés à l'algorithme de recherche MAC (qui maintient la propriété de cohérence d'arc généralisée lors d'une recherche arborescente). Nous avons effectué quelques tests sur des instances de différents problèmes distincts : Les mdds introduites dans [Cheng and Yap, 2010], les nonogrammes [Pesant et al., 2012], les Bdds, les mots croisés et les problèmes

| Instance | MDD | STR1 | STR2 | STR3 | STR$^c$ |
|---|---|---|---|---|---|
| *a7-v24-d5-ps0.5-psh0.7-9* | **17.819** | 879 | 334 | 367 | 780.318 (43.775% – 12.192) |
| *a7-v24-d5-ps0.5-psh0.9-2* | **5.536** | 256 | 145 | 143 | 283.828 (43.776% – 11.511) |
| *bdd-21-2713-15-79-9* | 77.171 | 80.2 | **23.3** | 60.0 | 84.608 (82,612% – 1.05) |
| *bdd-21-2713-15-79-11* | 55.752 | 78.5 | **23.5** | 48.5 | 74.796 (82,643% – 1.164) |
| *crossword-m1-ogd-23-04* | 104.485 | 82.7 | **78.2** | 103 | 80.085 (10.688% – 2.919) |
| *crossword-m1c-lex-vg5-7-* | 86.475 | 43.5 | **31.4** | 36.6 | 49.393 (20.134% – 0.5) |
| *nonogram-gp-108* | 55.732 | 290 | **78.7** | 118 | 319.613 (94.519% – 20.561) |
| *nonogram-gp-116* | 16.658 | 102 | **21.7** | 21.9 | 108.098 (95.7% – 7.948) |
| *rand-6-10-10-60-950-0* | 67.821 | 75.1 | 45.4 | **34.4** | 104.554 (34.317% – 21.768) |
| *rand-7-9-9-30-980-0* | 47.285 | 26.8 | **18.5** | 46.3 | 50.271 (57.469% – 19.383) |

Table 3: Le temps CPU (en secondes) pour quelques instances résolues avec MAC. Les ratios et le temps CPU de compression pour STR$^c$ sont donnés entre parenthèses.

aléatoires. Les résultats figurent en table 3.2 ; l'heuristique dom/ddeg est utilisée pour garantir le même parcours d'arbre (dom/wdeg est plus versatile).

L'algorithme STR$^c$ permet une économie spatiale d'au moins 50% par rapport à STR1 et STR2, et jusqu'à un facteur 4 par rapport à STR3. Nous décrivons le temps CPU ainsi que le ratio de compression et le temps de compression. Lorsqu'on considère le temps de recherche, il apparaît que STR$^c$ rivalise avec STR1, mais reste toutefois supplanté par STR2.

## STR-*slice*: Les contraintes table fragmentées

Dans la première approche, nous avons défini un motif comme un séquence de valeurs consécutives. Bien que cette définition nous ait permis d'obtenir une réduction importante de la complexité spatiale, la forme compressée de la contrainte table présente quelques inconvénients particulièrement pendant le processus de filtrage nous empêchant d'utiliser les variantes optimisées de STR. Pour chaque test de validité d'un tuple, il est nécessaire de réitérer sur toutes ses valeurs (si nous ne sommes pas dans le cas de réutiliser les résultats précédemment faits et enregistrés). Ces inconvénients sont dus à la définition du motif : un motif pourrait impliquer des variables différentes selon sa position dans le tuple.

Pour pouvoir profiter de variantes STR optimisées, nous avons proposé dans cette seconde approche une nouvelle définition du motif et, ainsi, une nouvelle forme compressée des contraintes table à l'intermédiaire des technique de fouilles de données.

Un **motif** $\mu$ d'une contrainte $c$ est une instanciation $I$ d'un sous ensemble de variables de $c$. On note $scp(\mu)$ sa portée, qui est $vars(I)$, $|\mu|$ sa longueur, qui est égale à $|scp(\mu)|$, et $nbOcc(\mu)$ son nombre d'occurrences dans $rel(c)$, qui est $|\{\tau \in rel(c) \mid \mu \subseteq \tau\}|$.

Une **sous-table** $T$ associée à un motif $\mu$ d'une contrainte $c$ est la table obtenue en ne conservant que les tuples de $c$ qui contiennent $\mu$ et en effaçant $\mu$ dans chacun de ces tuples.

$$T = \{\tau \setminus \mu \mid \tau \in rel(c) \wedge \mu \subseteq \tau\}$$

La portée de $T$ est $scp(T) = scp(c) - scp(\mu)$

Un **fragment** d'une contrainte $c$ est un couple $(\mu, T)$ tel que $\mu$ est un motif de la contrainte $c$ et $T$ est la sous-table associée au motif $\mu$.

Comme l'ensemble des tuples représentés par un fragment $(\mu, T)$ représente en fait le produit cartésien de $\mu$ par $T$, nous allons également utiliser la notation $\mu \otimes T$ pour désigner un fragment d'une contrainte. Après le processus de fragmentation d'une contrainte, l'ensemble des tuples qui ne sont associés à aucun motif sont regroupés dans un *fragment par défaut* noté $(\emptyset, T)$

Le motif $\mu$=$(x_1 = a, x_4 = c, x_5 = a)$ de la contrainte $c$, détecté en figure 3.16(a), apparaît dans les tuples $\tau_2$, $\tau_3$ et $\tau_7$. Donc le fragment résultant est composé du motif $\mu$ et sa sous-table correspondante extraite de $c$, comme décrit en figure 3.16(b).

| | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|---|
| $\tau_1$ | (c, | b, | c, | a, | c) |
| $\tau_2$ | (**a**, | a, | b, | **c**, | **a**) |
| $\tau_3$ | (**a**, | c, | b, | **c**, | **a**) |
| $\tau_4$ | (b, | a, | c, | b, | c) |
| $\tau_5$ | (b, | a, | a, | b, | b) |
| $\tau_6$ | (c, | c, | b, | c, | a) |
| $\tau_7$ | (**a**, | c, | a, | **c**, | **a**) |

(a) Une contrainte $c$.

| $x_1$ | $x_4$ | $x_5$ |
|---|---|---|
| $a$ | $c$ | $a$ |

$\otimes$

| $x_2$ | $x_3$ | |
|---|---|---|
| $a$ | $b$ | $\tau_2$ |
| $c$ | $b$ | $\tau_3$ |
| $c$ | $a$ | $\tau_7$ |

(b) Un fragment $(\mu, T)$ de $c$.

Figure 5: Un exemple de fragment de contrainte.

Le test de validité sur les tuples (classiques ou bien compressés) est une opération très importante dans les algorithmes de filtrage pour les contraintes table. Pour les contraintes table fragmentée, nous étendons la notion de validité à un fragment d'une contrainte.

Un fragment $(\mu, T)$ est *valide ssi* il existe au moins un tuple du produit cartésien $\mu \otimes T$ qui soit valide. De ce fait, un fragment est valide *ssi* son motif est valide et sa *sous-table* correspondante contient au moins un *sous-tuple* valide.

**Méthode de compression** Plusieurs algorithmes de fouille de données, tels que Apriori [Agrawal and Srikant, 1994]et FP -Growth [Han et al., 2000], peuvent être utilisés pour identifier les motifs les plus fréquents. Dans le cadre de notre approche, nous n'avons pas besoin d'identifier chaque motif fréquent possible mais seulement ceux qui sont utiles pour la compression, et en particulier au plus un motif par tuple. La construction d'un *FP-Tree* (*Frequent-Pattern Tree*) qui est la première étape dans l'algorithme *FP-Growth* est particulièrement bien adapté à cet objectif car elle identifie chaque motif long et fréquent. Cette construction ne nécessite que trois parcours de la table de la contrainte.

Nous expliquons brièvement la construction d'un *FP-Tree* dans notre contexte de compression de table, en utilisant la contrainte donnée par la table 3.16(a). L'algorithme prend comme paramètre *minSupport* qui est le nombre minimal d'occurrences d'un motif pour qu'il soit considéré comme fréquent. Dans notre exemple, nous allons utiliser *minSupport*= 2 pour identifier les motifs qui apparaîssent au moins deux fois.

Dans une première étape, nous collectons le nombre d'occurrences de chaque valeur. Par abus de langage, nous appellerons *fréquence* le nombre d'occurrences d'une valeur. Cette étape nécessite un parcours de la table. Le résultat sur notre exemple est donné par la figure 6(a). Ensuite, lors d'un deuxième parcours, nous trions l'ensemble des tuples par ordre décroissant de fréquence des valeurs. Le résultat est donné par la figure 6(b) où la fréquence d'une valeur est donnée entre parenthèses. Les valeurs qui ont une fréquence en dessous du seuil *minSupport* sont retirées du tuple (elles sont identifiés en caractères gras) parce qu'ils ne peuvent pas apparaître dans un motif fréquent. Une fois le tuple trié et éventuellement réduit, il est 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. La figure 7(a) représente le FP-Tree obtenu sur notre exemple. Le premier tuple inséré dans l'arbre est le début de $\tau_1$ qui est $(x_1 = c, x_3 = c, x_5 = c)$. Cela crée la branche la plus à gauche de l'arbre. Chaque nœud de cette branche a initialement une fréquence de 1. Le deuxième tuple inséré est $(x_4 = c, x_5 = a, x_1 = a, x_2 = a, x_3 = b)$ qui crée la troisième branche à gauche dans l'arborescence (chaque nœud ayant une fréquence de 1 à cette étape). Lorsque $\tau_3$ est insérée, la nouvelle branche $(x_4 = c, x_5 = a, x_1 = a, x_2 = c, x_3 = b)$ partage ses trois premières arêtes avec la dernière branche. De ce fait, les fréquences des nœuds correspondants sont incrémentées à 2. Les autres tuples sont insérés de la même façon. Finalement, les nœuds ayant une fréquence en dessous du seuil *minSupport* sont supprimés. L'arbre restant est représenté avec des lignes épaisses et entouré par une ligne en pointillés sur la figure 7(a). Il faut maintenant identifier les motifs du FP-tree qui sont utiles pour la compression. Chaque nœud de l'arbre correspond à un motif fréquent $\mu$ qui peut être lu sur

|   | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|---|
| a | 3 | 3 | 2 | 1 | 4 |
| b | 2 | 1 | 3 | 2 | 1 |
| c | 2 | 3 | 2 | 4 | 2 |

(a) Les fréquences.

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| $\tau_1$ | (2) $x_1 = c$ | (2) $x_3 = c$ | (2) $x_5 = c$ | **(1)** $\mathbf{x_2 = b}$ | **(1)** $\mathbf{x_4 = a}$ |
| $\tau_2$ | (4) $x_4 = c$ | (4) $x_5 = a$ | (3) $x_1 = a$ | (3) $x_2 = a$ | (3) $x_3 = b$ |
| $\tau_3$ | (4) $x_4 = c$ | (4) $x_5 = a$ | (3) $x_1 = a$ | (3) $x_2 = c$ | (3) $x_3 = b$ |
| $\tau_4$ | (3) $x_2 = a$ | (2) $x_1 = b$ | (2) $x_3 = c$ | (2) $x_4 = b$ | (2) $x_5 = c$ |
| $\tau_5$ | (3) $x_2 = a$ | (2) $x_1 = b$ | (2) $x_3 = a$ | (2) $x_4 = b$ | **(1)** $\mathbf{x_5 = b}$ |
| $\tau_6$ | (4) $x_4 = c$ | (4) $x_5 = a$ | (3) $x_2 = c$ | (3) $x_3 = b$ | (2) $x_1 = c$ |
| $\tau_7$ | (4) $x_4 = c$ | (4) $x_5 = a$ | (3) $x_1 = a$ | (3) $x_2 = c$ | (2) $x_3 = a$ |

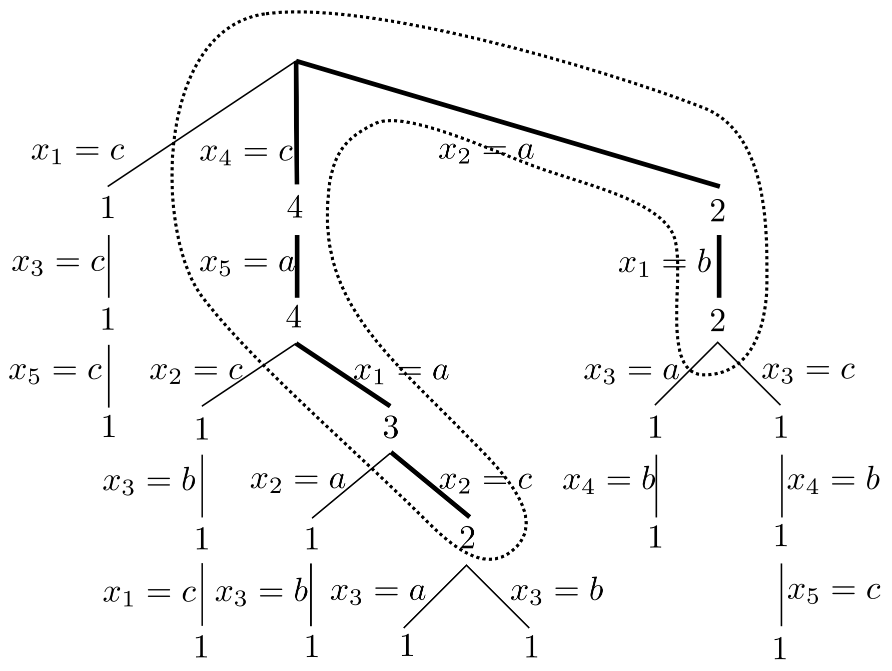
(b) Les tuples triées selon un ordre décroissant de fréquence.

Figure 6: Les deux première étapes de compression.

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 la factorisation de ce motif fréquent est $|\mu| \times (f - 1)$ valeurs (on peut supprimer toutes les occurrences du motif, sauf une). Dans notre exemple, pour la première branche en gras, nous pouvons voir qu'en utilisant le motif $(x_4 = c, x_5 = a)$, nous gagnons six valeurs, avec le motif $(x_4 = c, x_5 = a, x_1 = a)$ nous gagnons également 6 valeurs tandis qu'on ne gagne que 4 valeurs avec la branche complète $(x_4 = c, x_5 = a, x_1 = a, x_2 = c)$. Par conséquent, nous réduisons l'arbre en supprimant les nœuds qui offrent un gain moindre que leur père. Les feuilles de l'arbre obtenu représentent les motifs fréquents utilisé dans notre compression : $(x_4 = c, x_5 = a, x_1 = a)$ et $(x_2 = a, x_1 = b)$. Pour terminer, nous créons un fragment pour chaque motif fréquent que nous avons identifié. Ces fragments seront remplis lors d'un dernier parcours de la contrainte. Pour chaque tuple, nous utilisons le FP-tree pour identifier si le tuple (trié) commence par un motif fréquent. Dans ce cas, nous ajoutons le reste du tuple à la *sous-table* correspondante. Les tuples qui ne commencent pas par un motif fréquent sont ajoutés au fragment par défaut.

L'algorithme 27 résume les différentes étapes du processus de compression.

**Algorithme de filtrage** Afin d'appliquer GAC sur les contraintes table fragmentées, notre idée est d'adapter la technique (STR), et plus précisément la variante STR2 optimisée. Comme une contrainte table fragmentée est composée de plusieurs fragments, chacune composée d'un motif et d'une *sous-table*, le processus de filtrage que nous proposons agit à deux niveaux distincts. Au niveau



(a) L'arbre des péfixes fréquents (FP-tree).

| $x_1$ | $x_4$ | $x_5$ |
|---|---|---|
| $a$ | $c$ | $a$ |

$\bigotimes$

| $x_2$ | $x_3$ | |
|---|---|---|
| $a$ | $b$ | $\tau_2$ |
| $c$ | $b$ | $\tau_3$ |
| $c$ | $a$ | $\tau_7$ |

| $x_1$ | $x_2$ |
|---|---|
| $b$ | $a$ |

$\bigotimes$

| $x_3$ | $x_4$ | $x_5$ | |
|---|---|---|---|
| $c$ | $b$ | $c$ | $\tau_4$ |
| $a$ | $b$ | $b$ | $\tau_5$ |

$\varnothing \bigotimes$

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | |
|---|---|---|---|---|---|
| $c$ | $b$ | $c$ | $a$ | $c$ | $\tau_1$ |
| $c$ | $c$ | $b$ | $c$ | $a$ | $\tau_6$ |

(b) Table compressée.

Figure 7: FP-tree et la table compressée.

haut, la validité de chaque fragment est déterminée, et au niveau bas, la validité de chaque couple (motif, sous-tuple) est vérifiée. Rappelons qu'un fragment est valide *ssi* à la fois son motif est valide et au moins un sous-tuple de la sous-table est valide (voir la définition ).

Une contrainte table fragmentée $c$ est représentée par un tableau `entries`$[c]$

**Algorithm 2:** compress-STR-slice($c$: Constraint, $minSupport$: float)

```
   // Initialization of $frequencies$
1  foreach $i \in length(frequencies)$ do
2      $frequencies[i] \leftarrow 0$
   // First scan:compute the frequency of each value of $c$
3  foreach $\tau \in table(c)$ do
4      foreach $v \in \tau$ do
5          $frequencies[v]{+}{+}$
   // Second scan:build FP-Tree
6  foreach $\tau \in table(c)$ do
       // sorting $\tau$ and removing values less frequent than
          $minSupport$
7      $\tau \leftarrow$ sortTuple ($\tau$, $frequencies$, $minSupport$)
       // insert $\tau$ into FP-Tree by updating nodes frequency
8      addTuple($root(FP\text{-}Tree)$,$\tau$)
   // removing nodes less frequent than $minSupport$ or such that
      $|\mu| \times (f-1)$ is smaller than for their parents
9  pruneTree($root(FP\text{-}Tree)$,$minSupport$,0)
   // Third scan:compress $c$
10 foreach $\tau \in table(c)$ do
11     $\mu \leftarrow$ searchPattern($root(FP\text{-}Tree)$,$\tau$)
12     if $\mu \neq nil$ then
13         $entry \leftarrow$ getEntry($entries[c]$,$\mu$)
14         if $entry = nil$ then
               // There is no entry for this pattern
15             $entry \leftarrow$ addEntry($entries[c]$,$\mu$)
16         addSubTuple($entry$,$\tau \setminus \mu$)
17     else
18         addSubTuple($defaultEntry$,$\tau$)
```

de $p$ fragments. La gestion des fragments valides, appelés fragments *courants* [1], est effectuée comme suit :

- `entriesLimit`$[c]$ est l'indice du dernier fragment courant dans `entries`$[c]$. Les éléments de `entries`$[c]$ aux indices allant de 1 à `entriesLimit`$[c]$ sont les fragments courants de $c$.

[1] Les fragments courants correspondent aux fragments valides obtenus comme résultat du dernier appel de l'algorithme.

- la suppression d'un fragment (qui est devenu invalide) à l'indice $i$ est faite par l'appel à la fonction removeEntry$(c, i)$. Cet appel effectue une permutation entre les fragments d'indice $i$ et `entriesLimit`$[c]$, puis décrémente `entriesLimit`$[c]$. Notez que l'ordre initial de fragments n'est pas conservé.

- La restauration d'un ensemble de fragments se fait simplement en changeant la valeur de `entriesLimit`$[c]$.

Chaque fragment de `entries` peut être représenté comme un enregistrement composé d'un champ `pattern` et un champ `subtable`. Plus précisément :

- le champ `pattern` enregistre une instanciation partielle $\mu$, et peut être représenté en pratique comme une structure composée de deux tableaux : un pour les variables, la portée du motif, et l'autre pour les valeurs.

- la structure `subtable` stocke une sous-table $T$ et peut être représentée en pratique comme une structure composée de deux tableaux : un pour les variables, à savoir la portée de la sous-table $T$, et d'autre part, un tableau à deux dimensions, pour les sous-tuples.

Dans notre présentation, nous allons traiter directement $\mu$ et $T$ sans tenir compte de tous les détails d'implantation. Par exemple, $T$ sera considéré comme un tableau à deux dimensions. La gestion de l'ensemble des sous-tuples valides, appelés sous-tuples *courants* de $T$, est réalisée comme suit :

- *limit*$[T]$ est l'indice du dernier sous-tuple courant dans $T$. Les éléments de $T$ aux indices allant de 1 à `limit`$[T]$ sont les sous-tuples courants de $T$.

- la suppression d'un sous-tuple (qui est devenu invalide) à l'indice $i$ est effectuée par un appel à la fonction removeSubtuple$(T, i)$. Cet appel effectue une permutation entre les sous-tuples à indices $i$ et `limit`$[T]$, puis décrémente `limit`$[T]$. Notez que l'ordre initial de sous-tuples n'est pas conservé.

- la restauration d'un ensemble de sous-tuples se fait simplement en changeant la valeur de `limit`$[T]$.

Notez que la gestion des fragments courants et sous-tuples courants utilise les mêmes principes que ceux de STR. En outre, comme dans [Lecoutre, 2011], nous utilisons deux ensembles de variables, appelés $S^{val}$ et $S^{sup}$. L'ensemble $S^{val}$ contient des variables non affectées (et éventuellement, la dernière variable assignée) dont les domaines ont été réduits depuis le dernier appel de l'algorithme de filtrage sur $c$. Pour configurer $S^{val}$, nous avons besoin d'enregistrer la taille du domaine de chaque variable $x$ juste après l'exécution de STR-slice sur $c$ : cette valeur est enregistrée dans `lastSize`$[x]$. L'ensemble $S^{sup}$ contient des variables non affectées de la contrainte $c$ dont les domaines contiennent chacun au moins une valeur pour laquelle un support doit être trouvé. Ces deux ensembles permettent de restreindre les itérations sur les variables à celles qui sont uniquement pertinentes. Nous utilisons également un tableau `gacValues`$[x]$ pour chaque variable $x$. A tout moment, `gacValues`$[x]$ contient toutes les valeurs de $dom(x)$ pour lesquelles un support a déjà été trouvé : par conséquent, les valeurs d'une variable $x$ pour lesquelles on n'a pas trouvé de support sont exactement ceux de $dom(x) \setminus$ `gacValues`$[x]$. Notez que les ensembles $S^{val}$ and $S^{sup}$ sont initialement définis par rapport à la portée de $c$. Cependant, pour chaque sous-table, nous aussi allons utiliser des ensembles locaux $S^{lval}$ et $S^{lsup}$ de $S^{val}$ et $S^{sup}$ comme expliqué plus loin.

L'algorithme 30 est une procédure de filtrage, appelé STR-slice, qui établit GAC sur une contrainte table fragmentée $c$ appartenant à un CN $N$.

**Résultats expérimentaux** Dans la table 3.6, nous comparons les comportements respectifs de STR1, STR2, STR3, MDD et STR-*slice* sur différentes instances [2] impliquant des contraintes table positives d'arité supérieure à 2. Nous utilisons MAC avec *dom/ddeg* comme choix d'ordonnancement des variables et *lexico* comme heuristique de choix de valeur, pour résoudre tous ces problèmes. Une observation générale de cette expérimentation préliminaire est que STR-slice concurrence STR2 et STR3, sans toutefois l'emporter clairement.

## L'utilisation de la La cohérence d'arc singleton en parallèle pour l'amélioration de la recherche

Notre approche est basée sur une architecture de maître/esclaves où le maître est un solveur CSP séquentiel et les différents esclaves aident leur maître durant le processus de la recherche. Ce solveur principal transmet son instantiation courante à ses esclaves qui essayeront de déduire des informations pertinentes en exploitant des niveaux différents de cohérence. Dès que de nouveaux faits sont découverts, ils sont transmis au solveur principal qui les prend en compte dès que possible. Notre but est de minimiser la synchronisation entre le solveur principal et les esclaves. Dans notre approche, le solveur principal et les différents esclaves utilisent des cœurs différents du même hôte.

La cohérence d'arc singleton (SAC) [Debruyne and Bessiere, 1997] est une cohérence forte trop lourde à établir pendant la recherche [Lecoutre and Prosser, 2006]. Par contre, dans une solveur parallèle les cœurs disponible peuvent effectuer ces tests SAC gratuitement ( du point de vue wall-clock). Les littéraux que l'on découvre SAC incohérent sont transmis au maître pour les éviter dans ses décisions futures.

[2]disponibles sur `http://www.cril.univ-artois.fr/CSC09`.

**Algorithm 3:** STR-slice($c$: constraint):set of variables

```
   // Initialization of sets $S^{val}$ and $S^{sup}$, as in STR2
1  $S^{val} \leftarrow \emptyset$
2  $S^{sup} \leftarrow \emptyset$
3  if lastPast($P$) $\in scp(c)$ then
4      $S^{val} \leftarrow S^{val} \cup \{$lastPast($P$)$\}$
5  foreach variable $x \in scp(c) \mid x \notin$ past($P$) do
6      gacValues[$x$] $\leftarrow \emptyset$
7      $S^{sup} \leftarrow S^{sup} \cup \{x\}$
8      if $|dom(x)| \neq$ lastSize[$c$][$x$] then
9          $S^{val} \leftarrow S^{val} \cup \{x\}$
10         lastSize[$c$][$x$] $\leftarrow |dom(x)|$
   // Iteration over all entries of $c$
11 $i \leftarrow 1$
12 while $i \leq$ entriesLimit[$c$] do
13     $(\mu, T) \leftarrow$ entries[$c$][$i$]        // $ith$ current entry of $c$
14     if isValidPattern($\mu$) and then scanSubtable($T$) then
15         foreach variable $x \in scp(\mu) \mid x \in S^{sup}$ do
16             if $\mu[x] \notin$ gacValues[$x$] then
17                 gacValues[$x$] $\leftarrow$ gacValues[$x$] $\cup \{\mu[x]\}$
18                 if $|dom(x)| = |$gacValues[$x$]$|$ then
19                     $S^{sup} \leftarrow S^{sup} \setminus \{x\}$
20         $i \leftarrow i + 1$
21     else
22         removeEntry($c, i$)        // entriesLimit[$c$] decremented
   // domains are now updated and $X_{evt}$ computed, as in STR2
23 $X_{evt} \leftarrow \emptyset$
24 foreach variable $x \in S^{sup}$ do
25     $dom(x) \leftarrow gacValues[x]$
26     if $dom(x) = \emptyset$ then
27         throw INCONSISTENCY
28     $X_{evt} \leftarrow X_{evt} \cup \{x\}$
29     lastSize[$c$][$x$] $\leftarrow |dom(x)|$
30 return $X_{evt}$
```

La figure 4.1 décrit l'architecture de notre approche. En fait, chaque maître et esclaves ont leur propre copie du problème. Le maître résout le problème tandis

| Instance | STR1 | STR2 | STR3 | MDD | STR-slice |
|---|---|---|---|---|---|
| *a7-v24-d5-ps0.5-psh0.7-9* | 879 | 334 | 367 | **25.5** | 200 (69% – 5.41) |
| *a7-v24-d5-ps0.5-psh0.9-6* | 353 | 195 | 324 | **16.6** | 174 (62% – 5.82) |
| *bdd-21-2713-15-79-11* | 78.5 | **23.5** | 48.5 | 82.6 | 31.7 (88.05% – 0.28) |
| *crossword-ogd-vg12-13* | 799 | 342 | **208** | $> 1,200$ | 242 (73.46% – 0.74) |
| *crossword-uk-vg10-13* | 1,173 | **576** | 589 | $> 1,200$ | 598 (89.63% – 0.48) |

Table 4: Le temps CPU sur quelques instances.

que les esclaves établissent SAC sur quelques littéraux du problème. Une *pile de décisions* est associée à chaque maître et esclave. Le maître stocke toutes les décisions positives et négatives faites dans sa propre *pile de décisions* tandis que les esclaves les copient seulement dans leurs piles, chaque fois qu'il y a de nouvelles décisions. Cette pile décrit l'état du problème de chaque entité. Tous les esclaves testent la cohérence des littéraux qui sont stockés dans la *file de littéraux* qui est une structure de données partagée entre tous les acteurs. Établir la cohérence SAC sur ces littéraux produit de nouveaux faits, de nouveaux messages sont alors transférés dans la *file de messages* qui est une structure de données partagée entre le maître et les esclaves. Le maître à son tour extrait des messages de la *file de messages* pour les exploiter dans son processus de résolution en évitant ainsi des échecs.

**La pile de décisions** La pile de décision stocke toutes les décisions faites par le solveur. Elles peuvent être positives (assignation) ou négatives (réfutation). Chaque décision positive définit un niveau auquel correspond un horodatage qui indique l'instant ou cette décision a été prise. Cet horodatage est obtenu par un compteur global qui est initialisé à 0 et incrémenté chaque fois qu'une décision est prise. Ces informations sont utilisées pour identifier les modifications de la pile de décision. En fait, un horodatage est seulement associé aux décisions positives qui est une caractéristique du solveur AbsCon [Lecoutre and Tabary, 2007]. Des décisions positives sont stockées dans la pile dans un ordre consécutif. Quand un retour arrière est effectué, la valeur causant l'échec est supprimée du domaine de sa variable et cette réfutation est ajoutée au niveau précédent.

Les esclaves obtiennent initialement une copie $\mathscr{P}'$ du problème $\mathscr{P}$ qui est traité par le maître (des variables, des domaines, des contraintes). Pour atteindre le même état de recherche que le maître, les esclaves peuvent soit copier le nouvel état du problème ou obtenir seulement l'ensemble des décisions faites jusqu'ici et les ré-appliquer de nouveau. En fait, dans une itération, ils copient d'une manière incrémentale la pile de décisions du maître (c'est-à-dire la liste de décisions prises

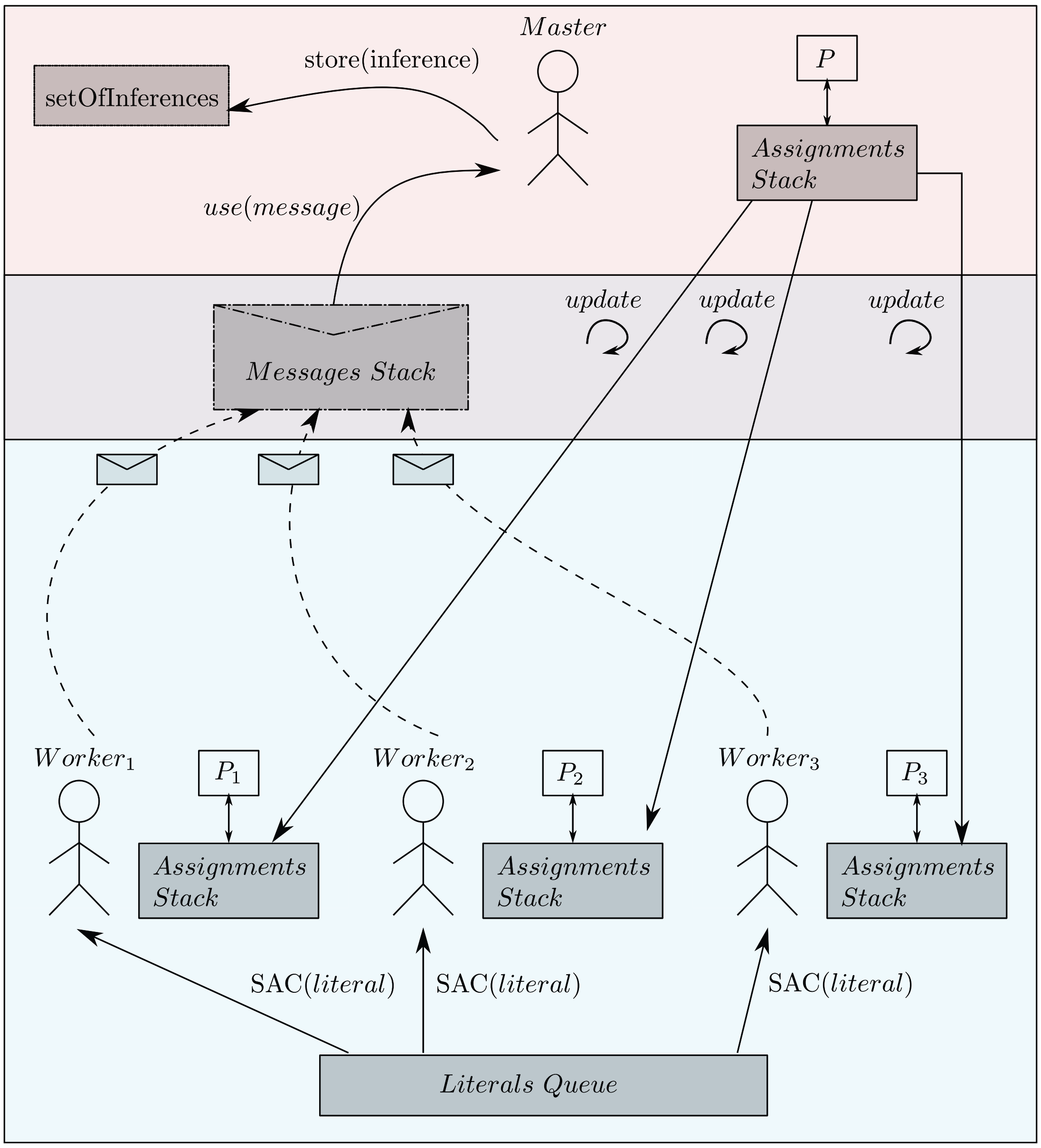


Figure 8: Vue d'ensemble de l'architecture.

par le maître), reproduisent le filtrage fait par le maître après ces décisions pour atteindre le même état que le maître et exécutent ensuite leur propre cohérence. La copie progressive de la pile de décisions consiste, d'abord, à identifier la partie de la pile qui est identique dans le maître et dans la copie de l'esclave et ensuite la reproduction de chaque décision prise par le maître après cette partie commune. . L'esclave obtient d'abord l'indicateur de pile actuel du maître et identifie ensuite la dernière décision qui a le même horodatage que le maître. Alors, l'esclave copie chaque décision du maître à partir de cette dernière décision commune. Puisque le maître peut effectuer un retour arrière entre temps, l'esclave vérifie alors que la dernière décision qu'il a copiée ait toujours le même horodatage que dans le maître. Si ce n'est pas le cas, le maître a fait un retour arrière pendant la copie et l'esclave reprend sa copie de *la pile de décisions*.

**La file de messages** Une fois qu'un esclave a une copie stable de l'état du maître, il choisit un nouveau littéral $(x, a)$ dans la file globale de littéraux et vérifie si le littéral $(x, a)$ est SAC-cohérent. Si ce n'est pas le cas, l'esclave produit un message contenant l'information $x \neq a$ et son niveau de décision actuel et un horodatage où ces informations ont été déduites. Le message est stocké dans la *file de Messages*. Le maître extraira ces informations quand il est prêt à les l'utiliser. Après que le test SAC a été effectué sur un littéral $(x, a)$, il est remis à la file de littéraux, en mettant à jour sa priorité, pour tester à nouveau sa cohérence plus tard. L'algorithme 36 illustre l'établissement de la cohérence SAC sur l'ensemble de littéraux de la file de littéraux.

**Algorithm 4:** enforceConsistencyOnLiterals($P_i$: Constraint Network of slave $i$ ,*LiteralsQueue*: Queue of literals)

```
1  $valueFound \leftarrow false$
2  while $\neg valueFound$ do
3      $literal \leftarrow getLiteral(literalsQueue)$
4      if $\neg assigned(literal.var)$ and $|dom(literal.var)| > 1$ then
5          $valueFound \leftarrow true$
6      else
7          $literal.countdown$ - -
8          insert $literal$ to $literalsQueue$
   // while the problem is not solved, there is always a value to
   test
9  assign $literal.val$ to $literal.var$
10 $consistent \leftarrow GAC(P, literal.var)$
11 $backtrack()$
   // going back to the previous level in order to test more
   pairs
12 if $\neg consistent$ then
13     $literal.priority{+}{+}$
14     $putMessage(currentLevel, timeStamp[level], literal)$
       // Putting the inferred result in the queue of messages
15 $literal.countdown$ - -
16 insert $literal$ to $literalsQueue$
```

Avant qu'une décision ne soit prise, le maître vérifie si quelques informations ont été déduites par les autres esclaves. Ceci est fait en vérifiant le contenu de *la file de messages*, qui est lue par le maître et écrite par les esclaves. Quand un message est extrait de cette file, le maître vérifie, en premier lieu, si l'inférence est toujours cohérente par rapport à l'état actuel du solveur. Si le message n'est pas cohérente, il est ignoré. Autrement, l'inférence est prise en compte.

En fait, un message contient deux informations principales :

- *le littéral* : la paire (variable, valeur) pour éviter dans des assignations futures;

- (*niveau*, *horodatage*) : le niveau de la paire (dans l'arbre de recherche) et son horodatage dans lequel la propriété "le littéral est SAC-incohérent" est toujours pertinente.

Les algorithmes 34 et 35 décrivent la gestion des messages par le maître. L'algorithme 35 illustre le processus d'extraction des messages par le maître avant chaque décision prise. La validité du message est retournée par la fonction 34. Une fois que le maître vérifie que le message est toujours pertinent, il le stocke dans sa propre structure de données, appelée *setOfInferences*. En fait, *setOfInferences* est une table où *setOfInferences[i]* définit les messages extraits et utilisés au niveau $i$. Grâce à cette structure de données, le maître peut réutiliser ces inférences dans des niveaux plus hauts, quand un retour arrière est effectué.

**Algorithm 5:** isValidMessage(*msg*: Message): Boolean

```
1 if $msg.level \leq currentLevel$ and
  $msg.timeStamp = timeStamp[msg.level]$ then
2     return $true$        // the message $msg$ is still relevant
3 return $false$
```

**La file de littéraux** Pour exécuter les différents tests SAC, les littéraux du problème sont regroupés dans une *file de Littéraux*. Les différents esclaves coopèrent pour examiner chaque littéral possible aussi souvent que possible. Puisque quelques littéraux vont plus probablement devenir SAC-incohérent, nous utilisons des priorités pour tester ces littéraux plus souvent que les autres. Donc, pour chaque littéral dans la *file de Littéraux* corresponds une priorité, qui est incrémentée quand le littéral est identifié comme SAC-incohérent (voir la ligne 13 de l'algorithme 36). Pour gérer cette priorité nous utilisons l'algorithme Completely Fair Scheduling (CFS) [Li et al., 2009] qui assure l'exécution du processus ayant utilisé le minimum de temps en premier lieu et donc les processus sont ordonnés en fonction du temps d'exécution utilisé.

**Algorithm 6:** applyMessages(*hasBacktracked*: Boolean)

```
1  if hasBacktracked then
2      foreach $i \in$ [currentLevel+1,levelBeforeBacktrack] do
3          foreach msg : setOfInferences[i] do
4              if isValidMessage(msg) then
5                  remove msg.literal.val from dom(lmsg.iteral.var)
6                  add msg to setOfInferences[currentLevel]
7          clear setOfInferences[i]
                       // removing the inferences stored at level i
8  foreach msg : msgQueue do
9      if isValidMessage(msg) then
10         remove msg.literal.val from dom(msg.literal.var)
11         add msg to setOfInferences[currentLevel]
12     remove msg
```

**Amélioration de la recherche** L'algorithme 37 décrit l'algorithme de recherche utilisé par le maître. En fait, nous utilisons l'algorithme de recherche classique MAC. En plus, on intègre l'extraction de message (ligne 4): un appel à l'algorithme 35 est effectué pour pouvoir tirer profit des faits découverts par l'ensemble des esclaves. De cette manière, les valeurs SAC-incohérents sont supprimés de leurs domaines respectifs en vue de les éviter dans les prochaines décisions.

**Résultats expérimentaux** Les tables 4.2 et 4.4 illustrent les temps moyens (en wall clock) pour quelques séries d'instances en utilisant respectivement les heuristiques *dom/wdeg* et *dom/ddeg*.

Contrairement à nos attentes, notre approche semble être moins efficace que l'algorithme de recherche le plus utilisé MAC. Ceci pourrait être dû au fait que les valeurs incohérentes sont, en fait, facilement déduites et par la suite supprimées par le maître par la propagation. Ceci implique que les informations envoyées ne sont pas aussi fortes que nos attentes pour pouvoir accélérer la recherche de maître. L'heuristique de choix de variable pourrait être aussi la raison derrière l'échec de notre approche. En fait, une analyse approfondie montre que même dans une résolution séquentielle établissant une cohérence forte gratuitement ne peut pas impliquer une réduction importante de l'espace de recherche ce qui explique les résultats obtenus par notre approche.

**Algorithm 7:** solveProblem($P$: Constraint network)

```
1  $finished \leftarrow false$
2  $hasBacktracked \leftarrow false$
3  while $\neg finished$ do
                                  // infer all not used messages
4      $applyMessages(hasBacktracked)$
                       // select new pair (variable,value) to assign
5      $var \leftarrow getNextVariable(variableOrderingHeuristic)$
6      $val \leftarrow getNextValue(valueOrderingHeuristic, var)$
7      assign $var$ to $val$
8      $levelBeforeBacktrack \leftarrow |past(P)|$
9      $consistent \leftarrow checkConsistencyAfterAssignment(P)$
10     if $consistent$ and $|past(P)| = n$ then
11         display $solution$
12         $finished \leftarrow true$
13     else
14         while $\neg consistent$ do
15             $backtrack()$
16             $hasBacktracked \leftarrow true$
17             remove $val$ from $dom(var)$
18             $consistent \leftarrow checkConsistencyAfterRefutation(P)$
```

| Series | #inst | MAC | MSAC | SAC+MAC | Par(7) |
|---|---|---|---|---|---|
| langford | 17 | **90.55** | 905.32 | 391.14 | 408.16 |
| queen | 7 | 785.85 | 4143 | 2382 | **235.42** |
| quennsKnight | 22 | 1473 | **162.39** | 165.97 | 1264 |
| rand-8-20-5-18-800 | 10 | **45.91** | 517.68 | 1124 | 93.20 |
| fapp25-2230 | 12 | 46.95 | 620.83 | 48.80 | **41.93** |
| ewddr2-10-by-5 | 10 | **1.545** | 6.589 | 2.336 | 3.026 |
| cc | 13 | **2.772** | 19.62 | 2.837 | 13.46 |
| BlackHole-4-4-e | 10 | **0.458** | 7.858 | 0.485 | 1.110 |

Table 5: Le temps wall claock moyen pour des séries d'instances en utilisant l'heuristique dom/wdeg.

| Series | #inst | MAC | MSAC | SAC+MAC | Par(7) |
|---|---|---|---|---|---|
| langford | 17 | 77.828 | 830,090 | **76.995** | 397.311 |
| queen | 7 | 845.799 | 4,659.207 | 24,443.690 | **319.756** |
| queensKnight | 22 | 3,446.051 | **160.044** | 166.904 | 1,475.197 |
| rand-8-20-5-18-800 | 10 | **20.221** | 204.801 | 22.192 | 34.837 |
| fapp25-2230 | 12 | 46.246 | 619.829 | 49.241 | **43.191** |
| ewddr2-10-by-5 | 10 | **1.518** | 6.888 | 2.362 | 3.046 |
| cc | 10 | 53.004 | 67.663 | 54.017 | **13.422** |
| BlackHole-4-4-e | 10 | **0.493** | 11.354 | 0.498 | 1.156 |

Table 6: Le temps wall claock moyen pour des séries d'instances en utilisant l'heuristique dom/ddeg.

# Conclusion

Cette thèse s'articule autour des techniques d'optimisation de la résolution des CSPs en raisonnant sur plusieurs axes.

Dans la première partie, nous traitons la compression des contraintes table. Vu l'expressivité offerte par ces contraintes aux utilisateurs non experts, son utilisation est courante dans le monde industriel. Au fil du temps, le volume des données manipulées ne cesse d'augmenter ce qui implique l'utilisation de structures de données compactes pour les stocker, et aussi des algorithmes de filtrage optimisés pour les gérer pendant le processus de résolution. Dans cette thèse, nous proposons deux méthodes différentes pour la compression des contraintes de table. Les deux approches sont basés sur la recherche des motifs fréquents pour éviter la redondance. Cependant, la façon de définir un motif, la détection des motifs fréquents et la nouvelle représentation compacte diffère significativement. Les résultats expérimentaux montrent que l'approche $STR^c$ permet d'avoir une compression importante, mais comparée aux variantes STR il rivalise seulement avec $STR1$. Quant à STR-*slice*, les expérimentations montrent des résultats plus compétitifs où STR-*slice* supplante $STR2$ et parfois $STR3$ malgré qu'elle est pénalisée parfois par le coût de compression.

La seconde partie est consacrée à une autre façon d'optimiser la résolution de CSP qui est l'utilisation d'une architecture parallèle. Nous proposons une méthode où nous utilisons une architecture parallèle pour améliorer le processus de résolution en établissant des cohérences parallèles. En fait, les esclaves envoient à leur maître le résultat obtenu après avoir établi la cohérence partielle en tant que nouveaux faits. Le maître, à son tour essaye de profiter d'eux en enlevant les valeurs correspondantes. Contrairement à ce qu'on pensait, le travail fait par les esclaves n'implique pas d'amélioration significative du processus de résolution.

# Introduction

Constraint Programming (CP) is a powerful paradigm used for modeling and solving combinatorial constraint problems that relies on a wide range of techniques coming from artificial intelligence, operational research, graph theory, . . . , etc. The basic idea of constraint programming is that the user expresses its constraints and a constraint solver seeks a solution. A constraint expresses a restriction on the combinations of possible values assigned to variables. A constraint network is defined by a set of variables, each one with its own domain, and by a set of constraints. CP can be used for the most of our daily problems such as Scheduling problems. For example, in order to make a school schedule, we should take into consideration various constraints such as professors and classrooms availability. In the Information Technology (IT) field, "*Constraint programming is an advance that IT has ever done that is closest to the Holy Grail of programming: the user defines the problem, the computer solves it.*" Eugene C. Freuder.

Constraint satisfaction problems (CSP) [Montanari, 1974], is a framework at the heart of CP problems. They correspond to decision problems where we seek for states or objects satisfying a number of constraints or criteria. These decision problems have two answers to the question they encode: $true$, if the problem admits a solution, $false$, otherwise. CSPs are the subject of intense research in both artificial intelligence and operations research. Many CSPs require the combination of heuristics and combinatorial optimization methods to solve them in a reasonable time.

Solving CSP instances can quickly become difficult when the order of problem instances grow. Solvers are composed of two important components: "search" and "inference". In fact, the "inference" mechanism refers to making deductions, whereas, the "search" mechanism consists in building a solution by extending iteratively a partial solution. In order to build a solution, or to prove that no solution exists, a solver alternates between search and inference. It makes a new decision to extend a partial solution and then makes as many deductions as possible. Deductions could be, for example, removing all values that could not appear in a solution. When there are no more deductions that could be inferred from the last decision, the solver makes a new decision. In fact, the decision is made with respect to a given heuristic, i.e., the solver assigns a value to a variable chosen by the heuristic supposing that this choice could speed up the search or lead to a solution. This is not always the case, that is the reason why the solver could return back to the previous decisions, concluding that the last decision leads to a failure and, thus, no solution could be found from there.

With the improvement of computers, larger and larger problems can be solved. However, the size of industrial problems grow faster which requires a vast amount of memory space to store them and entail great difficulties to solve them. In the late decades, different works were proposed in order to deal with large problems.

On the one hand, several approaches were proposed in order to reduce the memory space required to represent constraints and, in particular, table constraints which are an extensional form of constraints. Different approaches use compact data structures to represent table constraints such as Tries [Gent et al., 2007], Multi-valued Decision Diagrams (MDDs) [Cheng and Yap, 2010] and Deterministic Finite Automata (DFA) [Pesant, 2004]. Others proposed new compact representations such as Compressed tuples [Hubbe and Freuder, 1992], Short supports [Nightingale et al., 2011, Nightingale et al., 2013], Smart tuples [Mairy et al., 2015] and Data-mining based approaches [Jabbour et al., 2013a, Jabbour et al., 2013b].

On the other hand, different approaches were proposed in order to speed-up the resolution time. We distinguish the use of parallel computing. The problem is then solved using different cores. We distinguish different techniques such as Work Sharing [Schulte, 2000, Régin et al., 2013, Régin et al., 2014], Work Stealing [Sleep, 1981, Kotthoff and Moore, 2010, Kotthoff and Moore, 2010, Chu et al., 2009], Multi-Agent Search [Rao and Kumar, 1988, Bordeaux et al., 2009] and Portfolios [Gomes and Selman, 2001, O'Mahony et al., 2008, Amadini et al., 2015, Dasygenis and Stergiou, 2014].

## Outline

In this thesis, we focus on *Constraint Satisfaction Problems* and, in particular, the different techniques used to reduce both the required memory space and the resolution time. The thesis is organized as follows:

- In **Chapter 1**, we introduce the background for constraint satisfaction problems (CSP) that we use in this thesis. We present, then, various consistencies and the algorithms used to enforce such consistencies. Furthermore, we describe different search algorithms such as *Look-back* approaches, *Look-ahead* approaches and *Maintaining Arc Consistency* search algorithm. Finally, we present some heuristics that are used to guide the search to the most promising research areas.

- **Chapter 2** is divided into two main parts. In a first place, we investigate different representations of table constraints and the used algorithms to filter them. In a second place, we introduce the background for parallel computing that we use in this thesis. We present, then, various approaches of constraint programming using parallel computing.

- In **Chapter 3**, we propose two different compression techniques for table constraints. Each of them presents a different form of compressed table constraint aiming to reduce time and space complexity. For each approach we describe the compression method used to obtain the new form of table constraints and also the filtering algorithm used during search. The first work STR$^c$[Gharbi et al., 2013] was presented at *JFPC 2013* and *CP 2013* (Doctoral Program). The second work STR-*slice* [Gharbi et al., 2014] was presented at *CPAIOR 2014*.

- In **Chapter 4**, we give a new approach for solving a CSP instance by enforcing consistencies in parallel. In fact, we explore another way of using a parallel architecture in which a main solver is helped by side workers that partially establish consistencies, which are otherwise two heavy to be maintained by the main solver.

# Part I

# Background

# Chapter 1

# Constraint Satisfaction Problem

## Contents

Constraint Programming (CP) is a framework for the representing and solving combinatorial constrained problems. In this chapter, we present the different basis of this framework. Since constraint solvers rely on "inference" and "search" techniques, we introduce both of them.

This chapter is organized as follows:

- In Section 1.1, we introduce the different features of constraint satisfaction problems;
- In Section 1.2, we describe several consistencies;
- In Section 1.3, we present Generalized Arc Consistency (GAC), the most important consistency used in constraint satisfaction and the algorithms enforcing GAC on so-called table constraints;
- In Section 1.4, we describe another consistency, Singleton Arc Consistency (SAC), and its different variants;
- In Section 1.5, we describe the most used search algorithm, called MAC, as well as many other techniques used in order to look for a solution(s);
- Finally, we present in Section 1.6 the heuristics commonly used to guide the search .

## 1.1 Introduction to CSP

### 1.1.1 Definitions

In this section, we introduce variables and constraints before defining constraint networks.

**Definition 1** ($Variable$) *A* variable *generally denoted $x$ has a* domain*, denoted $dom(x)$, which is the set of values that can be taken by $x$. The current domain of $x$ may change, but it is always included in the initial domain denoted $dom^{init}(x)$ ($dom(x) \subseteq dom^{init}(x)$).*

We focus, in the context of our work, only on variables with finite domains.

A *variable* may take different states. For example, a variable can be "assigned" or "instantiated" to a value $a$ from its domain (it is"unassigned", otherwise). If the domain of a variable contains only one value, it is called a "singleton" variable.

For defining relations, we need Cartesian products.

**Definition 2** (*Cartesian Product*) *Let $D_1, D_2, \ldots, D_r$ be a sequence of $r$ sets. The Cartesian product $D_1 \times D_2 \times \cdots \times D_r$, also denoted $\prod_{i=1}^{r} D_i$, is the set $\{(a_1, a_2, \ldots, a_r) | a_1 \in D_1, a_2 \in D_2, \ldots, a_r \in D_r\}$.*

**Example 1** *We give an example of a Cartesian product built from domains of variables $x$, $y$ and $z$ with $dom(x) = dom(y) = dom(z) = \{a, b\}$.*

$$dom(x) \times dom(y) \times dom(z) = \left\{ \begin{array}{cc} (a,a,a), & (b,a,a), \\ (a,a,b), & (b,a,b), \\ (a,b,a), & (b,b,a), \\ (a,b,b), & (b,b,b) \end{array} \right\}$$

**Definition 3** (*Relation*) *A* relation *$R$ is defined over a sequence of $r$ sets $D_1, D_2, \ldots, D_r$. $R$ is a subset of the Cartesian product $\prod_{i=1}^{r} D_i$ ($R \subseteq \prod_{i=1}^{r} D_i$).*

**Example 2** *$R_{xyz}$ is a relation defined over $dom(x) \times dom(y) \times dom(z)$ (introduced in Example 1).*

$$R_{xyz} = \left\{ \begin{array}{c} (b,a,a), \\ (b,a,b), \\ (a,b,a), \\ (b,b,a) \end{array} \right\}$$

**Definition 4** (*Literal*) *A* literal *is a pair $(x, a_i)$ where $x \in \mathscr{X}$ and $a_i \in dom(x)$.*

**Definition 5** (*Constraint*) *A* constraint *$c$ involves a set of variables called the* scope *of $c$ (denoted $scp(c)$). A constraint $c$ is semantically defined by a* relation *denoted $rel(c)$ that exhibits the combinations of values allowed by $c$ for all variables of its scope ($rel(c) \subseteq \prod_{x \in scp(c)} dom^{init}(x)$). The* arity *of a constraint $c$ is the size of $scp(c)$, and is usually denoted by $r$.*

We distinguish three forms of constraints: *intentional* constraints, *extensional* constraints and *global* constraints.

Constraints defined in *intension* describe, *implicitly*, the relation between the variables of its scope by a predicate based on a mathematical expression or formula. A predicate maps any combination of literals to a Boolean value. The combination is allowed if the mapped Boolean value is *true*, otherwise it is forbidden. The constraint $x = y$ is such an intentional constraint.

Constraints defined in *extension*, also called table constraints, can be represented in different equivalent ways. For example:

- An *explicit relation* that lists the allowed or disallowed combinations of values (tuples).

**Example 3** $c_{xyz}$ *is a ternary constraint represented in extension as follows:*

| $x$ | $y$ | $z$ |
|---|---|---|
| $(b,$ | $a,$ | $a)$ |
| $(c,$ | $a,$ | $c)$ |
| $(a,$ | $b,$ | $a)$ |
| $(b,$ | $b,$ | $c)$ |

- A *compatibility hyper-graph* $\mathscr{H}$, $\mathscr{H} = (\mathscr{V}, \mathscr{E})$, where $\mathscr{V}$ the set of vertices (nodes) define the literals and $\mathscr{E}$ the set of hyper-edges linking them. Supported pairs are related by solid hyper-edges, whereas dotted links define unsupported ones on $c$. A 2-uniform hyper-graph is a called a graph. For simplicity, we use graphs in our examples.

**Example 4** *In Figure 1.1,* $c_{xy}$ *is a binary constraint represented by a compatibility graph. For example,* $(a, a)$ *and* $(c, b)$ *are pairs allowed by* $c_{xy}$ *whereas* $(b, a)$ *and* $(c, c)$ *are disallowed ones.*

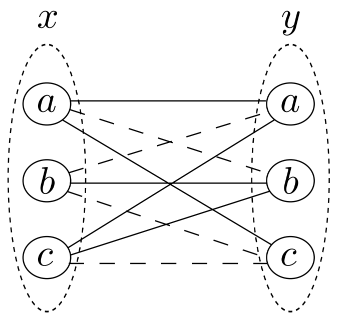


Figure 1.1: The compatibility graph of constraint $c_{xy}$ such that $dom(x) = dom(y) = \{a, b, c\}$.

*Global* constraints [Bessiere and van Hentenryck, 2003, Régin, 2011a] are constraints defined over an unfixed number of variables. This form of constraints facilitate the problem expression by a user providing a better view of the structure of the problem. For example, a well-known global constraint is *AllDifferent* [Régin, 1994]. This constraint means that all the involved variables must take different values, i.e. no variable can have the same value as another one. Global constraints associated with powerful filtering algorithms (presented in Section 1.3) are one of the main strengths of constraint programming.

**Definition 6** (*Constraint network*) *A finite* Constraint Network *(CN) $P$ is a triplet $< \mathscr{X}, \mathscr{C}, \mathscr{D} >$ where $\mathscr{X}$ is a finite set of variables (also denoted $vars(P)$), $\mathscr{C}$ a set of constraints (also denoted $cons(P)$) such that $\forall c \in cons(P), scp(c) \subseteq vars(P)$ and $\mathscr{D}$ is the domains of the variables; $\mathscr{D} = \{dom(x) | x \in \mathscr{X}\}$.*

The main numerical features of a constraint network $P$ are as follows:

- $n$ the number of variables;
- $d$ the size of the greatest domain of a variable;
- $e$ the number of constraints;
- $r$ the maximum arity of the constraints.

We define below instantiations and tuples.

**Definition 7** (*Instantiation*) *An* instantiation *$I$ of a set of variables $S$=$\{x_1, x_2, \ldots, x_r\}$ is a set $\{(x_i, a_i) \mid \forall i \in [1, r]; a_i \in dom^{init}(x_i)\}$*
*The set of variables $S$ involved in $I$ is denoted $vars(I)$ and each value $a_i$ is denoted $I[x_i]$.*
*An instantiation $I$ is* valid *iff $\forall (x, a_i) \in I, a_i \in dom(x)$.*

**Definition 8** (*Tuple*) *An $r$-*tuple *$\tau$ on a constraint $c$ such that $scp(c) = \{x_1, x_2, \ldots x_r\}$ is a sequence of values $(a_1, a_2, \ldots, a_r)$ such that $\forall i \in [1, r]; a_i \in dom^{init}(x_i)$. The value $a_i$ is denoted by $\tau[x_i]$.*

In the context of this thesis, for simplicity, we make no difference between a tuple $\tau = (a_1, a_2, \ldots, a_r)$ and an instantiation $I = \{(x_i, a_i) | \forall i \in [1, r]; a_i \in dom^{init}(x_i)\}$. We say that $I$ is the corresponding instantiation of $\tau$.

**Definition 9** (*Valid tuple*) *An $r$-tuple $\tau$ on a constraint $c$ such that $scp(c) = \{x_1, x_2, \ldots x_r\}$ is* valid *iff $\forall x \in scp(\tau), \tau[x] \in dom(x)$.*

**Example 5** *Let us consider a constraint $c_{xyz}$ with $dom(x)$=$dom(z)$=$\{a, b, c\}$ and $dom(y)$=$\{a, c\}$. $\tau_1$=$(a, a, b)$ is a valid tuple since $\tau_1[x] \in dom(x)$, $\tau_1[y] \in dom(y)$ and $\tau_1[z] \in dom(z)$. However, $\tau_2$=$(b, b, b)$ is not valid since $\tau_2[y] \notin dom(y)$.*

**Definition 10** (*Support tuple*) *An $r$-tuple $\tau$ on $scp(c)$ is a* support *on the $r$-ary constraint $c$ iff $\tau$ is a valid tuple which is allowed by $c$. If $\tau$ is a support on a constraint $c$ involving a variable $x$ and such that $\tau[x] = a$, we say that $\tau$ is a* support for $(x, a)$ *on $c$.*

In Example 5, $\tau_1$ is a support for $(x, a)$, $(y, a)$ and $(z, b)$ on $c_{xyz}$.

**Definition 11** (*Instantiation Projection*) *Let $S$ be a set of variables and an instantiation $I$. The projection of $I$ on $S$, denoted $I[S]$, is $I[S] = \{(x,a)|(x,a) \in I \wedge x \in S\}$.*

**Definition 12** (*Satisfaction of a Constraint*) *A constraint $c \in \mathscr{C}$ is said* satisfied *by an instantiation $I$ iff $scp(c) \subseteq vars(I)$ and $I[scp(c)] \in rel(c)$.*

The domains of variables evolve during search which affects the validity of instantiations. In a constraint network, we distinguish several types of instantiations.

**Definition 13** (*Partial/Complete Instantiation*) *Considering a constraint network $P$, an instantiation $I$ is said* partial *if $vars(I) \subset vars(P)$. If $vars(I) = vars(P)$, $I$ is a* complete *instantiation.*

**Definition 14** (*Solution*) *Considering a constraint network $P$, an instantiation $I$ is a* solution *of $P$ if $I$ is a complete instantiation that satisfies all the constraints of $P$.*

**Definition 15** (*Locally consistent Instantiation*) *Let $P =< \mathscr{X}, \mathscr{D}, \mathscr{C} >$ be a constraint network, and an instantiation $I$ of a subset $S$ of $\mathscr{X}$. $I$ is said* locally consistent *iff $\forall c \in \mathscr{C}$ such that $scp(c) \subseteq vars(I)$, $c$ is satisfied by $I$.*

**Definition 16** (*Globally consistent/inconsistent Instantiation*) *An instantiation $I$ is said* globally consistent *if it can be extended to a solution, otherwise it is* globally inconsistent.

### 1.1.2 Complexity

In this sub-section, we introduce three main complexity classes:

- *P problems* for which a polynomial time algorithm exists ($O(n^k)$) such that $n$ is the size of the problem and $k$ is a constant. In the complexity class $P$, problems can be solved on a deterministic Turing machine [Herken, 1995] in polynomial time. The AKS (Agrawal–Kayal–Saxena) primality test [Bornemann, 2003] is a problem belonging to the complexity class $P$. This problem consists in determining whether a number is prime or composite;

- *NP problems* for which a verification algorithm exists to verify, in a deterministic way, if a solution to the problem is valid or not in polynomial time. NP problems can be solved on a non-deterministic Turing machine in polynomial time. One of the well-known problems of this class is the integer factorization which is the decomposition of a composite number into a product of smaller integers.

The constraint satisfaction problem belongs to the *NP-complete* complexity class. The most notable characteristic of NP-complete problems is that the required time to solve such problems grows quickly as the size of the problem grows (under the assumption that P $\neq$ NP).

## 1.1.3 Example of a CSP: Map coloring

We take the example of a problem called "map coloring". This problem consists in coloring a map in a such way that there is no two adjacent regions having the same color. We mean by "adjacent" two regions sharing a same boundary line. The four-color theorem discovered by Francis Guthrie in the early 1850s (would-be proved by Alfred Bray Kempe in 1879 and proved in [Appel and Haken, 1977]), claims that "every planar map is four colorable". To illustrate that, we take the example of a Tunisian map illustrated in Figure 1.2.

Since our aim is to use only four different colors to color this map, we first express this problem as a CSP. The four colors that we can use are *white*($w$), *light gray*($lg$), *mid gray*($mg$) and *dark gray*($dg$) which compose the domain of the different variables. Figure 1.2 presents all variables of the problem which are the regions to be colored. We have 24 variables since there are 24 regions. For the constraint part, we add a binary constraint each time we have two regions sharing a boundary line. As a consequence, the map coloring problem is formalized as a constraint network:

- $P$=$< \mathscr{X},\mathscr{D},\mathscr{C} >$;
- $\mathscr{X}$=$\{x_i | i \in [0,23]\}$;
- $\mathscr{D}$=$\{dom(x_i) | i \in [0,23]\}$ such that $dom(x_i) = \{w, lg, mg, dg\}$;
- $\mathscr{C}$=
$$\left\{\begin{array}{cccc}
c_{x_0x_2} : x_0 \neq x_2 & c_{x_0x_3} : x_0 \neq x_3 & c_{x_0x_4} : x_0 \neq x_4 & c_{x_1x_2} : x_1 \neq x_2 \\
c_{x_1x_6} : x_1 \neq x_6 & c_{x_1x_7} : x_1 \neq x_7 & c_{x_2x_3} : x_2 \neq x_3 & c_{x_2x_7} : x_2 \neq x_7 \\
c_{x_3x_4} : x_3 \neq x_4 & c_{x_3x_5} : x_3 \neq x_5 & c_{x_3x_8} : x_3 \neq x_8 & c_{x_3x_9} : x_3 \neq x_9 \\
c_{x_4x_5} : x_4 \neq x_5 & c_{x_5x_9} : x_5 \neq x_9 & c_{x_6x_7} : x_6 \neq x_7 & c_{x_6x_{11}} : x_6 \neq x_{11} \\
c_{x_7x_7} : x_7 \neq x_8 & c_{x_7x_{11}} : x_7 \neq x_{11} & c_{x_7x_{12}} : x_7 \neq x_{12} & c_{x_7x_{15}} : x_7 \neq x_{15} \\
c_{x_9x_{10}} : x_9 \neq x_{10} & c_{x_{10}x_{13}} : x_{10} \neq x_{13} & c_{x_{11}x_{12}} : x_{11} \neq x_{12} & c_{x_{11}x_{15}} : x_{11} \neq x_{15} \\
c_{x_{11}x_{18}} : x_{11} \neq x_{18} & c_{x_{12}x_{13}} : x_{12} \neq x_{13} & c_{x_{12}x_{15}} : x_{12} \neq x_{15} & c_{x_{12}x_{16}} : x_{12} \neq x_{16} \\
c_{x_{12}x_{17}} : x_{12} \neq x_{17} & c_{x_{13}x_{14}} : x_{13} \neq x_{14} & c_{x_{14}x_{16}} : x_{14} \neq x_{16} & c_{x_{15}x_{17}} : x_{15} \neq x_{17} \\
c_{x_{15}x_{18}} : x_{15} \neq x_{18} & c_{x_{16}x_{17}} : x_{16} \neq x_{17} & c_{x_{17}x_{21}} : x_{17} \neq x_{21} & c_{x_{18}x_{19}} : x_{18} \neq x_{19} \\
c_{x_{18}x_{20}} : x_{18} \neq x_{20} & c_{x_{18}x_{21}} : x_{18} \neq x_{21} & c_{x_{19}x_{20}} : x_{19} \neq x_{20} & c_{x_{20}x_{21}} : x_{20} \neq x_{21} \\
c_{x_{20}x_{22}} : x_{20} \neq x_{22} & c_{x_{20}x_{23}} : x_{20} \neq x_{23} & c_{x_{21}x_{22}} : x_{21} \neq x_{22} & c_{x_{22}x_{23}} : x_{22} \neq x_{23}
\end{array}\right\}$$

Figure 1.2 shows a solution to the problem where:

- $x_0 = x_1 = x_5 = x_8 = x_{14} = x_{17} = x_{20} = w$

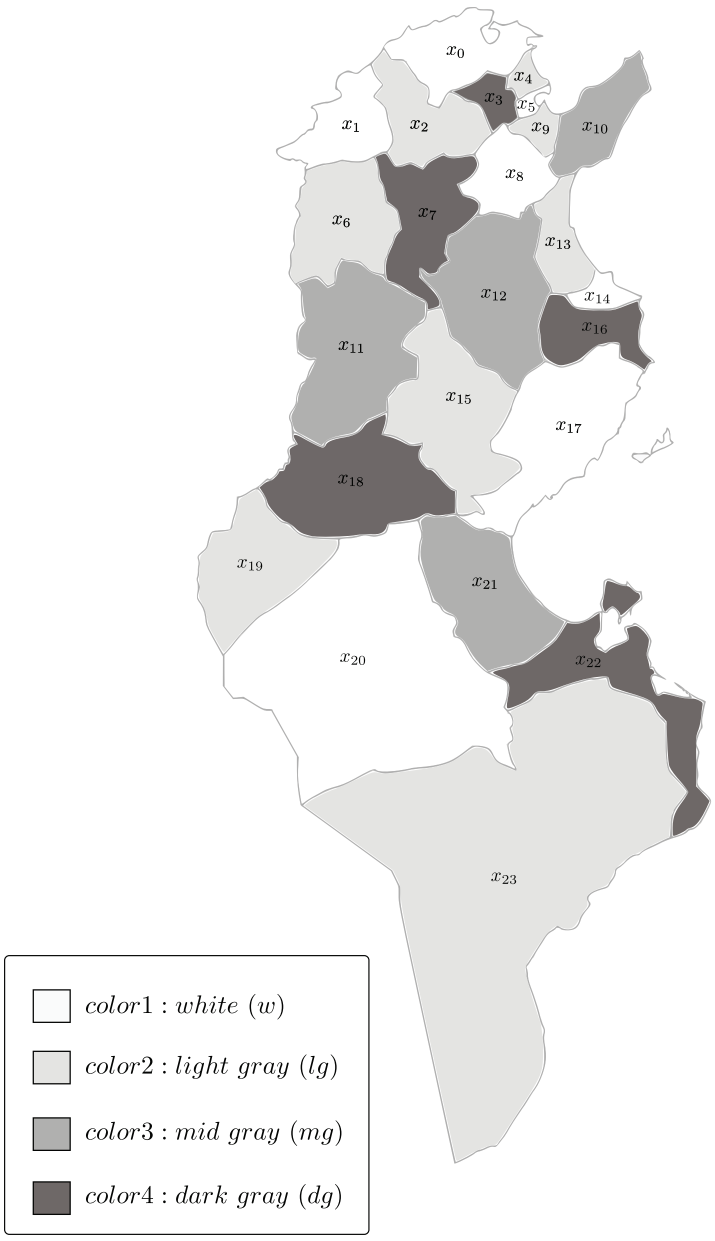


Figure 1.2: Complete instantiation of the Tunisian map coloring (Solution).

- $x_2 = x_4 = x_6 = x_9 = x_{13} = x_{15} = x_{19} = x_{23} = lg$
- $x_{10} = x_{11} = x_{12} = x_{21} = mg$
- $x_3 = x_7 = x_{16} = x_{18} = x_{22} = dg$

### 1.1.4 Solvers

In constraint programming, we distinguish two major steps: modeling problems and solving them. Modeling problems refers to expressing its variables and constraints and solving them means looking for a solution or proving that there does not exist one.

To model constraint satisfaction problems several modeling languages have been proposed such as *OPL* [van Hentenryck, 1999], *Zinc* [de la Banda et al., 2006] and *Essence* [Frisch et al., 2007]. Simple formats exist also such as XCSP based on XML [Roussel and Lecoutre, 2009].

Several solvers have been developed such as *Choco* [Rochart et al., 2006], *Gecode* [Schulte et al., 2006], *Comet* [Nyström et al., 2003], *Minion* [Gent et al., 2006] and *ILOG* [1] solver.

In the context of this thesis, we use *AbsCon* [Lecoutre and Tabary, 2007], the solver developed at CRIL. We used the instances benchmarks available at `http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html`

## 1.2 Consistency

In order to solve CSP instances, there are two major categories of algorithms and techniques: "Inference" and "Search". Since CSP is *NP-complete*, finding a solution can be difficult. To speed up this process, we can use inference techniques whose aim is to simplify the problem typically by reducing the search space. If we cannot benefit from inference techniques, an "exploration" of the search space, called "Search" (see Section 1.5), is needed in order to look for a solution.

### 1.2.1 Definitions

Some inference algorithms and techniques are based on "constraint filtering" which consists in removing combination(s) of values while preserving constraints semantics in order to speed up the search space exploration [Montanari, 1974]. This filtering process is described on the basis of "consistencies" which are constraint network properties. A consistency corresponds to a certain level of coherence. It can be "local", if it deals with a subset of variables, or "global" if it concerns the entire network.

**Definition 17** (*k-consistency*) *Let $P$ be a constraint network and $k$ be an integer such that $1 \leq k < n$; $n = |vars(P)|$. $P$ is* k-consistent *[Freuder, 1978] iff for every subset of size $k-1$ $S_{k-1} \subset vars(P)$ and every additional variable $y$ such that $y \in vars(P) \setminus S_{k-1}$, every locally consistent instantiation $I$ of $S_{k-1}$*

[1] www.ilog.com

*on $P$ can be extended to a locally consistent instantiation $I^{'}$ such that vars($I^{'}$)= $S_{k-1} \cup \{y\}$.*

A *2-consistent* constraint network $P$ is said *arc-consistent.*

It is important to note that a *k-consistent* constraint network $P$ is not necessarily *j-consistent* with $1 \leq j < k$. A stronger property can be defined.

**Definition 18** (*Strong k-consistency*) *Let $P$ be a constraint network and $k$ an integer such that $1 \leq k < n$ where $n = |vars(P)|$. $P$ is* strong k-consistent *iff it is* j-consistent $\forall j \in [1, k]$.

A stronger consistency can be defined over a constraint network $P$ (See Section 1.4.3).

**Definition 19** (*Global consistency*) *A constraint network $P$ is* globally consistent *iff it is strongly* n-consistent.

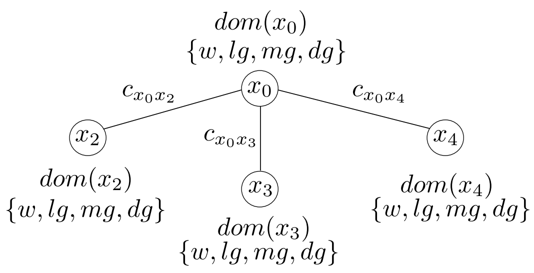
We focus, in the context of this thesis, on "Domain-filtering" consistencies which detect some inconsistent values, in the domains of variables, aiming to discard them in order to simplify the problem.

**Example 6** *Figure 1.3 illustrates the practical effect of enforcing a* domain-filtering consistency. *Taking the example of map coloring introduced in Section 1.1.3, we present three constraints $c_{x_0x_2}$, $c_{x_0x_3}$ and $c_{x_0x_4}$. The initial domains of variables $x_0$, $x_2$, $x_3$ and $x_4$ are the same; $dom(x_0) = dom(x_2) = dom(x_3) = dom(x_4) = \{w, lg, mg, dg\}$. After assigning $w$ to the variable $x_0$, enforcing* domain-filtering consistency *on variables $x_2$, $x_3$ and $x_4$ induces removing the value $w$ from their domains in order to not violate the constraints semantics (two neighbor regions cannot have the same color).*

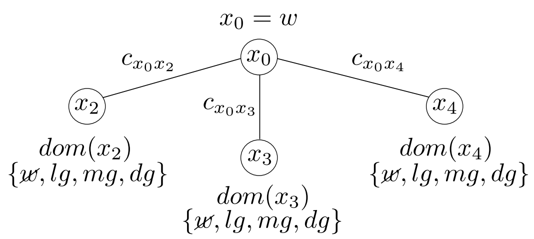
In the context of this thesis, we focus on local consistencies which are properties applied to a subset of variables or constraints. Historically [Montanari, 1974], distinguished three main consistencies.

- Node Consistency *1-consistency*
- Arc Consistency *2-consistency*
- Path Consistency

In the next sections, we introduce Arc Consistency and its generalization GAC and also a stronger consistency SAC.



(a) Initial domains.



(b) Current domains after the assignment $x_0 = w$.

Figure 1.3: Illustration of domain filtering.

## 1.3 Generalized Arc Consistency (GAC)

*Generalized Arc Consistency (GAC)* is the most important consistency, which is a property that corresponds to the maximum level of filtering, used in practice, when constraints are treated independently.

**Definition 20** (*Generalized Arc Consistency*) *Let $P$ be a constraint network, $c$ a constraint such that $c \in cons(P)$ and $x$ a variable such that $x \in scp(c)$. A literal $(x, a)$ is* GAC-consistent *iff there is a valid tuple $\tau \in rel(c)$ such that $\tau[x] = a$. A variable $x$ is* GAC-consistent *iff $\forall a \in dom(x)$, $(x, a)$ is* GAC-consistent*. A constraint $c$ is* GAC-consistent *iff $\forall x \in scp(c)$, $x$ is* GAC-consistent*. A constraint network $P$ is* GAC-consistent *iff $\forall c \in cons(P)$, $c$ is* GAC-consistent.

**Example 7** *Let us consider an* allDifferent *global constraint on three variables $x$, $y$ and $z$ such that $dom(x) = dom(z) = dom(y) = \{a, b\}$. If we suppose that $x$ is assigned to $a$ and $y$ to $b$, using the domain filtering technique the domain of $z$ does not contain anymore a possible value since $z$ should be different from $x$ and $y$. The* allDifferent *constraint is thus GAC-inconsistent.*

$$dom(x) = \{a, \cancel{b}\} \Longrightarrow dom(y) = \{\cancel{a}, b\} \Longrightarrow dom(z) = \{\cancel{a}, \cancel{b}\}$$

*Now, if we consider that $dom(z) = \{a, b, c\}$, by assigning $a$ to $x$ and $b$ to $y$, the variable $z$ could be assigned to $c$ and then the* allDifferent *constraint is GAC-consistent.*

$$dom(x) = \{a, \cancel{b}\} \Longrightarrow dom(y) = \{\cancel{a}, b\} \Longrightarrow dom(z) = \{\cancel{a}, \cancel{b}, c\}$$

In Section 1.3.2, we explore filtering algorithms which enforce *GAC* on table constraints.

## 1.3.1 Arc Consistency

*Arc Consistency* (AC) corresponds to GAC when constraints are binary.

**Example 8** *Let $P$ be a constraint network described by its compatibility graph in Figure 1.4. The variables of $P$ are $vars(P) = \{x, y, z\}$ such that $dom(x) = dom(y) = dom(z) = \{a, b, c\}$. The constraints of $P$ are $cons(P) = \{c_{xy}, c_{xz}, c_{yz}\}$. These constraints are defined in extension such that $c_{xy} = \{(a, a), (b, b), (c, c)\}$, $c_{xz} = \{(a, b), (b, a), (c, c)\}$ and $c_{yz} = \{(a, b), (c, a)\}$. $P$ is not arc-consistent since there are some values in the domain of $vars(P)$ which are not arc-consistent. This is the case of the literals $(z, c)$ and $(y, b)$ that have no supports on $c_{yz}$.*

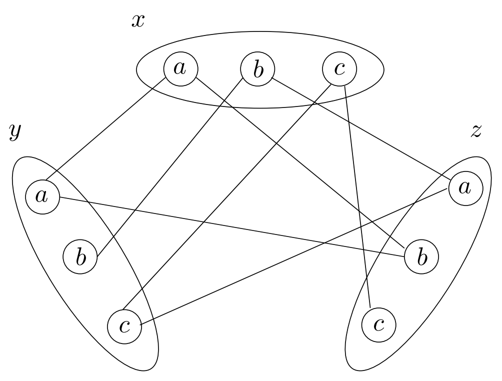
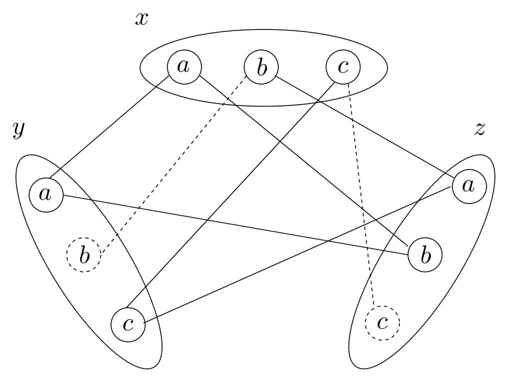


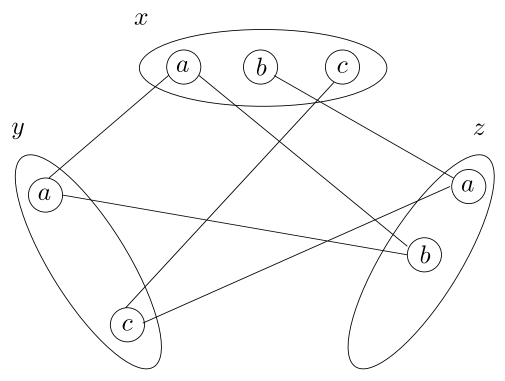
Figure 1.4: An arc-inconsistent network $P$.

*To enforce AC on $P$, we proceed to several constraint filterings. We start by enforcing AC on $c_{yz}$ described in Figure 1.5. The literal $(z, c)$ is not arc-consistent, so it is removed from the domain of the variable $z$. Hence, all values supported by $(z, c)$ lose their support (illustrated by a dotted link) since it is discarded: $c_{xz}$, originally arc-consistent, loses a support for $(x, c)$ and becomes not arc-consistent. The literal $(y, b)$ is also not arc-consistent: it is removed from $dom(y)$ and the literal $(x, b)$ has no more a support on $c_{xy}$ (originally arc-consistent). The obtained network, denoted AC(P) and illustrated in Figure 1.5(b), is still not arc-consistent.*

*As $c_{xy}$ and $c_{xz}$ become not arc-consistent, we enforce AC on them. The literal $(x, b)$ does not have a support on $c_{xy}$, so it is removed from $dom(x)$ causing the inconsistency of the literal $(z, a)$ on $c_{xz}$. This is described in Figure 1.6(a). The literals $(z, a)$ and $(x, c)$ are removed while enforcing AC on $c_{xz}$ (Figure 1.6(b)).*



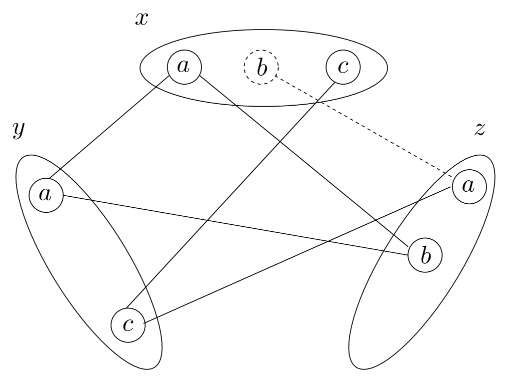
(a) $(y, b)$ and $(z, c)$ are removed.



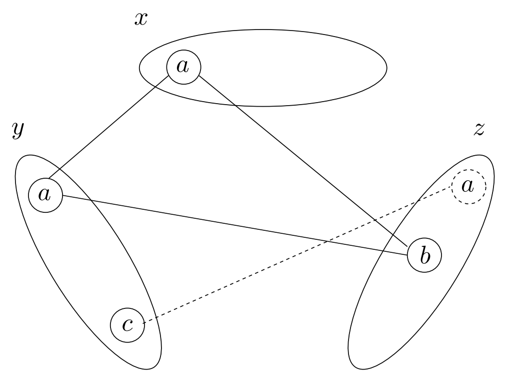
(b) An arc-inconsistent network.

Figure 1.5: Enforcing AC on $c_{yz}$.

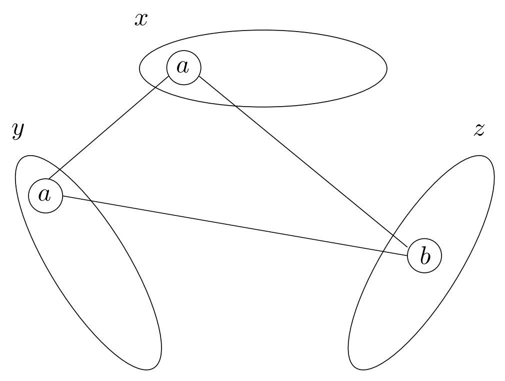
*The literal $(y, c)$ has then no support for all the constraints of $P$. By removing $(y, c)$, we obtain the final arc-consistent network (Figure 1.6(c)).*



(a) Enforcing AC on $c_{xy}$.



(b) Enforcing AC on $c_{xz}$.



(c) An arc-consistent network $P^{'} = AC(P)$.

Figure 1.6: Enforcing AC on constraints originally arc-consistent.

Example 8 shows that enforcing arc-consistency on a constraint network $P$ may lead to removing a literal $(x, a)$ which is in its turn a support for another literal $(y, b)$. As a consequence, a AC-consistent constraint may become not AC-consistent due to a support removal. The process of “constraint propagation” is repeated until it reaches a fixed point [Granas and Dugundji, 2003, Bessiere, 2006] where the constraint network is arc-consistent ($AC(P) = P$) or an inconsistency is detected due to a domain wipe-out ($AC(P) = \bot$). Several algorithms have been proposed to enforce

| **Algorithm** | **Time complexity** | **Space complexity** |
|---|---|---|
| AC1 | $O(end^3)$ | $O(n^2)$ |
| AC3 | $O(ed^3)$ | $O(e + nd)$ |
| AC4 | $O(ed^2)$ | $O(ed^2)$ |
| AC6 | $O(ed^2)$ | $O(ed)$ |
| AC7 | $O(ed^2)$ | $O(ed)$ |
| AC2001 | $O(ed^2)$ | $O(ed)$ |
| AC3$^{rm}$ | $O(ed^3)$ | $O(ed)$ |

Table 1.1: The worst-case complexity of several $AC$ algorithms on binary constraint networks.

$AC$: AC1 , AC3 [Mackworth, 1977], AC4 [Mohr and Henderson, 1986], AC6 [Bessiere, 1994], AC7 [Bessiere et al., 1999], AC2001 [Bessiere and Régin, 2001, Zhang and Yap, 2001] and AC3$^{rm}$ [Lecoutre and Hemery, 2007]. These algorithms have different complexities in terms of time and space (presented in Table 1.1).

We present only both AC3 and AC2001 algorithms. These two algorithms use the same *doAC* propagation algorithm (presented in Algorithm 8) calling Algorithm *revise* (in line 4) which differs from an $AC$ algorithm to another. Algorithm 8 takes as input a constraint network $P$. *doAC* returns a Boolean indication; “true” if the algorithm reaches an arc-consistent network and “false” if an inconsistency is detected (there is an empty domain), denoted $AC(P) = \bot$. *doAC* algorithm maintains a propagation queue $Q$ containing literals $(c, x)$ where $c \in \mathscr{C}$ and $x \in scp(c)$. These literals contained in $Q$ are revised by Algorithm *revise* which removes all values $a \in dom(x)$ that do not have a support on $c$. If *revise* removes at least one value, it checks if $dom(x) = \emptyset$. If it is the case, a global inconsistency is detected and “false” is returned, otherwise the queue $Q$ is updated by adding all literals ($c$’,$y$) where $c$’$\in \mathscr{C}$ is a constraint involving $x$. All these literals should be revised in the next iteration since it is possible that a removed literal $(x, a)$ is a support for another literal $(y, b)$ on a constraint $c$’ (as shown in Example 8). The constraint propagation achieves its fixed point when $Q$ is empty ($Q = \emptyset$: there is no literal $(c, x)$ to revise) and “true” is returned indicating that the constraint network respects the $AC$ property.

**Algorithm 8:** doAC ($P = (\mathscr{X}, \mathscr{D}, \mathscr{C})$: Constraint network): Boolean

```
1 $Q \leftarrow \{(c, x) | c \in \mathscr{C} \wedge x \in scp(c)\}$
2 while $Q \neq \emptyset$ do
3     choose then delete $(c, x)$ from $Q$
4     if $revise(c, x)$ then
5         if $dom(x) = \emptyset$ then
6             return false
7         $Q \leftarrow Q \cup \{(c', y) | c' \in \mathscr{C} \wedge x \in scp(c') \wedge y \in scp(c') \wedge y \neq x \wedge c \neq c'\}$
8 return true
```

**AC3 [Mackworth, 1977]** AC3 algorithm uses *revise*-3 revision technique (Algorithm 9). Each literal $(c, x)$ is revised in a manner that *revise*-3 checks if for each value in the current domain of $x$, there exists a support on $c$ (Line 3). If a support is found, the checking process continues until each value of $dom(x)$ is verified. If *revise*-3 does not find a support for $(x, a)$ on $c$, this value is removed from $dom(x)$ (Line 4). *revise*-3 algorithm returns *true* iff there is at least one value removed from $dom(x)$. AC3 algorithm has a time complexity of $O(ed^3)$ where $e$ is the number of constraints of $P$ and $d$ is the size of the greatest domain of a variable of the network (as presented in Section 1.1.1).

**Algorithm 9:** revise-3($c$: constraint, $x$:variable): Boolean

```
  // $scp(c) = \{x, y\}$
1 $modified \leftarrow false$
2 foreach $a \in dom(x)$ do
3     if $\nexists b \in dom(y)$ such that $(a, b) \in rel(c)$ then
4         $dom(x) \leftarrow dom(x) \backslash a$
5         $modified \leftarrow true$
6 return $modified$
```

**AC2001 [Bessiere and Régin, 2001, Zhang and Yap, 2001]** *AC2001* has an optimized *revise* algorithm in a manner that speeds up the support detection process. *revise-2001* (presented in Algorithm 10) uses the data structure, called *last*, which stores the last support found on a constraint $c$, such that $scp(c) = \{x, y\}$, for a variable $x$ and a value $a$. This structure will speed up the support searching. In other terms, it is not required anymore to iterate, *in each revision*, over all values of $dom(y)$ to find a support, but only checking if $last[c, x, a]$ (last found support) still exists in $dom(y)$ is enough (Line 8). If the last found support does not exist in $dom(y)$, the searching process continues from $last[c, x, a]$ position and so looking for a support does not restart from scratch

(Line 4). This explains why the condition ($b > last[c, x, a]$) is added. If there is no "new" support found, the literal $(x, a)$ is considered arc-inconsistent and it is removed from $dom(x)$ (Line 5), otherwise the "new" found support $(y, b)$ is stored in $last[c, x, a]$ (Line 8). The time complexity of *AC2001* algorithm is $O(ed^2)$ which is reduced compared to $AC3$ ($O(ed^3)$).

**Algorithm 10:** revise-2001($c$: constraint, $x$:variable): Boolean

```
  // $scp(c) = \{x, y\}$
1 $modified \leftarrow false$
2 foreach $a \in dom(x)$ do
3     if $last[c, x, a] \notin dom(y)$ then
4         if $\nexists b \in dom(y)$ such that $b > last[c, x, a] \wedge (a, b) \in rel(c)$ then
5             $dom(x) \leftarrow dom(x) \backslash a$
6             $modified \leftarrow true$
7         else
8             $last[c, x, a] \leftarrow b$
9 return modified
```

### 1.3.2 GAC for Table Constraints

Table constraints, i.e., constraints given in extension by listing the tuples of values allowed (positive table) or forbidden (negative table) for a set of variables, are widely studied in Constraint Programming. This is because such constraints are present in many real-world applications from areas such as design and configuration, databases, and preferences modeling. Sometimes, table constraints provide the unique natural or practical way for a non-expert user to express her constraints. So far, research on table constraints has mainly focused on the development of fast algorithms to enforce Generalized Arc Consistency and/or to compress their representation. Algorithms *AC3* and *AC2001* presented in the previous section are generalized for *n-ary* constraints as *GAC3* and *GAC2001* and could be used for table constraints. We introduce different variants of Simple Tabular Reduction (STR) technique used to manage table constraints during search.

For the sake of simplicity, we only consider positive table constraints.

**Classical schemes** We distinguish two classical schemes for support seeking. *GAC-valid* scheme iterates over valid tuples until an allowed one is found. *GAC-allowed* scheme iterates over allowed tuples until a valid one is found. Since iterating over allowed/valid tuple is expensive, several STR variants are proposed to speed up enforcing GAC on table constraints. *GAC-valid+allowed*

[Lecoutre and Szymanek, 2006] alternates in iteration over both valid and allowed tuples without any additional data-structure.

### Simple Tabular Reduction (STR)

STR [Ullmann, 2007], also called STR1, is one of the most efficient techniques to enforce GAC on table constraints. This approach differs significantly from the other techniques and aims to reduce dynamically, and during search, table constraints complexities. In fact, enforcing GAC consists in removing invalid tuples in a manner that the table maintains only the allowed tuples. This technique facilitates detecting generalized arc-inconsistent values inducing tuples removing. As a consequence, a table contains only supports. We present STR algorithm and its optimized variants. However, we will keep the backtracking issues (going back to the previous search state) for a further section (Section 1.5.1).

The table constraint *table*[c] is split into two major parts. The first one contains the allowed valid tuples, also called *current tuples*, forming the *current table*. The remaining part contains all tuples removed at different levels of search. In our context, a level of search corresponds to the number of positive decisions made. Managing the set of tuples is provided by:

- *position*[*c*] is an array of size $t$ that provides indirect access to the tuples of *table*[*c*]. In fact, this array is used in order to achieve an $O(1)$ permutation since we permute indexes instead of tuples. *table*[*c*][*position*[*c*][*i*]] represents the $i^{th}$ tuple of $c$. At any given time the values in *position*[*c*] are a permutation of $\{1, 2, \ldots, t\}$.

- *currentLimit*[*c*] is the index of the last current tuple in *table*[*c*]. The elements in *position*[*c*] at indexes ranging from 1 to *currentLimit*[*c*] are positions of the current tuples of $c$. As a consequence, the current table of c is composed of exactly *currentLimit*[*c*] tuples.

- *levelLimits*[*c*] is an array of size $n$+1 such that *levelLimits*[*c*][*p*] is the position of the first invalid tuple of *table*[*c*] removed at search level $p$. If there is no tuple removed at level $p$, *levelLimits*[*c*][*p*]=$-1$. All tuples removed at level $p$ (if *levelLimits*[*c*][*p*]$\neq -1$) can be accessed using indexes at locations in array *position*[*c*] ranging from *currentLimit*[*c*]+1 to *levelLimits*[*c*][*p*]. *levelLimits*[*c*] is exploited in Backtracking and it is indexed from 0 to $n$. *levelLimits*[*c*][0] is used to store the tuples removed at pre-processing [Benson et al., 1992] (the set of investigations done before searching for solution(s)) .

- *gacValues*[*c*] [Ullmann, 1977] is an array of size $r$, where $r$ is the arity of $c$. To each uninstantiated variable $x$ corresponds a set *gacValues*[*c*][*x*] containing all values in *dom*($x$) which are proved to have a support when GAC is enforced on $c$.

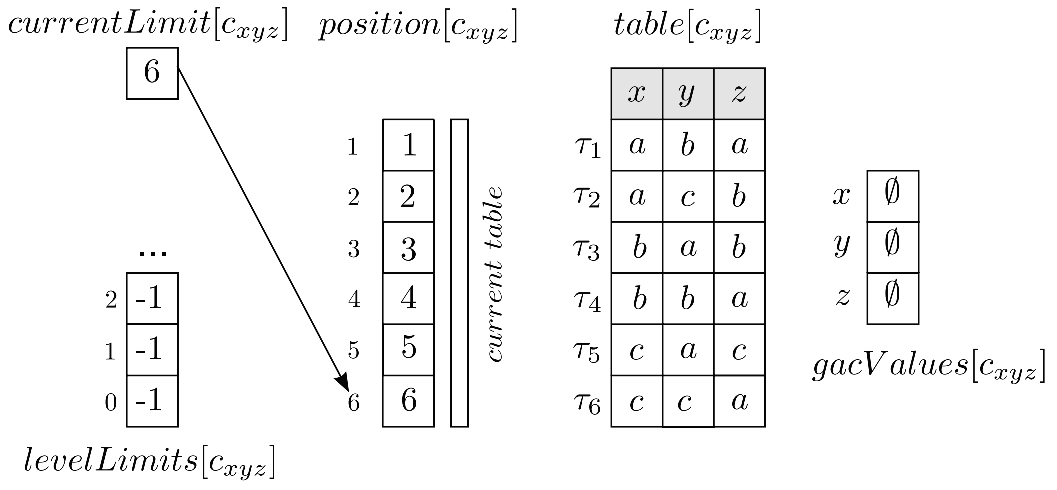
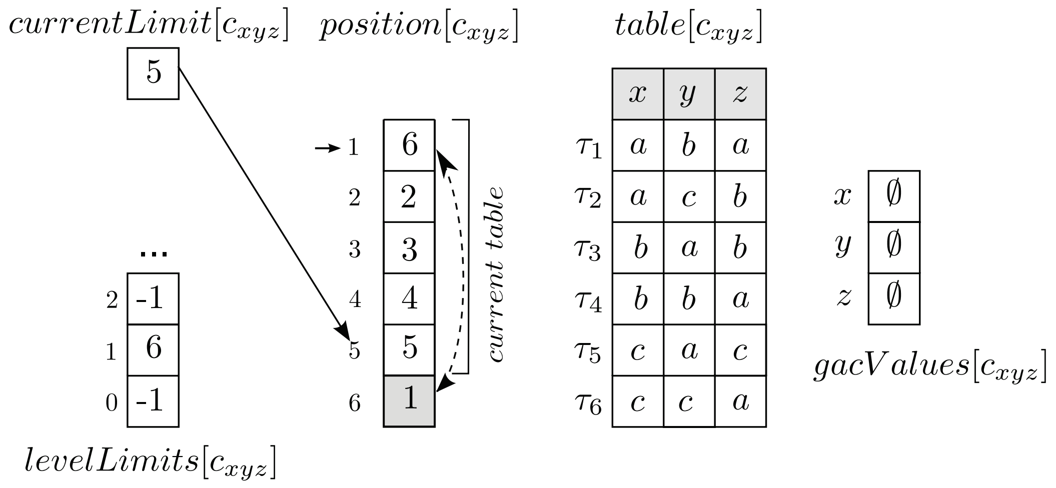


Figure 1.7: Initialization of STR data structures for a positive table constraint $c_{xyz}$; $dom(x) = dom(y) = dom(z) = \{a, b, c\}$.

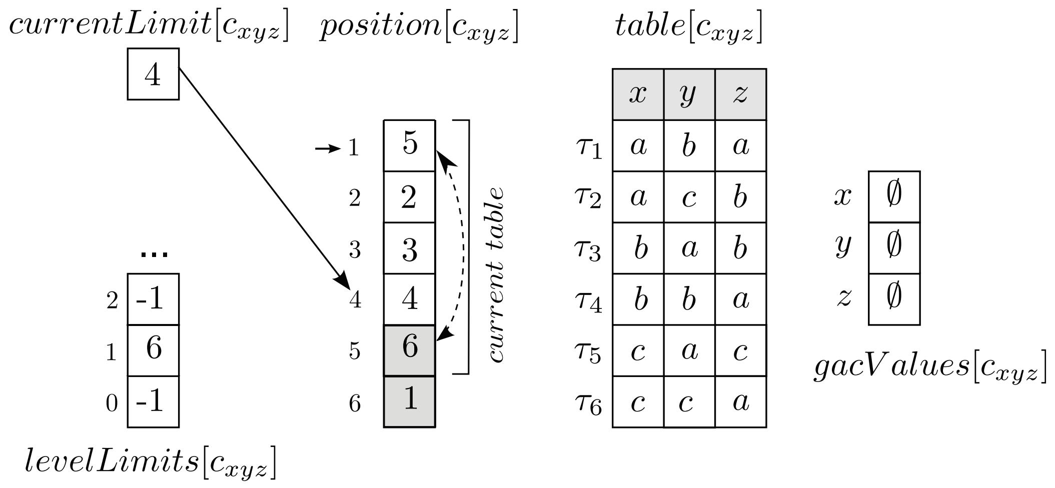
**Example 9** *To illustrate the use of the different data structures, we describe the different changes during search.* $c_{xyz}$ *is a ternary positive table constraint such that* $scp(c_{xyz}) = \{x, y, z\}$ *and* $dom(x) = dom(y) = dom(z) = \{a, b, c\}$. $rel(c_{xyz})$ *is presented in Figure 1.7 which describes the initial states of* position*[c],* levelLimits*[c],* currentLimit*[c] and* gacValues*[c].*

*We consider that a decision made at level* 1 *(for a variable of the constraint network involving* $c_{xyz}$*) induces (by constraint propagation) the removal of* $(z, a)$. *STR is then applied (Figure 1.8) such that* $dom^1(x) = dom^1(y) = \{a, b, c\}$ *and* $dom^1(z) = \{b, c\}$. *Tuple* $\tau_1$ *is no longer valid, so it is removed by swapping its position with the* currentLimit *position.* levelLimits*[*$c_{xyz}$*][1] is updated to* 6 *as it is the position of the first invalid tuple in this level. The* currentLimit *is also updated to* 5 *(the last current tuple)(Figure 1.8(a)). The process is continued by swapping* position*[*$c_{xyz}$*][1] and* position*[*$c_{xyz}$*][5]. The final result is described in 1.8(c). The array* gacValues*[*$c_{xyz}$*] is updated where* gacValues*[*$c_{xyz}$*][x]=*$dom^1(x)$, gacValues*[*$c_{xyz}$*][z]=*$dom^1(z)$ *but* gacValues*[*$c_{xyz}$*][y]*$\neq$ $dom^1(y)$. *The literal* $(y, b)$ *is not supported anymore and will be removed.* currentLimit*[*$c_{xyz}$*] is equal to* 3 *which means that there are three remaining valid tuples in* $c_{xyz}$ *after enforcing GAC. At level* 2*, the literal* $(x, a)$ *is removed by propagation. Enforcing GAC is then required. Figure 1.9 describes the final result where* levelLimits*[*$c_{xyz}$*][2] is turned to* 3 *referencing the first removed tuple. The variable y loses also the value c since* gacValues*[*$c_{xyz}$*][y]=*$\{a\} \neq dom^2(y) = \{a, c\}$ *meaning that this value is no longer supported by any current tuple of* $c_{xyz}$.

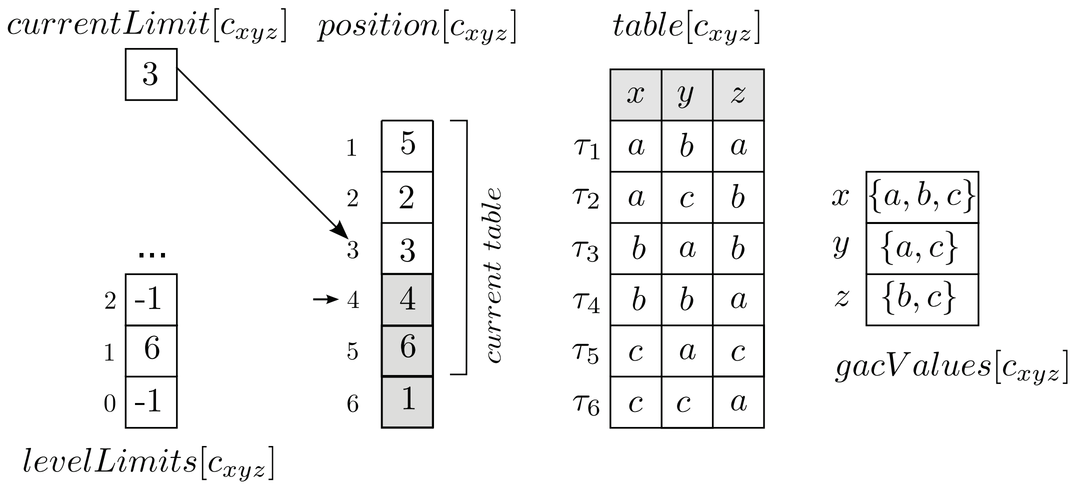
The worst-case time complexity of *enforceGAC-STR* is $O(r'd + rt')$ where $r'$ is the uninstantiated variables ($r' = |scp(c) \setminus past(P)|$) and $t'$ is the current



(a) Testing the validity of $\tau_1$.



(b) Testing the validity of $\tau_6$.



(c) STR applied on $c_{xyz}$; $(y, b)$ is not supported.

Figure 1.8: Enforcing GAC-STR on $c_{xyz}$ after the removal of $(z, a)$ at level 1.

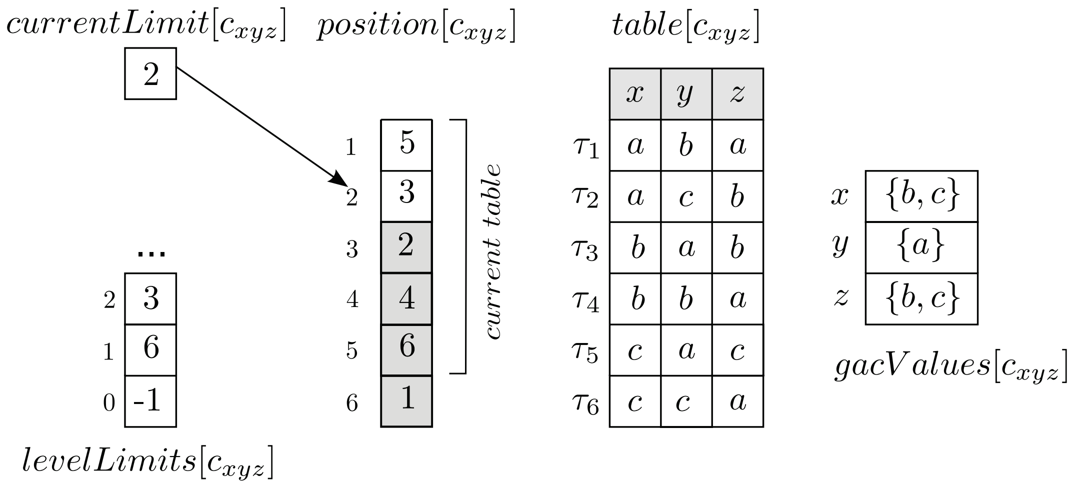


Figure 1.9: Enforcing GAC-STR on $c_{xyz}$ after the removal of $(x, a)$ at level 2; $(y, c)$ is not supported.

tuples ($past(P)$ is the last assigned variable). The worst-case space complexity of *enforceGAC-STR* is $O(n + rt)$.

**STR2 [Lecoutre, 2011]** An optimized variant of STR is proposed in [Lecoutre, 2011] where we can distinguish two optimizations.

In a first place, it is useless to continue looking for supports for domain values if these ones have already been detected GAC-consistent. To avoid such redundant operations, the $S^{sup}$ set is introduced. All uninstantiated variables in $scp(c)$ whose domain contains at least one unsupported value are stored in $S^{sup}$.

In a second place, useless validity operations are avoided. It is true that, for each variable $x$ in $scp(c)$, if there is no backtrack and $dom(x)$ does not change since the last GAC-STR call, the current value $\tau[x] \in dom(x)$. A second set, denoted $S^{val}$, is defined which contains all uninstantiated variables in $scp(c)$ whose domain has been reduced since the last call of GAC-STR.

The used GAC-STR is called *enforceGAC-STR2* (presented in Algorithm 11). The gain obtained through $S^{sup}$ use is obvious in Lines 16 and 25 where, instead of iterating over all variables in *enforceGAC-STR*, *enforceGAC-STR2* iterates only over those contained in $S^{sup}$. Initially contains all uninstantiated variables (Line 7), $S^{sup}$ set is updated in Line 20 where the variable $x$ is removed if it has been found that all the value of its domains are supported ($|dom(x)| = |\texttt{gacValues}[c][x]|$).

The set $S^{val}$ contains the last assigned variable if it is involved in $scp(c)$ (lastPast(P)) and, eventually all variables having a domain smaller than the last stored size ($lastSize[c][x]$) in the last call (Lines 4 and 9). The set $S^{val}$ is used in *isValidSTuple2* (introduced in Algorithm 12) to reduce tests to only variables contained in $S^{val}$ whose domains has not been changed since the last call.

**Algorithm 11:** enforceGAC-STR2($P$ :constraint network, $c$: constraint):set of variables

```
// Initialization of sets $S^{val}$ and $S^{sup}$
$S^{val} \leftarrow \emptyset$
$S^{sup} \leftarrow \emptyset$
if lastPast($P$) $\in scp(c)$ then
    $S^{val} \leftarrow S^{val} \cup \{\text{lastPast}(P)\}$
foreach variable $x \in scp(c) | x \notin \text{past}(P)$ do
    gacValues$[c][x] \leftarrow \emptyset$
    $S^{sup} \leftarrow S^{sup} \cup \{x\}$
    if $|dom(x)| \neq$ lastSize$[c][x]$ then
        $S^{val} \leftarrow S^{val} \cup \{x\}$
        lastSize$[c][x] \leftarrow |dom(x)|$
// Iteration over all current tuples of $c$
$i \leftarrow 1$
while $i \leq$ currentLimit$[c]$ do
    $index \leftarrow$ position$[c][i]$
    $\tau \leftarrow$ table$[c][index]$
    if isValidTuple($c, S^{val}, \tau$) then
        foreach variable $x \in scp(c) | x \in S^{sup}$ do
            if $\tau[x] \notin$ gacValues$[c][x]$ then
                gacValues$[c][x] \leftarrow$ gacValues$[c][x] \cup \{\tau[x]\}$
                if $|dom(x)| = |$gacValues$[c][x]|$ then
                    $S^{sup} \leftarrow S^{sup} \setminus \{x\}$
        $i \leftarrow i + 1$
    else
        removeTuple($c, i$)          // currentLimit$[c]$ decremented
// domains are now updated and $X_{evt}$ computed
$X_{evt} \leftarrow \emptyset$
foreach variable $x \in S^{sup}$ do
    $dom(x) \leftarrow gacValues[c][x]$
    if $dom(x) = \emptyset$ then
        throw INCONSISTENCY
    $X_{evt} \leftarrow X_{evt} \cup \{x\}$
    lastSize$[c][x] \leftarrow |dom(x)|$
return $X_{evt}$ // $X_{evt}$ constains variables with reduced domains
```

**Algorithm 12:** isValidSTuple($c$: constraint,$S^{val}$:variables, $\tau$: tuple): Boolean

```
1 foreach variable $x \in S^{val}$ do
2     if $\tau[x] \notin dom(x)$ then
3         return false
4 return true
```

According to [Lecoutre, 2011], there exists situations where applying *enforceGAC-STR2* is $O(t + rd)$, which means that *enforceGAC-STR2* is potentially $r$ times faster than *enforceGAC-STR* .

**STR3 [Lecoutre et al., 2012]** STR3 is also an optimized variant of STR which differs significantly from STR and STR2 in representing invalid tuples. Rather than discarding them explicitly from the table, invalid tuples are partitioned off in a different way avoiding duplicated effort in re-establishing the consistency of values across the search tree as it commonly happens with conventional GAC algorithms. The idea is similar to GAC4 [Mohr and Masini, 1988]: each element of a table is examined at most once along any path of the search tree.

The worst-case accumulated cost along a single path of length m in the search tree involving a positive r-ary table constraint containing t tuples is $O(rt + m)$ for STR3 whereas it is $O(rtm)$ for STR2. The space complexity of STR3 for a single table constraint is $O(rd + t)$ whereas the space complexity for STR2 is $O(r)$ (not counting space for the table representation itself).

Experiments in [Lecoutre et al., 2012] show that STR3 (which is greedy in memory space) is much faster than STR2 when the average size of the tables is not reduced drastically during search. However, where simple tabular reduction can eliminate many tuples from the tables that they become largely empty, STR2 is faster than STR3.

Christophe: à revoir

## 1.4 Singleton Arc Consistency(SAC)

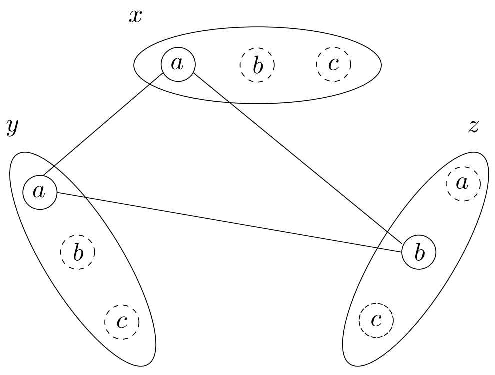
*Singleton arc consistency (SAC)* is a consistency stronger than arc consistency allowing to identify more inconsistent values before/during search. SAC is kind of looking one step in all directions (*Breadth-First Search* of level 1).

### 1.4.1 Definitions

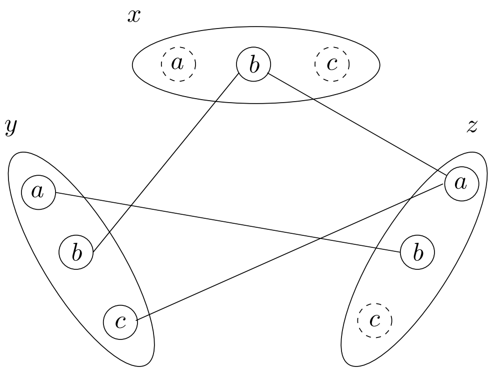
**Definition 21** (*Constraint network* **such that** $x = a$) *Let $P$ be a constraint network, a variable $x \in \mathscr{X}$ and a value $a \in dom(x)$, $P|_{x=a}$ is the constraint network obtained after reducing $dom(x)$ to $\{a\}$.*

**Definition 22** (*Singleton Arc-Consistent network*) *A constraint network $P$ is* singleton arc-consistent *iff $\forall x \in \mathscr{X}$, $\forall a \in dom(x)$, $GAC(P|_{x=a}) \neq \perp$.*

**Example 10** *We consider again the constraint network $P$ used in Example 8 (Figure 1.4). Figure 1.10 describes enforcing SAC on a constraint network $P$. Figure 1.10(a) is the obtained network after assigning $a$ to $x$. $dom(x)$ is, thus, reduced to only $a$. $P|_{x=a}$ is GAC, and, so $(x, a)$ is SAC. Figure 1.10(b) is the obtained network after assigning $b$ to $x$. $(z, a)$ is the only support of $(x, b)$ on $c_{xz}$ and $(y, b)$ is also the only one on $c_{xy}$. However ,$\{(y, b), (z, a)\} \notin rel(c_{yz})$. As a consequence, $GAC(P|_{x=b}) = \perp$ and $(x, b)$ is not SAC.*



(a) $P|_{x=a}$



(b) $P|_{x=b}$

Figure 1.10: Enforcing *Singleton Arc-Consistency* on $P$.

Singleton consistencies have received much attention: SAC1 [Debruyne and Bessiere, 1997], SAC2 [Bartak and Erben, 2004], SAC-Opt [Bessiere and Debruyne, 2004], SAC-SDS [Bessiere and Debruyne, 2005], SAC3 and SAC3+ [Lecoutre and Cardon, 2005]. Table 1.2 describes the time and space complexities of exploring consistencies on binary constraint networks of these different algorithms. For SAC3+ algorithm, the variable $b^+$ denotes the total number of times a branch is built or checked by the algorithm and $b_{max}$ denotes the maximum number of branches recorded at the same time by the algorithm.

| Algorithm | Time complexity | Space complexity |
|---|---|---|
| SAC1 | $O(en^2d^4)$ | $O(ed)$ |
| SAC2 | $O(en^2d^4)$ | $O(n^2d^2)$ |
| SAC-Opt | $O(end^3)$ | $O(end^2)$ |
| SAC-SDS | $O(end^4)$ | $O(n^2d^2)$ |
| SAC3 | $O(en^2d^4)$ | $O(ed)$ |
| SAC3+ | $O(b^+ed^2)$ | $O(ed + b_{max}nd)$ |

Table 1.2: The worst-case complexity of the *SAC* algorithms on binary constraint networks.

### 1.4.2 SAC1

SAC1 [Debruyne and Bessiere, 1997] is the first proposed algorithm in order to enforce singleton arc consistency. Algorithm 13 starts by enforcing GAC on the whole constraint network (Line 1). *SAC* algorithm iterates then over all literals $(x, a)$ such that $x \in \mathscr{X}$ and $a \in dom(x)$ (Line 6). If a literal is detected to be SAC-inconsistent, it is removed from its domain and GAC is enforced to propagate this removal (Line 8). If enforcing GAC returns an empty domain (Lines 2 and 9) $false$ is returned meaning that $P$ is proved to be SAC-inconsistent (SAC($P$) $=\perp$); otherwise *true* is returned.

**Algorithm 13:** SAC($P$: constraint network):Boolean

```
1  $P \leftarrow GAC(P)$                // GAC is initially enforced
2  if $P = \perp$ then
3      return $false$
4  repeat
5      $modified \leftarrow false$
6      foreach $(x, a)$ such that $x \in vars(P) \wedge a \in dom(x)$ do
7          if $GAC(P|_{x=a}) = \perp$ then
8              $P \leftarrow GAC(P|_{x\neq a})$   // $a$ is removed from $dom(x)$ and GAC is enforced
9              if $P = \perp$ then
10                 return $false$
11             $modified \leftarrow true$
12 until $\neg modified$
13 return $true$
```

SAC2 [Bartak and Erben, 2004], an optimized variant of SAC, is based on the fact that if $GAC(P|_{x=a}) \neq \perp$ then the literal $(x, a)$ remains SAC-consistent as long as its supports exists. SAC-Opt [Bessiere and Debruyne, 2004], unlike SAC1 and SAC2, doesn't restart checking values from scratch but uses the stored sub-problem for each SAC-consistent values. In order to avoid some useless tests, SAC-Opt requires a huge memory space especially on large constraints. Ensuring a trade-off between time and space complexity, SAC-SDS [Bessiere and Debruyne, 2005], an optimized variant, shares data structures required for establishing GAC on the different sub-problems. Unlike the previous variants, SAC3 [Lecoutre and Cardon, 2005] does not use a *Breadth-First Search* but builds fewer branches of greater length maintaining GAC at each step. In fact, a current branch is extended until a dead-end (a domain is wiped-out so it impossible to make a new assignment) is reached. SAC3+, also presented in [Lecoutre and Cardon, 2005], is an improvement associating with each branch a domain so it is easier to determine which previously built branch(s) must be reconsidered when a value is removed.

### 1.4.3 Weak k-SAC

Weak $k$-Singleton Arc Consistency [van Dongen, 2006] is a stronger form of SAC. It is equal to singleton arc consistency when $k = 1$ and stronger than SAC when $k > 1$. By analogy to Definition 17, we define *Weak k-SAC*.

**Definition 23** (*Weak k-Singleton Arc-Consistency*) *Let $P$ be a constraint network and $1 \leqslant k \leqslant n$ be an integer. A literal $(x, a)$ of $P$ is weakly k-SAC-consistent iff there exists an instantiation $I$ on a set of $k-1$ variables such that $x \notin scp(I)$ and $I$ is a valid instantiation such that $GAC(P|_{\{(x,a)\} \cup I}) \neq \bot$. A variable $x$ is weakly k-SAC-consistent iff $\forall\ a \in dom(x)$, $a$ is weakly k-SAC-consistent. $P$ is weakly k-SAC-consistent iff $\forall\ x \in \mathscr{X}$, $x$ is weakly k-SAC-consistent.*

Algorithm 14 illustrates the enforcement of weak $k$-SAC consistency on a constraint network $P$. Following SAC, weak k-SAC starts by enforcing GAC on the whole constraint network (Line 1). All literals $(x, a)$ are collected into a queue $Q_{wsac}$. Treated one by one, WSAC calls the function *extendable* to verify if it is possible to instanciate $k$ variables. If it is not the case, the literal $(x, a)$ is removed and the process continues with another literal.

## 1.5 Search

Several algorithms are proposed ensuring the "search" mechanism. We distinguish two major categories: *look-back* and *look-ahead* techniques.

**Algorithm 14:** WSAC($P$: constraint network, $k$: integer):Boolean

```
1  $P \leftarrow GAC(P)$            // GAC is initially enforced
2  if $P = \bot$ then
3      return $false$
4  $Q_{wsac} \leftarrow \{(x,a) | x \in vars(P) \wedge a \in dom(x)\}$
5  while $Q_{wsac} \neq \emptyset$ do
6      pick and delete $(x,a)$ from $Q_{wsac}$
7      if $\neg extendable(P|_{x=a}, k)$ then
8          $P \leftarrow GAC(P|_{x \neq a})$
                       // $a$ is removed from $dom(x)$ and GAC enforced
9          if $P = \bot$ then
10             return $false$
11         $Q_{wsac} \leftarrow \{(x,a) | x \in vars(P) \wedge a \in dom(x)\}$
12 return $true$
```

### 1.5.1 Look-back approaches

**Standard Backtrack (SBT)** *Backtrack* technique is usually used in order to solve constraint satisfaction problem instances. It extends, incrementally, a partial instantiation aiming to reach a possible solution. To do that, a depth-first search is performed assigning variables one by one. If a conflict is detected when assigning $x = a$, i.e. an instantiation $I$ violates a constraint $c$, another value of $x$ must be found (edge 1 in Figure 1.11). If there is no value in $dom(x)$ satisfying $\mathscr{C}$ of $P$, a chronological backtrack is performed inducing looking for another value for the previous variable (edge 2 in Figure 1.11). This process continues until all variables of the constraint network $P$ are assigned. In the other case, and after exploring the whole search space, we say that $P$ is "unsatisfiable".

There are also other methods using backtracking that differs from SBT. We distinguish:

- **Gaschnig-backjumping [Gaschnig, 1979]:** this approach uses backtracking in a different way. In fact, if for a variable $x$ there is no value from its domain that, added to a partial instantiation $I$, satisfies a constraint $c$ of $P$, the backtrack is not performed for the last assigned variable. This approach uses "culprit" variables, which are the recent assigned variables causing this incompatibility. As a consequence, Backtracking to the recent "culprit" variable, could lead to a solution.

- **Conflict-directed backjumping (CBJ)[Prosser, 1993]:** this approach avoids "Thrashing" which is exploring the same dead-ending sub-trees. Using nogoods, which are assignment sets that are not contained in any

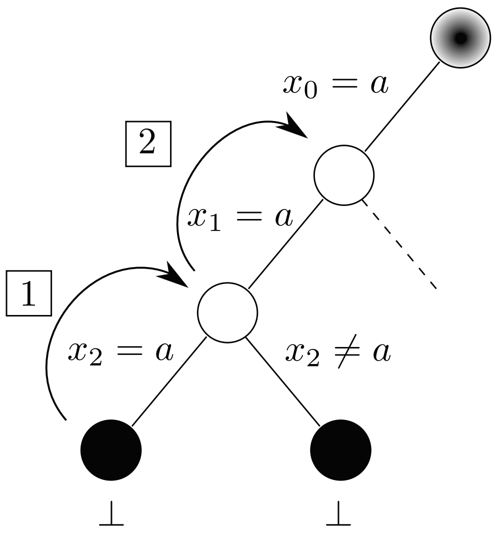


Figure 1.11: Possible standard backtracks on $P$.

solution, this technique avoids the "wrong" assignments that could lead to a failure. For each conflict, the set of nogoods leading to this conflict are stored, and thus, the backjumping is performed at the level of the recent decision leading to the conflict.

- **Dynamic Backtracking (DBT)[Ginsberg, 1993]:** Dynamic Backtracking uses a set of nogoods in order to detect the decision in charge of the last conflict. However, unlike the previous techniques, DBT removes the culprit decisions while maintaining the other decisions.

### 1.5.2 Look-ahead approaches

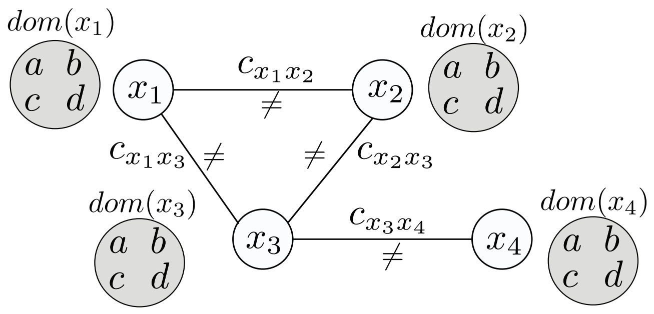
Unlike *look-back* approaches that look back on the made decisions in order to revise them and extend them to a solution, *look-ahead* approaches, introduced in [Haralick and Elliott, 1980], enhance exploring the search space by anticipating failures and reducing the search space. We distinguish the *Partial-Lookahead* and *Full-Lookahead*.

These inference techniques evaluate the impact of each decision on the domain of each unassigned variable. After each decision, a consistency is maintained on the constraint network in order to remove the inconsistent values. As a consequence, the search space is reduced and it contains only the consistent values avoiding by this filtering possible failures.

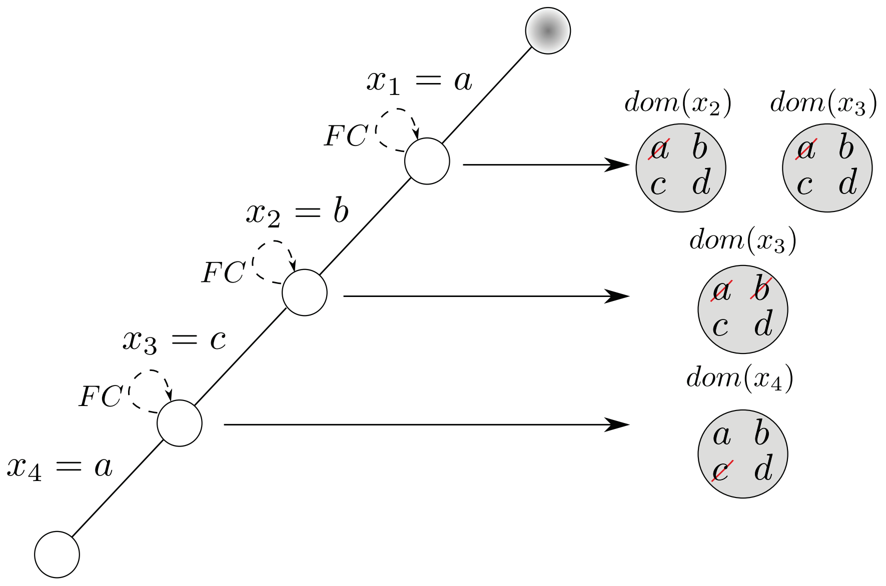
We present Forward Checking (FC), a Partial-Lookahead approach, introduced in [Haralick and Elliott, 1980].

**Forward Checking (FC)** After each decision, FC technique applies a partial arc-consistency form on the constraint network. Only the neighborhood (variables involved in the same constraints as the variable $x$) of the assigned variable is revised.

**Example 11** *Figure 1.12 illustrates FC approach. After the first decision $x_1 = a$, the domain of $x_1$ neighborhood is revised: the value $a$ is removed from $dom(x_2)$ and $dom(x_3)$ to respect the semantic of constraints $c_{x_1x_2}$ and $c_{x_1x_3}$. Assigning $x_2$ to $b$, induces the removal of the literal $(x_3, b)$. After the decision $x_3 = c$ ($c$ is one of the coherent values left in $dom(x_3)$), $dom(x_4)$ loses the value $c$.*



(a) A constraint network $P$



(b) FC applied after each decision

Figure 1.12: Forward Checking on $P$.

### 1.5.3 Maintaining Arc Consistency (MAC)

In order to enhance the effectiveness of search approaches, we may maintain the consistency of the constraint network associated to each node by applying one of the filtering algorithm (seen in Section 1.3 and 1.4). MAC [Sabin and Freuder, 1994] is the backtrack algorithm that maintains GAC during search. MAC is considered to be the most efficient complete approach to solve CSP instances.

**Algorithm 15:** MAC($P$: constraint network)

```
1  $consistent \leftarrow GAC(P)$          // GAC is initially enforced
2  if $\neg consistent$ then
3      return
4  $I \leftarrow \emptyset$
5  $finished \leftarrow false$
6  while $\neg finished$ do
7      pick a literal $(x, a)$ of $P$ such that $x \notin past(P)$
8      $I.push(x, a)$
9      $dom(x).reduceTo(a, |I|)$       // $x$ is assigned the value $a$
10     $consistent \leftarrow GAC(P)$
11     if $consistent \wedge |I| = n$ then
12         $print(I)$              // a solution has been found
13         $consistent \leftarrow false$          // inserted to keep looking for
           solutions
14     while $\neg consistent \wedge |I| \neq \emptyset$ do
15         $(x, a) \leftarrow I.pop()$
16         foreach variable $y \in vars(P) \setminus vars(I)$ do
17             $dom(y).restoreUpto(|I| + 1)$    // domains are restored
18         $dom(x).removeValue(a, |I|)$    // $a$ is removed from $dom(x)$
19         $consistent \leftarrow dom(x) \neq \emptyset \wedge GAC(P)$
20     if $\neg consistent$ then
21         $finished \leftarrow true$
```

Algorithm 15 enforces MAC on a constraint network $P$. GAC is initially enforced at each node (Line 1). MAC select a new unassigned variable to assign (Line 7), GAC is then enforced on $P$ (unlike FC). If after the assignment of $x$, $P$ is consistent and all variables are assigned, a solution is then found. Otherwise ($P$ is not GAC-consistent), a backtrack is performed (Lines 14 - 19) until $P$ becomes GAC-consistent. Inconsistent values are removed to avoid such failures in the future.

## 1.6 Heuristics

To guide a CSP instance solving, we may ensure at each node of the search tree that the "right" decisions are made. Algorithms presented in Section 1.5 don't propose any order in which the variable are assigned or the values are chosen. At a given node of search, the choice of a literal $(x, a)$ is determined by an heuristic that evaluates the "best" variable and the "best" value among all the possible candidates.

According to [Wallace, 2005], two principles are used in heuristics: "fail first" and "promise" principles. Usually, variable ordering heuristics use the "fail first" principle and "promise" principle for value ordering heuristics. For the variables heuristics, we distinguish three categories:

### 1.6.1 Static Variable Ordering (SVO)

For these heuristics, the order is defined before search from the problem structures. As a consequence, the same priority order is conserved during the search. For example:

- *lexico*: the variables are ordered by a lexicographical order of their names.
- *deg*: the variables are ordered by their degrees. A variable degree of variable corresponds to the number of constraints in which the variable is involved. We distinguish *maxDegree* (respectively *minDegree*) which corresponds to a decreasing order (respectively increasing order) of degrees.

### 1.6.2 Dynamic Variable Ordering (DVO)

These heuristics exploit different information on the problem current state in order to determine the most promising variable at a search node. For example:

- *ddeg*: Variables are ordered in a decreasing order of their "current" degree. In other words, *ddeg* chooses first the variable connected to the greatest number of unassigned variables.
- *dom/ddeg* [Bessiere and Régin, 1996]: This dynamic heuristic chooses first the variable having the smallest ratio:

$$\frac{current\ domain\ size}{ddeg} \tag{1.1}$$

### 1.6.3 Adaptive Variable Ordering (AVO)

AVO heuristics combine the dynamic and static information of the problem. They exploit information about past states of the search, which are not only about the current branch but also on the branches already explored.

- *wdeg*: Variables are ordered in a decreasing order of their "current" *weighted* degree. In fact, for each constraint corresponds a weight which is incremented each time it is violated during search. As a consequence, variables involved in the constraints that are often violated, have the highest priority.

- *dom/wdeg* [Boussemart et al., 2004]: This dynamic heuristic chooses first the variable having the smallest ratio:

$$\frac{current\ domain\ size}{wdeg} \tag{1.2}$$

- *Impact* [Refalo, 2004]: This heuristic is based on the impact of an assignment. In fact, the impact is evaluated with respect to the average space search reduction after an assignment. The variable impact is the sum of the assignments impacts of each value of its domain. This impact is maintained during search.

- *Counting-Based heuristic* [Pesant et al., 2012]: This heuristic aims to keep most of solutions when assigning variables. This is done by determining what proportion of solutions to each constraint agrees with that assignment.

- *Activity-Based heuristic* [Michel and Hentenryck, 2012]: This heuristic chooses variables based on their activity during propagation. In fact, for each variable $x$ corresponds a counter which measures its activity that corresponds to how often the domain of $x$ is reduced during the search. This counter is maintained during search.

# Chapter 2

# Compression and Parallel computing

## Contents

In this chapter, we introduce the state-of-the-art of both compression and parallel computing. This chapter is organized as follows:

- we describe several compression-based approaches for table constraints in Section 2.1;
- we describe parallel computing features and models and we present related parallel CSP approaches in Section 2.2.

## 2.1 Compression techniques

Table constraints are important for modeling parts of many problems, but they admit practical limitations because they are sometimes too large to be represented in a direct way. In order to reduce space and/or time complexity, researchers have focused on various forms of compression. In Section 2.1.1, we present several compact data structures used in the representation of table constraints. Moreover, in Section 2.1.2, we introduce some compression-based approaches using both compact representation and optimized filtering.

### 2.1.1 Compact representation of table constraints

Several compact data structures aim to reduce the space required to represent these constraints. We distinguish tries [Gent et al., 2007], Multi-valued Decision Diagrams (MDDs) [Cheng and Yap, 2010] and Deterministic Finite Automata (DFA) [Pesant, 2004]. These approaches use general structures to represent table constraints in a compact way, so as to facilitate the filtering process.

#### Tries

A trie [Fredkin, 1960], also called “prefix” tree, is an ordered data structure used to store and retrieve, usually, strings. All the children of a node have a common prefix which allows to represent a large set of values composing a language. The term *trie* comes from the word *re**trie**val*. In a trie, edges are labeled with a symbol of the language. As a result, to read a word, it is required to go through all the path starting from the root to the leaf or to the node ending this word. Each node contains some information. Values are associated with leaves or inner nodes ending a word. These values correspond to keys of interests such as frequency or weight. Seeking for a word in a trie is done in $O(k)$ time; where $k$ is the length of a search string.

**Example 12** *In Figure 2.1, nodes represent the word read form the root until that node and keys are listed for the nodes corresponding to a word form the English language (bold circles). Each complete English word has an arbitrary integer value associated with it.*

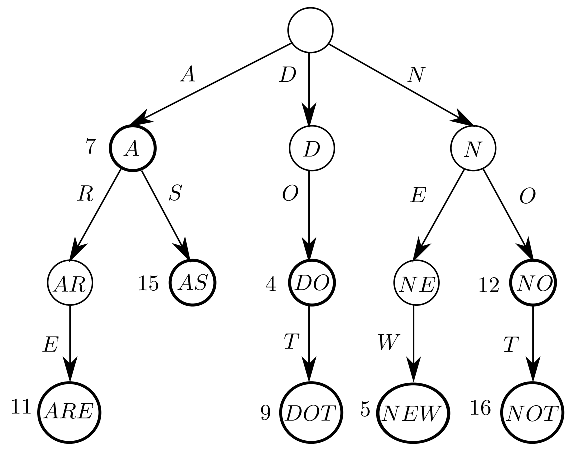


Figure 2.1: A trie of English words.

In [Gent et al., 2007] tries are first used to represent a table of *n-ary* constraint $c$. The different levels of the trie are the different variables in $scp(c)$. To each variable corresponds a level. The variables are assigned to levels in the same order in which they appear in the scope of the constraint. The language used to label the edges is the domain of the variables. All paths from the root have the same length since all tuples are a combination of the same number of values ($r$).

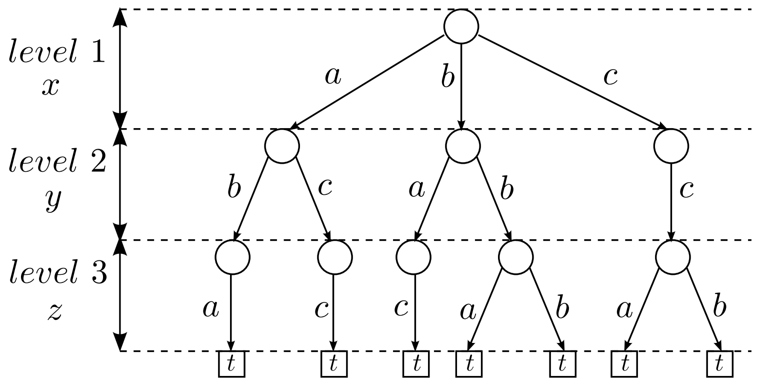
**Example 13** *Figure 2.2 illustrates the trie (Figure 2.2(b)) built from the positive table constraint $c_{xyz}$ defined in extension (Table 2.2(a)) as in [Gent et al., 2007]. The three variables involved in $c_{xyz}$ are the three levels of the trie. Tuples sharing the same first value(s) share the same first edge(s) in the trie. For example, tuples $\tau_1$ and $\tau_2$ have the same literal $(x, a)$ in $c_{xyz}$ which is represented by the edge labeled by $a$ in level 1. Moreover, tuples $\tau_6$ and $\tau_7$ have two literals in common which are $(x, c)$ and $(y, c)$. This is the reason that they share the edge labeled $c$ in the first level and its child edge labeled $c$ in the second level. The total number of edges of this* trie *is* 15*. In this figure, all paths are ending by a true node* $\boxed{t}$ *meaning that all illustrated tuples are allowed by the constraint.*

Since tries allow us to share prefixes which reduce space complexity, different tests of validity can be avoided at the first levels which may be advantageous to reduce time complexity. In some cases, one single validity test may prevent us from useless tests for a huge number of tuples, but this depends on the order of the scope of the constraint. However, in [Gent et al., 2007], authors used $r$ tries for each constraint; each one corresponds to a variable (the order of the scope is changed: the first level is dedicated to this variable); to speed up support seeking which penalizes the space complexity required to represent such constraints.

Algorithm 16 describes GAC enforcement on a trie. In fact, this algorithm looks for a valid support $\tau$ for a literal $(x, a)$ and returns it if it exists. $trie(x, a)$ is

| | $x$ | $y$ | $z$ |
|---|---|---|---|
| $\tau_1$ | $a$ | $b$ | $a$ |
| $\tau_2$ | $a$ | $c$ | $c$ |
| $\tau_3$ | $b$ | $a$ | $c$ |
| $\tau_4$ | $b$ | $b$ | $a$ |
| $\tau_5$ | $b$ | $b$ | $b$ |
| $\tau_6$ | $c$ | $c$ | $a$ |
| $\tau_7$ | $c$ | $c$ | $b$ |

(a) Positive table constraint $c_{xyz}$.



(b) Trie built from table constraint in 2.2(a).

Figure 2.2: A trie representation of a table constraint.

```
Algorithm 16: SeekValidSupportTrie($x$: variable, $a$: value): Tuple
  // $\tau$ is the valid support which is the output
1 $\tau[x] \leftarrow a$
2 return $extendSupport(trie(x, a), \tau)$
```

the root node of the sub-trie that contains all levels except the one corresponding to variable $x$. Algortihm 17 is a recursive function that performs a depth-first search (Lines 3 - 9). If this fuction returns *nil* (Line 10), there doesn't exit any support for the literal $(x, a)$ on the constraint represented by the trie. Otherwise, the support is returned at Line 2 when the search achieves a *true* node $\boxed{t}$ .

## Multi-valued Decision Diagrams

Using a trie data structure allows to discard prefix redundancy. Decision Diagrams allows to discard both prefix and suffix redundancy which is used in [Cheng and Yap, 2010, Cheng and Yap, 2006]. This reduces the size of represented data and also prevents from redundant tests especially in the lower levels

**Algorithm 17:** extendSupport(*node*: Node, $\tau$: Tuple): Tuple

```
1  if $node = \boxed{t}$ then
2      return $\tau$
3  $x \leftarrow node.variable$
4  foreach $child \in node.getChildren()$ do
5      if $child.value \in dom(x)$ then
6          $\tau[x] \leftarrow child.value$
7          $\tau' \leftarrow extendSupport(child, \tau)$
8          if $\tau' \neq nil$ then
9              return $\tau'$
10 return $nil$
```

which is the case for tries. A Multi-valued Decision Diagram (MDD) is an arc-labeled *Directed Acyclic Graph* (DAG). An MDD contains $n$ levels such that each level corresponds to a variable. Each edge is labeled by a symbol of the language represented by the MDD. An MDD is either the t-terminal (allowed values) or f-terminal (disallowed values). If all domains are binary the decision diagram is called "binary" (Binary Decision Diagram (BDD) [Bryant, 1986]).

**Example 14** *Figure 2.3 illustrates the t-terminal MDD corresponding to the positive table constraint $c_{xyz}$ defined in extension (Table 2.2(a)). The three variables involved in $c_{xyz}$ are the three levels of the* MDD. *Tuples sharing the same first value(s) or/and the same last value(s) have the same edge(s) in the depicted* MDD. *For example, tuples $\tau_2$ and $\tau_3$ have the same suffix $(z, c)$ that is why they share the same edge labeled $c$ in the level* 3. *Tuples $\tau_4$, $\tau_5$, $\tau_6$ and $\tau_7$ end either by $(z, a)$ or $(z, b)$ which is translated in* MDD *by the two shared edges labeled $a$ and $b$ in the third level. The total number of edges of this* MDD *is* 12.

Algorithm 18 describes *mddc* algorithm. This algorithm doesn't use the revision-based technique meaning that rather than looking for a support for each value, GAC is enforced globally by discarding all GAC-inconsistent values contained in the MDD starting each time from scratch. This may supplant enforcing GAC on tries. In fact, *mddc* keeps for each variable $x_i$ a set $S_i$ of unsupported values of its domain. mddcSeekSupports algorithm goes through all nodes of MDD using a Depth First Search (DFS) for traversing the MDD from the root node and updates the sets $S_i$: two main sets are maintained $\xi^{YES}$ for visited and consistent edges (having a positive terminating node: t-terminal), and $\xi^{NO}$ for inconsistent edges due to pruned values (having a negative terminating node: f-terminal). Since each edge represents a value, after doing a DFS, the values that have not been kept are removed since there is no longer any path from the root

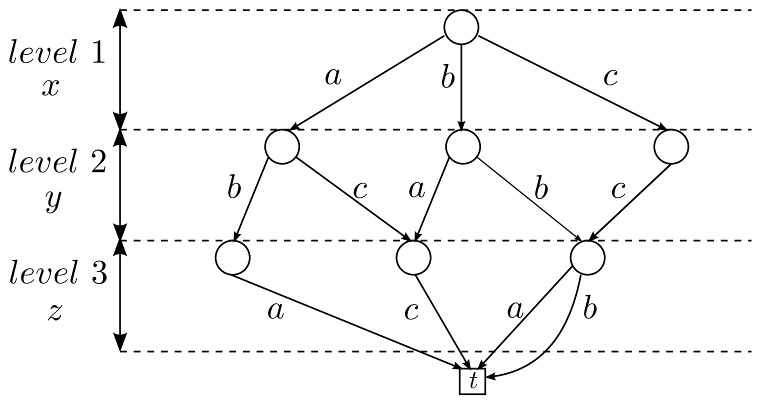


Figure 2.3: MDD built from the table constraint in Figure 2.2(a).

**Algorithm 18:** mddc($G$: MDD root, $st$: state): (boolean, state)

```
1  $\xi^{YES} \leftarrow \emptyset$
2  $restore(\xi^{NO}, st)$
3  foreach $i \in [1..r]$ do
4      $S_i \leftarrow dom(x_i)$          // values that have no support yet
5  $\Delta \leftarrow r + 1$
6  $mddcSeekSupports(G, 1)$
7  foreach $i \in [1..\Delta - 1]$ do
8      $dom(x_i) \leftarrow dom(x_i) \setminus S_i$
9  $st' \leftarrow state(\xi^{NO})$
   // Maintaining GAC during search
10 if $\exists S_i \in \mathscr{S}$ such that $S_i \neq \emptyset$ then
11     return(YES, st')              // domains have changed
12 else
13     return(No, st')
```

to a t-terminal node corresponding to this value. During search some additional data structures and optimization are used to achieve efficient operations (Lines 1, 2 and 9).

Clearly, in terms of space complexity using MDDs is more advantageous than using trie knowing that, in [Gent et al., 2007], for each constraint $r$ tries are required but just one MDD is enough.

In [Perez and Régin, 2014], the authors present MDD4 algorithm, which adapts GAC4 [Mohr and Masini, 1988] algorithm to MDD constraints. The two main improvements are:

- the way MDD is maintained during search: Tuples are not represented explicitly in an MDD constraint. They are the different paths from the root to a positive terminal. To enforce GAC, *mddc* checks each time it goes through an edge if its value is still valid. MDD4 algorithm, proposes to delete the inconsistent values, thus its corresponding edges in order to reduce these redundant tests;

- the way $S$ lists are maintained: MDD4 establishes a relation between the values in domains and their corresponding edges of the MDD through $S$ lists. Hence, for each literal $(x, a)$, $S(x, a)$ contains the set of edges in the MDD corresponding to this value (labeled with the value $a$ in the level $x$). If $S(x, a)$ becomes empty that means that there is no more an edge for this value and then $dom(x)$ can be reduced by removing $a$.

### Deterministic Finite Automata

A *deterministic finite automata* (DFA) $M$ is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of states;
- $\Sigma$ is a finite set of input symbols called the alphabet;
- $\delta$ is a transition function $(\delta : Q \times \Sigma \rightarrow Q)$;
- $q_0$ is the initial state $(q_0 \in Q)$;
- $F$ is a set of final states $(F \subseteq Q)$.

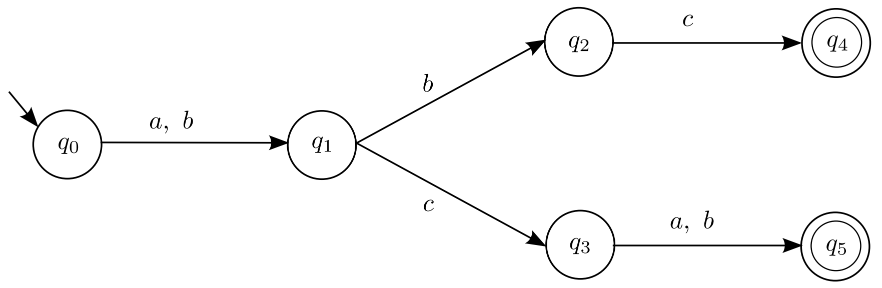
DFA is a deterministic finite state machine which accepts/refuses a finite set of symbols contained in the alphabet $\Sigma$. Starting from the initial state $q_0$, the state machine transition for each symbol, considering the transition function $\delta$, from a state to another. If one of the final states included in $F$ is reached, the input word is said "accepted" by the state machine. As a consequence, the set of accepted words compose the "regular" language recognized by $M$ and denoted $L(M)$.

**Example 15** *Figure 2.4 illustrates the DFA representation of the positive table constraint $c_{xyz}$ defined in extension (table in Figure 2.4(a)). The regular expression representing* table$[c_{xyz}]$ *is* $(a + b)(bc + c(a + b))$.

DFA is used in [Pesant, 2004] to represent a global constraint called *regular* which is a kind of global constraint generalizing the stretch constraint introduced in [Pesant, 2001]. This type of constraint is used in rostering problem where every team of a working group has an identical schedule as the others but it is out of phase. A regular expression can represent a valid sequence of activities. However, for the *allDifferent* global constraint, the number of states of the associated DFA is exponential in the number of symbols in the alphabet $(|Q| = 2^{|\Sigma|})$.

| | $x$ | $y$ | $z$ |
|---|---|---|---|
| $\tau_1$ | $a$ | $c$ | $a$ |
| $\tau_2$ | $a$ | $b$ | $c$ |
| $\tau_3$ | $a$ | $c$ | $b$ |
| $\tau_4$ | $b$ | $b$ | $c$ |
| $\tau_5$ | $b$ | $c$ | $a$ |
| $\tau_6$ | $b$ | $c$ | $b$ |

(a) Positive table constraint $c_{xyz}$.



(b) DFA.

Figure 2.4: DFA representation of the table constraint in Figure 2.4(a).

The filtering algorithm used to enforce GAC (presented in [Pesant, 2004]) constructs, in a first stage, a directed multi-graph, and then, collects the set of states that support a value $(x, a)$. In [Tiedemann et al., 2007], the authors prove that MDD reduction gives smaller and more efficient graphs in terms of propagation time than straight DFA unfolding used for the regular constraint.

## 2.1.2 Compression-based approaches

Tries [Gent et al., 2007], Multi-valued Decision Diagrams (MDDs) [Cheng and Yap, 2010] and Deterministic Finite Automata (DFA) [Pesant, 2004] are such general structures, which are used to represent table constraints compactly, so as to speedup filtering. These data structures were introduced previously in Section 2.1.1.

Several approaches propose different compressed representations of table constraints and different ways to filter them. Compressed tuples [Hubbe and Freuder, 1992], Short supports [Nightingale et al., 2011, Nightingale et al., 2013] and Smart tuples [Mairy et al., 2015] present new definitions of supports that differ from the classic ones. Data-mining based approaches [Jabbour et al., 2013a, Jabbour et al., 2013b] compress table constraints giving another representation using data mining features. In this section, we introduce four different categories of compressed tables:

- Short Supports;
- Cartesian-product based approaches;
- Smart tuples;
- Data-mining based approaches.

In each category, we present an approach that enforces GAC using STR algorithm.

### Short Supports

As presented in Definition 10 in Chapter 1, a literal $(x, a)$ is valid if there exists a valid tuple $\tau$ in the table constraint $c$ such that $\tau[x] = a$. Based on this idea, [Nightingale et al., 2011, Nightingale et al., 2013] proposes a new representation of table constraints in which for each literan $(x, a)$ corresponds the set of its supporting tuples. This form is obtained before search.

In fact, a short support (i.e. tuple) is a generalization of a support tuple defined in Definition 10.

**Definition 24** *A short support for constraint c is a valid tuple s built on a subset of scp(c). Whereas, when a support is built on scp(c), it is called a full-length support.*

**Example 16** *Let us consider the* Element *[Colton and Miguel, 2001] constraint* $z = x_k$ *such that* $dom(x_0) = dom(x_1) = dom(x_2) = dom(z) \in \{a, b, c\}$ *and* $dom(k) \in \{0, 1, 2\}$. *This constraint is satisfied when the* $k^{th}$ *variable* $x_k$ *takes the same value as the variable* $z$. $s = \{(x_1, a), (k, 1), (z, a)\}$ *is a short support satisfying this constraint. s does not contain a value for each variable, which is the case for variables* $x_0$ *and* $x_2$. *As a consequence, any extension of s with valid values for* $x_0$ *and* $x_2$ *is a support such as* $fs = \{(x_1, a), (k, 1), (z, a), (x_0, b), (x_2, c)\}$.

In [Nightingale et al., 2013], authors use a new symbol to represent the missing variables. In fact, considering a constraint $c$ such that $scp(c) = \{x_1, x_2, x_3, x_4\}$ and a short support (tuple) $\tau = \{(x_1, a), (x_2, b)\}$, $\tau$ is represented as $\tau = \{a, b, *, *\}$ where $*$ indicates that variables $x_3, x_4$ are not mentioned in $\tau$.

SHORTGAC is the algorithm that enforces GAC using short supports. SHORTGAC maintains a set of short supports supporting all the valid values of the variables involved in the scope of this constraint. Classically, enforcing GAC on table constraints means looking for supports for unsupported values. SHORTGAC looks for support in two different ways. First, as for classic tuples, SHORTGAC assumes that a value $(x, a)$ is supported if there exists a short support $s$ such that $s[x] = a$. Second, SHORTGAC considers that a value $(x, a)$ is supported by a short support $s$ if $s$ does not involve $x$, i. e. $\forall v \in dom(x), (x, v) \notin s$.

In [Nightingale et al., 2011], three main data structures are used in SHORTGAC algorithm:

- *numSupports*: the total number of valid supports.
- $supportsPerVar[x]$: an array indicating the number of supports supporting each variable $x$.
- $supportsPerLit[x][a]$: a 2-dimensional array indicating the set short supports containing the literal $(x, a)$.

SHORTGAC algorithm considers that each variable $x$ for which $supportsPerVar[x] < numSupport$ is *fully supported* and there is no need to look for supports for this variable. In fact, $supportsPerVar[x] < numSupport$ for a variable $x$ means that there exists a support $s$ that does not involve $x$. As a consequence, $s$ is a support for all values in $dom(x)$ and so, $x$ is supported and the algorithm does not have to look for supports for the literals $(x, a)$. Otherwise, in the case where $supportsPerVar[x] = numSupport$ meaning that $x$ is involved in all short supports, SHORTGAC seeks for a support for each value $a \in dom(x)$ for which $|supportsPerLit[x][a]|$=0 (i.e. the value has no support).

**Example 17** *Let us consider a table constraint $c_{x_1x_2x_3}$ such that the current domains of the variables involved in $scp(c_{x_1x_2x_3})$ are: $dom(x_1) = dom(x_3) = \{0, 1, 2\}$ and $dom(x_2) = \{0, 1\}$. $table(c_{x_1x_2x_3})$ contains two short supports $s_0 = \{(x_1, 0), (x_2, 1)\}$ and $s_1 = \{(x_1, 1), (x_3, 0)\}$. Hence, $numSupports = 2$ as depicted in Figure 2.5(c). The data structure supportsPerLit is depicted in Figure 2.5(a) where for each value $(x, a)$ corresponds the list of short supports involving it. For example, for the variable $x_3$ only the value 0 is involved in a support of $table(c_{x_1x_2x_3})$. The two remaining values are not contained in any short support. The symbol $\times$ indicates that a value $(x, a)$ is not valid, which is the case for the value $(x_2, 2)$. The $supportsPerVar$ data structure (Figure 2.5(b)) indicates the number of short supports supporting each variable x which is the sum of* supportsPerLit[x][a] *for each value $a \in dom(x)$. In our case,* supportsPerVar[$x_2$] = supportsPerVar[$x_3$]=1 *which is less than numSupports meaning that both $x_2$ and $x_3$ are fully supported. We have* supportsPerVar[$x_1$]=*numSupports, so we have to look for a support for $(x_1, 2)$ since $|supportsPerLit[x_1][2]|$=0. $table(c_{x_1x_2x_3})$ does not contain any support for $(x_1, 2)$ so this value is removed and* supportsPerLit[$x_1$][2] *is updated to $\times$.*

Experimental results show that in three case studies (Element [Colton and Miguel, 2001], Lex-ordering [Frisch et al., 2002] and Rectangle Packing [Simonis and O'Sullivan, 2008]), SHORTGAC is faster than other methods such as GAC-Schema.

| | $x_1$ | $x_2$ | $x_3$ |
|---|---|---|---|
| 0 | $\{s_0\}$ | $\emptyset$ | $\{s_1\}$ |
| 1 | $\{s_1\}$ | $\{s_0\}$ | $\emptyset$ |
| 2 | $\emptyset$ | $\times$ | $\emptyset$ |

(a) *supportsPerLit*

| $x_1$ | $x_2$ | $x_3$ |
|---|---|---|
| 2 | 1 | 1 |

(b) *supportsPerVar*

| |
|---|
| 2 |

(c) *numSupports*

Figure 2.5: SHORTGAC data structures.

Interestingly, Simple Tabular Reduction technique is used in [Nightingale et al., 2013] to enforce GAC on short supports (SHORTSTR2). The main feature is that when a short support does not involve some variable $x$, it supports all values of $x$, this is called *implicit support*. SHORTSTR2 is a slight modified version of STR2 taking into consideration this new form of a tuple. Experimental results do not show a huge compression ratio, but a GAC algorithm that competes with STR2.

### Cartesian-Product based approaches

Cartesian product is another classical mechanism to represent compactly large sets of tuples. For instance, it has been applied successfully for handling sets of solutions [Hubbe and Freuder, 1992, Régin, 2011b], symmetry breaking [Focacci and Milano, 2001, Fahle et al., 2001], and learning [Katsirelos and Bacchus, 2005, Lecoutre et al., 2007]. So far, this form of compression has been used in two distinct GAC algorithms for table constraints: by revisiting the general GAC-schema [Katsirelos and Walsh, 2007] and by combining compressed tuples with STR [Xia and Yap, 2013]. In fact, rather than listing allowed tuples in $rel(c)$, Cartesian products are used (See Definition 2)

**Example 18** *We give an example of a relation $rel(c)$ defined over three variables $x$, $y$ and $z$ with $dom(x) = dom(y) = dom(z) = \{a, b\}$.*

$$rel(c) = \left\{ \begin{array}{c} (b,a,a), \\ (b,a,b), \\ (b,b,a), \\ (b,b,b) \end{array} \right\}$$

*Using Cartesian product $rel(c)$ could be represented by one compressed tuple as follows:*

$$rel(c) = \{\ (\{b\},\{a,b\},\{a,b\})\ \}$$

In [Xia and Yap, 2013], Cartesian product is used to compress tuples, called c-tuples, in table constraints. Figure 2.6 illustrates the table constraint representation using c-tuples. The first eight tuples are compressed in $\tau_{c0}$ as a Cartesian product listing the possible values that could be taken by each variable.

|  | $x_1$ | $x_2$ | $x_3$ | $x_4$ |
|---|---|---|---|---|
| $\tau_0$ | $a$ | $a$ | $a$ | $a$ |
| $\tau_1$ | $a$ | $a$ | $a$ | $b$ |
| $\tau_2$ | $a$ | $a$ | $b$ | $a$ |
| $\tau_3$ | $a$ | $a$ | $b$ | $b$ |
| $\tau_4$ | $a$ | $b$ | $a$ | $a$ |
| $\tau_5$ | $a$ | $b$ | $a$ | $b$ |
| $\tau_6$ | $a$ | $b$ | $b$ | $a$ |
| $\tau_7$ | $a$ | $b$ | $b$ | $b$ |
| $\tau_8$ | $c$ | $c$ | $c$ | $c$ |

(a) table($c_{x_1x_2x_3x_4}$).

|  | $x_1$ | $x_2$ | $x_3$ | $x_4$ |
|---|---|---|---|---|
| $\tau_{c0}$ | $\{a\}$ | $\{a,b\}$ | $\{a,b\}$ | $\{a,b\}$ |
| $\tau_{c1}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ | $\{c\}$ |

(b) table($c_{x_1x_2x_3x_4}$) for STR2-C.

Figure 2.6: c-tuple representation.

To enforce GAC on such tables, authors proposed two algorithms STR2-C (compressed tuples STR2 variant) and STR3-C based on STR2 and STR3 variants. We only illustrate the STR2-C algorithm. The validity of a c-tuple [Focacci and Milano, 2001, Katsirelos and Walsh, 2007] is defined as: a c-tuple is valid if the Cartesian product contains at least one valid tuple. As a consequence, unlike STR2 which considers that all values belonging to a valid tuple are GAC consistent, the valid c-tuple, in STR2-C, may contain GAC-inconsistent values which implies more consistency tests for that c-tuple. That is the reason why, a pointer is used to separate the inconsistent values from the unchecked ones for each variable (similar to *currentLimit* pointer).

Let us consider the example in Figure 2.6 and assuming that the value $(x_4, a)$ is removed, $\tau_0$,$\tau_2$, $\tau_4$ and $\tau_6$ (in Figure 2.6(a)) are no more valid. However, in the compressed table (in Figure 2.6(b)) using c-tuple, we cannot remove tuple $\tau_{c0}$ since it contains tuples $\tau_1$, $\tau_3$, $\tau_5$ and $\tau_7$ that are still valid which implies the validity of $\tau_{c0}$.

Experimental results show that STR2-C and STR3-C are competitive with STR variants when the table can be compressed enough to outweigh the additional costs. However, encoding table constraints as a set of c-tuples remains difficult since it is not done in a natural way.

### Smart Tables

Smart table constraints [Mairy et al., 2015] are a new compact representation of table constraints using arithmetic constraints which reduce significantly the memory space required to encode such combination of values.

**Example 19** *Let us consider the table constraint $c_{x_1x_2x_3}$ depicted in Figure 2.7(a) such that dom($x_1$)=dom($x_2$)=dom($x_3$)=$\{0, 1, 2\}$. This table constraint could be represented using arithmetic constraints as in the Smart table constraint depicted in Table 2.7(b). In fact, in all tuples of $c_{x_1x_2x_3}$ the value of the variable $x_1$ is always equals to the value of the variable $x_3$. This explains the first arithmetic constraint $x_1 = x_3$ in $sc_{x_1x_2x_3}$. All the values taken by the variable $x_2$ are less or equal to 1 ($x_2 \leq 1$). The symbol $*$ taken by the variable $x_3$ is the same used in [Nightingale et al., 2013].*

| | $x_1$ | $x_2$ | $x_3$ |
|---|---|---|---|
| $\tau_0$ | 0 | 0 | 0 |
| $\tau_1$ | 1 | 0 | 1 |
| $\tau_2$ | 2 | 0 | 2 |
| $\tau_3$ | 0 | 1 | 0 |
| $\tau_4$ | 1 | 1 | 1 |
| $\tau_5$ | 2 | 1 | 2 |

(a) Table Constraint $c_{x_1x_2x_3}$.

| | $x_1$ | $x_2$ | $x_3$ |
|---|---|---|---|
| $\tau_0$ | $= x_3$ | $\leq 1$ | $*$ |

(b) Smart Table Constraint $sc_{x_1x_2x_3}$.

Figure 2.7: Smart table representation.

Smart table constraints could be considered a disjunction of conjunctions of basic arithmetic constraints since each smart tuple contains a conjunction of basic arithmetic constraints. Smart table constraints are an interesting tool to express different global constraints.

The filtering algorithm used for smart table constraints, called smartSTR, is inspired from STR. In fact, we associate for each smart tuple *sc* a sub-problem $P_{sc}$ which is represented by tree-like structure (forest of trees) collecting all valid values supporting the arithmetic constraints of *sc*. smartSTR checks sequentially all the valid smart tuples. However, the only difference between smartSTR and the classic STR is the way that valid values (i. e. supports) set is maintained. Since the supported values in smart tuples are stored in trees, a smart tuple is considered valid iff each tree (corresponding to each arithmetic constraint) has at least one solution.

Experimental results show an important spatial and time complexity reduction using smart table constraints instead of classic table constraints. These results competes with SHORTGAC and outperforms it especially when table constraints are larger.

Short supports, compressed tuples and smart tuples introduced in the previous sections are a generalization of classical tuples in tables of constraints.

These new representations don't offer the same fluency in the expression of such constraints especially to non-expert users. C-tuples and short supports need a compression algorithm to transform a classical table constraint. However, smart table offers a facility of expression of constraints. In fact, it can be considered a tool that combines intentional, extensional and global constraints. It is also possible to use C-tuples and short supports as smart tuples. For example, a short support $\{a, b, *, *\}$ is also a smart tuple since both techniques use the "*" symbol. A c-tuple can be represented as a smart tuple as follows: $\tau_{c0} = \{\{a\}, \{a, b\}, \{a, b\}, \{a, b\}\}$ is written as $sc = \{\in \{a\}, \in \{a, b\}, \in \{a, b\}, \in \{a, b\}\}$.

**Data-mining based approaches**

Other techniques of compression based on data-mining techniques are used in order to reduce the size of table constraints. We first introduce the different features used in data mining that are used in our contribution in Chapter 3 and then we explore the related works using this technique.

**Data-mining features** Data mining is the process of discovering interesting knowledge from a large amount of data. Since the development of computer science, there has been an information explosion, which is increasing in a fast way. The problem was not any more a simple problem of storage of information, but rather the way to analyze this important heap of data, which led to the development of data mining techniques (since the beginning of 1990s). That is why the main objective of data mining consists in looking for and extracting "useful" information from a big amount of data stored in databases or warehouses.

In the field of data mining, a pattern is composed of a set of items, called **itemset**. This term comes from the domain of the supermarkets systems (the market basket) [Agrawal et al., 1993] which was the source of inspiration for the domain of data mining. In the analysis of a basket, the data consist in a number of purchases realized by the customers. Every transaction consists of a number of articles, which a customer bought together. In Data mining, a pattern could be of different types: a word, a sentence, a sequence of molecules or nucleotides in the case of looking for a sequence DNA [Witten and Frank, 2005], etc. In the context of our contribution, a pattern is a set of values taken by variables.

**Definition 25 (Transactional Database)** *Let $I = \{a_1, a_2, \ldots, a_m\}$ be a set of* ***items*** *such as products (Juice, Bread, Milk, ... , and so on). DB is a set of transactional database $DB = \langle T_1, T_2, \ldots, T_n \rangle$ where each transaction $T_i$ is a set of items such that $T_i \subseteq I$. Each transaction is associated with a unique identifier, transaction identifier (TID)*

The concept of *frequent itemset* was introduced for transactions databases.

**Definition 26 (Frequent pattern)** *Let $I = \{a_1, a_2, \ldots, a_m\}$ be a set of* ***items****, and a transaction database $DB$. The* ***support count*** *$\sigma$ (or occurrence frequency) of a pattern $\mu$, which is a set of items, is the number of transactions containing $\mu$ in $DB$. The* ***support s*** *of a pattern $\mu$ is the fraction of transactions that contains $\mu$, $s(\mu) = \sigma(\mu)/|DB|$.*
*$\mu$ is a* ***frequent pattern*** *if $\mu$'s support is greater than a predefined minimum support threshold minSup such that $minSup \in [0,1]$.*

A *k-itemset* $\mu$ consists of $k$ items of $I$. The set of frequent patterns is defined:

$$F(DB, min\_sup) = \{\mu \subseteq I | s(\mu) \geq min\_sup\} \tag{2.1}$$

**Example 20** *Let us consider the transaction database depicted in Table 2.1, which represents different market baskets and their contents:*

| *TID* | *Items* |
|---|---|
| 1 | Bread, Milk |
| 2 | Bread, Cookies, Juice, Eggs |
| 3 | Milk, Cookies, Juice, Coke |
| 4 | Bread, Milk, Cookies, Juice |
| 5 | Bread, Milk, Cookies, Coke |

Table 2.1: Transaction database of different market baskets.

*As said before, an itemset is a collection of one or more items. {Bread, Milk, Cookies} is a 3-itemset.*
***Support count ($\sigma$)****: $\sigma$({Bread, Milk, Cookies})=2, since this itemset appears two times in transaction database.*
***Support****: s({Bread, Milk, Cookies})=2/5 represents the fraction of transactions that contains this itemset.*
*Let minSup be equal to 2/5, the pattern {Bread, Milk, Cookies} is frequent since $s(\{Bread, Milk, Cookies\}) \geq minSup$. For simplicity, we use the support terminology to represent a* ***support count ($\sigma$)*** *in the next examples. It is important to note that the term support refers, in data mining context, to the frequency of a pattern which is different from the one used in a CSP context.*

Several data mining algorithms, such as Apriori [Agrawal and Srikant, 1994] and Frequent-Pattern Growth (FP-Growth) [Han et al., 2000] among others (See more in [Wu et al., 2008]), can be used to identify frequent patterns. We are only introducing FP-Growth [Han et al., 2000, Han et al., 2004] algorithm since we use it in our contribution (Chapter 3). This choice is made since FP-Growth seems to be the suitable algorithm to use for our compression process since it offers the best frequent patterns detection in terms of space and time. In fact, FP-Growth constructs a Frequent Pattern Tree (FP-Tree) from a transaction database, which satisfy the minimum support (*minSup*) [Pramod and Vyas, 2010]. This algorithm does not use the candidate generation technique, which is a feature that consists in generating $k$-itemsets from $(k-1)$-itemsets, and due to the compact structure used it does not require a huge memory space. Moreover, this algorithm requires only *two* scans of the transaction database.

The FP-Growth algorithm is composed of two main steps:

**Step 1: FP-Tree construction** This step consists of building the compact data structure called FP-Tree. This construction is done using one pass over the data-set. The algorithm:

1. detects the frequent items;
2. orders the transactions with respect to a decreasing order based on the items frequencies;
3. removes the infrequent items taking into consideration the minimal support threshold fixed.

**Example 21** *Let us consider a transaction database DB presented in Figure 2.8(a):*

| *TID* | *Items* |
|---|---|
| 1 | {B, A} |
| 2 | {D, C, B} |
| 3 | {A, C} |
| 4 | {A, D, B} |
| 5 | {A, B, C} |
| 6 | {D, B} |
| 7 | {A} |
| 8 | {C, A, B} |
| 9 | {D, B, A} |
| 10 | {E, B, C} |

(a) A data-set.

| *TID* | *Items* |
|---|---|
| 1 | {B, A} |
| 2 | {B, C, D} |
| 3 | {A, C} |
| 4 | {B, A, D} |
| 5 | {B, A, C} |
| 6 | {B, D} |
| 7 | {A} |
| 8 | {B, A, C} |
| 9 | {B, A, D} |
| 10 | {B, C, E} |

(b) An ordered data-set.

| *TID* | *Items* |
|---|---|
| 1 | {B, A} |
| 2 | {B, C} |
| 3 | {A, C} |
| 4 | {B, A} |
| 5 | {B, A, C} |
| 6 | {B} |
| 7 | {A} |
| 8 | {B, A, C} |
| 9 | {B, A} |
| 10 | {B, C} |

(c) A frequent data-set.

Figure 2.8: Transaction database evolution.

1. *calculating support: s(B)=8; s(A)=7; s(C)=5; s(D)=4; s(E)=1.*

2. *ordering itemsets: all transactions in the Figure 2.8(b) are ordered taking into account a decreasing order of items support.*

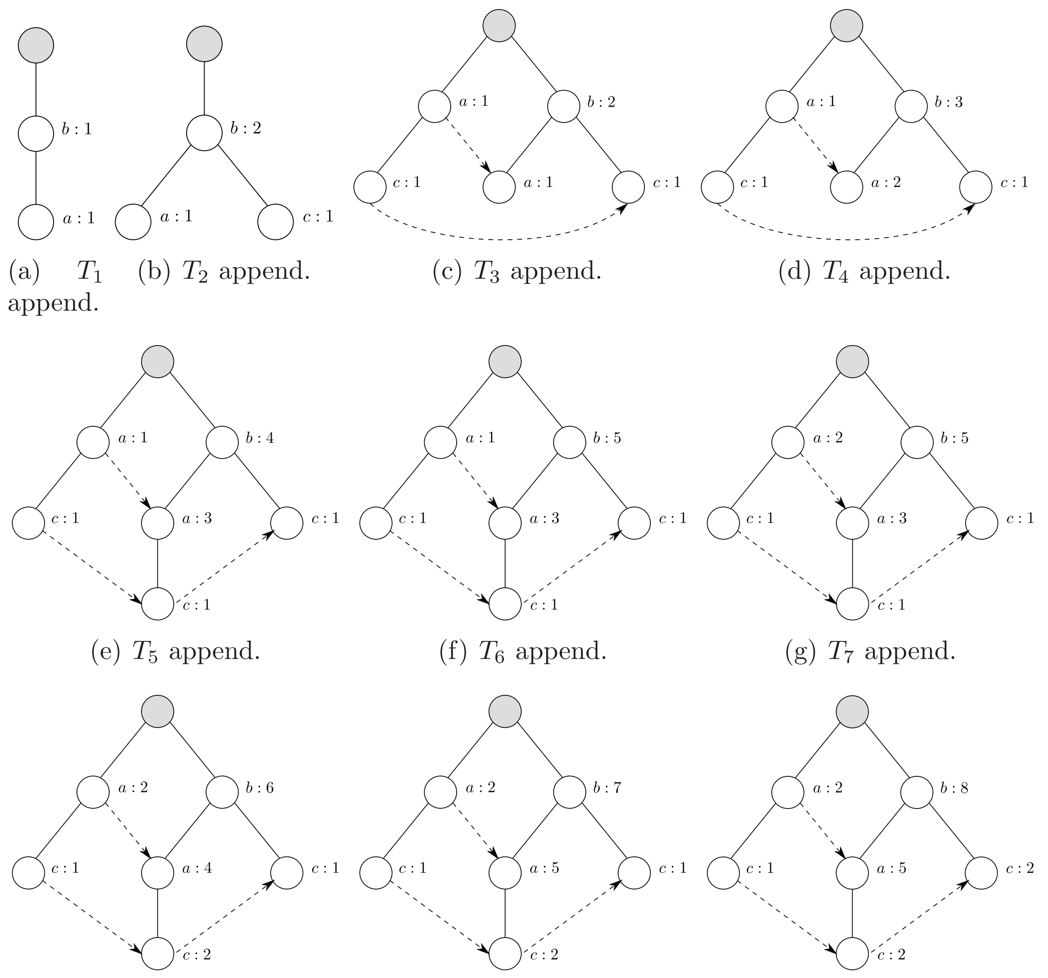
3. *removing infrequent items: if we consider minSup=*5*, we have to remove items D and E since they are not frequent. Figure 2.8(c) illustrates the final result.*

After ordering the transaction database and removing from it infrequent items, the algorithm builds an FP-Tree using one pass over the transaction database. An FP-Tree is, in fact, a trie (see Section 2.1.1) where each branch represents the frequent part of a transaction. Each node of an FP-Tree contains the number of branches which share that node, and labeled with the concerned item. The root node does not have any label. FP-Growth reads one transaction at a time and maps it to a path: a path, from the root, composed of the frequent items of the transaction is added to the FP-Tree. The decreasing order of support is used. Paths can overlap when a transaction $T_i$ shares the same items as a *prefix* with a transaction $T_j$. In this case, the nodes counters are incremented. The more paths are overlapped, the higher the compression is. The nodes labeled with the same value are linked with a dashed link. This is used in the patterns extraction step.

The FP-Tree allows us to obtain a smaller size of data than the uncompressed data especially when many transactions share several items. The size of the tree depends on how the items are ordered. Data-mining techniques experts usually use a decreasing order based on support, but it does not always lead to the smallest tree [Han et al., 2004].

**Example 22** *Figure 2.9 illustrates the different step of the FP-Tree construction. In Figure 2.9(a), we add a path corresponding to transaction* 1 *of the database. In Figure 2.9(b), transaction* 2 *shares prefix "b" with the path added previously which explains the fact that the two paths share node "b". The counter of this node is, thus, incremented. In Figure 2.9(c), we add the path corresponding to transaction* 3*. The value "a" already exists in the FP-Tree, that is why the two nodes labeled with the value "a" are linked with a dashed link. This is the case for the nodes labeled with the value "c". Adding transaction* 4*, increments the two nodes "b" and "a" since the path exists already (Figure 2.9(d)). Transaction* 5 *adds the node "c" at the end of the path* $b \rightarrow a$ *while incrementing the nodes counters (Figure 2.9(e)). All the remaining transactions correspond to existing path (see Figures 2.9(f), 2.9(g), 2.9(h), 2.9(i) and 2.9(j)). Figure 2.9(j) depicts the final FP-Tree obtained at the end of step* 1.

**Step 2: Frequent Itemset Generation** This step consists in extracting the frequent itemsets from the FP-tree. This extraction is done using a bottom-up algorithm, which detects frequent itemsets starting from leaves to the root.



(a) $T_1$ append. (b) $T_2$ append. (c) $T_3$ append. (d) $T_4$ append.

(e) $T_5$ append. (f) $T_6$ append. (g) $T_7$ append.

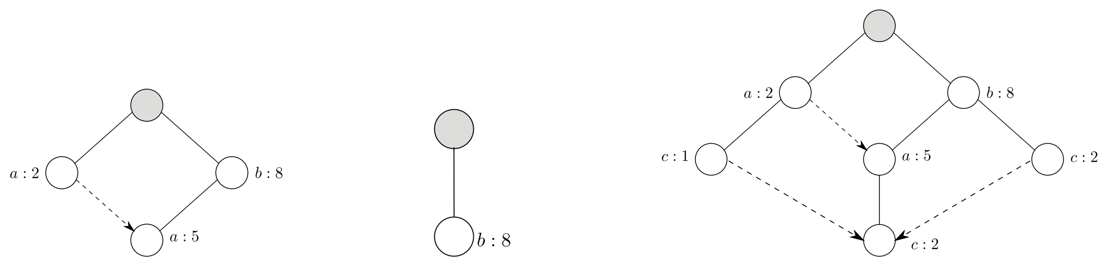
(h) $T_8$ append. (i) $T_9$ append. (j) $T_{10}$ append.

Figure 2.9: FP-tree construction by adding the different transactions.

First, we extract prefix path sub-trees ending in an item or an itemset. Then, each prefix path sub-tree is processed recursively to extract the frequent itemsets. To check if the ending item is frequent, FP-Growth accumulates linked counters and verify if the sum is greater than minSup threshold. If it is the case, the endings are removed from the sub-tree and the process is repeated this time to check if we can add another item to this suffix. To do that, the algorithm uses the conditional FP-Tree, which is an FP-Tree that would be built if we only consider transactions containing a particular itemset.

**Example 23** *Figure 2.10 represents all sub-trees ending respectively with "a", "b" and "c". In fact, to obtain such sub-trees, the edges not ending by such values are removed. All these sub-trees are extracted from the complete FP-Tree* *represented in Figure 2.9(j).*

*For example, Figure 2.11 represents conditional FP-Trees on "a" (Figure 2.11(d)) and "c" (Figure 2.11(c)), which is the updated sub-tree of the ones presented in Figures 2.10(a) and 2.10(c). In fact, to obtain Figure 2.11(c), we only consider transactions that contain the item "c". In each remaining transaction, only the items before the item "c" are represented. It is equivalent to keeping only the paths ending by the node "c", from the FP-Tree represented in Figure 2.9(j), and removing all edges pointing to the node "c". This step aims to look for eventual frequent items, added to "c", we obtain a frequent itemset. This is not the case for this conditional FP-Tree since* $s(a) = 1 + 2 = 3 < minSupport = 5$. *In the case of condition FP-Tree on "a", we have* $s(b) = 5 = minSupport$. *As a consequence, "a" and "ba" are two frequent itemsets ending in "a" extracted from FP-Tree. The result of this step is represented in Table 2.2 where the results of sub-trees are merged for the same suffix.*



(a) prefix sub-tree ending in $a$.
(b) prefix sub-tree ending in $b$.
(c) prefix sub-tree ending in $c$.

Figure 2.10: Prefix path sub-trees.

| Suffix | Frequent Itemsets |
|---|---|
| a | {a}, {b, a} |
| b | {b} |
| c | {c} |

Table 2.2: The frequent itemsets (minSupport=5).

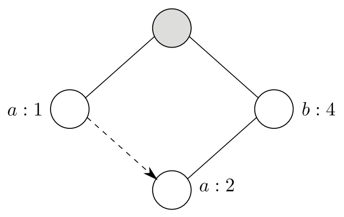
**Data-mining based approaches: related works** In [Jabbour et al., 2013b], data-mining techniques are used to reduce the size of propositional formulae in Conjunctive Normal Form (CNF). This approach is extended to CSP [Jabbour et al., 2013a]. In fact, additional variables and values are needed, and constraints are reformulated using Tseitin extension principle [Tseitin, 1983]. This principle consists in converting a Disjunctive Normal Form (DNF) formula which is a disjunction of conjunctions to a CNF formula which is a conjunction

| *TID* | *Items* |
|---|---|
| 1 | ~~{B, A}~~ |
| 2 | {B, ~~C~~} |
| 3 | {A, ~~C~~} |
| 4 | ~~{B, A}~~ |
| 5 | {B, A, ~~C~~} |
| 6 | ~~{B}~~ |
| 7 | ~~{A}~~ |
| 8 | {B, A, ~~C~~} |
| 9 | ~~{B, A}~~ |
| 10 | {B, ~~C~~} |

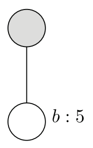
(a) Data-set: transactions ending in $c$.

| *TID* | *Items* |
|---|---|
| 1 | {B, ~~A}~~ |
| 2 | ~~{B, C~~} |
| 3 | ~~{A, C~~} |
| 4 | {B, ~~A}~~ |
| 5 | {B, ~~A, C~~} |
| 6 | ~~{B}~~ |
| 7 | ~~{A}~~ |
| 8 | {B, ~~A, C~~} |
| 9 | {B, ~~A}~~ |
| 10 | ~~{B, C~~} |

(b) Data-set: transactions ending in $a$.



(c) FP-Tree conditional on $c$.



(d) FP-Tree conditional on $a$.

Figure 2.11: Conditional FP-Tree.

of disjunctions by introducing new variables in order to prevent combinatorial explosion due to such conversion. Let us consider a DNF:

$$(x_1 \wedge \cdots \wedge x_l) \vee (y_1 \wedge \cdots \wedge y_m) \vee (z_1 \wedge \cdots \wedge z_n)$$

The conversion of this DNF to a CNF using the distributive characteristic of disjunction over conjunction causes a combinatorial explosion:

$$(x_1 \vee y_1 \vee z_1) \wedge (x_1 \vee y_1 \vee z_2) \wedge \cdots \wedge (x_l \vee y_m \vee z_n)$$

Applying Tseitin extension principle, by using additional variables $(t_1, t_2, t_3)$, we get:

$$(t_1 \vee t_2 \vee t_3) \wedge (t_1 \rightarrow (x_1 \wedge \cdots \wedge x_l)) \wedge (t_2 \rightarrow (y_1 \wedge \cdots \wedge y_m)) \wedge (t_3 \rightarrow (z_1 \wedge \cdots \wedge z_n))$$

Authors extend this technique to CSP and especially table constraints. This is introduced by Figure 2.12 which illustrates the Rewriting Rule used to reduce table constraints based on Tseitin extension principle. Let us consider the positive table constraint $c$ in Figure 2.12(a) and the itemset $I = \{(x_1, 0), (x_3, 0)\}$ which occurs four times. In order to compress $c$, authors propose to add a new variable $z$ and two new values $c_1$ and $c_2$, such that $dom(z) = \{c_1, c_2\}$. In fact, these values will replace the possible combination of values taken by $x_1$ and $x_3$ in the compressed table constraint such that $c_1 \rightarrow (0, 0)$ and $c_2 \rightarrow (0, 1)$. The Rewriting Rule presented in this paper generate two new constraints $c_0$ which is defined in Figure 2.12(b), such that $scp(c_0) = \{z, x_1, x_3\}$ and $c$' which is the compressed form of table constraint defined in Figure 2.12(c). The space required to represent the standard table constraint, not including the cost of the scope and variables, is $8 \times 4 = 32$, however, for the compressed table constraint it is equal to $3 \times 2 + 8 \times 3 = 30$. To the best of our knowledge, no experimentation on this approach was conducted, which prevents us from concluding about the efficiency of this approach. In other terms it is not clear if the compression is high enough to outweigh the cost of additional variables and values.

## 2.2 Parallel computing

Parallel computing has evolved during these last decades. It is no longer the matter of high equipped data centers since computers keep evolving to a higher number of processors. With the emergence of multicore processors [1], application need parallelism to exploit all available power of a machine offering this type of architecture. The Central Processing Unit (CPU) of computers has more and more cores which speeds up computing tasks. Graphics Processing Unit (GPU) is another unit which manipulates computer graphics and image processing. This

[1]A multi-core processor is an integrated circuit to which two or more processors have been attached in order to enhance performance by processing simultaneously multiple tasks.

| | $x_1$ | $x_2$ | $x_3$ | $x_4$ |
|---|---|---|---|---|
| $\tau_0$ | 0 | 0 | 0 | 0 |
| $\tau_1$ | 0 | 0 | 0 | 1 |
| $\tau_2$ | 0 | 0 | 1 | 0 |
| $\tau_3$ | 0 | 0 | 1 | 1 |
| $\tau_4$ | 0 | 1 | 0 | 0 |
| $\tau_5$ | 0 | 1 | 0 | 1 |
| $\tau_6$ | 0 | 1 | 1 | 0 |
| $\tau_7$ | 0 | 1 | 1 | 1 |

(a) Positive table constraint $c$.

| $z$ | $x_1$ | $x_3$ |
|---|---|---|
| $c_1$ | 0 | 0 |
| $c_2$ | 0 | 1 |

(b) $c_0$.

| | $z$ | $x_2$ | $x_4$ |
|---|---|---|---|
| $\tau_0$ | $c_1$ | 0 | 0 |
| $\tau_1$ | $c_1$ | 0 | 1 |
| $\tau_2$ | $c_2$ | 0 | 0 |
| $\tau_3$ | $c_2$ | 0 | 1 |
| $\tau_4$ | $c_1$ | 1 | 0 |
| $\tau_5$ | $c_1$ | 1 | 1 |
| $\tau_6$ | $c_2$ | 1 | 0 |
| $\tau_7$ | $c_2$ | 1 | 1 |

(c) $c^{'}$.

Figure 2.12: Reducing table constraint: $c_0$ and $c^{'}$.

unit offers also a highly parallel structure making it more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel.

Several parallel architectures have been proposed in order to benefit from these hardware advancements and to parallelize classical sequential algorithms.

In this section, we introduce parallel models and measures to evaluate parallel systems, and then, present the use of parallel algorithm in the context of constraint programming.

## 2.2.1 Definitions

**Definition 27 (Process)** *A* process *is a computer program that is being executed and has its own memory space.*

**Definition 28 (Thread)** *A* thread *is the execution in parallel of a part of the code of a process program. Contrary to a process, the threads of a same process share the same memory space and environment.*

One of the reasons behind using threads is that they are less expensive than creating processes.

Different measures are used to report the time spent by each thread, each process or the entire system. We distinguish the wall-clock time and the CPU time used in our experimentations.

**Definition 29 (Wall-clock time)** *The Wall-clock (wck) time is the time spent from the start to the completion of a task. This time includes programmed delays and the time spent waiting for resources to become available.*

**Definition 30 (Central Processing Unit time)** *The CPU time is the time during which a CPU was used. This includes processing instructions. The CPU time is useful to quantify the overall computing effort.*

## 2.2.2 Parallel performance measurements

To evaluate parallel solutions several metrics are used, but the most important one is speedup.

### Speedup

*Speedup* describes how much faster a parallel algorithm runs with respect to the best sequential one. For a problem of size $n$, the *speedup* $S_p$ is expressed as follows:

$$S_p = \frac{T_s}{T_p} \tag{2.2}$$

where $T_s$ is the resolution time of the sequential algorithm and $T_p$ is the resolution time of a parallel algorithm on $p$ processors. We mean by resolution time the wall clock time required to finish the resolution.

If the speedup increases *linearly* as a function of $p$, then we speak of *linear speedup*. Linear speedup means that the overhead of the algorithm is always in the same proportion with its running time, for all $p$. In the particular case of $T_p = T_s/p$, we then speak of *perfect linear* speedup. In practice, most programs achieve sub-linear speedup ($T_p \geq T_s/p$). When $T_p \leq T_s/p$, we achieve a super-linear speedup. Figure 2.13 shows the four possible speedups. If a problem cannot be

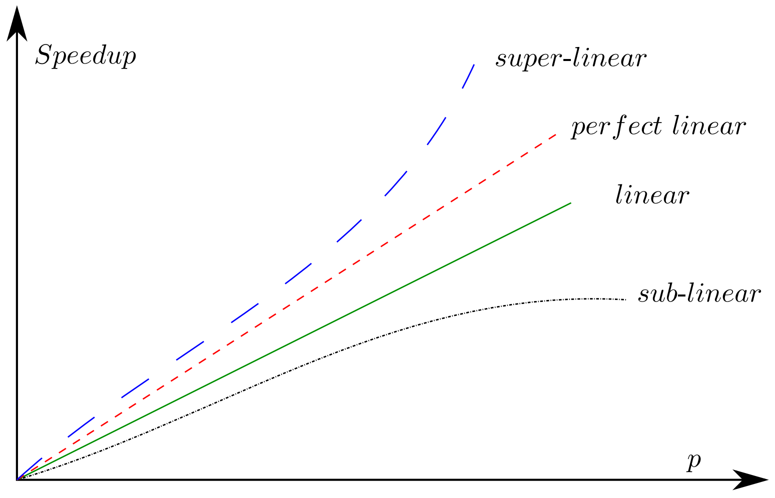


Figure 2.13: Possible speedup curves.

completely parallelized (one of the causes for sub-linear speedup), only a partial speedup is expected. Hence, rather than considering the speedup expressed in Equation 2.2 Amdahl and Gustafson proposed each one an expression, called a law of speedup. The two law's give different points of view about the execution time scale with the number of processors for computing partial speedup.

**Amdahl's law** Amdahl's law [Amdahl, 1967] expresses the maximum expected overall speedup of a sequential algorithm using $p$ processors as follows:

$$S(p) = \frac{T_s}{T_p} = \frac{t_s + t_p}{t_s + \frac{t_p}{p}} \tag{2.3}$$

where $t_s$ is the computation time needed for the sequential part and $t_p$ is the computation time needed for the parallel part.
Let $\alpha$ denote the sequential portion of the computation:

$$\alpha = \frac{t_s}{t_s + t_p} \tag{2.4}$$

We get, then

$$S(p) = \frac{1}{\alpha + \frac{(1-\alpha)}{p}} \tag{2.5}$$

If $p \mapsto \infty$:

$$S(p) = \frac{1}{1-\alpha} \tag{2.6}$$

which means that if we assume that a computer has as many processors as we want the maximum expected speedup is $\frac{1}{1-\alpha}$. If we take the example of a sequential algorithm requiring 10 hours using a single processor core and only 9 hours could be parallelized, the maximum expected speedup is 10 ($\frac{1}{\frac{1}{10}}$). Amdahl's law is useful for algorithms that need to scale its performance as a function of the number of processors, fixing the problem size $n$. This type of scaling is known as *strong scaling*. However, this law does not fully exploit the computing power that becomes available as the number of cores increases. Figure 2.14 illustrates the total work evolution with respect to Amdahl's law where $\varepsilon \mapsto 0$ when the number of cores $\mapsto \infty$. The assumption is "the problem size does not change with the number of CPUs".

**Gustafson's law** Rather than fixing a problem size, Gustafson's law [Gustafson, 1988] uses fixed-time model. This model considers that the work per processor is fixed when increasing the number of processors $p$ and the size of the problem $n$. Gustafson's law states that the time of a parallel system is composed of the computation time needed for the sequential part $t_s^*$ and the computation time needed for the parallel part $t_p^*$ executed by $p$ processors ($T(p) = t_s^* + t_p^*$). The expected speedup is expressed as follows:

$$S(p) = \frac{t_s^* + t_p^* p}{t_s^* + t_p^*} \tag{2.7}$$

Let $\alpha^*$ denote the sequential portion of the computation on the parallel system:

$$\alpha^* = \frac{t_s^*}{t_s^* + t_p^*} \tag{2.8}$$

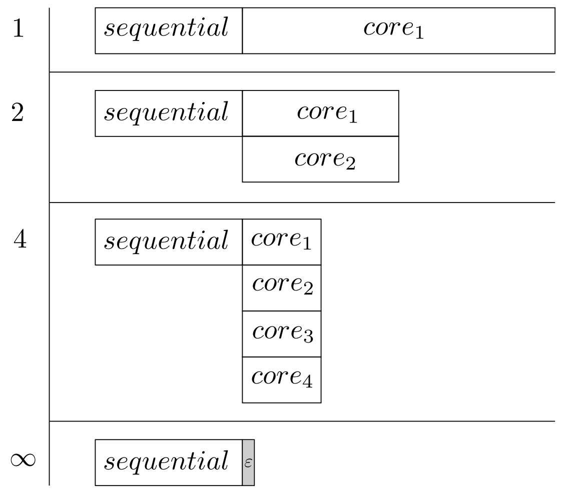


Figure 2.14: Illustrating Amdahl's Law (total work).

We get:

$$S(p) = \alpha^* + p(1 - \alpha^*) = p - \alpha^*(p - 1) \tag{2.9}$$

Thus, if $\alpha$ is small enough (i.e. the parallelizable part is big enough), the speedup is approximately $p$. This type of scaling is known as *weak scaling*. Figure 2.15 illustrates the total work evolution with respect to Gustafson's law. The assumption is "the computation time is constant".

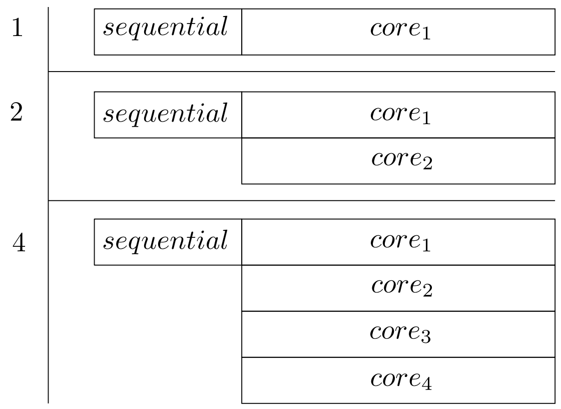


Figure 2.15: Illustrating Gustafson's Law (total work).

It is true that speedup might be one of the most important measures to evaluate a parallel solution. However, there are also other metrics that provide additional information about the quality of a parallel algorithm, such as the *efficiency*.

#### Efficiency

Efficiency metric of an algorithm using $p$ processors estimates how well the processors are being used in solving a problem, compared to how much effort is wasted in communication and synchronization. It is expressed as follows:

$$E_p = \frac{S_p}{p} \tag{2.10}$$

In fact, mostly, $E_p \leq 1$ except for a super-linear speedup which is difficult to obtain in practice (except for some cases like satisfiable instances in CP). It is equal to 1 when we can achieve a perfect linear. Efficiency metric is as important as speedup since it tells how well the algorithm exploits machine power.

### 2.2.3 Parallel programming models

Several parallel programming models are used to express a parallel algorithm with an abstraction of a computer system. They basically describe the way the different processors interacts or not with each other.

#### Communication model

**Shared memory** In a shared memory model, several threads (or processes) share a common memory, i.e threads can read and write asynchronously within this memory. Parallel algorithms using this model need to manage concurrent access to the memory when reading/writing data. Synchronization features are used to control concurrent threads, such as monitors [Hoare, 1974], semaphores [Dunstan, 1991], atomic operations [Herlihy and Wing, 1987] and mutexes to control inputs/ outputs of a common memory.

Symmetric Shared Memory multiprocessor (SMP), depicted in Figure 2.16, is a parallel hardware and software architecture which owns identical processors, so as to increase the computing power, while maintaining a single shared memory.

Non-uniform memory access (NUMA), depicted in Figure 2.17, is a computer memory model. While SMP is a “share everything” system, NUMA adds an intermediate level of memory. In fact, under NUMA, each processor has its own local memory to which it can access faster than non-local ones.

**Message passing** This model is basically used in distributed systems in which components are located on networked computers. These components communicate and coordinate their actions by asynchronously or synchronously passing (sending or/and receiving) messages. Different data structures are used depending on the way messages are sent. Several algorithms use queue data structure to manage the different messages. Dijkstra introduced many new ideas for distributed systems based on exclusion mechanisms [Dijkstra, 2001]. Figure 2.18

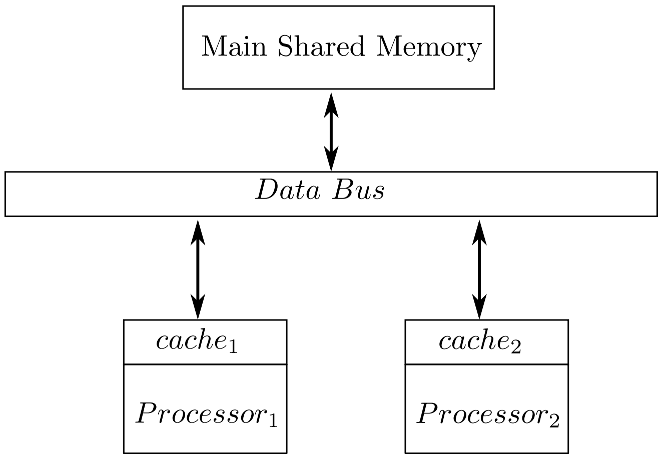


Figure 2.16: Symmetric Shared Memory multiprocessor architecture.

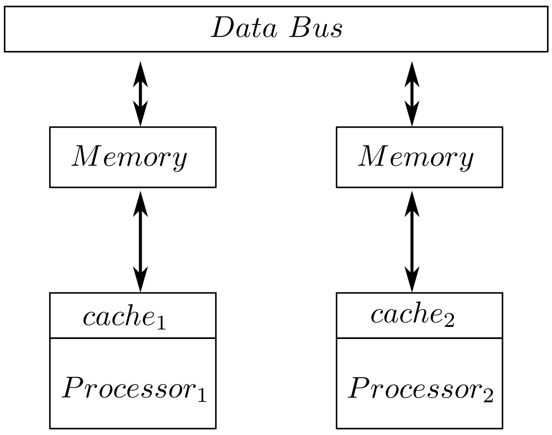


Figure 2.17: Non-uniform memory access model.

describes the message passing model where queues are used for each component to manage messages.

The main disadvantage of this model is the expensive cost of communication between the different components and especially managing messages.

### Parallel strategies

**Divide and conquer** Figure 2.19 describes Divide and conquer model which is a two-phase skeleton:

- *divide phase* consists in splitting the work up into sub-problems or tasks until sub tasks may be directly solved;

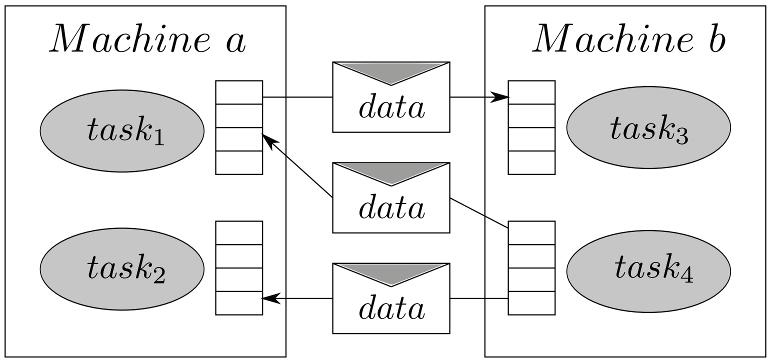


Figure 2.18: Message passing model.

- *conquer phase* where the result is recursively rebuild from partial results; we merge the results from that work serially again.

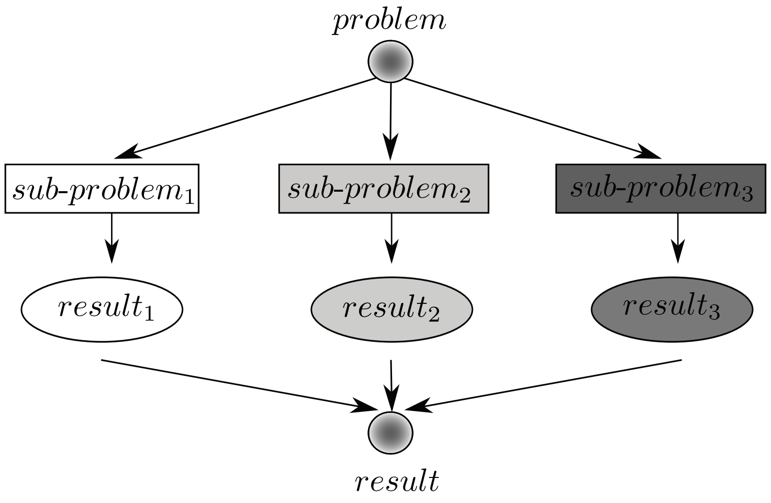


Figure 2.19: Divide and conquer model.

**MapReduce** MapReduce [Dean and Ghemawat, 2008] is a well-known parallel programming tool. In fact, MapReduce model is composed of *Map()* and *Reduce()* procedures. *Map()* performs filtering and sorting operations on data whereas *Reduce()* procedure performs a summary operation of the treated data. Figure 2.20 describes MapReduce model for word counting problem. In fact, words are divided to be treated in parallel. In a first step, the different words are tagged: each world belongs to a category, such as “Friend” or “Sun”. Then, the same words are gathered together and counted: the words of each category are brought together. Finally, all results are regrouped as depicted in the final result of Figure 2.20. MapReduce model is different from divide and conquer systems in tagging the intermediate results. In MapReduce, we divide the work up serially, execute tasks in parallel, and then, tag the results to indicate which results go with which other results. The merging is then serial for all the results with the same tag, but can be executed in parallel for results that have different tags.

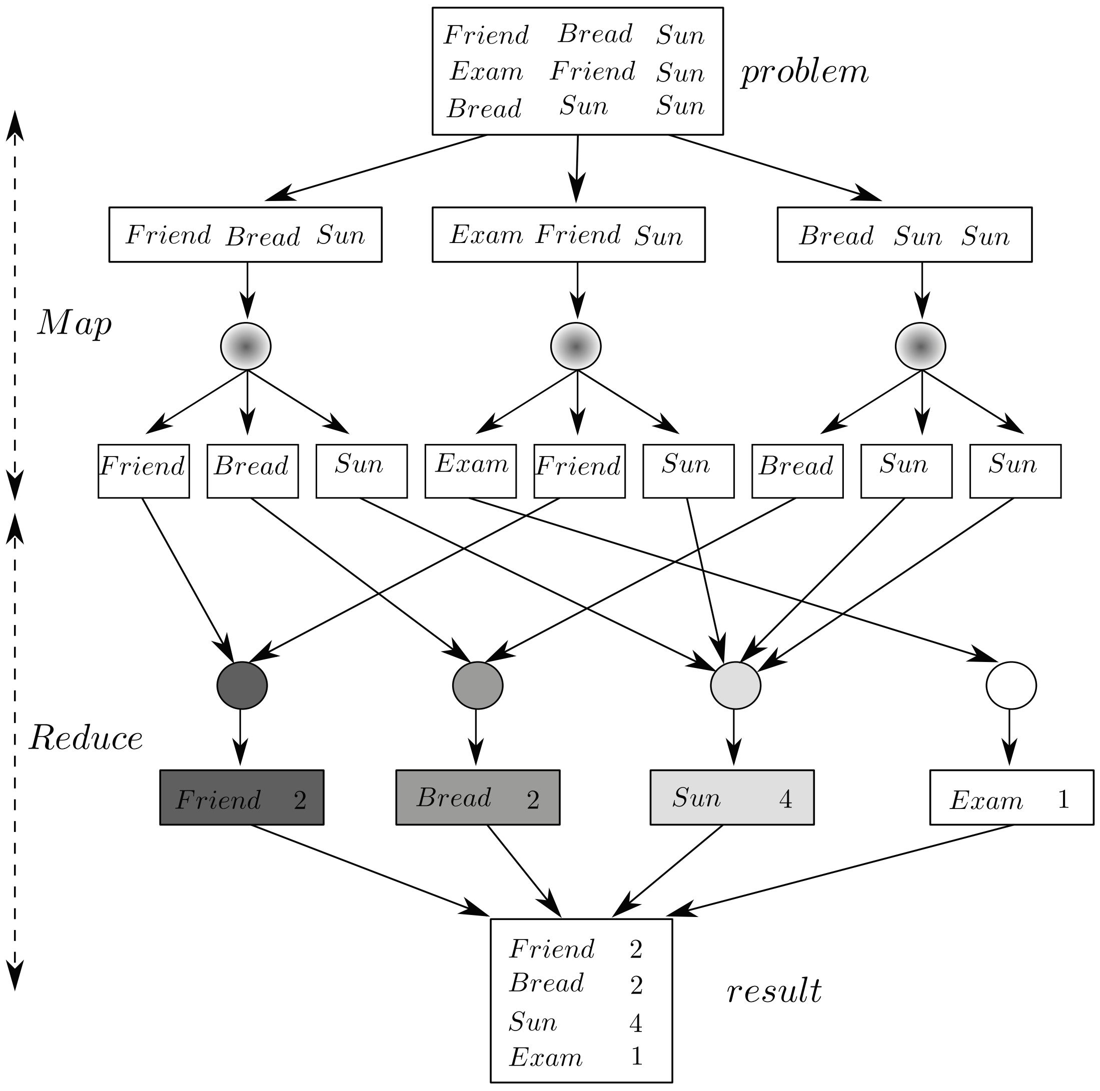


Figure 2.20: MapReduce model.

### 2.2.4 CSP solving using parallel programming models

We introduced previously different models of parallel programming used in several fields. In constraint programming, several search/inference algorithms were proposed in order to speed up the search process sequentially. With the development of parallel programming techniques, the CSP community tried to benefit from such parallel architecture when solving constraint problems. In this section, we introduce:

- Distributed Arc Consistency
- Work Sharing
- Work Stealing
- Multi-Agent Search
- Portfolios

An interesting survey of parallel approaches in the literature for constraint solving is proposed in [Gent et al., 2011].

#### Distributed Arc Consistency

In order to speed up the solving CSP process, several works were proposed in order to distribute computing to a set of parallel processes. Another justification is that the problem itself may be distributed geographically or due to organizational structures.

Consistencies correspond to a certain level of coherence maintained during search as introduced in Section 1.2. Enforcing consistency when the problem is distributed is different from the classical sequential way to enforce consistencies.

Using a shared memory model was a first natural way to enforce AC in a parallel way, but the proposed algorithms using such model were penalized with communication/synchronization costs. In a distributed context, problems cannot be solved by a centralized solver. We call such problems: Distributed Constraint Satisfaction Problems (DisCSP). We distinguish: Sensor networks [Domshlak et al., 2005], distributed resource allocation problems [Prosser et al., 1992] and distributed meeting scheduling [Maheswaran et al., 2004]. Several distributed arc consistency algorithms were proposed in the 1960s using asynchronous message passing (Section 2.2.3).

Asynchronous Backtracking algorithm (ABT) [Yokoo et al., 1992, Yokoo et al., 1998] is the first algorithm that maintains arc consistency in DisCSP. Consistency is maintained based on nogood exchange between agents. Each agent stores a set of received nogoods as justification of inconsistent values. When an agent cannot make any assignment with respect to its current domains, either due to the original constraints or due to the received nogoods, new nogoods are generated by resolution of its set of nogoods and each one is sent to the closest agent involved, causing backtracking.

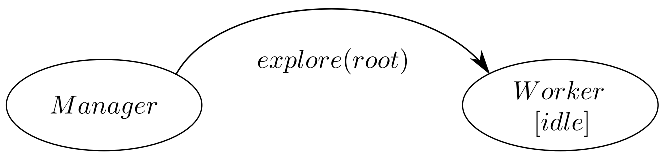
In [Nguyen and Deville, 1998], Distributed Arc Consistency ($DisAC4$) algorithm is based on $AC4$ (time-optimal sequential algorithm [Mohr and Henderson, 1986]). $DisAC4$ is a coarse-gained parallel algorithm designed for distributed memory models. In fact, $AC4$ is maintained in each node where a list of domains deletion is stored. After reaching a fixed point, a node broadcasts its list of domains deletion and waits for messages from other nodes. When the whole system reaches a fixed point, each node should proceed to domain deletion. This algorithm has $O(n^2d^2/p)$ time complexity. $DisAC4$ reaches, on some instances, a speedup close to linear with respect to the number of processors $p$. However, with the other instances $DisAC4$ reaches a $1,5$ speedup over 8 processors. The strongest feature of $DisAC4$, as the authors claim, is its suitability to be implemented on very common hardware infrastructures.

$DisAC9$ [Hamadi, 1999] improves the asynchronous message passing process since it is a difficult issue due to dependencies between deletions. $DisAC9$ is optimal according to the number of message passing operations and it is considered the quickest processing algorithm of the Distributed Constraint Satisfaction Problems (DCSP) since it minimizes the number of messages transmitted between workers ($O(n^2d^3)$ complexity). In fact, a worker informs other workers about domain deletion *only if* it has the unique support for their values considering a constraint $c$. In the other case, it is useless to send a message and the worker stores it in the list of domains deletion that will be sent iff other *significant* (unique support) deletion occurs. The crucial point while distributing CSPs is the distribution of information among workers. In particular, partitioning variables in order to obtain a distributed problem is the main issue.

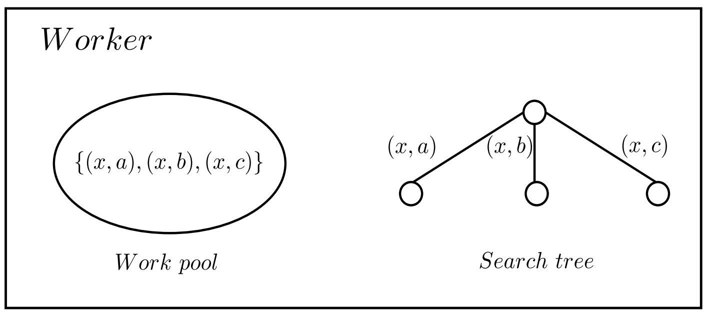
### Work Sharing

*Parallel search made simple* [Schulte, 2000] is one of the well-known approaches where the authors used *work splitting* technique in order to share work. They assume that for simple and reusable parallel constraint solving, we should separate: *search*, *concurrency* and *distribution.* Using concurrent language Oz [Henz et al., 1993] (Mozart implementation), they are referring to search nodes by the term *computation spaces.* The main feature of their approach is assuming that resources are cheap and mostly idle, which differs from the classical idea, in order to obtain a good speedup rather than a good resource use. In fact, workers use *work sharing* to dynamically distribute work on them. Each worker maintains a *work pool* which includes all nodes to be explored. The work is managed by a single manager through the usage of messages described in Figure 2.21. We distinguish three major steps: initialization, finding work and collecting a solution.

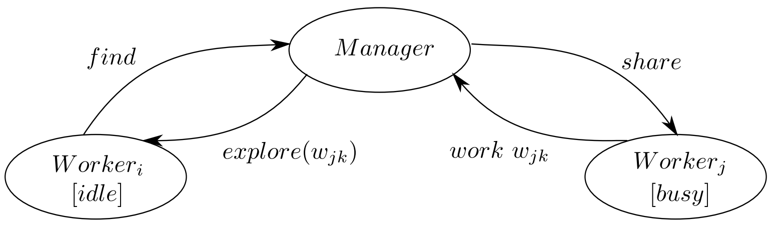
In fact, in a first step a manager sends an *explore*-message to an idle worker corresponding to the root of the search tree. This worker becomes busy and starts to explore the search tree. The worker generates in turn new work which corresponds to the unexplored branches of the search tree. The available work is stored in its work pool. An idle worker contacts the manager in order to obtain work by sending a *find*-message. The manager in its turn sends a *share*-message to the first busy worker (a list of busy workers is maintained in the manager). In case where the contacted worker has a work in its pool to give, it sends to the manager which sends it back to the idle worker by an *explore*-message. Finally, when a worker finds a solution it sends it to the manager by a *collect*-message. This approach reaches an almost linear speedup.



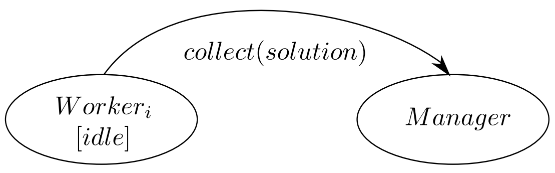
(a) Initialization.



(b) Exploration.



(c) Finding work.



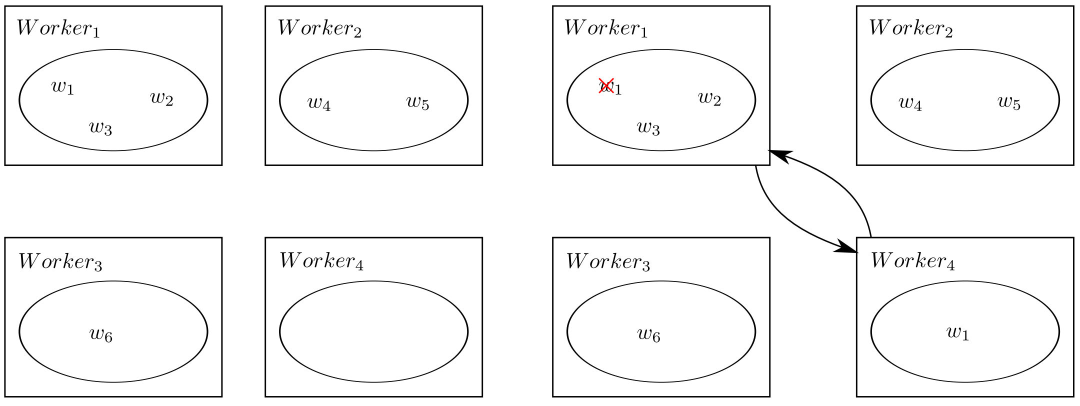
(d) Collecting a solution.

Figure 2.21: Messages communicated between workers and manager.

More recently, the authors in [Régin et al., 2013] proposed another approach called "embarrassingly parallel search", which statically decomposes the initial problem into many small sub-problems that are available to the workers and puts them in a queue. When a worker is idle, it takes dynamically a job from the job queue and solves it. The master maintains the concurrent access of the queue. The resolution is finished when all sub-problems are solved. In this approach there is no communication between workers so when a worker finds a better solution, the other workers cannot benefit from it to improve their current resolution. A good balancing is observed in practice. The decomposition algorithm is improved in [Régin et al., 2014].

### Work Stealing

*Work stealing*, first proposed in [Sleep, 1981], is the most popular architecture for parallel search where nodes steal work from each other if they are out of work. In fact, the problem is dynamically split during the solving process. This guarantees that workers are always busy and ensure a dynamic load balance. Figure 2.22 describes this architecture. In Figure 2.22(a) the work is not split in an equitable way: we suppose that $Worker_4$ has finished its work. $Worker_4$ remains without work that is why it steals work $w_1$ from $Worker_1$ in Figure 2.22(b). The disadvantage of this approach is the communication costs between workers in order to obtain work especially at the end of the problem solving when many workers have no sub-problems to solve. Moreover, when asking for work, a worker should not be given an easy work so it will not steal another work again almost instantly. This architecture is also used in [Chu et al., 2008, Schubert et al., 2008, Chu et al., 2009]



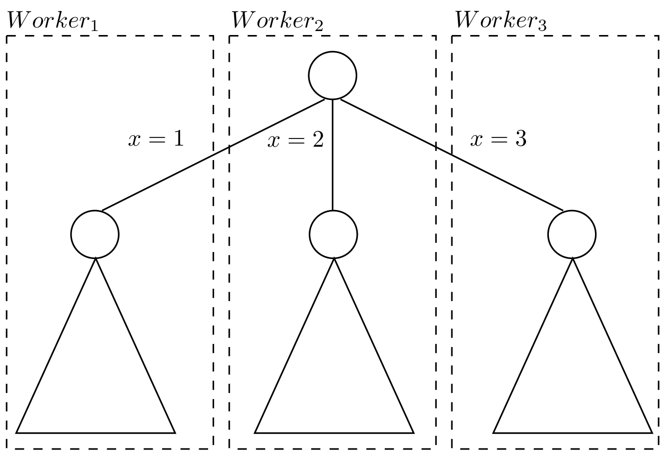
(a) Work distribution. (b) $Worker_4$ steals work from $Worker_1$.

Figure 2.22: Work stealing architecture.

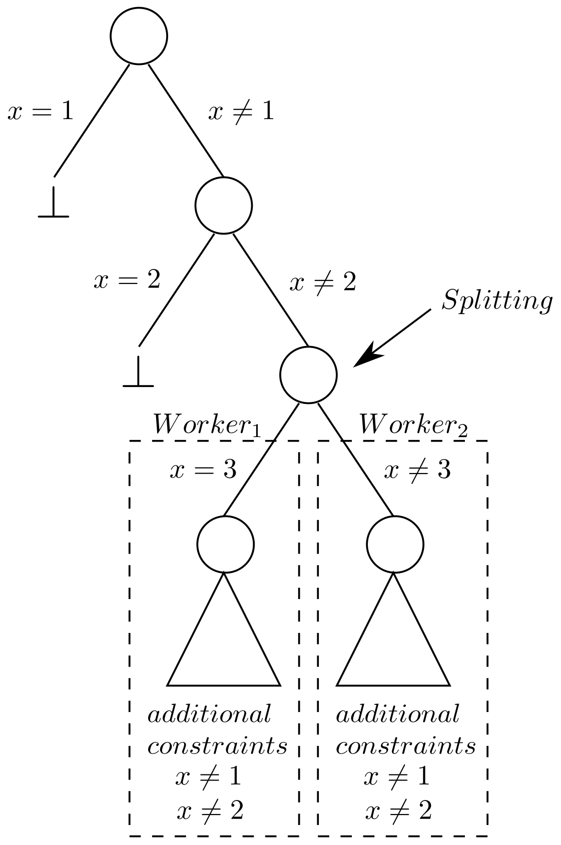
In [Xie and Davenport, 2010], a masters/workers approach is proposed where the search space is divided between the different masters. Each master puts its attributed sub-trees in a work pool in order to dispatch them to the workers. In fact, an idle worker requests its master for work and does not steal work directly from a busy worker. If there is no unexplored nodes the worker remains idle until new jobs become available.

Another *work stealing* approach is presented in [Kotthoff and Moore, 2010]. The authors describe a model splitting approach which consists in modifying the constraint model of the problem. In fact, the problem model is converted into new models, when assigning variables for example. Added to that, several constraints used to partition the search space are appended to the models. This approach does not share any information between workers. The approach uses the work stealing technique, while outputting restart nogoods for the problem in order to prevent the worker from exploring the space just explored before (In Figure 2.23(a), the restart nogood is $x \neq 1$ and $x \neq 2$). A job server is also used to avoid implementing distribution. Clearly, parents give up all their search space and split themselves into $n$ parts. To the best of our knowledge, there is no experiments done. Figure 2.23 describes this architecture where the search space is split in two different conditions. In Figure 2.23(a), the search space is split before search where the 3 branches corresponding to the assignment of variable $x$ is partitioned between workers. We call this an *n-way* branching. The main problem of such splitting is that it is impossible to predict the size of the search space for each of the splits: some parts could lead to a failure and then one of the workers will be idle while the others do most of the work. The search-space splitting in Figure 2.23(b) is done in a different way. In fact, during search and after the two decisions, $x \neq 1$ and $x \neq 2$, the search space is split between worker. In [Kotthoff and Moore, 2010], authors add restart nogoods which are additional constraints to the search space of each worker.

In [Chu et al., 2009], different work stealing strategies are described which are based on a *confidence* measure. This latter is the estimated ratio of solution densities between the sub-trees at each node obtained with respect to the branching heuristic used at each node and it is updated during search. The authors present a quantitative analysis discussing the effect of the different work stealing strategies on the work amount performed. A function can be used to determine the relation between the strength of the branching heuristic and the density of solutions and, thus, ensure automatically an optimal work stealing. For example, sometimes it appears that stealing *left* and *low* could be much better, but sometimes stealing *high* can be better. The authors show that in practice stealing low increases the communication cost. This is the reason why they defined a bound above the average fail depth below which work cannot be stolen. These different strategies show an effectiveness ranging from speedups of 7 times to a super linear speedup on the benchmarks in the paper using 8 threads for the parallel search algorithm.



(a) Search-space splitting before search (*n-way* branching).



(b) Search-space splitting during search (*2-way* branching).

Figure 2.23: Search-space splitting architecture.

### Multi-Agent Search

In multi-agent search architecture, we have one problem and different agents. In fact, each agent solves the problem independently and may communicate with the other agents. This approach could achieve super-linear speed up even-though the diversity among agents increases communication costs. This idea was, first, explored in [Rao and Kumar, 1988].

In [Bordeaux et al., 2009], the authors present some "Experiments with Massively Parallel Constraint Solving" where they explore approaches avoiding communication between workers. In fact, the problem is split over workers with respect to hashing constraints. The hashing constraint of each core is unique. The authors present promising results on more than 64 cores.

### Portfolios

In this multi-core architecture, a portfolio of solvers use a variety of algorithms to have a globally better solver. This architecture could be considered a multi-agent search. The idea of *portfolio-based search* was explored in [Gomes and Selman, 2001]. Portfolios have been extremely used by the SAT community and a variety of solvers were proposed. To the best of our knowledge, there are two CSP portfolio solvers CPHydra [O'Mahony et al., 2008] and Sunny-CP [Amadini et al., 2015]. CPHydra, a sequential portfolio, won the 2008 CSP solver competition [2]. It uses a case-based reasoning and solves a problem based on the $k$ most similar instances (K-nearest neighbor) from a base. In fact, CPHydra computes a schedule of the portfolio constituent solvers to be run in sequence. This schedule of solvers allocates a duration to each solver. Three main solvers are used in CPHydra: AbsCon, Choco and Mistral.

Sunny-CP attended the MiniZinc Challenge (MZC) [Stuckey et al., 2010] with respectable results ($4^{th}$ out of 18). Sunny-CP is a CSP/COP portfolio solver using MiniZinc Language. Using different solvers, features, data-sets and parameters tuning, Sunny-CP is able to outperform state-of-the-art constraint solvers. For an input problem, Sunny-CP selects the k closest instances using the $k$-Nearest Neighbors (k-NN) algorithm. Based on these instances, Sunny-CP selects the promising solvers to run.

In [Dasygenis and Stergiou, 2014], the authors proposed a way to build portfolios for parallel solving by varying the local consistency during search. In fact, authors use different heuristics for adaptive propagator selection proposed in [Stergiou, 2008] which switch dynamically between a weak and a strong propagator for each individual constraint during search. A constraint $c$ is made *strong* or *weak* with respect to user predefined thresholds considering different propagation events such as Domain Wipe Out (DWO) and domain deletions. Two main thresholds are used in this portfolio: $l_{dwo}$ and $l_{del}$. A constraint $c$ is made

[2]http://cpai.ucc.ie/08/

*strong* if the number of revisions of $c$ since the last time it caused a DWO (respectively at least one value deletion) is less or equal to $l_{dwo}$ (respectively $l_{del}$). Experiments in this paper use $H_1(l_{dwo})$, $H_2(l_{del})$ or $H_4$ heuristics, introduced in [Stergiou, 2008], or a combination of them while varying thresholds for each solver. Each solver uses MAC. Experimental results demonstrated that this approach constantly outperforms the sequential solver and competes with the other portfolio approaches using randomized variable ordering heuristic.

# Part II

# Contributions

# Chapter 3

# Compressing Table Constraints

## Contents

## 3.1 Introduction

In Chapter 2, we introduced the context of compressing table constraints. Several approaches and data structures are presented. Moreover, for each of them, we introduced the corresponding filtering algorithm.

In this chapter, we propose two different compression techniques for table constraints. Each of them presents a different form of compressed table constraint aiming to reduce time and space complexity.

This chapter is organized as follows:

- In the first Section, we introduce a first work (STR$^c$) [Gharbi et al., 2013] that was presented at *JFPC 2013* and *CP 2013 (Doctoral Program)*;
- The second Section presents a second work (STR-*slice*) [Gharbi et al., 2014] presented at *CPAIOR 2014*;
- Finally, we conclude.

## 3.2 STR$^c$

The first approach, called compressed STR (STR$^c$), consists in combining STR with a compression algorithm based on "tries", which is different from the ones described in [Katsirelos and Walsh, 2007] and [Xia and Yap, 2013] where a Cartesian Product representation is used for compression. The idea is to identify the recurrent patterns (sub-tuples) in the tuples of each constraint, and replace their occurrences by references to a patterns table. The filtering process is an adaptation of STR in order to take into consideration the patterns appearing at different positions.

This section is organized as follows. In Section 31, we describe the way patterns are defined. Then, we give details about the compression process in Section 3.2.2. Next, we present in Section 3.2.3 the filtering algorithm used for the "compressed" form of table constraints. Finally, we present some experimental results in Section 3.2.4.

### 3.2.1 Patterns Definition

In our context, a pattern is defined as follows:

**Definition 31** *(pattern) A pattern $\mu$ is a sequence of consecutive values (a sub-tuple) in a tuple $\tau$ of a table constraint. We note $|\mu|$ the length of a pattern $\mu$, nbOcc($\mu$) the number of occurrences of the pattern in the tuples of a given constraint.*

**Example 24** *Let $\mu_1$ be a pattern such that $\mu_1$=$(a, a, b)$ and $\mu_1$ appears 4 times in a table constraint. We say that $|\mu_1| = 3$, nbOcc$(\mu_1) = 4$.*

Since the main idea of this approach is to reduce the spatial complexity of table constraints through using the most frequent patterns, we defined a pattern in a way that maximizes its frequency. It is important to note that the patterns extracted in our approach are independent from their position in the scope of the table constraint. According to that, a pattern does not correspond necessarily to the assignment of the same values to the same variables, but rather the same sequence of values. This choice was made to hopefully maximize the frequency of possible patterns, and, thus, to obtain a better compression. Therefore, the longer the patterns and the more frequent they are, the smaller the table constraint representation will be.

**Example 25** *Figure 3.1 describes the detection of the pattern $(c, b, c)$. This pattern which appears in **four** tuples ($\tau_1$, $\tau_3$, $\tau_4$ and $\tau_6$), does not involve the same variables. In tuple $\tau_1$ this pattern involves $x_1$, $x_2$ and $x_3$, whereas, in the case of tuple $\tau_4$ the pattern involves $x_3$, $x_4$ and $x_5$. It is evident that if we consider the same variables when looking for a pattern and that the variables must be consecutive, we will not have the same patterns neither the same number of occurrences as the patterns found applying Definition 31. This is shown by colors in Figure 3.1. The pattern $\{(x_1, c), (x_2, b), (x_3, c)\}$ appears **just one time**. The pattern $\{(x_2, c), (x_3, b), (x_4, c)\}$ appears **two times**.*

| | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|---|
| $\tau_1$ | (**c**, | **b**, | **c**, | a, | c) |
| $\tau_2$ | (a, | a, | b, | b, | a) |
| $\tau_3$ | (a, | **c**, | **b**, | **c**, | a) |
| $\tau_4$ | (b, | a, | **c**, | **b**, | **c**) |
| $\tau_5$ | (b, | a, | a, | b, | b) |
| $\tau_6$ | (a, | **c**, | **b**, | **c**, | b) |
| $\tau_7$ | (a, | c, | a, | c, | a) |

Figure 3.1: Patterns detection according to positions in the table constraint $C_{x_1,x_2,x_3,x_4,x_5}$.

**Example 26** *Through Tables 3.2(a) and 3.2(b), we illustrate the effect of scope order on patterns detection. Let us consider, for example, the pattern $(c, b, c)$ in Table 3.2(a) which appears four times. If we change the scope order arbitrarily in Table 3.2(b) this pattern does not appear anymore. On the other hand, the pattern $(a, b, b)$ that appears two times in Table 3.2(a), happens to be more frequent in Table 3.2(b) (appearing three times).*

|  | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|---|
| $\tau_1$ | (**c**, | **b**, | **c**, | a, | c) |
| $\tau_2$ | (a, | **a**, | **b**, | **b**, | a) |
| $\tau_3$ | (a, | **c**, | **b**, | **c**, | a) |
| $\tau_4$ | (b, | a, | **c**, | **b**, | **c**) |
| $\tau_5$ | (b, | a, | **a**, | **b**, | **b**) |
| $\tau_6$ | (a, | **c**, | **b**, | **c**, | b) |
| $\tau_7$ | (a, | c, | a, | c, | a) |

(a) Patterns detection with a lexicographical-ordered scope.

|  | $x_2$ | $x_4$ | $x_1$ | $x_3$ | $x_5$ |
|---|---|---|---|---|---|
| $\tau_1$ | (b, | a, | c, | c, | c) |
| $\tau_2$ | (a, | b, | a, | b, | a) |
| $\tau_3$ | (c, | c, | a, | b, | a) |
| $\tau_4$ | (**a**, | **b**, | **b**, | c, | c) |
| $\tau_5$ | (**a**, | **b**, | **b**, | a, | b) |
| $\tau_6$ | (c, | c, | **a**, | **b**, | **b**) |
| $\tau_7$ | (c, | c, | a, | a, | a) |

(b) Patterns detection with an arbitrary-ordered scope.

Figure 3.2: Patterns detection according to the scope in the table constraint $C_{x_2,x_4,x_1,x_3,x_5}$.

From Example 26, we conclude that the scope order of the constraint can affect advantageously or not patterns detection: the detected patterns change and also their number of occurrences. We note that in our first approach (STR$^c$), we consider that the scope is ordered in a lexicographical order. Whereas, for the second approach (Section 3.3) we use another patterns definition that depends only on variables, and so, the scope order doesn't influence patterns detection.

### 3.2.2 Compression Method

**Collecting patterns** In this step, we identify the relevant patterns from the various tuples of a given table constraint. To do that, we use tries (Section 2.1.1) as data structures to store all these patterns. We first introduce the different algorithms for manipulating this data structure.

Algorithms 19, 20 and 21 describes the way that patterns are managed in the trie data structure. In Algorithm 19, when a new pattern candidate $\mu$ is extracted, we check first if it is already stored into *patternsTrie* or there exists a pattern $\mu^{'} \subset \mu$ such that $\mu^{'}$ is already in the trie (Lines 3 - 9). If $\mu$ does not exist in the trie or only a part of $\mu$ exists *nbOcc* counters of the new added nodes are initially fixed to 1 (Line 11). If $\mu$ shares its first value(s) with a path the trie, the *nbOcc* counters of the shared nodes are incremented (Line 7) and the remaining part is inserted at the right place. Algorithm 20 describes the way a pattern is extracted from the trie. In a ascending way (from a node to the root), we extract the pattern corresponding to the path going from the root to a fixed node. Algorithm 21 describes the way that the length of a path is calculated.

In order to collect all possible patterns, we read each tuple in the table constraint. For each possible pattern, we first verify if it already exists in the trie. If it is the case, the corresponding nodes counters are updated (incremented). If the pattern shares its first values with an existing path in the trie, only the

**Algorithm 19:** addPattern($root$:Node,$\mu$:Pattern)

```
node ← root
i ← 0
while $i < \mu.length$ do
    $node \leftarrow node.getChild(\mu[i])$
    if $node \neq nil$ then
        $i \leftarrow i + 1$
        $node.nbOcc \leftarrow node.nbOcc + 1$
    else
        break
            // $\mu$ is partially or entirely added
while $i < \mu.length$ do
    $addChild(node, \mu[i])$          // $nbOcc$ is initialized to 1
    $node \leftarrow node.getChild(\mu[i])$
    $i \leftarrow i + 1$
```

**Algorithm 20:** getPattern($node$:Node):pattern

```
$pattern \leftarrow \prec\succ$
while $node \neq root$ do
    $pattern \leftarrow label(node.getParent(), node) + pattern$
    $node \leftarrow node.getParent()$
return $pattern$
```

**Algorithm 21:** getLengthPath($node$:Node):Integer

```
$length \leftarrow 0$
while $node \neq root$ do
    $length \leftarrow length + 1$
    $node \leftarrow node.getParent()$
return $length$
```

corresponding nodes counters are updated and the remaining part of the pattern is added at its right place of the trie. If the pattern does not exist in the trie and it does not share any first value with the trie prefixes, a new path corresponding to this pattern is created from the root. A trie contains, thus, all the existing sequences of values and their number of occurrences.

**Example 27** *This example shows the different steps of the trie construction for the first tuple $\tau_1$ of the table constraint in Figure 3.1. We detect all possible* *patterns of scope at least equal to* 3 *and at most* 4 *(we explain this choice in the next paragraph). Starting from the first position,*$(c, b, c)$ *is the first extracted pattern. Since the trie is empty, it is inserted in the trie creating a first path from the root (Figure 3.3(a)). The nodes' counters are initially equal to* 1. $(c, b, c, a)$ *is also a pattern candidate of length* 4 *starting from position* 1. $(c, b, c, a)$ *is not included in the trie but it is the continuity of the path* $(c, b, c)$. *Thus, only the value* $a$ *is added at the end of this path (Figure 3.3(b)). The nodes' counters are not updated in this case because there are not two different paths sharing their first values (edges). Starting from position* 2, $(b, c, a)$ *a pattern of length* 3 *is inserted in Figure 3.3(c).* $(b, c, a, c)$ *a pattern of length* 4 *is added in the same way as the pattern* $(c, b, c, a)$*: the value* $c$ *is put at the end of the path (3.3(d)). Starting from position* 3 *the only possible pattern candidate is* $(c, a, c)$ *which shares its first value* $c$ *with the path* $c \to b \to c \to a$. *The node's counter corresponding to the value* $c$ *is updated to* 2. *The path* $(a, c)$ *is put as a child of the ending node of edge* $c$.

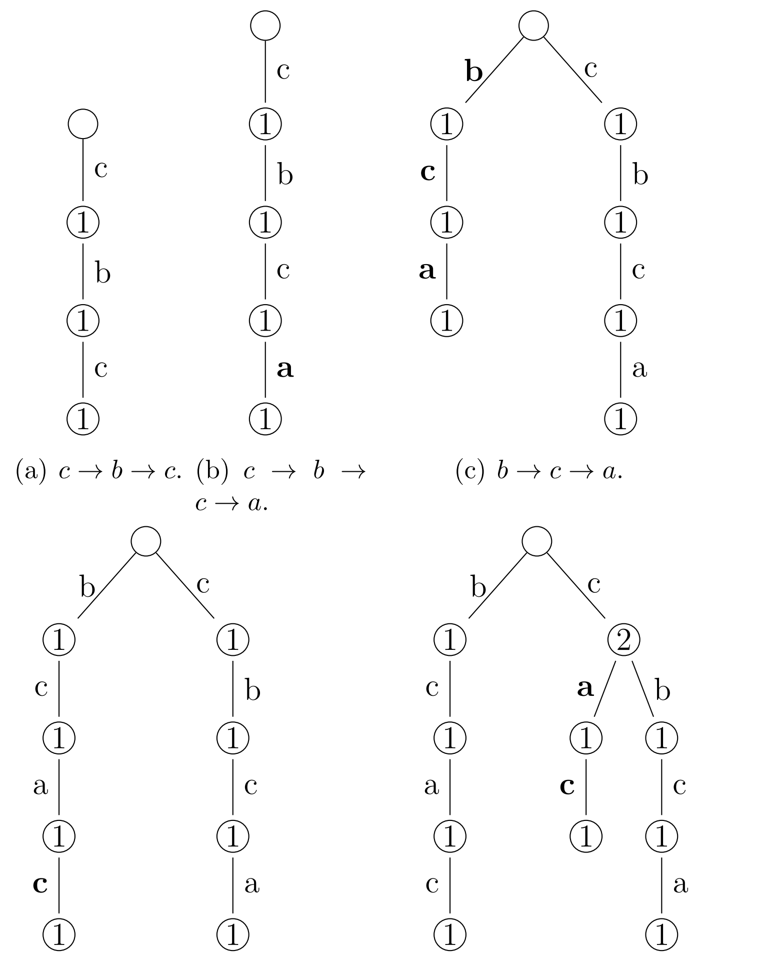
| | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|---|
| $\Rightarrow \tau_1$ | (c, | b, | c, | a, | c) |
| $\tau_2$ | (a, | a, | b, | b, | a) |
| $\tau_3$ | (a, | c, | b, | c, | a) |
| $\tau_4$ | (b, | a, | c, | b, | c) |
| $\tau_5$ | (b, | a, | a, | b, | b) |
| $\tau_6$ | (a, | c, | b, | c, | b) |
| $\tau_7$ | (a, | c, | a, | c, | a) |

Table 3.1: Table constraint $C_{x_1,x_2,x_3,x_4,x_5}$.

First experiments show that choosing 1 or 2 as minimal length for patterns, is not effective to obtain a high ratio of compression. That is why the minimal length for patterns is fixed to 3 in our approach. The maximal possible length of a pattern, in our approach, is defined as the constraint arity minus 1.

Algorithm 22 describes the different steps used to build all possible patterns. Lines 3-6 of the algorithm iterate over all tuples of the table and collect all candidate patterns of different sizes starting from different positions of the tuple. The size of patterns is between the minimal length of patterns, which is fixed in our approach to 3, and the maximal length, which represents the constraint arity minus 1. All these patterns are stored in *patternsTrie*. The remaining steps (Line 8) of Algorithm 22 are introduced in the next paragraph.

**Detecting frequent patterns** In a second step, it is necessary to identify the most relevant patterns for the compression process. To do that, we introduce



(a) $c \to b \to c$. (b) $c \to b \to c \to a$. (c) $b \to c \to a$.

(d) $b \to c \to a \to c$. (e) the trie obtained after patterns extraction on $\tau_1$.

Figure 3.3: The patterns trie built from $\tau_1$ of the constraint given in Table 3.1.

the notion of **score** of a pattern $\mu$ as follows:

$$score(\mu) = |\mu| \times nbOcc(\mu) \tag{3.1}$$

In fact, the score of each pattern expresses the space occupied by all of its occurrences in the table constraint. The bigger the score is, the bigger the occupied space is and, thus, the smaller the table constraint will be if we replace each occurrence of the pattern by a reference towards it. Consequently, this definition of score allows us to choose the patterns that make the table smaller and then obtain a better compression. A selection threshold is fixed, and only the patterns having a score greater than this threshold are retained in the compression

**Algorithm 22:** buildPatterns($c$: Constraint): Queue

```
1 $maxSize \leftarrow |scp(c)|$ - 1
  // Collecting all possible patterns in $patternsTrie$
2 $patternsTrie \leftarrow \emptyset$
3 foreach $\tau_i \in table(c)$ do
4     foreach $position \in [0, |scp(c)| - minSize]$ do
5         foreach $size \in [minSize, min(maxSize, |scp(c)| - position)]$ do
              // Adding pattern in $patternsTrie$
6             $\mu \leftarrow \prec \tau_i[position], \ldots, \tau_i[position + size] \succ$
              $addPattern(root(patternsTrie), \mu)$
  // Maintaining only the frequent patterns
7 $FP\text{-}Queue \leftarrow \emptyset$
8 $extractBestFrequentPatterns\ (patternsTrie.root,\ FP\text{-}Queue)$
9 return $FP\text{-}Queue$
```

algorithm and stored in the patterns table. In the experimental section, Figures 3.14 and 3.15 show the impact of the threshold score variation on the number of used patterns and the compression ratio.

For efficiency reasons, the total number of retained patterns is limited by a second parameter called *nbMaxPatterns* in order to control the compression time and the memory space used to store the different patterns. This parameter is useful in the case of class of instances having an important number of frequent patterns.

Both parameters *scoreThreshold* and *nbMaxPatterns* enable us to obtain the most frequent patterns having a score greater than *scoreThreshold*. Both of them are important in the compression process.

Algorithm 23 describes the frequent patterns detecting step. In fact, after collecting all candidate patterns in *patternsTrie* we calculate for each pattern its score. Considering the pattern's score, we decide if we put it in the *FP-Queue*, which is a priority queue where patterns are stored in a decreasing order of their score. This score is calculated as mentioned before. In the case that *FP-Queue* is already full (Line 8), meaning that the number of patterns is equal to *nbMaxPatterns*, every time we calculate the score of a new pattern we compare it to the worst pattern which is the pattern having the smallest score meaning that it is the last element of the queue (Line 9). This comparison (Line 10) allows us to decide whether we keep the worst pattern in *FP-Queue* (in case that the new pattern's score is less than the worst pattern) or replace it by the new pattern, otherwise (Lines 11 - 13).

**Algorithm 23:** extractBestFrequentPatterns(*node*: Node, *FP-Queue*: Queue)

```
1  $score \leftarrow 0$
2  if $level(node) \geqslant minSize$ then
3      $score \leftarrow getLengthPath(node) \times node.nbOcc$
4      if $score > scoreThreshold$ then
5          if $size(FP\text{-}Queue) < nbMaxPatterns$ then
               // Extract pattern from patternsTrie: the path from
                  root to node
6              $pattern \leftarrow$ getPattern($node$)
7              addPatternQ($pattern$, $score$, $FP\text{-}Queue$)
8          else
9              $worstPattern \leftarrow$ getWorstPattern($FP\text{-}Queue$)
10             if $score > score(worstPattern)$ then
11                 delete($worstPattern$)
12                 $pattern \leftarrow$ getPattern($node$)
13                 addPatternQ($pattern$, $score$, $FP\text{-}Queue$)
14 foreach $child \in node.getChildren()$ do
15     extractBestFrequentPatterns($child$, $FP\text{-}Queue$);
```

**Compressing table constraints** After detecting the most relevant patterns, iterating over the constraint table is necessary to detect the presence of such patterns and replacing them with indexes. An illustration of the compressed data structure is described in Figure 3.4. In fact, once a pattern is found, it is replaced by a reference towards a patterns table where the frequent patterns are stored. The reference is encoded by a negative integer (-*patternId*). In fact, the tuple will no more contain the values composing the pattern, but rather a reference that points to the pattern stored in the patterns table.

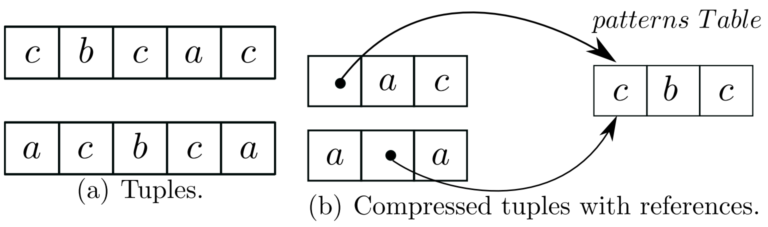


Figure 3.4: Compressing tuples.

Algorithm 24 describes the compression step. First, for each tuple in the table constraint we look for the best contained pattern in *FP-Queue*. When searching

**Algorithm 24:** compress-STR$^c$($c$: Constraint,$FP$-$Queue$: Queue)

```
1  foreach $\tau_i \in table(c)$ do
2      repeat
           // searching for the best frequent pattern contained in $\tau_i$
3          $bestPattern \leftarrow \prec\succ$
4          foreach $\mu_i \in FP\text{-}Queue$ do
5              if $contains(\mu_i,\tau_i)$ then
6                  $bestPattern \leftarrow \mu_i$
7                  $bestPattern.position \leftarrow$ index($\mu_i$,$\tau_i$)
8                  break
9          if $bestPattern \neq nil$ then
               // Compressing each tuple $\tau_i$ of $c$: $\tau_i^c$ is the
                  prospective compressed tuple which is encoded as
                  an array of maximal length $|scp(c)|$. Its length is
                  denoted $length^c$
10             $position \leftarrow 0$
11             $length^c \leftarrow 0$
12             while $position < \tau_i.length$ do
13                 if $position = bestPattern.position$ then
14                     $\tau_i^c[length^c] \leftarrow -bestPattern.id$ // a reference
                       towards the pattern
15                     $position \leftarrow position + bestPattern.length$
16                 else
17                     $\tau_i^c[length^c] \leftarrow \tau_i[position]$
18                     $position \leftarrow position + 1$
19                 $length^c \leftarrow length^c + 1$
20     until $bestPattern = nil$
21     if $length^c < \tau_i.length$ then
22         $\tau_i \leftarrow \tau_i^c$
```

for the best pattern, we respect a decreasing order of patterns score. If a pattern is found, we add it to the patterns table and then replace its occurrence in the tuple by a reference towards it in the patterns table. This process continues until there is no possible pattern that could be replaced in the tuple. We should note that when searching for the best pattern contained in a tuple, we take into consideration the replacement done before. As a consequence, patterns don't overlap since we choose to replace always the occurrence of the best one. Once we have iterated over all values of a tuple we check if we have already replaced at least one pattern. If it is the case, the tuple $\tau_i$ of the constraint is removed and replaced by the compressed one $\tau_i^c$. It is important to note that our approach does not take into consideration the dynamic score variation. In other terms, the trie of frequent pattern is built statically before compression. However, if patterns share some values and we decide to use one of them in compression, the score of others will change during compression. Let us take the example of patterns "cacb" and "acb". If we suppose that $nbOcc(cacb) = 3$ and $nbOcc(acb) = 4$. If we decide to use the pattern *cacb*, $nbOcc(acb)$ should be updated to 1 and in this case it might be less frequent compared to the remaining patterns.

**Illustration** We give now an illustration of the whole process from patterns detection to table compression. Table 3.1 represents a positive table constraint of arity 5, involving variables $x_1, x_2, x_3, x_4$ and $x_5$. We can notice that several patterns are repeated among tuples such as *cbc*, *aab* and *abb* as patterns of length 3 and *aabb*, *acbc* and *cbca* as patterns of length 4. Through visiting all tuples of our constraint, we can build the trie illustrated in Figure 3.5, where the number of occurrences is given in each node. Every node identifies a path $\mu$ (going from the root to the leaf) associated with the nbOcc($\mu$) counter. In order to make clearer the frequent patterns detection step, in Figure 3.6, we present the same trie depicted in Figure 3.5 but by mentioning the *score* in each node rather than the *nbOcc* counter.

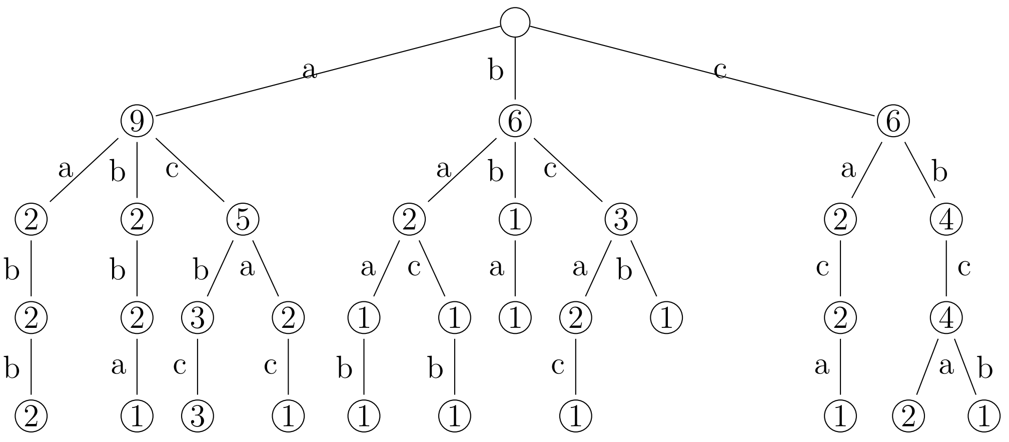


Figure 3.5: The trie of patterns built from the constraint given in Table 3.1: *nbOcc* are given in nodes.

The compression algorithm allows us to have a compressed version of the table constraint; its logical representation is given in Figure 3.7 whereas the patterns table is given on the right of the table constraint. Tuples $\tau_1$, $\tau_3$, $\tau_4$, $\tau_6$ of the compressed table respectively reference the pattern $\mu_1$ at positions 1, 2, 3

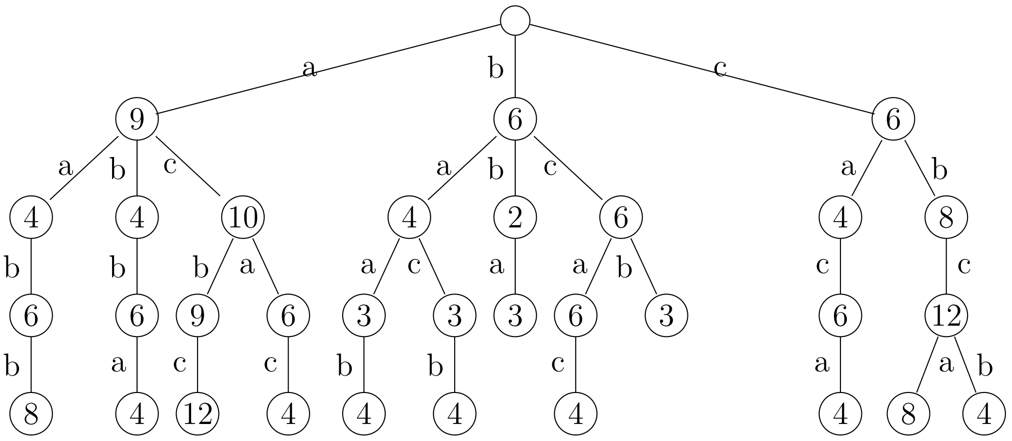


Figure 3.6: The trie of patterns built from the constraint given in Table 3.1: *score* are given in nodes.

and 2, and tuples $\tau_2$, $\tau_5$ respectively reference the pattern $\mu_2$ at positions 1 and 2.

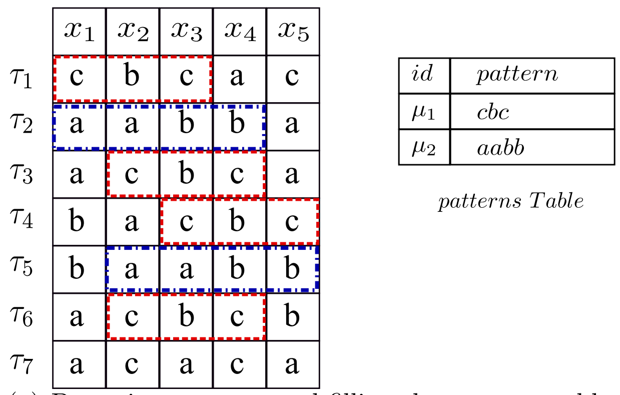
This mechanism of compression can entail an important reduction of the space occupied by the table. This is illustrated by the physical view given in Figure 3.8. The compressed tuples reference their corresponding patterns stored in the patterns table. The table constraint is, thus, composed of tuples of different sizes since they contain patterns of different sizes.

Figure 3.9 shows the space occupied by the two forms of table constraints: the classic and the compressed one. The space required to represent a table constraint is:
$\Sigma_i$ length($\tau_i$)= number of tuples $\times$ length of a tuple = $t \times |scp(c)|= 7 \times 5 = 35$
The space required by the compressed form of the table constraint is:
$\Sigma_i$ length($\tau_i$) + $\Sigma_i$ length($\mu_j$)= (3+2+3+3+2+3+5) + (3+4)=29

## 3.2.3 Filtering algorithm

During the filtering of a table constraint, it is necessary to check the validity of tuples, which implies to verify the validity of the patterns. When a pattern appears several times in the table at the same position, we wish to test the validity of the pattern only once, which allows us to speed up the filtering.

To do this, we use a counter `currentTime`, which is incremented every time we filter the table constraint, and an array of timestamps `stamps`$[\mu_i, j]$. For a pattern $\mu_i$, which occurs at a position $j$, `stamps`$[\mu_i, j]$ gives the result of the last test of validity $(\mu_i, j)$ (field `valid`) as well as the value of the counter `time` when this test was made. Every time the validity of $(\mu_i, j)$ must be tested, we

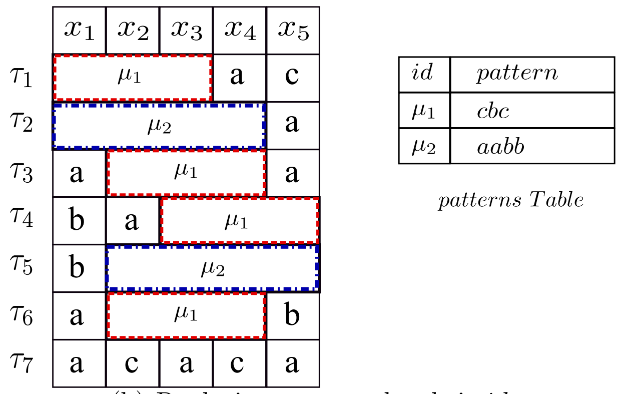


| | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|---|
| $\tau_1$ | c | b | c | a | c |
| $\tau_2$ | a | a | b | b | a |
| $\tau_3$ | a | c | b | c | a |
| $\tau_4$ | b | a | c | b | c |
| $\tau_5$ | b | a | a | b | b |
| $\tau_6$ | a | c | b | c | b |
| $\tau_7$ | a | c | a | c | a |

| *id* | *pattern* |
|---|---|
| $\mu_1$ | *cbc* |
| $\mu_2$ | *aabb* |

*patterns Table*

(a) Detecting patterns and filling the patterns table.



| | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|---|
| $\tau_1$ | $\mu_1$ | | | a | c |
| $\tau_2$ | $\mu_2$ | | | | a |
| $\tau_3$ | a | $\mu_1$ | | | a |
| $\tau_4$ | b | a | $\mu_1$ | | |
| $\tau_5$ | b | $\mu_2$ | | | |
| $\tau_6$ | a | $\mu_1$ | | | b |
| $\tau_7$ | a | c | a | c | a |

| *id* | *pattern* |
|---|---|
| $\mu_1$ | *cbc* |
| $\mu_2$ | *aabb* |

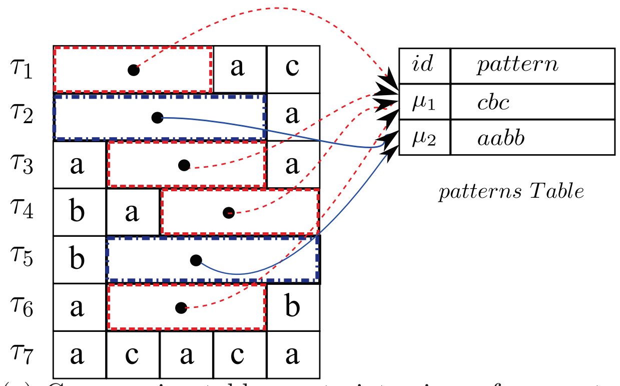
*patterns Table*

(b) Replacing patterns by their *id*.

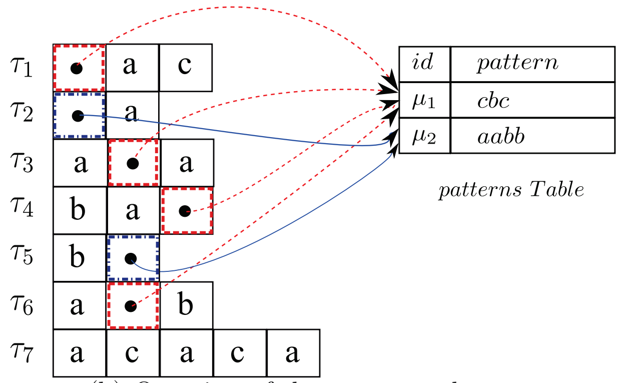
Figure 3.7: Compressed table (logical view).

verify first if the value `time` of `stamps`[$\mu_{\mathtt{i}}$, `j`] is equal to the current value of `currentTime`. If it is the case, the validity was already tested in the current step of filtering and, thus, the value `valid` of `stamps`[$\mu_{\mathtt{i}}$, `j`] directly supplies the answer, which avoids useless calculations. Otherwise, it is necessary to test the validity of $(\mu_i, j)$ and to store the result in `stamps`[$\mu_{\mathtt{i}}$, `j`].

Figure 3.10 presents an example of the evolution of the structure `stamps`. First, all elements are initialized at `currentTime`=0 and the fields `valid` remain unassigned (see Figure 3.10(a)). During the first filtering of the table, the global counter `currentTime` is set to 1. To determine whether tuple $\tau_3$ (for example) is valid, it is necessary to ensure that $\mu_1$ at position $j = 2$ is valid. As the value `time` of `stamps`[$\mu_1$,2] is not equal to the current value of `currentTime`, we know that this test was not already made in the current filtering step. Thus, we verify whether $c$, $b$ and $c$ are still present in the domains of $x_2$, $x_3$ and $x_4$ respectively. We suppose here that the result is positive. Hence, the values 1 and *true* are stored respectively in the fields `time` and `valid` of `stamps`[$\mu_1$,2]. Later, when we test the validity of tuple $\tau_6$, we notice immediately (our assumption) that the

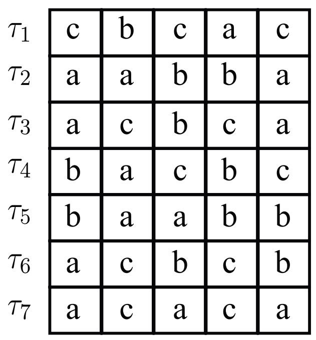


(a) Compressing table constraint using references towards the patterns table.



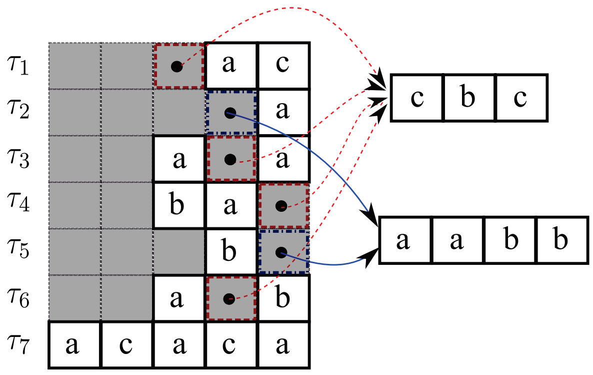
(b) Overview of the compressed space.

Figure 3.8: Compressed table (physical view).



|  |  |  |  |  |  |
|---|---|---|---|---|---|
| $\tau_1$ | c | b | c | a | c |
| $\tau_2$ | a | a | b | b | a |
| $\tau_3$ | a | c | b | c | a |
| $\tau_4$ | b | a | c | b | c |
| $\tau_5$ | b | a | a | b | b |
| $\tau_6$ | a | c | b | c | b |
| $\tau_7$ | a | c | a | c | a |

(a) Before compression.



(b) Overview of the compressed space.

Figure 3.9: Compressed table (Gained space).

validity of $\mu_1$ was tested during the current filtering and it is enough to use the stored result in the field `valid` of `stamps`[$\mu_1$,2]. Assuming that $\mu_1$ is invalid at

| *id* \ *position* | 1 | 2 | 3 |
|---|---|---|---|
| $\mu_1$ | $time = 0$ $valid =?$ | $time = 0$ $valid =?$ | $time = 0$ $valid =?$ |
| $\mu_2$ | $time = 0$ $valid =?$ | $time = 0$ $valid =?$ | $time = 0$ $valid =?$ |

(a) Initialisation (`currentTime`=0).

| *id* \ *position* | 1 | 2 | 3 |
|---|---|---|---|
| $\mu_1$ | $time = 1$ $valid = F$ | $time = 1$ $valid = T$ | $time = 1$ $valid = F$ |
| $\mu_2$ | $time = 1$ $valid = T$ | $time = 1$ $valid = T$ | $time = 0$ $valid =?$ |

(b) At the end of `currentTime`=1.

Figure 3.10: Evolution of the `stamps` structure.

position 1 and 3, valid at position 2 and $\mu_2$ valid at position 1 and 2, we obtain at the end of the filtering the result presented in Figure 3.10(b).

For a constraint of arity $r$ and a pattern $\mu$, there is at most $r - |\mu| + 1$ possible pairs of $(\mu, j)$ (because 1$\leq$ j$\leq$ r-$|\mu|$+ 1). So, the structure `stamps` has a size $O(mr)$ where $m$ is the number of detected patterns.

Algorithm 25 describes the filtering process based on STR. It first goes through all current (valid) compressed tuples. In fact, current tuples are delimited by a pointer called `currentLimit`. This technique is used in almost all STR algorithms to separate valid tuples from the invalid ones in the table constraint (Section 1.3.2). The test of validity of compressed tuples is done by Algorithm 26, which indicates whether the compressed tuple is still valid from the last test or not. If the tuple is valid at `currentTime`, Algorithm 25 updates the set of values to be supported by the future (unassigned) variables of the constraint scope (Lines 8 - 10) as in STR. Otherwise, if the tuple remains not valid, it is removed from the valid tuples set by updating the `currentLimit` pointer. At the end of the algorithm, updating domains is required.

To test the validity of a compressed tuple, we should test every value composing it including the prospective patterns. A value is considered valid if it is present in the current domain of its variable. If the tuple references a pattern (Line 7), we verify if the `stamps` data structure is updated at `currentTime`. If it is the case, we benefit from the result of the test of validity stored in the field `valid` (Lines 9 - 11), otherwise, we proceed to the test of validity and we store it for further tests (Lines 13 - 19).

**Illustration** Figure 3.11 describes the evolution of the `stamps` data structure by the call of Algorithm 25 due to the event $x_3 \neq c$. In Figure 3.11(a), we test the validity of $\tau_1$. Since $\tau_1$ references the pattern $\mu_1$ at its first position, we start by testing the validity of the different values composing $\mu_1$. The pattern $\mu_1$ involves the literal ($x_3$,$c$) which implies its invalidity. The result is, thus, stored in `stamps`[$\mu_1$,1], $\tau_1$ is considered invalid (swapping it with the last valid tuple) and *currentLimit* is decremented. Figures 3.11(b), 3.11(c), 3.11(d), 3.11(e), 3.11(f) and 3.11(g) illustrate the different tests of validity done respectively over tuples

**Algorithm 25:** STR$^c$($P$ :constraint network, $c$: constraint):set of variables

```
   // Initialization of sets $gacValues$, as in STR
1  foreach variable $x \in scp(c) \mid x \notin \text{past}(P)$ do
2      gacValues[$x$] $\leftarrow \emptyset$
   // Iteration over all current compressed tuples of $c$
3  $i \leftarrow 1$
4  while $i \leq$ currentLimit[$c$] do
5      $index \leftarrow$ position[$c$][$i$]
6      $\tau \leftarrow$ table[$c$][$index$]
7      if $isValidTuple^{comp}(c, \tau)$ then
8          foreach variable $x \in scp(c) \mid x \notin \text{past}(P)$ do
9              if $\tau[x] \notin$ gacValues[$x$] then
10                 gacValues[$x$] $\leftarrow$ gacValues[$x$] $\cup \{\tau[x]\}$
11         $i \leftarrow i + 1$
12     else
13         $removeTuple(c, i, |\text{past}(P)|)$   // currentLimit[$c$] decremented
   // domains are now updated and $X_{evt}$ computed
14 $X_{evt} \leftarrow \emptyset$
15 foreach variable $x \in scp(c) \mid x \notin \text{past}(P)$ do
16     if gacValues[$x$] $\subset dom(x)$ then
17         $dom(x) \leftarrow gacValues[x]$
18         if $dom(x) = \emptyset$ then
19             throw INCONSISTENCY
20         $X_{evt} \leftarrow X_{evt} \cup \{x\}$
21 return $X_{evt}$
```

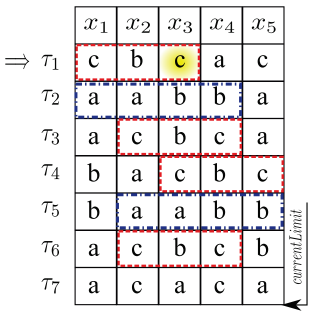
**Algorithm 26:** $isValidTuple^{comp}$($c$: Constraint, $\tau$: Tuple): Boolean

```
1  $oldPosition \leftarrow 0$
2  foreach $i \in [0..|scp(c|]$ do
       // compressed tuples contain values and references to
          patterns
3      if $isValue(\tau[i])$ then
4          if $\tau[i] \notin dom(scp(c)[oldPosition])$ then
5              return $false$
6          $oldPosition \leftarrow oldPosition + 1$
7      else
           // The case of a reference towards a pattern
8          $\mu \leftarrow patterns(|\tau[i]|)$
9          if $stamps[\mu, i].time = currentTime$ then
               // The pattern was tested at the instant currentTime
                  and it is not valid
10             if $stamps[\mu, i].valid = false$ then
11                 return $false$
12         else
               // The pattern is not tested yet at $currentTime$
13             $stamps[\mu, i].time \leftarrow currentTime$
14             foreach $j \in [0..|\mu|]$ do
15                 if $\mu[j] \notin dom(scp(c)[oldPosition])$ then
16                     $stamps[\mu, i].valid \leftarrow false$
17                     return $false$
18                 $oldPosition \leftarrow oldPosition + 1$
19             $stamps[\mu, i].valid \leftarrow true$
20 return $true$
```

$\tau_7$, $\tau_2$, $\tau_3$, $\tau_4$, $\tau_6$ and $\tau_5$. We can also observe the evolution of the `stamps` structure after each test. In Figure 3.11(f), we test the validity of tuple $\tau_6$, which references pattern $\mu_1$ at position 2. Thanks to the result stored previously in `stamps`[$\mu_1$,2] when testing the validity of $\tau_3$, we avoid re-checking the validity of the different values composing $\mu_1$. The final result of validity tests at $currentTime$=1 is given by Figure 3.11(g).

### 3.2.4 Experimental results

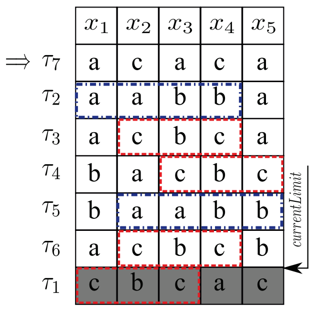
In order to show the practical interest of our approach (STR$^c$), we compared the behavior of STR1 [Ullmann, 2007], STR2 [Lecoutre, 2011], STR3



| | | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|---|---|
| $\Rightarrow$ | $\tau_1$ | c | b | c | a | c |
| | $\tau_2$ | a | a | b | b | a |
| | $\tau_3$ | a | c | b | c | a |
| | $\tau_4$ | b | a | c | b | c |
| | $\tau_5$ | b | a | a | b | b |
| | $\tau_6$ | a | c | b | c | b |
| | $\tau_7$ | a | c | a | c | a |

| *position* / *id* | 1 | 2 | 3 |
|---|---|---|---|
| $\mu_1$ | $time = 1$ $valid = F$ | $time = 0$ $valid = ?$ | $time = 0$ $valid = ?$ |
| $\mu_2$ | $time = 0$ $valid = ?$ | $time = 0$ $valid = ?$ | $time = 0$ $valid = ?$ |

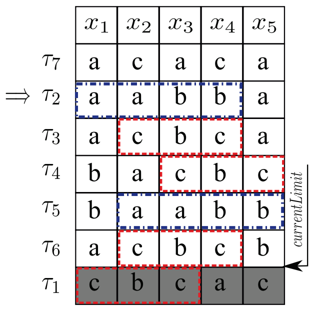
(a) $\tau_1$ validity test.



| | | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|---|---|
| $\Rightarrow$ | $\tau_7$ | a | c | a | c | a |
| | $\tau_2$ | a | a | b | b | a |
| | $\tau_3$ | a | c | b | c | a |
| | $\tau_4$ | b | a | c | b | c |
| | $\tau_5$ | b | a | a | b | b |
| | $\tau_6$ | a | c | b | c | b |
| | $\tau_1$ | c | b | c | a | c |

| *position* / *id* | 1 | 2 | 3 |
|---|---|---|---|
| $\mu_1$ | $time = 1$ $valid = F$ | $time = 0$ $valid = ?$ | $time = 0$ $valid = ?$ |
| $\mu_2$ | $time = 1$ $valid = T$ | $time = 0$ $valid = ?$ | $time = 0$ $valid = ?$ |

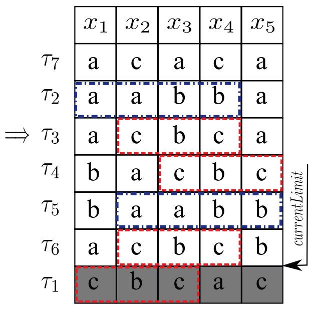
(b) $\tau_7$ validity test.



| | | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|---|---|
| | $\tau_7$ | a | c | a | c | a |
| $\Rightarrow$ | $\tau_2$ | a | a | b | b | a |
| | $\tau_3$ | a | c | b | c | a |
| | $\tau_4$ | b | a | c | b | c |
| | $\tau_5$ | b | a | a | b | b |
| | $\tau_6$ | a | c | b | c | b |
| | $\tau_1$ | c | b | c | a | c |

| *position* / *id* | 1 | 2 | 3 |
|---|---|---|---|
| $\mu_1$ | $time = 1$ $valid = F$ | $time = 0$ $valid = ?$ | $time = 0$ $valid = ?$ |
| $\mu_2$ | $time = 1$ $valid = T$ | $time = 0$ $valid = ?$ | $time = 0$ $valid = ?$ |

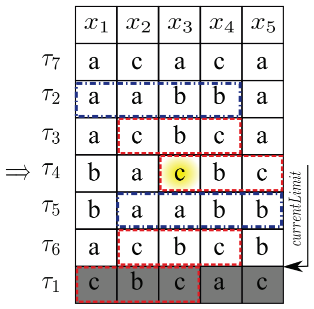
(c) $\tau_2$ validity test.



| | | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|---|---|
| | $\tau_7$ | a | c | a | c | a |
| | $\tau_2$ | a | a | b | b | a |
| $\Rightarrow$ | $\tau_3$ | a | c | b | c | a |
| | $\tau_4$ | b | a | c | b | c |
| | $\tau_5$ | b | a | a | b | b |
| | $\tau_6$ | a | c | b | c | b |
| | $\tau_1$ | c | b | c | a | c |

| *position* / *id* | 1 | 2 | 3 |
|---|---|---|---|
| $\mu_1$ | $time = 1$ $valid = F$ | $time = 1$ $valid = T$ | $time = 0$ $valid = ?$ |
| $\mu_2$ | $time = 1$ $valid = T$ | $time = 0$ $valid = ?$ | $time = 0$ $valid = ?$ |

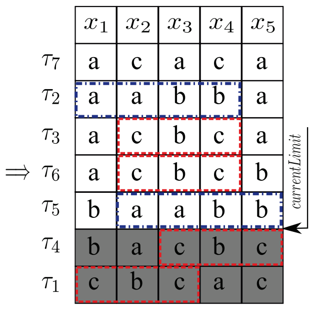
(d) $\tau_3$ validity test.



| | | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|---|---|
| | $\tau_7$ | a | c | a | c | a |
| | $\tau_2$ | a | a | b | b | a |
| | $\tau_3$ | a | c | b | c | a |
| $\Rightarrow$ | $\tau_4$ | b | a | c | b | c |
| | $\tau_5$ | b | a | a | b | b |
| | $\tau_6$ | a | c | b | c | b |
| | $\tau_1$ | c | b | c | a | c |

| *position* / *id* | 1 | 2 | 3 |
|---|---|---|---|
| $\mu_1$ | $time = 1$ $valid = F$ | $time = 1$ $valid = T$ | $time = 1$ $valid = F$ |
| $\mu_2$ | $time = 1$ $valid = T$ | $time = 0$ $valid = ?$ | $time = 0$ $valid = ?$ |

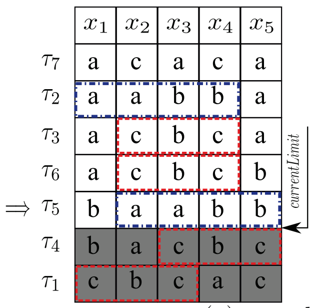
(e) $\tau_4$ validity test.



| | | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|---|---|
| | $\tau_7$ | a | c | a | c | a |
| | $\tau_2$ | a | a | b | b | a |
| | $\tau_3$ | a | c | b | c | a |
| $\Rightarrow$ | $\tau_6$ | a | c | b | c | b |
| | $\tau_5$ | b | a | a | b | b |
| | $\tau_4$ | b | a | c | b | c |
| | $\tau_1$ | c | b | c | a | c |

| *position* / *id* | 1 | 2 | 3 |
|---|---|---|---|
| $\mu_1$ | $time = 1$ $valid = F$ | $time = 1$ $valid = T$ | $time = 1$ $valid = F$ |
| $\mu_2$ | $time = 1$ $valid = T$ | $time = 0$ $valid = ?$ | $time = 0$ $valid = ?$ |

(f) $\tau_6$ validity test.



| | | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|---|---|
| | $\tau_7$ | a | c | a | c | a |
| | $\tau_2$ | a | a | b | b | a |
| | $\tau_3$ | a | c | b | c | a |
| | $\tau_6$ | a | c | b | c | b |
| $\Rightarrow$ | $\tau_5$ | b | a | a | b | b |
| | $\tau_4$ | b | a | c | b | c |
| | $\tau_1$ | c | b | c | a | c |

| *position* / *id* | 1 | 2 | 3 |
|---|---|---|---|
| $\mu_1$ | $time = 1$ $valid = F$ | $time = 1$ $valid = T$ | $time = 1$ $valid = F$ |
| $\mu_2$ | $time = 1$ $valid = T$ | $time = 1$ $valid = T$ | $time = 0$ $valid = ?$ |

(g) $\tau_5$ validity test.

Figure 3.11: Example of the `stamps` data structure evolution at $currentTime = 1$ ($event\ x_3 \neq c$).

[Lecoutre et al., 2012] and STR$^c$ algorithms when they are integrated into the search algorithm MAC (which maintains the property of generalized arc consistency during search) on one hand and enforcing GAC using MDD on the other hand. To validate our approach, we made some tests on instances of four distinct problems available at `http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html`: mdd [Cheng and Yap, 2010], bdd, crosswords, nonograms [Pesant et al., 2012] and random series. The experimental results on representative instances are given in Table 3.2; the heuristic *dom/ddeg* is used to ensure the same search path and *lexico* as value ordering heuristic.

We have conducted an experimentation with our solver AbsCon (Section 1.1.4) using a cluster of bi-quad cores Xeon processors at 2.66 GHz node with 16GiB of RAM under Linux. A time-out of $1,200$ seconds was set per instance.

On such instances (given in Table 3.5), STR$^c$ allows an important spatial reduction that may achieve 90 % (the crossword case) compared to the constraints used by the other approaches. For STR$^c$ we give resolution times, the compression ratio defined as the size of compressed tables over the size of initial tables and the time required to build the instance (including compression). The size of a compressed table is the size of compressed tuples and the patterns table. When we consider the time of search, it seems that STR$^c$ competes with STR1, but stay, however, supplanted by STR2. Compared to MDD, in the case where MDD supplants STR approaches, STR$^c$ doesn't give better times (mdd and nonograms). However, in the other case our approach competes with MDD and could even give better times (bdd-21-133, crossword).

Figures 3.12 and 3.13 give the number of patterns used per length for two constraints of two different problems as an example to show the distribution of used patterns based on their length. Figure 3.12 gives the distribution of patterns used for a constraint of arity 9 on a *crossword*-*m*1-*ogd*-23-04 instance. 69% of patterns are of length 5 which allows an efficient compression. Figure 3.13 gives the number of patterns used for a bigger constraint of arity 38 on a $nonogram - gp - 108$ instance. The most used patterns have lengths between 17 and 21.

Figure 3.14 shows two different curves: the dashed one for the detected frequent patterns using the trie data structure and the continuous one representing the patterns used in compressing the table constraint. There is a big difference that reaches the double, between the number of detected patterns and the used ones in the beginning of the curves. This means that when *scoreThreshold* is small there are many patterns detected, but when replacing their occurrences the algorithm uses only few of them. This could be explained by the decreasing order of scores used when compressing: the more frequent pattern is replaced first. Coming to the less frequent pattern, there is no tuples left. It is important to note that the number of used and detected patterns decreases when the score starts to be bigger. More and more patterns tends to be infrequent. The patterns

| Instance | MDD | STR1 | STR2 | STR3 | STR$^c$ |
|---|---|---|---|---|---|
| *a7-v24-d5-ps0.5-psh0.7-9* | **17.819** | 879 | 334 | 367 | 780.318 (43.775% – 12.192) |
| *a7-v24-d5-ps0.5-psh0.9-2* | **5.536** | 256 | 145 | 143 | 283.828 (43.776% – 11.511) |
| *a7-v24-d5-ps0.5-psh0.9-6* | **11.397** | 353 | 195 | 324 | 420.739 (43,776% – 11.628) |
| *bdd-21-2713-15-79-9* | 77.171 | 80.2 | **23.3** | 60.0 | 84.608 (82,612% – 1.05) |
| *bdd-21-2713-15-79-11* | 55.752 | 78.5 | **23.5** | 48.5 | 74.796 (82,643% – 1.164) |
| *bdd-21-133-18-78-11* | 47.157 | 38.4 | **11,0** | 250 | 33.538 (83.975% – 1.751) |
| *crossword-m1-ogd-23-04* | 104.485 | 82.7 | **78.2** | 103 | 80.085 (10.688% – 2.919) |
| *crossword-m1c-lex-vg5-7-* | 86.475 | 43.5 | **31.4** | 36.6 | 49.393 (20.134% – 0.5) |
| *crossword-m1c-ogd-vg10-13* | TO | TO | 897 | **765** | TO |
| *nonogram-gp-108* | 55.732 | 290 | **78.7** | 118 | 319.613 (94.519% – 20.561) |
| *nonogram-gp-116* | 16.658 | 102 | **21.7** | 21.9 | 108.098 (95.7% – 7.948) |
| *nonogram-gp-137* | 8.359 | 166 | **45.4** | 56.8 | 146.142 (92.967% – 42.072) |
| *rand-6-10-10-60-950-0* | 67.821 | 75.1 | 45.4 | **34.4** | 104.554 (34.317% – 21.768) |
| *rand-7-9-9-30-980-0* | 47.285 | 26.8 | **18.5** | 46.3 | 50.271 (57.469% – 19.383) |
| *rand-8-20-5-18-800-8* | 83.815 | 86.6 | **45.8** | 574 | 111.949 (62,960% – 8.744) |

Table 3.2: CPU time (in seconds) on some selected instances solved by MAC. Compression ratio and CPU time are given for STR$^c$ between parentheses.

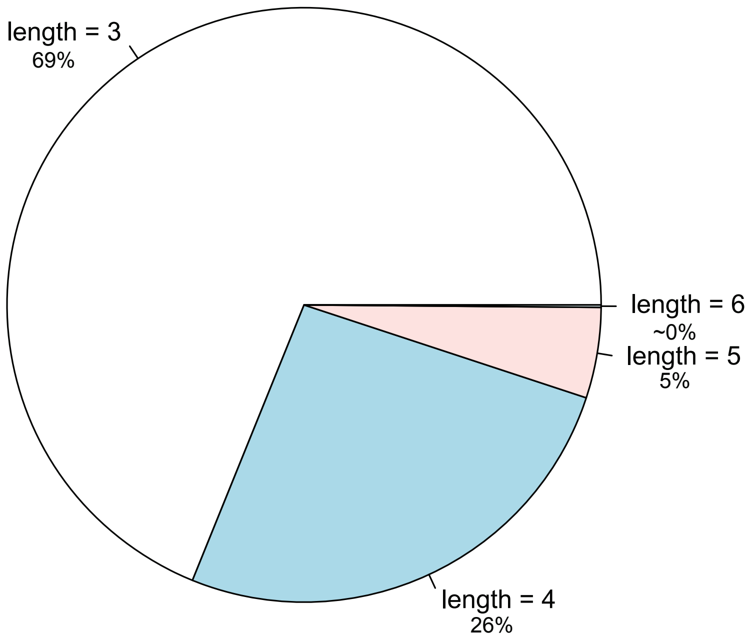


Figure 3.12: Pie chart of patterns (totalPatterns=771) used per length for a constraint of arity 9 (nbTuples=63138) of the instance problem *crossword-m1-ogd-23-04*.

number reaches 0 at $scoreThreshold$=282.

Figures 3.15 illustrates the variation of the compression ratio according to the $scoreThreshold$ for the instance *crossword-lex-vg*5-7. Since the number of used patterns detected is affected by the score rise, the compression ratio is also concerned by this factor. The compression ratio decreases with the score rise until there is no frequent patterns detected (at $scoreThreshold$=282), and thus, the table constraint cannot be compressed. In fact, when the $scoreThreshold$ rise that means that either the patterns detected are more and more frequent or the patterns are longer which implies an important reduction when replacing their occurrences.

## 3.3 STR-slice

In the first work, we defined a pattern as a sequence of consecutive values. Although this definition allowed us to obtain an important space complexity reduction, the compressed form of the table constraint presents some drawbacks especially during the filtering process preventing us to use optimized STR techniques. For each test of tuple validity, iterating over all its values is necessary (if

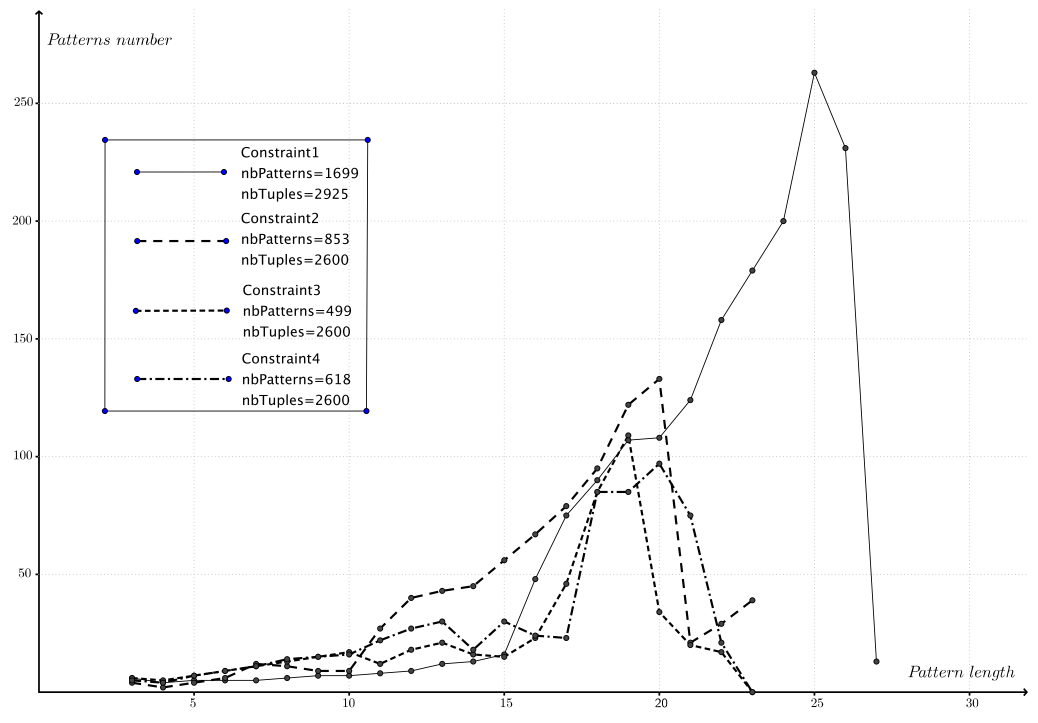


Figure 3.13: Number of patterns used per length for 4 constraints of arity 32 of the instance problem $nonogram - gp - 108$.

we are not in the case of re-using the results previously done and stored in the specified data structure). These drawbacks are due to the pattern definition: a pattern could involve different variables depending on the position in which it occurs.

Aiming to be able to benefit from optimized STR techniques, we proposed in this second work a new definition of pattern and, thus, a new compressed form of table constraints by means of data-mining techniques.

This section is organized as follows: after explaining the relaxation of pattern definition in Section 3.3.1, we present, in Section 3.3.2, a compression process for table constraints, introducing the algorithm used to obtain the new form of table constraints (called "sliced" table constraints). Next, we describe, in Section 3.3.3, an optimized algorithm to enforce GAC on "sliced" table constraints. Finally, we conclude by giving some experimental results in Section 3.3.4.

## 3.3.1 Pattern Definition

In this subsection, we introduce the new concepts of pattern and sub-table that are useful for the compression process.

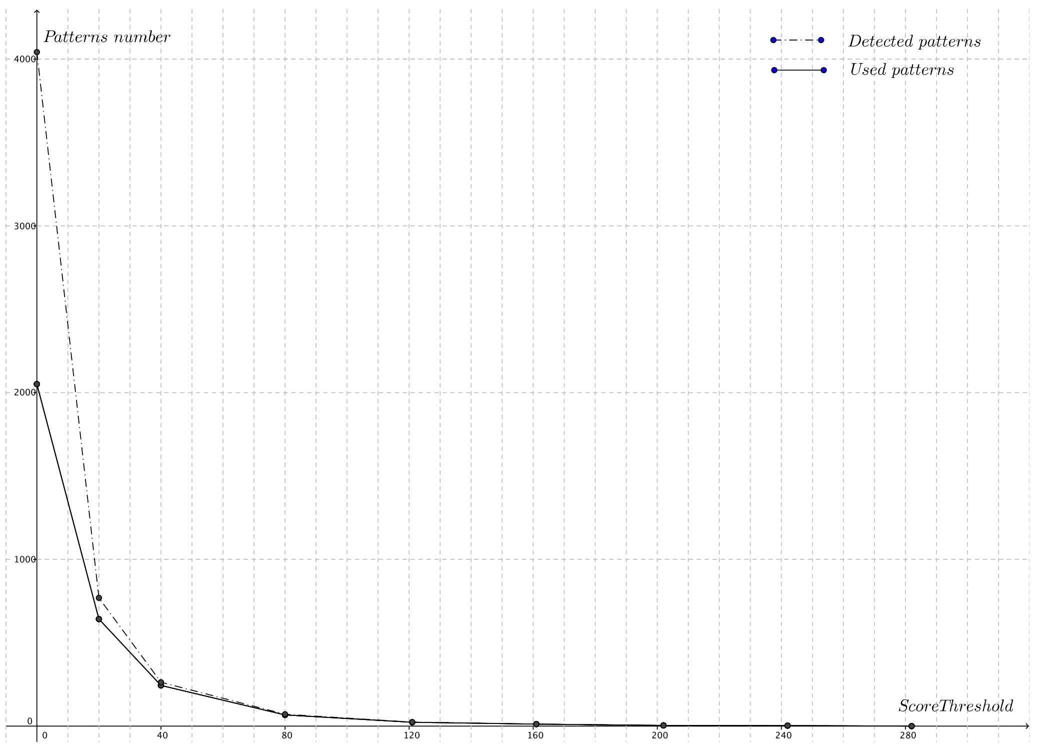


Figure 3.14: Number of patterns (detected and used) per score threshold for a constraint of arity 7 of the instance problem *crossword-lex-vg*5-7.

**Definition 32** *A* ***pattern*** $\mu$ *of a constraint* $c$ *is* ***an instantiation*** $I$ ***of some variables*** *of* $c$*. We note scp*$(\mu)$ *its scope, which is equal to vars*$(I)$*,* $|\mu|$ *its length, which is equal to* $|scp(\mu)|$*, and nbOcc*$(\mu)$ *its number of occurrences in rel*$(c)$*, which is* $|\{\tau \in rel(c) \mid \mu \subseteq \tau\}|$.

**Example 28** *Let* $c$ *be a positive table constraint on variables* $x_1$*,* $x_2$*,* $x_3$*,* $x_4$ *and* $x_5$ *with* $dom(x_1) = dom(x_2) = dom(x_3) = dom(x_4) = dom(x_5) = \{a, b, c\}$*. Table 3.3 represents a constraint* $c$ *with* 7 *allowed tuples.*

*In Table 3.3,* $\mu_1$=$\{(x_1, a), (x_4, c), (x_5, a)\}$ *and* $\mu_2$=$\{x_1 = b, x_2 = a\}$ *are patterns of length* 3 *and* 2*, with scp*$(\mu_1)$=$\{x_1, x_4, x_5\}$ *and scp*$(\mu_2)$=$\{x_1, x_2\}$*. Their number of occurrences are respectively* 3 *and* 2.

We present above a new definition of pattern different from the one chosen in the first work (Section 3.2). The pattern depends no more on a consecutive sequence of values, but rather on variables assignments, which are no more expected to be consecutive.

For our new compressed form, we chose to present in another way the patterns. They are no more gathered in a patterns table, but presented with the

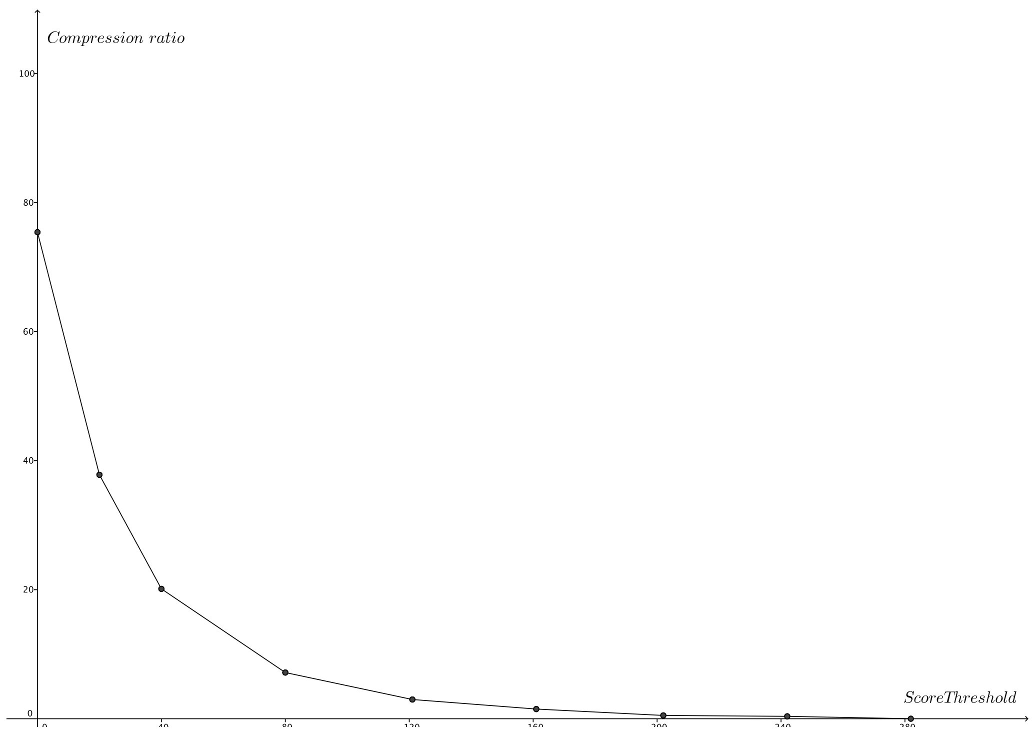


Figure 3.15: Compression ratio per score threshold for a constraint of arity 7 of the instance problem *crossword-lex-vg*5-7.

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|
| c | b | c | a | c |
| a | a | b | c | a |
| a | c | b | c | a |
| b | a | c | b | c |
| b | a | a | b | b |
| c | c | b | c | a |
| a | c | a | c | a |

Table 3.3: Table constraint $c$ on $x_1, x_2, x_3, x_4, x_5$.

remaining part of the tuples ("sub-table") as a unique entity called "entry" composing the final form of the constraint. These two data structures are introduced in Definitions 33 and 34.

**Definition 33** *(Sub-table) The* ***sub-table*** *$T$ associated with a pattern $\mu$ of a constraint $c$ is obtained by removing $\mu$ from tuples of $c$ that contain $\mu$ and ignoring other tuples.*

$$T = \{\tau \setminus \mu \mid \tau \in rel(c) \wedge \mu \subseteq \tau\}$$

*The scope of $T$ is $scp(T) = scp(c) - scp(\mu)$.*

**Example 29** *Table 3.4 represents the sub-table associated with the pattern $\mu_1$=$\{x_1 = a, x_4 = c, x_5 = a\}$ of $c$, presented in Table 3.3.*

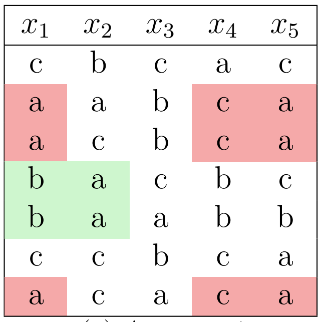
| $x_2$ | $x_3$ |
|---|---|
| a | b |
| c | b |
| c | a |

Table 3.4: The sub-table $T_1$ associated with the pattern $\mu_1$ of $c$.

**Definition 34** *(entry) An **entry** for a constraint $c$ is a pair $(\mu, T)$ such that $\mu$ is a pattern of $c$ and $T$ is the sub-table associated with $\mu$.*

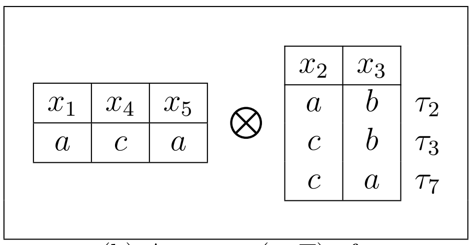
Since the set of tuples represented by an entry $(\mu, T)$ represents in fact the Cartesian product of $\mu$ by $T$, we shall also use the notation $\mu \otimes T$ to denote a constraint entry. Notice that after the compressing ("slicing") process of a constraint into a set of entries, the set of tuples, which are not associated with any pattern can be stored in a so called *default entry* denoted by $(\emptyset, T)$.

**Example 30** *The pattern $\mu_1$=$\{(x_1, a), (x_4, c), (x_5, a)\}$ of constraint $c$, depicted in Figure 3.16(a), appears in tuples $\tau_2$, $\tau_3$ and $\tau_7$. $\mu_1$ and the resulting sub-table form an entry for $c$, as shown in Figure 3.16(b).*



| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|
| c | b | c | a | c |
| a 🟥 | a | b | c 🟥 | a 🟥 |
| a 🟥 | c | b | c 🟥 | a 🟥 |
| b 🟩 | a 🟩 | c | b | c |
| b 🟩 | a 🟩 | a | b | b |
| c | c | b | c | a |
| a 🟥 | c | a | c 🟥 | a 🟥 |

(a) A constraint $c$.



| $x_1$ | $x_4$ | $x_5$ |
|---|---|---|
| $a$ | $c$ | $a$ |

$\otimes$

| $x_2$ | $x_3$ | |
|---|---|---|
| $a$ | $b$ | $\tau_2$ |
| $c$ | $b$ | $\tau_3$ |
| $c$ | $a$ | $\tau_7$ |

(b) An entry $(\mu, T)$ of $c$.

Figure 3.16: Example of a constraint entry.

**Definition 35** *("Sliced" table constraint) A "sliced" table constraint is a compressed form of table constraint composed of a set of entries.*

Testing the validity of classical or compressed tuples is an important operation in filtering algorithms of (compressed) table constraints. For sliced table constraints, we extend the notion of validity to constraint entries.

**Definition 36** *(Pattern) A pattern $\mu$, is* valid *iff* $\{\forall x \in \mu, \mu[x] \in dom(x)\}$

A table constraint is valid iff all its current tuples are valid. This validity is extended to *sub-table*, since it is considered the same data structure having a scope included in the scope of the original table constraint.

**Definition 37** *(Validity of an entry) An entry $(\mu, T)$ is* valid *iff at least one tuple of the Cartesian product $\mu \otimes T$ is valid. Equivalently, an entry is valid iff its pattern is valid and its sub-table contains at least one valid sub-tuple.*

### 3.3.2 Compression Method

In order to build the "sliced" table constraint, we apply the *FP-Growth* algorithm (Section 2.1.2) which is a technique used in data mining. To do this, we just use the first step of the algorithm (the FP-Tree construction), and then, extract classically the frequent patterns. In the context of table constraints, a transaction is a tuple; an item is, in fact, an assignment of value to a variable and a frequent itemset is a pattern, which is a partial instantiation.

We give now an example of the construction of the *FP-Tree* in the context of table compression, using the constraint given in Table 3.3 as an example. In our example, we shall use *minSupport*=2 to identify patterns that occur at least twice.

**First step: *FP-Tree* building** In this step, the algorithm goes through all tuples of a table constraint:

- we calculate for each value its support;
- we order the tuples in decreasing order of values support;
- we remove the infrequent values;
- we put the different tuples one by one in an FP-Tree.

**Example 31** *We collect the number of occurrences of each value. By abuse of terminology, we shall call frequency the support of a value (number of occurrences). The result on our example is given in Figure 3.17.*

| s($x_i$,$a_i$) | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|---|
| a | 3 | 3 | 2 | 1 | 4 |
| b | 2 | 1 | 3 | 2 | 1 |
| c | 2 | 3 | 2 | 4 | 2 |

Figure 3.17: Frequencies.

*We sort each tuple in decreasing order of frequency of values. The result is given in Figure 3.18 where the frequency of a value is given between parentheses. Values, which have a frequency below the threshold* minSupport *are removed from the tuple (they are identified in bold face) because they cannot appear in a frequent pattern.*

| | | | | | |
|---|---|---|---|---|---|
| $\tau_1$ | (2) $x_1 = c$ | (2) $x_3 = c$ | (2) $x_5 = c$ | **(1) $\mathbf{x_2 = b}$** | **(1) $\mathbf{x_4 = a}$** |
| $\tau_2$ | (4) $x_4 = c$ | (4) $x_5 = a$ | (3) $x_1 = a$ | (3) $x_2 = a$ | (3) $x_3 = b$ |
| $\tau_3$ | (4) $x_4 = c$ | (4) $x_5 = a$ | (3) $x_1 = a$ | (3) $x_2 = c$ | (3) $x_3 = b$ |
| $\tau_4$ | (3) $x_2 = a$ | (2) $x_1 = b$ | (2) $x_3 = c$ | (2) $x_4 = b$ | (2) $x_5 = c$ |
| $\tau_5$ | (3) $x_2 = a$ | (2) $x_1 = b$ | (2) $x_3 = a$ | (2) $x_4 = b$ | **(1) $\mathbf{x_5 = b}$** |
| $\tau_6$ | (4) $x_4 = c$ | (4) $x_5 = a$ | (3) $x_2 = c$ | (3) $x_3 = b$ | (2) $x_1 = c$ |
| $\tau_7$ | (4) $x_4 = c$ | (4) $x_5 = a$ | (3) $x_1 = a$ | (3) $x_2 = c$ | (2) $x_3 = a$ |

Figure 3.18: Tuples sorted according to decreasing frequencies.

*Once a tuple is sorted and possibly reduced, it is inserted in the FP-Tree. We note that we don't use the dotted links between values of different paths since we don't use the second step of the FP-Growth algorithm to extract the frequent patterns. Moreover, each edge from a parent to its child is labeled with a value rather than representing the value in nodes.*

*Figure 3.19 represents the* FP-Tree *obtained on our running example. The first tuple inserted in the tree is the beginning of $\tau_1$, that is $(x_1, c), (x_3, c), (x_5, c)$. This creates the leftmost branch of the tree. Each node of this branch is given a frequency of* 1. *The second tuple inserted is $(x_4, c), (x_5, a), (x_1, a), (x_2, a), (x_3, b)$ which creates the third leftmost branch in the tree (each node having a frequency of 1 at this step). When $\tau_3$ is inserted, the new branch $(x_4, c), (x_5, a), (x_1, a), (x_2, c), (x_3, b)$ shares its first three edges with the last branch, hence the frequency of the corresponding nodes is incremented and becomes 2. The other tuples are inserted in the same way. In the end, nodes with a frequency below the threshold* minSupport *are pruned. The remaining tree is depicted with thick lines and circled by a dashed line in Figure 3.19.*

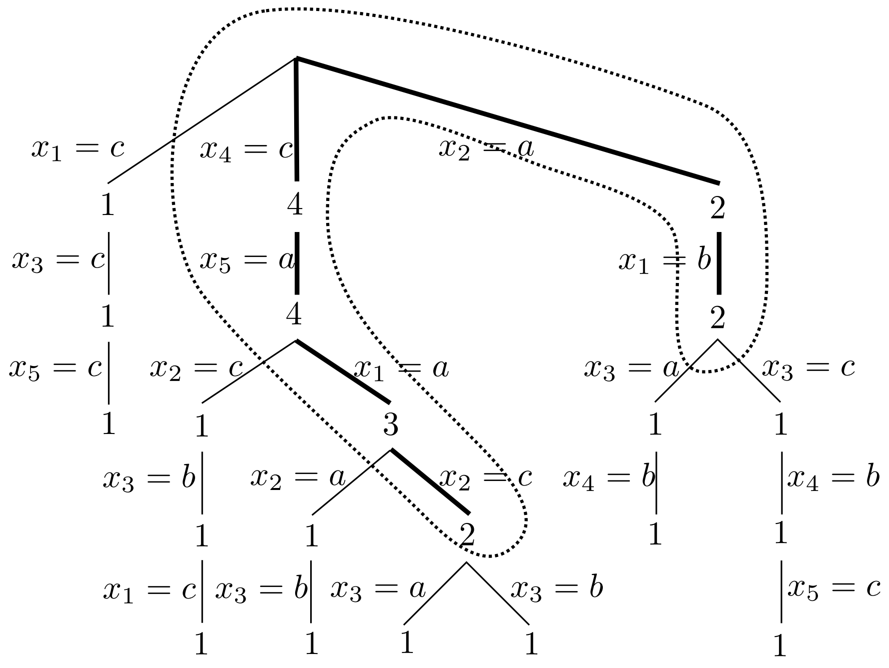


Figure 3.19: FP-tree built for the table constraint in Table 3.3.

**Second step: Frequent patterns detection** We now have to identify patterns in the *FP-Tree* which are relevant for compression. Each node of the tree corresponds to a frequent pattern $\mu$ which, can be read on the path from the root to the node. The frequency $nbOcc(\mu)$ of this pattern is given by the node itself. We choose to identify patterns based on the obtained savings while factoring this frequent pattern. In fact, the savings for a pattern are $|\mu| \times (nbOcc(\mu) - 1)$ values (we can save each occurrence of the pattern but one). In our example, we can see that the pattern $(x_4, c), (x_5, a)$ can save six values, the pattern $(x_4, c), (x_5, a), (x_1, a)$ can also save six values but the pattern $(x_4, c), (x_5, a), (x_1, a), (x_2, c)$ can save only four values. Therefore, we further prune the tree by removing nodes that save less values than their parents. The leaves of the tree we obtain represent the frequent pattern used in the compression: $(x_4, c), (x_5, a), (x_1, a)$ and $(x_2, a), (x_1, b)$.

To complete the compression, we create an entry for each frequent pattern we have identified and fill them in a last scan of the table. For each tuple, we use the *FP-Tree* to identify if the (sorted) tuple starts with a frequent pattern, in which case we add the rest of the tuple to the corresponding sub-table. Tuples which do not start with a frequent pattern are added to the default entry.

Algorithm 27 summarizes the different steps of the compression process. In a first scan of the table constraint $c$, we calculate the frequencies of each value. In a second scan, we sort each tuple by decreasing order of value frequency

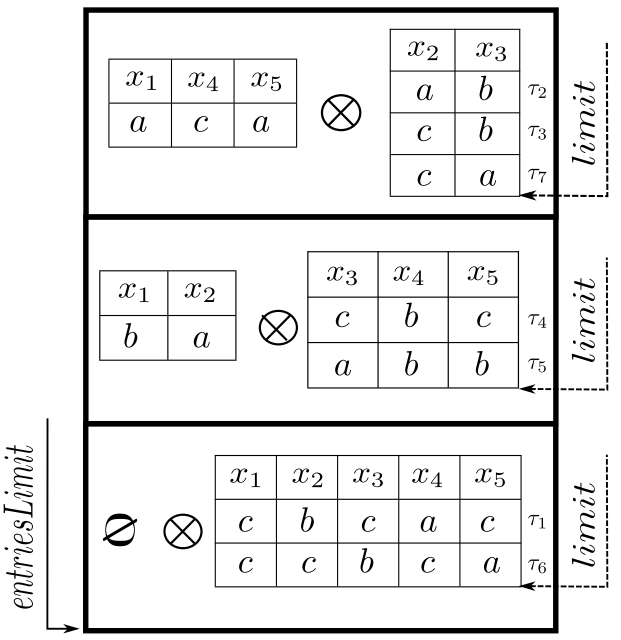


Figure 3.20: Sliced table constraint.

and remove each value having a frequency less than *minSupport*. Once a tuple is sorted, the ordered value are inserted into the *FP-Tree* (using Algorithm 19 but with a tuple rather than a pattern), if a path starting with these values does not already exist. If a path already exists, the node frequency counters for these values are updated. After building the whole *FP-Tree*, we prune all nodes having frequencies beyond *minSupport* or also those having less savings than their parents which means $|\mu_i| \times (frequency - 1) < (|\mu_i| - 1) \times (frequency' - 1)$ (Algorithm 28). To compress $c$ a last scan is required. For each tuple $\tau$ of constraint $c$, we look for a pattern $\mu$ sharing with it its first values (already ordered in the second scan) in *FP-Tree* (Algorithm 29). If there is a pattern $\mu$ found, we look in the "sliced" table constraint if there is an entry having as pattern $\mu$. If it is the case, we add the sub-tuple $(\tau \setminus \mu)$ to the sub-table of the identified entry, otherwise, we create a new entry having as pattern $\mu$ and then we put the sub-tuple $(\tau \setminus \mu)$ to the sub-table (Line 15). If there is no pattern found in the *FP-Tree*, we add the tuple $\tau$ to the $defaultEntry$ (Line 18).

Figure 3.21 describes the gained space obtained thanks to the compression process. The gained space, without considering *position* data structure (Section 1.3.2), is obtained as follows:
$\Sigma_i$ [gainedSpace(entry($i$))] $= \Sigma_i\ [|\mu_i| \times (|subTable(i)| - 1)]$
For our "sliced" table, the gained space is $3 \times 2 + 2 \times 1 = 6$

**Algorithm 27:** compress-STR-slice($c$: Constraint, $minSupport$: float)

```
   // Initialization of $frequencies$
1  foreach $i \in length(frequencies)$ do
2      $frequencies[i] \leftarrow 0$
   // First scan:compute the frequency of each value of $c$
3  foreach $\tau \in table(c)$ do
4      foreach $v \in \tau$ do
5          $frequencies[v]{+}{+}$
   // Second scan:build FP-Tree
6  foreach $\tau \in table(c)$ do
       // sorting $\tau$ and removing values less frequent than
          $minSupport$
7      $\tau \leftarrow$ sortTuple ($\tau$, $frequencies$, $minSupport$)
       // insert $\tau$ into FP-Tree by updating nodes frequency
8      addTuple($root(FP\text{-}Tree)$,$\tau$)
   // removing nodes less frequent than $minSupport$ or such that
      $|\mu| \times (f-1)$ is smaller than for their parents
9  pruneTree($root(FP\text{-}Tree)$,$minSupport$,0)
   // Third scan:compress $c$
10 foreach $\tau \in table(c)$ do
11     $\mu \leftarrow$ searchPattern($root(FP\text{-}Tree)$,$\tau$)
12     if $\mu \neq nil$ then
13         $entry \leftarrow$ getEntry($entries[c]$,$\mu$)
14         if $entry = nil$ then
               // There is no entry for this pattern
15             $entry \leftarrow$ addEntry($entries[c]$,$\mu$)
16         addSubTuple($entry$,$\tau \setminus \mu$)
17     else
18         addSubTuple($defaultEntry$,$\tau$)
```

### 3.3.3 Filtering Sliced Table Constraints

In order to enforce GAC on sliced table constraints, our idea is to adapt STR, and more specifically the optimized variant STR2, on the compressed form of this kind of constraint. As a sliced table constraint is composed of several entries, each one composed of both a pattern and a sub-table, the filtering process we propose acts at two distinct levels. At a high level, the validity of each entry is checked, and at a low-level, the validity of each pattern and each sub-tuple is checked. Remember that an entry is valid iff both its pattern is valid and at least

**Algorithm 28:** pruneTree(*node*: Node,*minSupport*: Float,*parentSavings*: Integer)

```
1 $savings \leftarrow (node.value - 1) \times node.level$
2 if $node \neq nil$ then
3     if $node.value < minSupport$ or $savings < parentSavings$ then
4         $remove(node)$
5 foreach $child \in node.getChildren()$ do
6     $pruneTree(child, minSupport, savings)$
```

**Algorithm 29:** searchPattern(*root*: Node,$\tau$: Tuple): pattern

```
1 $node \leftarrow root$
2 $\mu \leftarrow \prec\succ$
3 for $i \in [0..|\tau|]$ do
4     $node \leftarrow node.getChild(\tau[i])$
5     if $node \neq nil$ then
6         $\mu \leftarrow \mu + \tau[i]$
7     else
8         break
9 return $\mu$
```

one tuple from its sub-table is valid (see Definition 37). In this section, we first describe the employed data structures, then we introduce our GAC algorithm, and finally we give an illustration.

### Data structures

A sliced table constraint $c$ is represented by an array `entries`$[c]$ of $p$ entries. Managing the set of valid entries, called *current*[1] entries, is performed as follows:

- `entriesLimit`$[c]$ is the index of the last current entry in `entries`$[c]$. The elements in `entries`$[c]$ at indexes ranging from 1 to `entriesLimit`$[c]$ are the current entries of $c$.

- removing an entry (that has become invalid) at index $i$ is performed by a call of the form removeEntry$(c, i)$. Such a call swaps the entries at indexes $i$ and `entriesLimit`$[c]$, and then decrements `entriesLimit`$[c]$. Note that the initial order of entries is not preserved.

[1]Current entries correspond to valid entries at the end of the previous invocation of the algorithm.

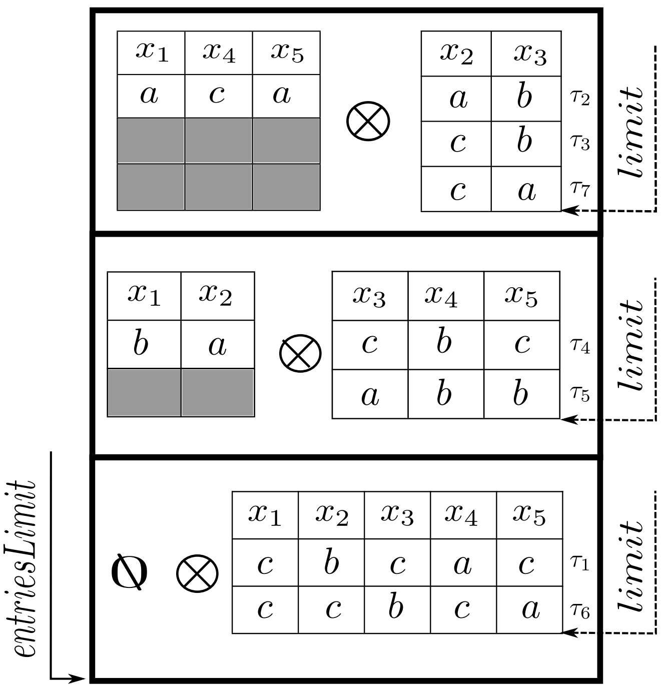


Figure 3.21: Gained Space after compression.

- restoring a set of entries can be performed by simply changing the value of `entriesLimit`[$c$].

Each entry in `entries` can be represented as a record composed of a field `pattern` and a field `subtable`. More precisely:

- the field `pattern` stores a partial instantiation $\mu$, and can be represented in practice as a record of two arrays: one for the variables, the scope of the pattern, and the other for the values.
- the field `subtable` stores a sub-table $T$, and can be represented in practice as a record of two arrays: one for the variables, i.e., the scope of the sub-table $T$, and the other, a two-dimensional array, for the sub-tuples.

Managing the set of valid sub-tuples, called *current* sub-tuples, of $T$, is performed as follows:

- *limit*[$T$] is the index of the last current sub-tuple in $T$. The elements in $T$ at indexes ranging from 1 to `limit`[$T$] are the current sub-tuples of $T$.
- removing a sub-tuple (that has become invalid) at index $i$ is performed by a call of the form removeSubtuple($T, i$). Such a call swaps the sub-tuples at indexes $i$ and `limit`[$T$], and then decrements `limit`[$T$]. Note that the initial order of sub-tuples is not preserved.

- restoring a set of sub-tuples can be performed by simply changing the value of `limit`$[T]$.

Note that the management of both current entries and current sub-tuples is in the spirit of STR. Also, as in [Lecoutre, 2011], we introduce two sets of variables, called $S^{val}$ and $S^{sup}$. The set $S^{val}$ contains uninstantiated variables (and possibly, the last assigned variable) whose domains have been reduced since the previous invocation of the filtering algorithm on $c$. To set up $S^{val}$, we need to record the domain size of each modified variable $x$ right after the execution of STR-slice on $c$: this value is recorded in `lastSize`$[x]$. The set $S^{sup}$ contains uninstantiated variables (from the scope of constraint $c$) whose domains contain each at least one value for which a support must be found. We also use an array `gacValues`$[x]$ for each variable $x$. At any time, `gacValues`$[x]$ contains all values in $dom(x)$ for which a support has already been found: hence, values for a variable $x$ without any proved support are exactly those in $dom(x) \setminus$ `gacValues`$[x]$. Note that the sets $S^{val}$ and $S^{sup}$ are initially defined with respect to the full scope of $c$. However, for each sub-table we also shall use local sets $S^{lval}$ and $S^{lsup}$ of $S^{val}$ and $S^{sup}$ as explained later.

## Algorithm

Algorithm 30 is a filtering procedure, called STR-slice, that establishes GAC on a specified sliced table constraint $c$ belonging to a CN $N$. Lines 1–10, which are exactly the same as those in Algorithm 5 of [Lecoutre, 2011], allow us to initialize the sets $S^{val}$, $S^{sup}$ and `gacValues`. Recall that $S^{val}$ must contain the last assigned variable, denoted by lastPast$(P)$, if it belongs to the scope of $c$. Lines 11–22 iterate over all current entries of $c$. To test the validity of an entry, we check first the validity of the pattern $\mu$ (Algorithm 31), and then, only when the pattern is valid, we check the validity of the sub-table $T$ by scanning it (Algorithm 32). If an entry is no more valid, it is removed at Line 22. Otherwise, considering the values that are present in the pattern, we have to update `gacValues` as well as $S^{sup}$ when a first support for a variable is found. Lines 23–30, which are exactly the same as those in Algorithm 5 of [Lecoutre, 2011], manage the reduction of domains: unsupported values are removed at line 25 and if the domain of a variable $x$ becomes empty, an exception is thrown at line 27. Also, the set of variables $X_{evt}$ reduced by STR-slice is computed and returned so that these “events” can be propagated to other constraints.

Algorithm 32 is an important function, called scanSubtable, of STR-slice. Its role is to iterate over all current (sub)tuples of a given sub-table, in order to collect supported values and to remove invalid tuples. Note that when this function is called, we have the guarantee that the pattern associated with the sub-table is valid (note the “and then” short-circuit operator at Line 14 of Algorithm 30). The first part of the function, lines 1–8, allow us to build the local sets

**Algorithm 30:** STR-slice($c$: constraint):set of variables

```
// Initialization of sets $S^{val}$ and $S^{sup}$, as in STR2
$S^{val} \leftarrow \emptyset$
$S^{sup} \leftarrow \emptyset$
if lastPast($P$) $\in scp(c)$ then
    $S^{val} \leftarrow S^{val} \cup \{$lastPast($P$)$\}$
foreach variable $x \in scp(c) \mid x \notin$ past($P$) do
    gacValues[$x$] $\leftarrow \emptyset$
    $S^{sup} \leftarrow S^{sup} \cup \{x\}$
    if $|dom(x)| \neq$ lastSize[$c$][$x$] then
        $S^{val} \leftarrow S^{val} \cup \{x\}$
        lastSize[$c$][$x$] $\leftarrow |dom(x)|$
// Iteration over all entries of $c$
$i \leftarrow 1$
while $i \leq$ entriesLimit[$c$] do
    $(\mu, T) \leftarrow$ entries[$c$][$i$]            // $ith$ current entry of $c$
    if isValidPattern($\mu$) and then scanSubtable($T$) then
        foreach variable $x \in scp(\mu) \mid x \in S^{sup}$ do
            if $\mu[x] \notin$ gacValues[$x$] then
                gacValues[$x$] $\leftarrow$ gacValues[$x$] $\cup \{\mu[x]\}$
                if $|dom(x)| = |$gacValues[$x$]$|$ then
                    $S^{sup} \leftarrow S^{sup} \setminus \{x\}$
        $i \leftarrow i + 1$
    else
        removeEntry($c, i$)          // entriesLimit[$c$] decremented
// domains are now updated and $X_{evt}$ computed, as in STR2
$X_{evt} \leftarrow \emptyset$
foreach variable $x \in S^{sup}$ do
    $dom(x) \leftarrow gacValues[x]$
    if $dom(x) = \emptyset$ then
        throw INCONSISTENCY
    $X_{evt} \leftarrow X_{evt} \cup \{x\}$
    lastSize[$c$][$x$] $\leftarrow |dom(x)|$
return $X_{evt}$
```

$S^{lval}$ and $S^{lsup}$ from $S^{val}$ and $S^{sup}$. Such sets are obtained by intersecting $S^{val}$ with $scp(T)$ and $S^{sup}$ with $scp(T)$, respectively. Once the sets $S^{lval}$ and $S^{lsup}$

**Algorithm 31:** isValidPattern($\mu$: pattern): Boolean

```
1 foreach variable $x \in scp(\mu)$ do
2     if $\mu[x] \notin dom(x)$ then
3         return false
4 return true
```

are initialized, we benefit from optimized operations concerning validity checking and support seeking, as in STR2. The second part of the function, lines 9–21, consists in iterating over all current sub-tuples of $T$. This is a classical STR2-like traversal of a set of tuples. Finally, Line 22 returns true when there still exists at least one valid sub-tuple.

It is interesting to note the lazy synchronization performed between the global unique set $S^{sup}$ and the specific local sets $S^{lsup}$ (one such set per sub-table). When a variable $x$ is identified as "fully supported", it is immediately removed from $S^{sup}$ (see Line 19 of Algorithm 30 and Line 18 of Algorithm 32). Consequently, that means that the next sub-tables (entries) will benefit from such a reduction, but the information is only transmitted at initialization (lines 6–8 of Algorithm 32). On the other hand, once initialized, the global set $S^{val}$ is never modified during the execution of STR-slice.

**Backtracking issues:** In our implementation, entries and tuples can be restored by modifying the value of the limit pointers (`entriesLimit`[$c$] and `limit`[$T$] for each sub-table $T$ of $c$), recorded at each search depth. Restoration is then achieved in $O(1 + p)$ (for each constraint) where $p$ is the number of entries. However, by introducing a simple data structure, it is possible to only call the restoration procedure when necessary, limiting restoration complexity to $O(1)$ in certain cases: it suffices to register the limit pointers that need to be updated when backtracking, and this for each level. When the search algorithm backtracks, we also have to deal with the array `lastSize`. As mentioned in [Lecoutre, 2011], we can record the content of such an array at each depth of search, so that the original state of the array can be restored upon backtracking. As STR-slice is a direct extension of STR2, it enforces GAC.

### Illustration

Figures 3.22 and 3.23 illustrate the different steps for filtering a sliced table constraint, when STR-slice is called after an event. In Figure 3.22, considering that the new event is simply $x_3 \neq a$ (i.e., the removal of the value $a$ from $dom(x_3)$), STR-slice starts checking the validity of the current entries (from 1 to `entriesLimit`). So, for the first entry, the validity of the pattern $\mu$=$\{(x_1, a), (x_4, c), (x_5, a)\}$ is first checked. Since $\mu$ remains valid (our hypothesis

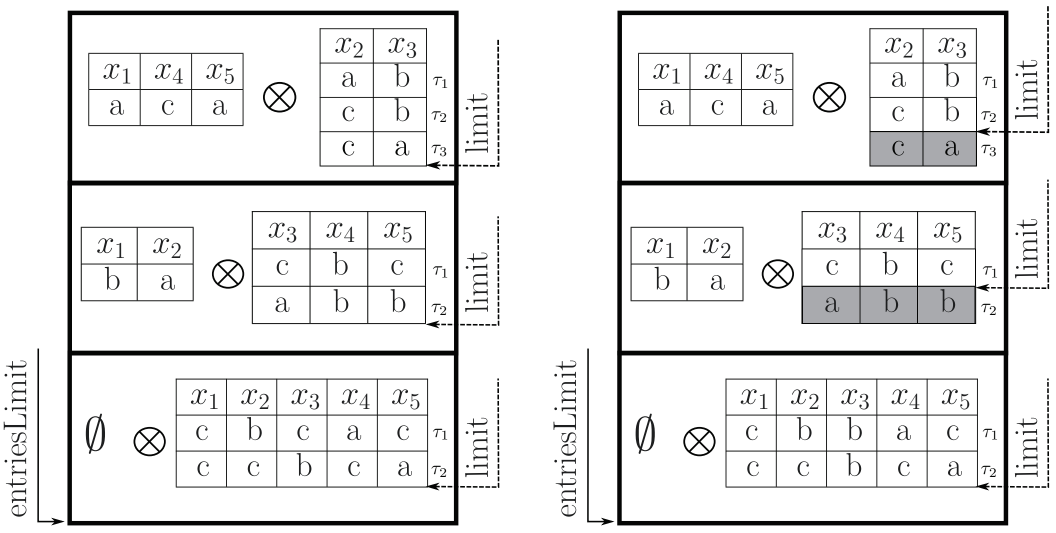
**Algorithm 32:** scanSubtable($T$: sub-table): Boolean

```
   // Initialization of local sets $S^{lval}$ and $S^{lsup}$ from $S^{val}$ and
       $S^{sup}$
 1 $S^{lval} \leftarrow \emptyset$
 2 foreach variable $x \in S^{val}$ do
 3     if $x \in scp(T)$ then
 4         $S^{lval} \leftarrow S^{lval} \cup \{x\}$
 5 $S^{lsup} \leftarrow \emptyset$
 6 foreach variable $x \in S^{sup}$ do
 7     if $x \in scp(T)$ then
 8         $S^{lsup} \leftarrow S^{lsup} \cup \{x\}$
   // Iteration over all (sub)tuples of $T$
 9 $i \leftarrow 1$
10 while $i \leq$ limit$[T]$ do
11     $\tau \leftarrow T[i]$                 // $ith$ current sub-tuple of $T$
12     if isValidSubtuple($S^{lval}, \tau$) then
13         foreach variable $x \in S^{lsup}$ do
14             if $\tau[x] \notin$ gacValues$[x]$ then
15                 gacValues$[x] \leftarrow$ gacValues$[x] \cup \{\tau[x]\}$
16                 if $|dom(x)| = |$gacValues$[x]|$ then
17                     $S^{lsup} \leftarrow S^{lsup} \setminus \{x\}$
18                     $S^{sup} \leftarrow S^{sup} \setminus \{x\}$
19         $i \leftarrow i + 1$
20     else
21         removeSubtuple($T, i$))          // limit$[T]$ decremented
22 return limit$[T] > 0$
```

**Algorithm 33:** isValidSubtuple($S^{lval}$: variables, $\tau$: tuple): Boolean

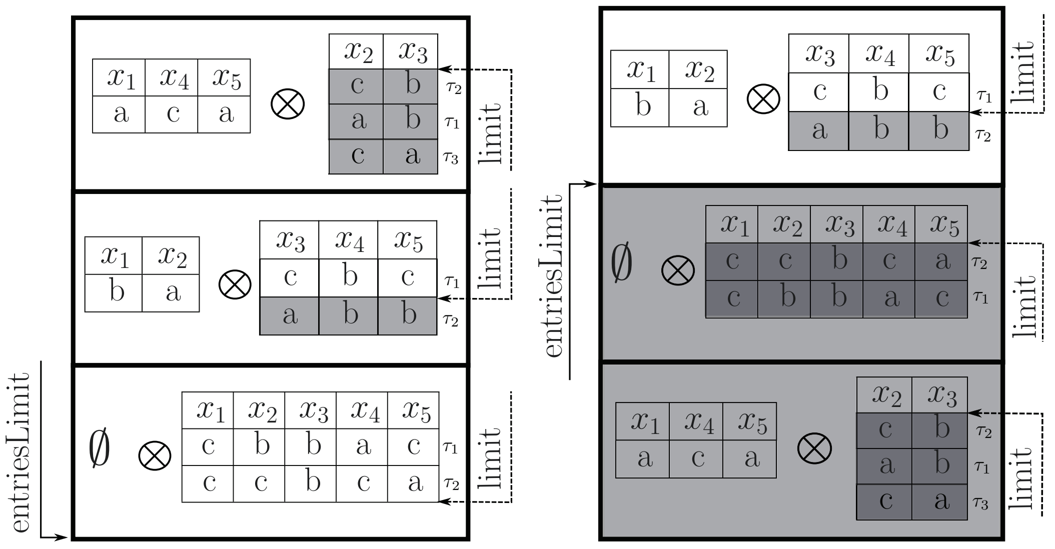
```
1 foreach variable $x \in S^{lval}$ do
2     if $\tau[x] \notin dom(x)$ then
3         return false
4 return true
```

is that the event was only $x_3 \neq a$), the sub-table of the first entry is scanned. Here, only the sub-tuple $\{(x_2, c), (x_3, a)\}$ is found invalid, which modifies the value of `limit` for the sub-table of this first entry. After the call to STR-slice, the constraint is as in Figure 3.22(b).



(a) Before the call. (b) After the call.

Figure 3.22: STR-slice called on a slice table constraint after the event $x_3 \neq a$.



(a) After the scan of the first entry. (b) After the call.

Figure 3.23: From Figure 3.22(b), STR-slice called after the event $x_3 \neq b$.

In Figure 3.23, considering now that the new event is $x_3 \neq b$, we start again with the first current entry. Figuring out that the pattern is still valid, we check the validity of the associated sub-tuples. Since the sub-tuple $\{x_2 = a, x_3 = b\}$ is no more valid, it is swapped with $\{x_2 = c, x_3 = b\}$. This latter sub-tuple is then also found invalid, which sets the value of `limit` to 0. This is illustrated in

| Instance | #ins | STR1 | STR2 | STR3 | STR-slice |
|---|---|---|---|---|---|
| *a7-v24-d5-ps05* | 11 | 298.05 | 147.73 | 189.14 | **115.30** (66% – 5.74) |
| *bdd* | 70 | 44.53 | **13.44** | 99.21 | 20.35 (86% – 0.59) |
| *crossword-ogd* | 43 | 90.05 | 39.35 | **25.69** | 29.59 (75.51% – 0.36) |
| *crossword-uk* | 43 | 95.20 | 45.88 | **44.33** | 47.21 (88.69% – 0.18) |
| *renault* | 46 | 19.66 | 14.39 | **13.37** | 17.20 (47.15% – 0.67) |

Table 3.5: Mean CPU time (in seconds) to solve instances from different series with MAC. Mean compression ratio and CPU time are given for STR-slice between parentheses.

Figure 3.23(a). As the sub-table of the first entry is empty, the entry is removed by swapping its position with that of last current entry. After the call to STR-slice, the constraint is as in Figure 3.23(b) (note that a second swap of constraint entries has been performed).

## 3.3.4 Experimental results

In order to show the practical interest of our approach to represent and filter sliced table constraints, we have conducted an experimentation in the same condition as the first approach (see Section 3.2.4). Since STR1, STR2 and STR3 belong to the state-of-the-art GAC algorithms for table constraints, we compare the respective behaviors of STR variants and STR-slice on the different series of instances used in Section 3.2.4

Table 3.5 shows mean results (CPU time in seconds) per series. For each series, the number of tested instances is given by #ins; it corresponds to the number of instances solved by all three variants within $1,200$ seconds. Note that the mean compression ratios and CPU times (in seconds) are also given for STR-slice between parentheses. We define the compression ratio as the size of the sliced tables over the size of the initial tables, where the size of a (sliced) table is the number of values over all patterns and (sub-)tables. The used minSupport is equal to 10% of the number of tuples in the table. The results in Table 3.5 show that STR-slice is competitive with both STR2 and STR3. Surprisingly, although the compression ratio obtained for the instances of the series *renault* is rather encouraging, the CPU time obtained for STR-slice is disappointing. We suspect that the presence of many constraints with small tables in the *renault* instances is penalizing for STR-slice because, in that case, the overhead of managing constraint entries is not counterbalanced by the small absolute spatial reduction. Table 3.6 presents the results obtained on some instances. A general observation

| Instance | MDD | STR1 | STR2 | STR3 | STR-slice |
|---|---|---|---|---|---|
| *a7-v24-d5-ps0.5-psh0.7-9* | **17.819** | 879 | 334 | 367 | 200 (68.75% – 5.406) |
| *a7-v24-d5-ps0.5-psh0.9-2* | **5.536** | 256 | 145 | 143 | 93 (62.5% – 5.957) |
| *a7-v24-d5-ps0.5-psh0.9-6* | **11.397** | 353 | 195 | 324 | 174 (62.5% – 5.82) |
| *bdd-21-2713-15-79-9* | 77.171 | 80.2 | **23.3** | 60.0 | 35.4 (88.230% – 0.283) |
| *bdd-21-2713-15-79-11* | 55.752 | 78.5 | **23.5** | 48.5 | 31.7 (88.05% – 0.28) |
| *bdd-21-133-18-78-11* | 47.157 | 38.4 | **11,0** | 250 | 19.0 (86.12% – 0.866) |
| *crossword-m1-ogd-23-04* | 104.485 | 82.7 | **78.2** | 103 | 80.1 (73.90% – 1.039) |
| *crossword-m1c-lex-vg5-7-* | 86.475 | 43.5 | **31.4** | 36.6 | 34.5 (96.17% – 0.097) |
| *crossword-m1c-ogd-vg10-13* | TO | TO | 897 | 765 | **750** (74.84% – 0.801) |
| *nonogram-gp-108* | 55.732 | 290 | **78.7** | 118 | 83.6 (74.93% – 2.475) |
| *nonogram-gp-116* | **16.658** | 102 | 21.7 | 21.9 | 22.8 (79.99% – 1.455) |
| *nonogram-gp-137* | **8.359** | 166 | 45.4 | 56.8 | 29.4 (82.96% – 3.5) |
| *rand-6-10-10-60-950-0* | 67.821 | 75.1 | 45.4 | **34.4** | 42.5 (77.77% – 4.950) |
| *rand-7-9-9-30-980-0* | 47.285 | 26.8 | **18.5** | 46.3 | 22.9 (75% – 7.489) |
| *rand-8-20-5-18-800-8* | 83.815 | 86.6 | **45.8** | 574 | 64.2 (72.36% – 4.787) |

Table 3.6: CPU time (in seconds) on some selected instances solved by MAC. Compression ratio and CPU time are given for STR-slice between parentheses.

from this experimentation is that STR-slice is a competitor to STR2 and STR3, but a not a competitor that takes a real advantage compared to MDD.

### Comparison with related approaches

In this section we compare both contributions with the approaches presented in Section 2.1. Table 3.7 compares, in a first time, our two contributions with ShortSTR2 algorithm (see Section 2.1.2). This table presents both resolution time (CPU time) and compression ratio as defined before (compressed table / original table). It is important to note that the resolution time for ShortSTR2 are given considering that the algorithm run on Minion solver [Gent et al., 2006]. We gathered the common benchmarks. Obviously from the given results, Short-STR2 outperforms our contribution only on bddSmall problem but the other compression ratios are too small (100% means that the algorithm does not compress the table). For the running time, we cannot make any comparison since two different solvers are used to solve these problems.

| Instance | STR$^c$ | | STR-slice | | ShortSTR2 | |
|---|---|---|---|---|---|---|
| rand-8-20 | 111.949 | 62.960% | 64.2 | 72.36% | 3,207 | 99% |
| bddSmall | 64.314 | 83.076% | 28.7 | 87.46% | 521.10 | 52.63% |
| crosswordVg | 49.393 | 20.134% | 34.5 | 96.17% | 396.25 | 100% |

Table 3.7: STR$^c$ and STR-slice compared to ShortSTR2.

Table 3.8 makes also a comparison considering the common benchmarks between the approaches: STR$^c$, STR-slice and STR2-C (Section 2.1.2). All these problems run oven AbsCon solver. STR2-C outperforms our contribution especially on the random problems (*MDD0.7* and *MDD0.9*) considering both compression ratio and resolution time.

| Instance | STR$^c$ | | STR-slice | | STR2-C | |
|---|---|---|---|---|---|---|
| rand-8-20 | 111.949 | 62.960% | 64.2 | 72.36% | 26.0 | 60.2% |
| MDD 0.7 | 780.318 | 43.775% | 200 | 68.75% | 171.3 | 5% |
| MDD 0.9 | 352.281 | 43.776% | 133.5 | 62.5% | 28.5 | 1.5% |

Table 3.8: STR$^c$ and STR-slice compared to STR2-C.

Due to the lack of experiments for data-mining based approaches we could not make any comparison.

## 3.4 Conclusions and discussion

Through the first work (STR$^c$), we combine both techniques of simple tabular reduction and compression. Identifying recurring patterns in tuples of different tables, reduces memory space and also the CPU time by avoiding redundant validity tests. From our tests, the STR$^c$ algorithm we propose seems to be competitive with STR1 (in the search step) but supplanted by STR2 and sometimes STR3.

For the second approach, we combined a new compression technique using the concept of sliced table constraints and an optimized adaptation of the tabular reduction (as in STR2) for the filtering process. Our experimentation shows that STR-slice is a competitor to the state-of-the-art STR2 and STR3 algorithms.

For the first compression technique just one scan of the table is required to build the trie of frequent patterns which is not the case for the second compression technique requiring two scans to build the trie. Even if we use the same data structure (trie) during the compression process for both approaches, but each trie have a different representation. In the first one we gather all paths starting with the same value. However, in the second one we gather all paths sharing the same literal.

Both of the two compression methods enable us to obtain high ratios of total space reduction. Nevertheless, the second compression technique supplants the drawback of the first one. In fact, the issue of compressing extensional constraints has two faces: the first one is to achieve a high reduction ratio of the required memory space which is the easiest step since there are several proposed compression approaches aiming to that especially in the data mining field. However, the most difficult step is how to represent the compressed constraint in a suitable way avoiding slowing down the filtering process, but also, if it is possible, to speed it up taking advantages from the several dynamic compression techniques like STR.

Since our first compression technique defines a pattern as a sequence of values independently from their positions which implies the independence from the variables, we were not able to use the STR2 and STR3 techniques which are based on variables domains changes. That is why the first approach could not compete with to the state-of-the-art STR2 and STR3 algorithms. This is not the case of the second approach ("sliced" table constraints) that is based on an optimized adaptation of the tabular reduction for the filtering process. The experimental results show that STR-*slice* algorithm is competitive with STR2 and STR3 even if it is sometimes sanctioned due to the compression time. Compared to MDD, the two approaches competes with MDD only in the case where STR variants supplant it.

# Chapter 4

# Using Singleton Arc Consistency in Parallel to Improve Search

## Contents

## Introduction

In Section 2.2, we introduced the main categories of the state-of-the-art of parallel approaches that have been proposed in order to solve CSP instances within a parallel architecture. Some approaches distribute the search tree over workers while others use a portfolio of workers which compete in order to solve an instance of a problem.

In this chapter, we explore another way of using a parallel architecture in which a main solver is helped by side workers that partially establish consistencies, which are otherwise two heavy to be maintained by the main solver.

This chapter is organized as follows:

- In Section 4.1, we describe our approach and the used parallel architecture;
- In Section 4.2, we introduce the data structures and algorithms used from the side of the master;
- In Section 4.3, we explore the workers side and how they interact with each others and with the master;
- In Section 4.4, we explain how our architecture is supposed to enhance the search process;
- Finally, we conclude with experimental results in Section 4.5.

## 4.1 Architecture

Our approach is based on a master/workers architecture where the master is a sequential CSP solver and the different workers help their master during the search process. This main solver transmits its current instantiation to its side workers, which will try to infer relevant information by exploiting different levels of consistencies. As soon as new facts are discovered, they are transmitted to the main solver that takes them into account as soon as possible. Our goal is to have a synchronization between the main solver and the side workers as lightweight as possible. In our current work, both the main solver and the side workers (threads running on different cores) run on the same host.

Different consistencies were introduced in Chapter 1 that could be ranked with respect to their strength. Singleton Arc Consistency (SAC) [Debruyne and Bessiere, 1997] is a strong consistency that is very expensive to enforce during search [Lecoutre and Prosser, 2006]. Performing each SAC test in a sequential solver is too heavy. But in a parallel solver available cores can perform these SAC tests for free (at least from a wall-clock point of view). The literals that are discovered SAC-inconsistent are transmitted to the master in order to avoid them in its future assignments.

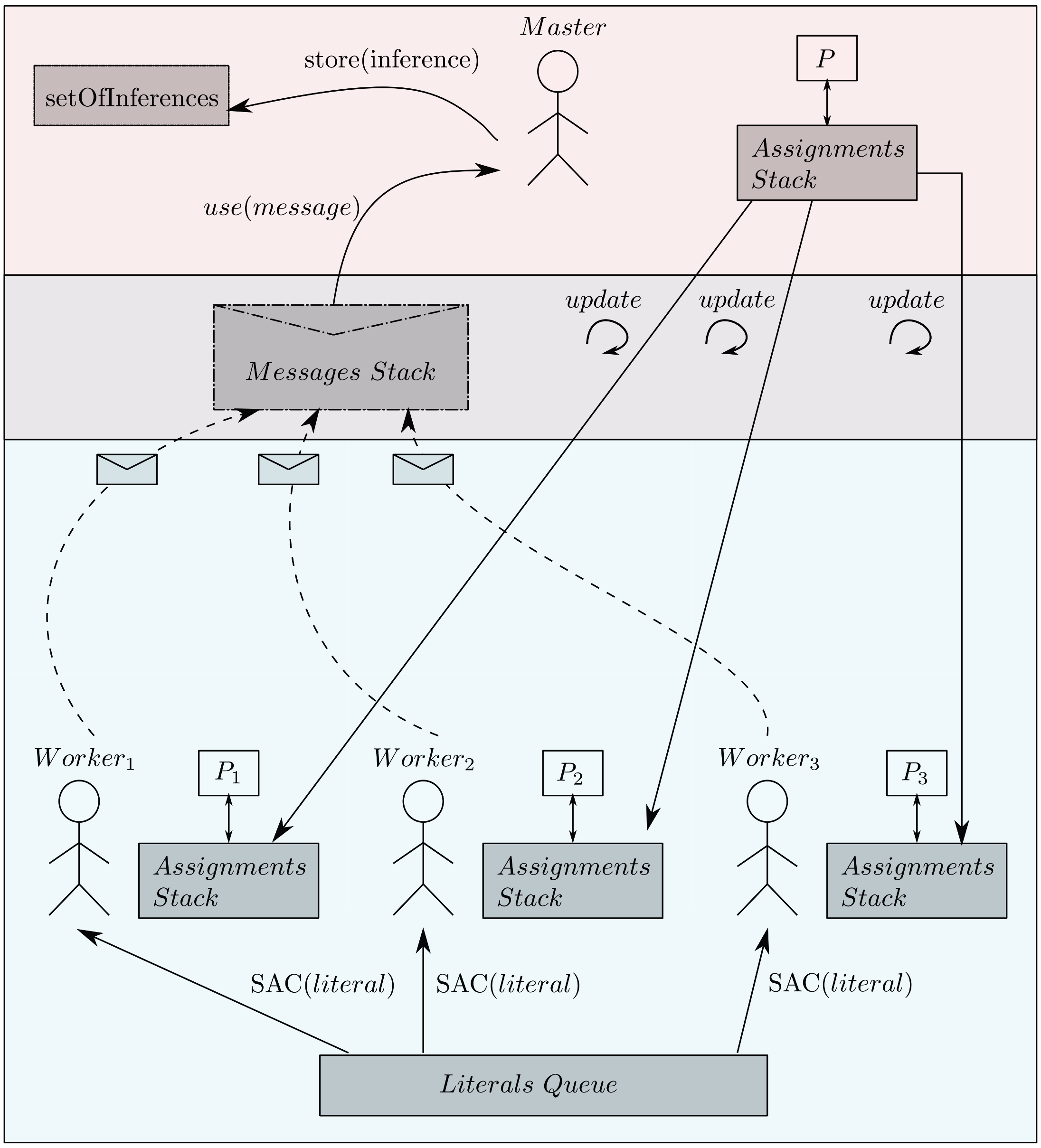


Figure 4.1: Overview of the architecture.

Figure 4.1 describes the architecture of our system. In fact, each master and worker has its own copy of the problem. The master solves the problem whereas the workers enforce SAC on some literals of the problem. An *Assignments Stack* is associated to each master and worker. The master stores all positive and negative decisions made in its own *Assignments Stack* while the workers only copy them to their stack, each time there are new decisions. This stack describes the state of the problem of each entity. All workers get literals to test for consistency from a *Literals Queue* which is a shared data structure between all of them. If enforcing SAC on these literals produces new facts, new messages are put into the *Messages Queue* which is a shared data structure between the master and the workers. The master in his turn extracts messages from the *Messages Queue* in order to exploit them in its solving process by avoiding failures.

We detail our approach from both sides: the master's side in Section 4.2 and the workers one in Section 4.3.

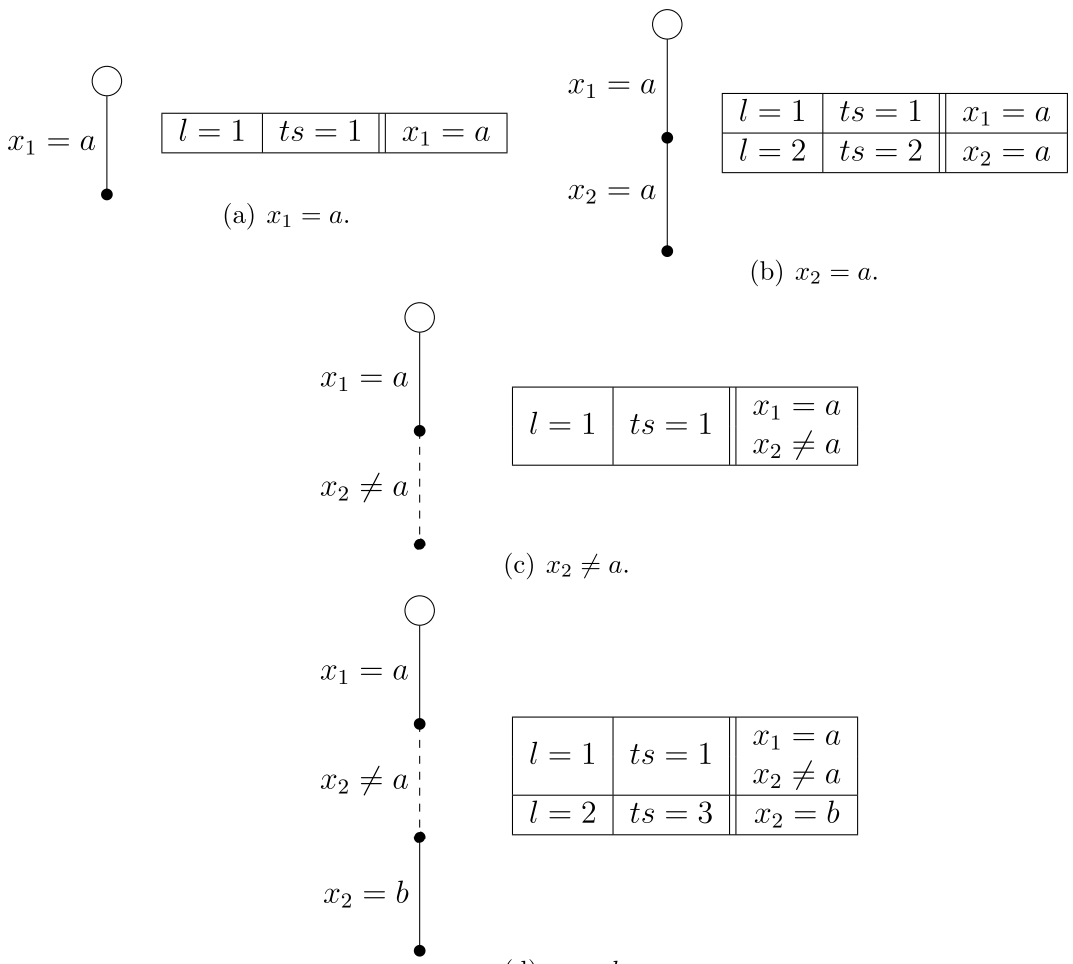
## 4.2 Master's side

In our approach, the master, runs a classical CSP solver using only two additional data structures: the *Messages Queue* which is shared with the workers and a *Set of Inferences* which is its own data structure. The *Assignments Stack* is accessible by the workers in order to get the problem state. All these used data structures are detailed in this Section from the master's side.

### 4.2.1 The Assignments Stack

In our context, a search tree is composed of different levels which represents the number of variables composing the problem. In the next algorithms, we consider that each level corresponds to an assignment of a variable. The *Assignments Stack* stores all decisions made by the solver. They might be positive (assignment) or negative (refutation) decisions. Each positive decision defines a level to which corresponds a time-stamp indication at which time this assignment was taken. This time-stamp is given by a global counter (on 64 bits to avoid any overflow) which is initialized to 0 and incremented each time a decision is taken. This information is used to identify the modifications of the *Assignments Stack* and is further described from the workers side. In fact, a time-stamp is only associated with positive decisions (assignments) which is a feature of AbsCon solver [Lecoutre and Tabary, 2007]. Positive decisions are stored in the stack in a consecutive order. When a backtrack occurs, the value causing the backtrack is removed from the domain of its variable and this refutation is added to the previous level.

**Example 32** *Figure 4.2 gives an illustration of the Assignments Stack management when different decisions are made. Positive decisions are stored in the stack in a consecutive order. When a backtrack occurs, e.g. $x_2 \neq a$ in Figure 4.2(c), the value causing the backtrack is removed from the domain of its variable and this refutation is added to the previous level. The time-stamp of the last positive decision before backtrack does not change.*



$x_1 = a$
$l = 1$ | $ts = 1$ | $x_1 = a$
$x_1 = a$
$x_2 = a$
$l = 1$ | $ts = 1$ | $x_1 = a$
$l = 2$ | $ts = 2$ | $x_2 = a$
$x_1 = a$
$x_2 \neq a$
$l = 1$ | $ts = 1$ | $x_1 = a$, $x_2 \neq a$
$x_1 = a$
$x_2 \neq a$
$x_2 = b$
$l = 1$ | $ts = 1$ | $x_1 = a$, $x_2 \neq a$
$l = 2$ | $ts = 3$ | $x_2 = b$

(a) $x_1 = a$.

(b) $x_2 = a$.

(c) $x_2 \neq a$.

(d) $x_2 = b$.

Figure 4.2: Master's side: managing the *Assignments Stack*.

### 4.2.2 The Messages Queue

Before a decision is taken, the master checks if some information has been inferred by the other workers. This is done by checking the content of the *Messages Queue*, which is read by the master and written by the workers. When a message is extracted from this queue, the master first checks if the inference is still relevant in the current state of the solver (the message could be outdated because the master backtracked in between). If the message is not relevant, it is ignored. Otherwise, the inference is taken into account.

In fact, a message contains two main information:

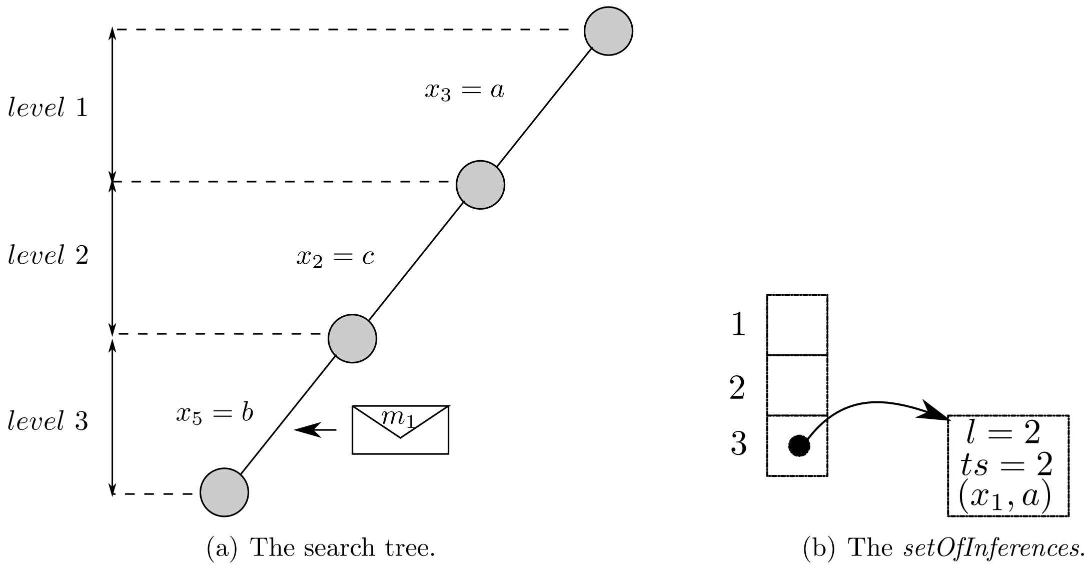
- *literal*: the pair (variable,value) to avoid in future assignments;
- (*level*, *timeStamp*): the pair level and its time-stamp in which the property

"the literal is SAC-inconsistent" is still relevant.

Algorithm 34 describes how the master judges the validity of the messages sent by workers to use them in the inference process. According to the master, a message is still relevant **iff** the level of the test is **higher** (in the search tree) than the master solver level *and* the time-stamps are **coherent**.

**Algorithm 34:** isValidMessage(*msg*: Message): Boolean

```
1 if $msg.level \leq currentLevel$ and
  $msg.timeStamp = timeStamp[msg.level]$ then
2     return true        // the message msg is still relevant
3 return false
```



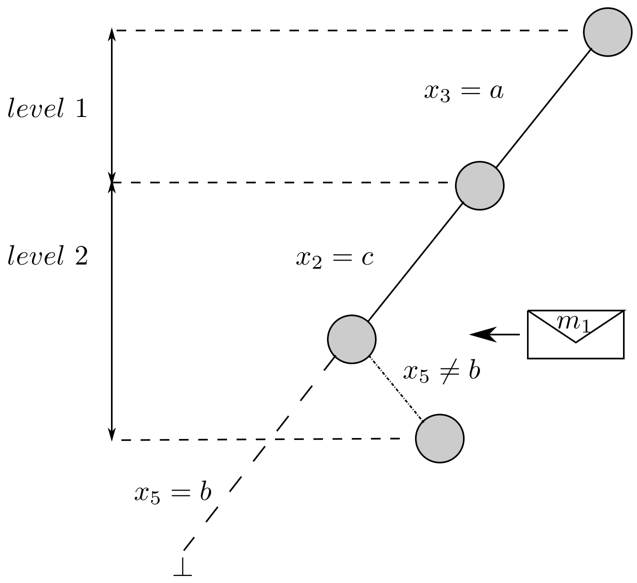
(a) The search tree. (b) The *setOfInferences*.

| level | time-stamp | decision |
|---|---|---|
| 1 | 1 | $x_3 = a$ |
| 2 | 2 | $x_2 = c$ |
| 3 | 3 | $x_5 = b$<br>$x_1 \neq a$ |

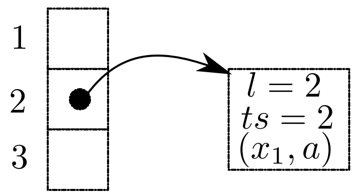
(c) The *Assignment stack*.

Figure 4.3: A master's search tree at level 3.

Once the master figures out that the message is still relevant, it stores it in its own data structure, called *setOfInferences*. In fact, *setOfInferences* is a table where *setOfInferences*[i] defines the messages extracted and used at level $i$. Thanks to this data-structure, the master can re-use these inferences in higher



(a) The search tree.



(b) The *setOfInferences*.

| level | time-stamp | decision |
|---|---|---|
| 1 | 1 | $x_3 = a$ |
| 2 | 2 | $x_2 = c$<br>$x_5 \neq b$<br>$x_1 \neq a$ |

(c) The *Assignment stack*.

Figure 4.4: The master backtracking to level 2.

levels, when a backtrack occurs. Figure 4.3 describes the use of *setOfInferences*. At level 3, the master receives a new message $m_1$=\{*level*=2, *timeStamp*=2, *literal*=$(x_1, a)$\}. This message is relevant with respect to the master current state (*level* $\leq$ master current level and *timeStamp*= timeStamp[2]). The master can, thus, benefit from this message by deleting this value as long as it has not backtracked. Value $a$ is then removed from $dom(x_1)$ and the message is stored in *setOfInferences*. It happens that a backtrack occurs to the level 2 which implies restoration of the previous state of the problem at level 2: the inference application of the message $m_1$ is undone when backtracking. Since $m_1$ remains relevant after backtracking, the master re-applies it.

Algorithm 35 describes how the master manages the messages received from the different workers and the inferences already stored in *setOfInferences*. In fact, the master checks *setOfInferences* only when a backtrack occurs (Lines 1 - 7). The parameter *hasBacktracked* indicates if the master has backtracked or not since the last decision made. If it is the case, the master re-applies all the inferences stored starting from its current level after backtracking. If the master

**Algorithm 35:** applyMessages(*hasBacktracked*: Boolean)

```
1  if hasBacktracked then
2      foreach $i \in$ [currentLevel+1,levelBeforeBacktrack] do
3          foreach msg : setOfInferences[i] do
4              if isValidMessage(msg) then
5                  remove msg.literal.val from dom(lmsg.iteral.var)
6                  add msg to setOfInferences[currentLevel]
7          clear setOfInferences[i]
                       // removing the inferences stored at level i
8  foreach msg : msgQueue do
9      if isValidMessage(msg) then
10         remove msg.literal.val from dom(msg.literal.var)
11         add msg to setOfInferences[currentLevel]
12     remove msg
```

hasn't backtracked, it checks the new messages in *Messages Queue*. It, first, verifies if the message is relevant. If it is the case, the literal is removed and the inference is added to *setOfInferences* at the current level of search (Lines 5 and 6). If the message is no more relevant, it is removed from the Messages Queue (Line 12).

## 4.3 Workers' side

The **workers** use three main data structures:

### 4.3.1 The Assignments Stack

The workers obtain initially a copy $\mathscr{P}'$ of the problem instance $\mathscr{P}$ that is handled by the master (variables, domains, constraints). In order to reach the same search state as the master, the workers can either copy the new state of the problem or get only the decision made so far and re-apply them. Since copying the new state of the problem enables the workers to get only the new domains state and not the constraints state (involving the handled supports for each constraint), we choose in our approach to make an incremental copy. In fact, in a loop, they copy incrementally the *Assignments Stack* of the master (i.e. the list of decisions taken by the master), reproduce the filtering made by the master after these decisions in order to reach the same state as the master and then run their own (partial) consistency. The incremental copy of the *Assignments Stack* consists in, first, identifying the part of the *Assignments Stack* which is identical in the master and in the worker's copy and then copying every decision taken by the master after this common part. It is performed in the following way. The worker first obtains the current stack pointer of the master and then identifies the last decision which has the same time-stamp in the master and in the worker. Then, the worker copies each decision of the master after this last common decision. Since the master may have backtracked in between, the worker then checks that the last decision that it copied still has the same time-stamp as in the master. If this is not the case, the master backtracked during the copy and the worker restarts its copy of the *Assignments Stack*. In practice, this does not happen too often (less than 1% of the total copies). Therefore, our approach guarantees that inconsistent copies are never used because when such a copy is obtained, the worker simply makes another one.

Figures 4.5 and 4.6 describe two cases of the copying process. In Figure 4.5, the master continued its solving process and, thus, there are new decisions to be copied in the worker's *Assignments Stack*. The last common positive decision between the master and the worker is $x_1 = a$. All decisions made after this one are added to the worker's stack.

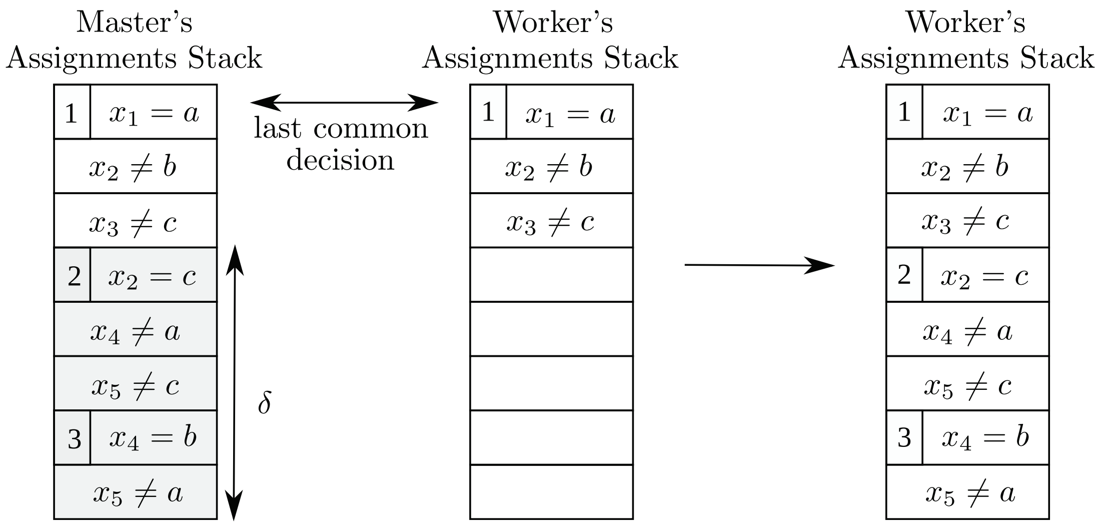


Figure 4.5: Copying process (case 1: the same branch).

In the case of Figure 4.6 the master has backtracked and started new other decisions. The worker has to remove all decisions until the common one. In fact, a backtrack on the variable $x_2$ has occurred and, thus, a negative decision is added to the first positive decision $x_1 = a$ (the last positive decision). The time-stamp of the decision $x_1 = a$ does not change when adding the refutation ($x_2 \neq c$). However, the worker has to update its stack in order to cope with these new facts. First of all, the worker removes the old decisions that are no more relevant. Then, its checks if there are new negative decisions of the last common positive decision ($x_1 = a$ in our case). This is done by comparing the number of negative decisions of the last common positive decision. Finally, the worker adds incrementally the decisions of the new levels (including refutations).

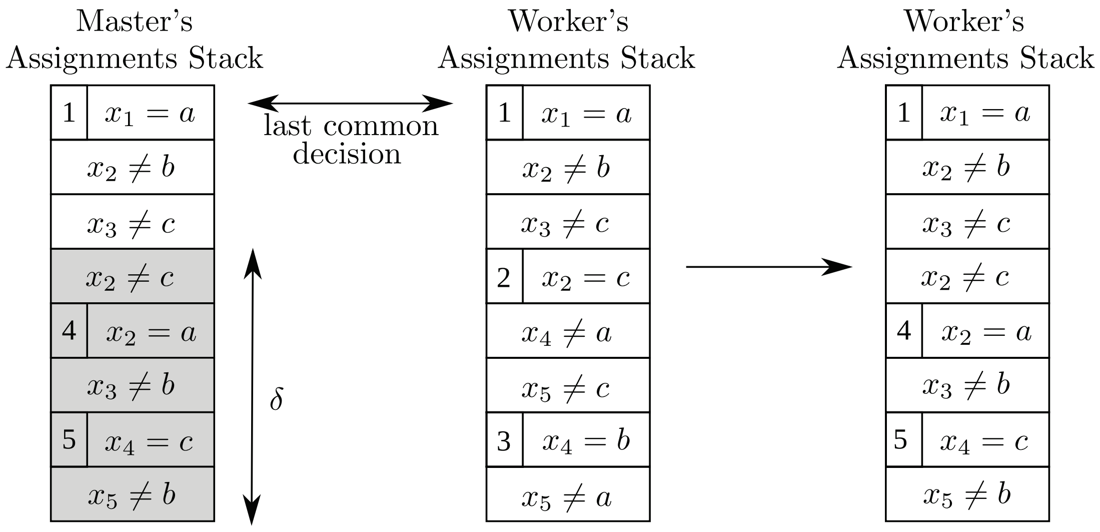


Figure 4.6: Copying process (case 2: backtracking).

### 4.3.2 The Messages Queue

Once a worker has a stable copy of the master's state, it picks a new literal $(x, a)$ from a global queue of literals (described in Section 4.3.3) and checks if $(x, a)$ is SAC-consistent, i.e., if no domain wipe-out (global inconsistency) is detected when taking the decision $x = a$ and running constraint propagation (we assume that GAC is achieved). If it is not, the worker places in the *Messages Queue* the data $x \neq a$ together with its current decision level and a time-stamp where this information could be inferred (as detailed in Section 4.2.2). The master will extract this information when it is ready to use it. After testing SAC on a literal $(x, a)$, it is put back to the literals queue, while updating its priority (as described in Section 4.3.3), in order to test its consistency later.

Algorithm 36 describes the SAC enforcing on the set of literals of the queue. We just focus, in this section, on the message producing process. When a literal (*variable*,*value*) is SAC-inconsistent (the SAC test returns *false*), it means that removing this literal enables the master to avoid some useless search sub-tree and, thus, speeds up the solving process. The worker transmits this information to its master (Line 14). A message is then composed indicating the literal to avoid, on which level the SAC test is still relevant and the time-stamps to identify the last positive decision related to this test.

**Algorithm 36:** enforceConsistencyOnLiterals($P_i$: Constraint Network of slave $i$ ,*LiteralsQueue*: Queue of literals)

```
1  $valueFound \leftarrow false$
2  while $\neg valueFound$ do
3      $literal \leftarrow getLiteral(literalsQueue)$
4      if $\neg assigned(literal.var)$ and $|dom(literal.var)| > 1$ then
5          $valueFound \leftarrow true$
6      else
7          $literal.countdown$ - -
8          insert $literal$ to $literalsQueue$
   // while the problem is not solved, there is always a value to
   test
9  assign $literal.val$ to $literal.var$
10 $consistent \leftarrow GAC(P, literal.var)$
11 $backtrack()$
   // going back to the previous level in order to test more
   pairs
12 if $\neg consistent$ then
13     $literal.priority{+}{+}$
14     $putMessage(currentLevel, timeStamp[level], literal)$
       // Putting the inferred result in the queue of messages
15 $literal.countdown$ - -
16 insert $literal$ to $literalsQueue$
```

### 4.3.3 The Literals Queue

In order to perform the SAC tests, the literals of the problem are put together in a *Literals Queue*. The different workers cooperate to examine each possible literal as often as possible. Since some literals are more likely to become SAC-inconsistent, we use priorities in order to test these literals more often than the other ones. Therefore, each literal in the *Literals Queue* is assigned a priority, which is incremented when the literal is identified as SAC-inconsistent (Algorithm 36 Line 13).

To manage this priority, we use the approach of the Completely Fair Scheduling (CFS) algorithm [Li et al., 2009] used in Linux. The CFS scheduler ensures executing the process that has used the least amount of time at the first place and, thus, the processes are ordered according to the execution time spent. The CFS uses a red–black tree data structure that is a balanced binary search tree whose key is based on the value of the running time. According to red–black tree property, tasks that have been given less processing time are on the left side, whereas tasks that have been given more processing time are on the right side. In Figure 4.7, node $N_7$ has the smallest value of running time and $N_{10}$ has the largest value. CFS scheduler accords the highest priority to the node that has the smallest running time value, and thus, left tasks are runned the first. After picking a process, the CFS scheduler removes it from the tree, executes it and updates its global execution time counter and then returns it back to the tree taking into account its new execution time counter. One advantage of this algorithm is that each value is tested as many times as its priority.

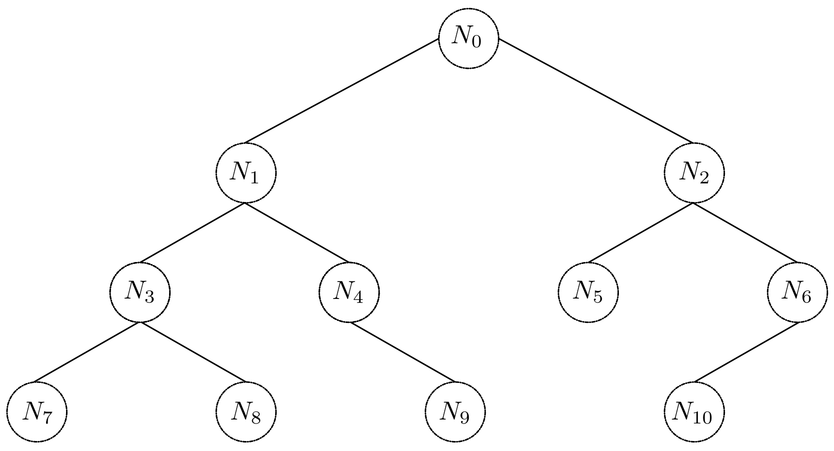


Figure 4.7: A red-black tree.

In our approach a literal has two main properties:

- *countdown* counter: it indicates the number of times a literal could be tested. Each time we test a literal, we decrease its *countdown* counter (Line 15 in the Algorithm 36) to ensure a balanced test between all the literals of the queue. When all the literals are tested, the *countdown* is automatically reset for all the literals. Each time we reset the *Literals Queue*, the value *countdown* of each literal takes the value of its *priority*;

- *priority* counter: it indicates how interesting a literal is. If a SAC test proves the inconsistency of a literal, its priority counter is increased (Line 13 in the Algorithm 36) because such a literal is important to be tested in other branchs of the search tree and may infer important information to be exploited by the master. The *priority* counter is initialized to 1 when the workers start their jobs: the literals have the same importance at the root of the search tree.

Since the *Literals Queue* is managed according to the CFS algorithm, in order to perform the SAC tests we first get a literal and remove it from the queue. Then, we check if the variable is already assigned or is a singleton variable. If it is the case, it is useless to enforce SAC on this literal and we put back the literal in the end of the queue (by updating its priority). Otherwise, we perform the test.

## 4.4 Enhancing the search process

Algorithm 37 describes the search algorithm used by the master. In fact, we use the classic MAC search algorithm. Added to that, we include extracting messages step (Line 4): a call to Algorithm 35 is made in order to benefit from the facts discovered by workers. In this way, the SAC-inconsistent values are removed from their respective domains which enables to avoid them (Line 6).

**Algorithm 37:** solveProblem($P$: Constraint network)

```
1  $finished \leftarrow false$
2  $hasBacktracked \leftarrow false$
3  while $\neg finished$ do
                                            // infer all not used messages
4      $applyMessages(hasBacktracked)$
                              // select new pair (variable,value) to assign
5      $var \leftarrow getNextVariable(variableOrderingHeuristic)$
6      $val \leftarrow getNextValue(valueOrderingHeuristic, var)$
7      assign $var$ to $val$
8      $levelBeforeBacktrack \leftarrow |past(P)|$
9      $consistent \leftarrow checkConsistencyAfterAssignment(P)$
10     if $consistent$ and $|past(P)| = n$ then
11         display $solution$
12         $finished \leftarrow true$
13     else
14         while $\neg consistent$ do
15             $backtrack()$
16             $hasBacktracked \leftarrow true$
17             remove $val$ from $dom(var)$
18             $consistent \leftarrow checkConsistencyAfterRefutation(P)$
```

Using the introduced data structures, Figure 4.8 illustrates our architecture. On this example, the master has taken three decisions so far. The first decision $x = a$ has time-stamp 1. Then, the master tried $y = c$ at time-stamp 2, but immediately obtained an inconsistency. Therefore, it removes this decision and adds the negative decision $y \neq c$ to the previous positive decision. The latest decision is $y = a$ at time-stamp 3. $Worker_1$ has copied the master's stack at

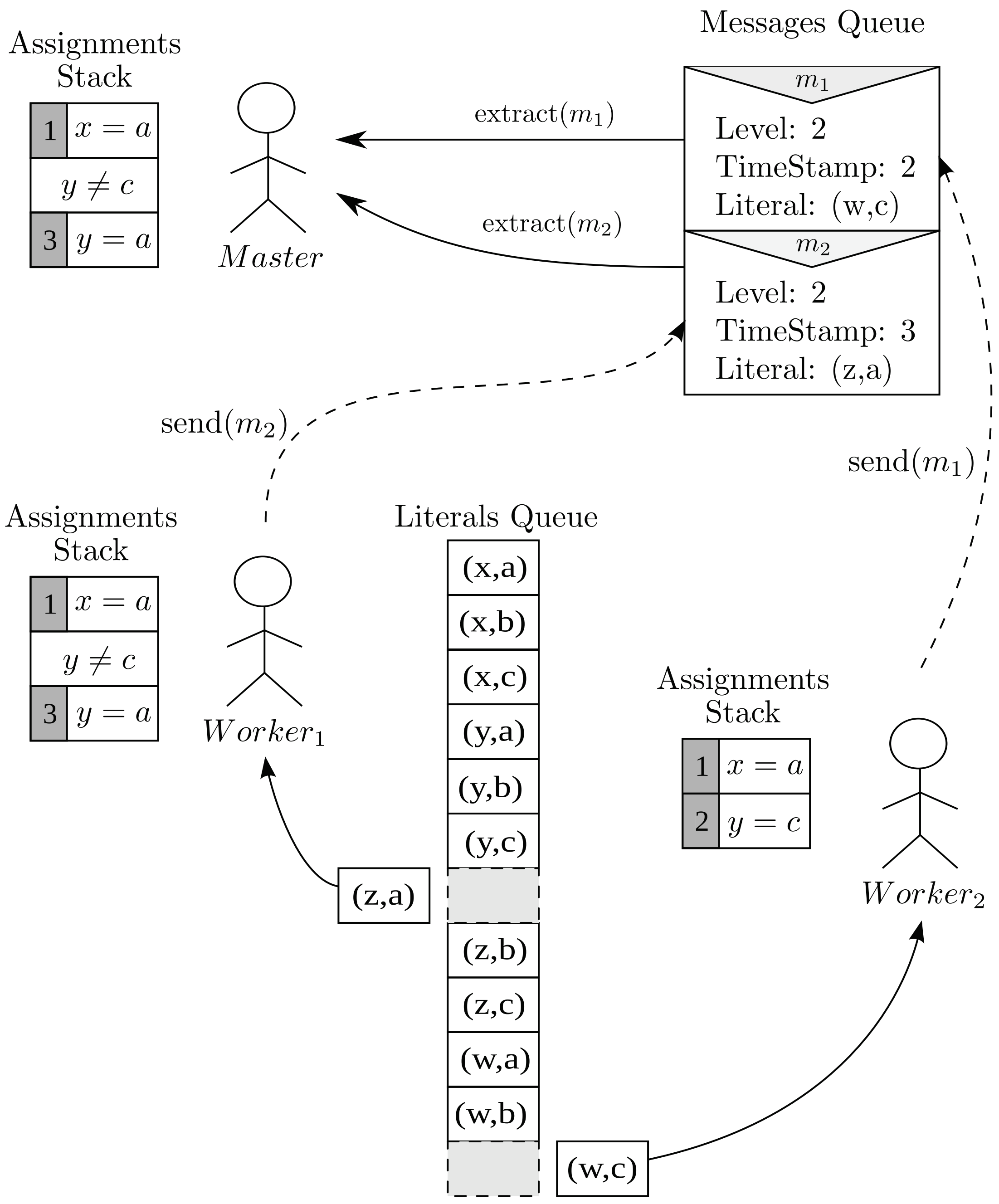


Figure 4.8: Illustration of the architecture.

this point, while $Worker_2$ has copied the stack at an earlier point where the decision $y = c$ was taken. Each worker picks a literal from a global list of literals that should be tested. $Worker_2$ picks the literal $(w, c)$ and detects that is SAC-inconsistent. A message $m_1$ indicating that $w \neq c$ can be inferred at decision level 2 (with time-stamp 2) is posted. $Worker_1$ tests $(z, a)$ and proves that it is inconsistent with the previous assignments. Therefore, it posts a message $m_2$ indicating than $z \neq a$ can be inferred at decision level 2 (with time-stamp 3).

The master finds out, when extracting messages, that $m_1$ is outdated since it has backtracked since $Worker_2$ has copied the *Assignments Stack*. The message $m_1$ is then removed. The message $m_2$ is relevant with respect to the problem state of the master. This latter can benefit from this information until it changes its decision at level 2.

## 4.5 Experimental results

In order to test the practical interest of the approach we propose, we have conducted a preliminary experimentation (with our solver AbsCon) using a cluster of bi-quad cores Xeon processors at 2.66 GHz node with 16GiB of RAM under Linux. We have used the search algorithm MAC (which maintains the property of generalized arc consistency during search), equipped with *dom/ddeg* [Bessiere and Régin, 1996] and *dom/wdeg* [Boussemart et al., 2004] as variable ordering heuristic and *lexico* as value ordering heuristic, to solve instances of different problems [1].

We have compared the wall-clock time (wck) in second(s), CPU time (cpu) in second(s) and the number of nodes (#nodes) for both classical sequential approaches and the parallel ones. For the sequential approaches, we have used MAC, SAC+MAC where SAC is enforced only at preprocessing, and MSAC that enforces SAC during the search (at each node). For the parallel approach, we have used 1, 3 and 7 workers (and of course, one master). The number of total messages (#tMsgs) and useful ones (#uMsgs) sent from the workers to the master is displayed. The speed-up ratio of resolution time compared to MAC is also calculated as follows:

$$\frac{MAC\ wck\ time}{parallel\ wck\ time} \tag{4.1}$$

A time-out of $1,800$ seconds was set per instance ; when the instance cannot be solved within this limit, TO is indicated in the table. In our tests, we suppose that CPU resources are available (for free). Under this assumption, we just focused on *wck time* rather than *CPU time*

Table 4.1 describes the results obtained when using the heuristic *dom/wdeg* on some selected instances. Obviously, our approach does not overweight the

[1] available at http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html

classical MAC solver, but it gives better resolution times (in wall-clock) than the other sequential approaches (SAC+MAC, MSAC).

| Instance | | #nodes | wck time | cpu time | #tMsgs | #uMsgs | Speedup ratio |
|---|---|---|---|---|---|---|---|
| | MAC | 3,411 | **1.188** | 1.375 | | | |
| | SAC+MAC | 3,411 | 1.211 | 1.414 | | | |
| cc-10-10-2 | MSAC | **367** | 2.997 | 3.199 | | | |
| | Par(7) | 3138 | 4.913 | 18.559 | 4,123 | 332 | 0.28 |
| | Par(3) | 3129 | 2.510 | 10.226 | 2,854 | 92 | 0.50 |
| | Par(1) | 3730 | 1.448 | 4.18 | 920 | 24 | 0.82 |
| | MAC | 5,309 | **81.047** | 80.943 | | | |
| | SAC+MAC | 3,723 | 446.735 | 445.572 | | | |
| cw-ogd-vg12-15 | MSAC | | TO | | | | |
| | Par(7) | 5,824 | 346.884 | 2255.207 | 11,956 | 2,092 | 0.23 |
| | Par(3) | 5,072 | 162.165 | 746.163 | 6,264 | 850 | 0.50 |
| | Par(1) | **5,008** | 134.122 | 247.401 | 1,878 | 90 | 0.60 |
| | MAC | 891,850 | **30.449** | 30.353 | | | |
| | SAC+MAC | 891,850 | 31.181 | 31.339 | | | |
| langford-3-13 | MSAC | | TO | | | | |
| | Par(7) | **864,631** | 57.564 | 231.143 | 1,102,392 | 36,544 | 0.53 |
| | Par(3) | 884,835 | 34.483 | 140.035 | 84,7136 | 16,079 | 0.88 |
| | Par(1) | 880,091 | 32.523 | 68.443 | 356,678 | 5,071 | 0.94 |
| | MAC | 660,504 | **17.753** | 17.825 | | | |
| | SAC+MAC | 660,504 | 18.334 | 18.25 | | | |
| queen-12-12-14 | MSAC | **18,651** | 200.428 | 199.81 | | | |
| | Par(7) | 664,789 | 41.884 | 123.137 | 574,521 | 41,555 | 0.42 |
| | Par(3) | 693,670 | 23.371 | 77.039 | 420,752 | 16,996 | 0.76 |
| | Par(1) | 742,988 | 18.748 | 39.429 | 224,975 | 6,723 | 0.70 |
| | MAC | 5,573 | 0.993 | 0.944 | | | |
| | SAC+MAC | **0** | 0.426 | 0.633 | | | |
| qK-12-5-mul | MSAC | **0** | **0.418** | 0.639 | | | |
| | Par(7) | 10,594 | 3.821 | 12.026 | 31,455 | 2,917 | 0.72 |
| | Par(3) | 6,299 | 1.410 | 5.309 | 30,601 | 2,886 | 1.11 |
| | Par(1) | 7,938 | 3.856 | 2.959 | 5,818 | 5,015 | 0.44 |

Table 4.1: Results for the sequential and parallel approaches for solving a few selected instances with dom/wdeg.

Table 4.2 describes the average wall-clock time for several series of instances. Times are mentioned for a classical MAC solver, MSAC, SAC+MAC and also our parallel approach using 7 workers. The wck-time of instances that reach time-out is penalized by $1,800 * 3$ seconds. On the 8 series of instances, our approach has the best average wall-clock time on only 2 series. The classical MAC solver is the best solver when using dom/wdeg heuristic compared to our parallel approach.

| Series | #inst | MAC | MSAC | SAC+MAC | Par(7) |
|---|---|---|---|---|---|
| langford | 17 | **90.55** | 905.32 | 391.14 | 408.16 |
| queen | 7 | 785.85 | 4143 | 2382 | **235.42** |
| quennsKnight | 22 | 1473 | **162.39** | 165.97 | 1264 |
| rand-8-20-5-18-800 | 10 | **45.91** | 517.68 | 1124 | 93.20 |
| fapp25-2230 | 12 | 46.95 | 620.83 | 48.80 | **41.93** |
| ewddr2-10-by-5 | 10 | **1.545** | 6.589 | 2.336 | 3.026 |
| cc | 13 | **2.772** | 19.62 | 2.837 | 13.46 |
| BlackHole-4-4-e | 10 | **0.458** | 7.858 | 0.485 | 1.110 |

Table 4.2: The average wall clock time for series of instances with dom/wdeg.

Disappointed by these results, we used *dom/ddeg* to compare our parallel approach with the sequential ones. This variable ordering heuristic guarantees that the different approaches have the same search path, which could by the reason behind the failure of our approach using *dom/wdeg*. Results are given in Table 4.3.

Compared with a classical solver using MAC, the interest of using a parallel approach is visible on these five instances:

- For some instances (*cw-ogd-vg12-15*, *langford-3-13* and *qk-12-5-mul*), the number of visited nodes is highly decreased;
- For some instances (*cc-10-10-2* and *qk-12-5-mul*), the wall clock time is decreased.

One interest of our approach is that SAC tests are interleaved with search, which means that we do not have to wait for the completion of the preprocessing to start the actual search, and still, we benefit from the discovery of inconsistent literals. In a sense, we use a kind of *anytime* version of SAC.

We investigate the efficiency of used messages on the speed-up obtained when using our parallel approach compared to the classical MAC solver. For the Crossword instance (cw-ogd-vg12-15), the inferences made by the workers are too limited for being useful compared with a MAC solver. Even though, using a parallel approach remains interesting compared to SAC+MAC or MSAC in term of wall clock time. On this instance, it appears that, for SAC+MAC, the most of CPU time (404 seconds) is spent at preprocessing. This explains why MSAC is

| Instance | | #nodes | wck time | cpu time | #tMsgs | #uMsgs | Speedup ratio |
|---|---|---|---|---|---|---|---|
| cc-10-10-2 | MAC | 542,776 | 47.90 | 47.479 | | | |
| | SAC+MAC | 542,776 | 48.23 | 47.946 | | | |
| | MSAC | **4960** | 49.10 | 48.73 | | | |
| | Par(7) | 73,684 | 114.889 | 456.136 | 46,111 | 23,979 | 2.3 |
| | Par(3) | 139,351 | **19.49** | 154.4 | 160.77 | 22,240 | 2.46 |
| | Par(1) | 315,035 | 33.77 | 81.087 | 268,379 | 16,512 | 1.42 |
| cw-ogd-vg12-15 | MAC | 4,224 | **70.71** | 70.528 | | | |
| | SAC+MAC | 4,648 | 545.48 | 543.951 | | | |
| | MSAC | | TO | | | | |
| | Par(7) | **4116** | 231.988 | 1590.632 | 6,469 | 412 | 0.292 |
| | Par(3) | 4241 | 125.783 | 616.308 | 3,361 | 158 | 0.536 |
| | Par(1) | 4216 | 94.94 | 180.118 | 1,352 | 71 | 0.745 |
| langford-3-13 | MAC | 764,944 | 29.14 | 29.148 | | | |
| | SAC+MAC | 764,944 | **28.53** | 28.689 | | | |
| | MSAC | | TO | | | | |
| | Par(7) | **750,105** | 47.01 | 195.52 | 10,617,44 | 33,987 | 0.620 |
| | Par(3) | 759,166 | 30.46 | 116.039 | 822,626 | 15,073 | 0.957 |
| | Par(1) | 763,373 | 28.80 | 60.192 | 300,048 | 4,994 | 1.019 |
| queen-12-12-14 | MAC | 2,211,140 | **64.12** | 64.123 | | | |
| | SAC+MAC | 2,211,140 | 65.650 | 65.427 | | | |
| | MSAC | **61,179** | 170.87 | 468.701 | | | |
| | Par(7) | 2,000,032 | 97.09 | 512.836 | 2,297,683 | 164,826 | 0.660 |
| | Par(3) | 2,083,863 | 68.49 | 265.991 | 1,739,529 | 110,223 | 0.936 |
| | Par(1) | 2,173,009 | 70.39 | 133.688 | 886,580 | 44,101 | 0.910 |
| qK-12-5-mul | MAC | 2,017,288 | 44.88 | 44.937 | | | |
| | SAC+MAC | **0** | 0.438 | 0.629 | | | |
| | MSAC | **0** | **0.401** | 0.629 | | | |
| | Par(7) | 1,647,766 | 87.391 | 533.816 | 10,505,678 | 1,778,671 | 0.513 |
| | Par(3) | 1,838,425 | 70.161 | 272.171 | 5,535,294 | 857,197 | 0.640 |
| | Par(1) | 1,928,055 | 59.009 | 115.684 | 1,880,410 | 365,341 | 0.760 |

Table 4.3: Results for the sequential and parallel approaches for solving a few selected instances with dom/ddeg.

not able to solve the problem in $1,800$ seconds. For the Chessboard Coloration instance (cc-10-10-2), the parallel approach achieves the best wall clock time with an average of messages number around $20,000$. For the Langford and Queen instances, the number of messages is greater than the Chessboard Coloration instance. However, this does not imply in anyway that the parallel approach is quicker. To conclude, analyzing these results shows clearly that the speed-up obtained is not correlated with to the number of used messages when using our parallel approach compared to the classical MAC solver. Instances can use an important number of messages but not reduce their search tree and, contrary to that, other instances could benefit from a few number of messages to reduce significantly their search tree.

| Series | #inst | MAC | MSAC | SAC+MAC | Par(7) |
|---|---|---|---|---|---|
| langford | 17 | 77.828 | 830,090 | **76.995** | 397.311 |
| queen | 7 | 845.799 | 4,659.207 | 24,443.690 | **319.756** |
| queensKnight | 22 | 3,446.051 | **160.044** | 166.904 | 1,475.197 |
| rand-8-20-5-18-800 | 10 | **20.221** | 204.801 | 22.192 | 34.837 |
| fapp25-2230 | 12 | 46.246 | 619.829 | 49.241 | **43.191** |
| ewddr2-10-by-5 | 10 | **1.518** | 6.888 | 2.362 | 3.046 |
| cc | 10 | 53.004 | 67.663 | 54.017 | **13.422** |
| BlackHole-4-4-e | 10 | **0.493** | 11.354 | 0.498 | 1.156 |

Table 4.4: The average wall clock time for series of instances with dom/ddeg.

Table 4.4 describes the average wall-clock time for several series of instances. Times are mentioned for a classical MAC solver, MSAC, SAC+MAC and also our parallel approach using 7 workers. The wck-time of instances that reach time-out is penalized by $1,800 * 3$ seconds. Our parallel approach has the best solving wall-clock time on 3 series out of 8 which outweigh the results obtained with MSAC and SAC+MAC. However, we have been disappointed in not reducing significantly the solving time compared to a MAC solver. In fact, using such a strong consistency is supposed to infer important information that can be used by a classical solver to reduce its search tree.

In order to understand the reasons behind the fruitless results (compared to MAC) of our approach, we decide to make more experiments. On the one hand, we used an oracle of inferences: we wanted to know what we would get if the SAC tests were costless (*costlessMSAC*). To do that, we stored the inferences made when maintaining classical SAC during search (MSAC). We then use the previous stored inferences (oracle of inferences) while maintaining arc consistency (MAC). We get, thus, the best case times of maintaining SAC during search. On the other hand, we compared *costlessMSAC* with a classical MAC solver and estimated the speed-up ratio as follows:

$$\frac{MAC\ wck\ time}{costlessMSAC\ wck\ time} \tag{4.2}$$

Figure 4.9 describes the distribution of 876 instances of several problems according to the speed-up ratio. In fact, only 3% of the instances of the conducted experimentation have a speed-up ratio greater than 10. Whereas, the other 94% of the instances have a speed-up ratio less than 2. In this case, it is difficult to overweight the costs of maintaining SAC in parallel. This leads us to assume that maintaining SAC during search is not as effective as supposed in reducing the search tree.

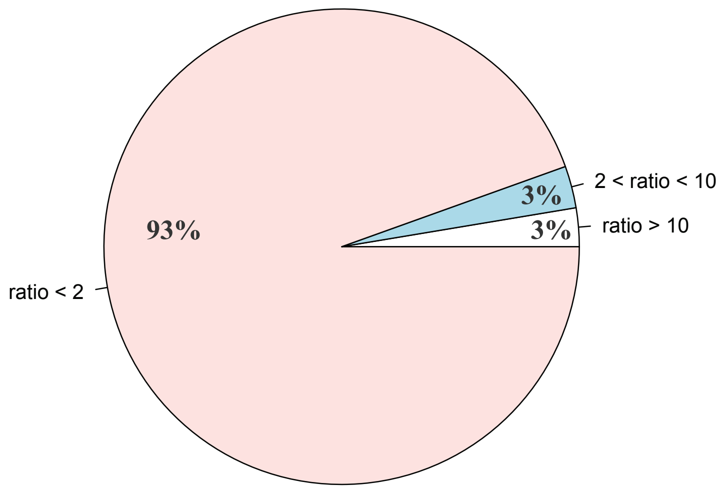
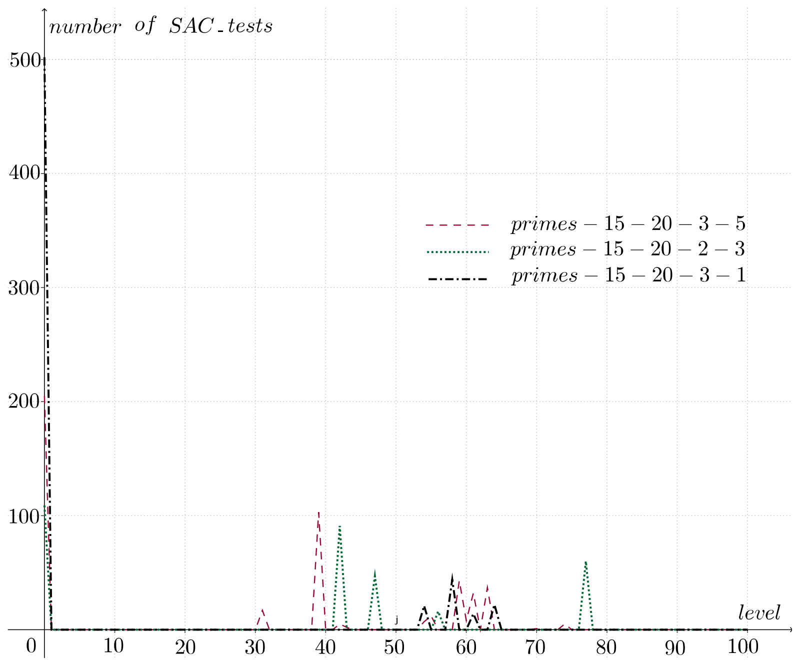


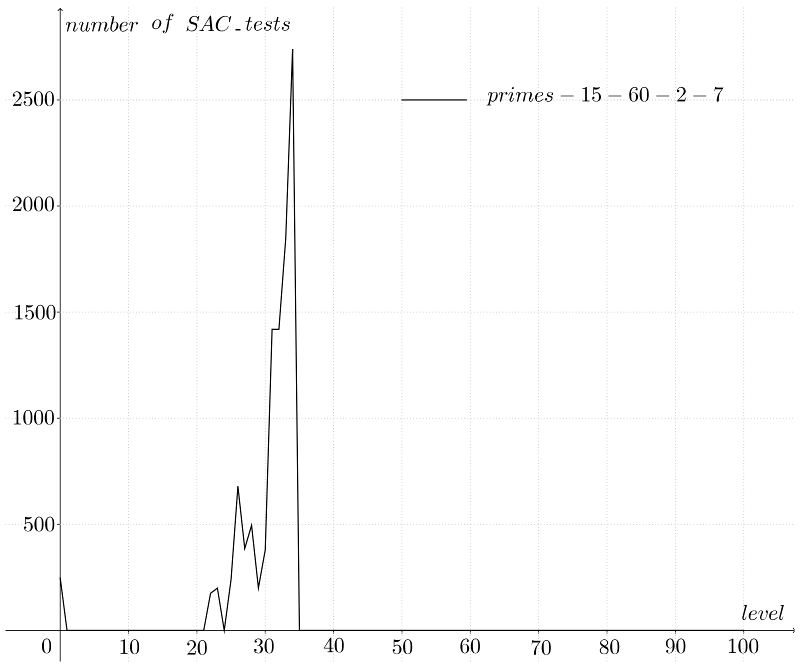
Figure 4.9: Pie chart of instances according to the speed-up ratio.

Table 4.5 describes the results of some selected instances of the previous experimentation: comparing MAC to costlessMSAC. These instances are presented in an increasing order of their speed-up. Obviously, the difficulty of instance resolution does not affect the obtained speed-up ratio. For example, the *langford-3-14* instance is more difficult than *val10-43*, whereas the speedup ratio for *langford-3-14* is significantly less than *val10-43*.

Figure 4.10(a) describes the number of effective SAC tests per level that contribute to the reduction of the search tree and, thus, helping the solver during its solving process. These results are more detailed in Table 4.6. In fact, we chose instances composed of the same number of variables (100 variables) and all variables have the same domain size of 46. However, these instances have different speed-up. Obviously, the *primes-15-60-2-7* instance which has the greatest speed-up ratio, has also the greatest number of effective SAC tests during the 32 first levels. This number reaches 2740 tests for the level 32. For the other instances, the greatest number of effective SAC tests are mostly during pre-processing whereas the remaining SAC tests are distributed during the search in few amounts. Obviously, some instances for which maintaining *costlessSAC* is efficient use a large number of SAC tests in order to reduce the search space.



(a) The effective SAC tests per level.



(b) The effective SAC tests per level.

Figure 4.10: Comparison between costlessMSAC and MAC for some selected instances of the *primes* problem using dom/ddeg.

| Instance | costlessMSAC | MAC | speed-up ratio |
|---|---|---|---|
| 2-fullins-5-4 | 11.44 | 13.683 | 1.196 |
| langford-3-14 | 93.257 | 229.071 | 2.456 |
| crossword-m1c-lex-vg6-7 | 16.527 | 51.253 | 3.101 |
| bdd-21-133-18-78-11 | 2.427 | 12.224 | 5.037 |
| rand-8-20-5-18-800-13 | 3.857 | 51.407 | 13.328 |
| cc-10-10-2 | 3.088 | 75.784 | 24.541 |
| half-n25-d5-e56-r7-1 | 4.703 | 198.238 | 42.151 |
| primes-10-20-2-5 | 1.202 | 51.973 | 43.239 |
| val10-43 | 0.772 | 43.983 | 56.973 |

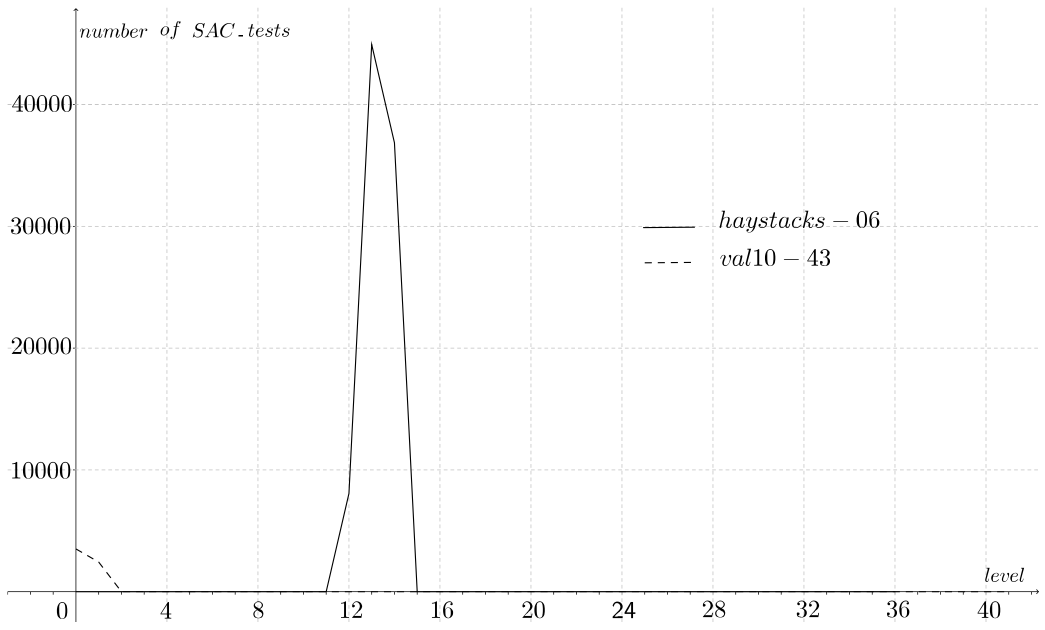
Table 4.5: The wall clock time (in seconds) for some selected instances with costlessMSAC and MAC (dom/ddeg).

Figure 4.11(a) describes two instances with different speed-up ratio (wck-time and speed-up are detailed in 4.11(b)). These results show, clearly, that the previous assumption is not always true. In fact, *haystacks-06* instance benefits from the greatest number of SAC tests during search reaching almost 45000 tests in its 13th level but does not reach an important speed-up compared to MAC. However, *val10-43* instance reaches a speed-up of almost 57 compared to MAC benefiting from a few amount of SAC tests at pre-processing and its first level.

Furthermore, using a high number of SAC tests at the first level of the search tree does not guarantee a high speed-up which the case for *primes-15-20-2-3* and *primes-15-20-3-1* instances (Figure 4.10(a)) compared to *val10-43* instance.

| Instance | costlessMSAC | MAC | Speed-up ratio |
|---|---|---|---|
| primes-15-60-2-7 | 1.194 | 37.384 | 31.310 |
| primes-15-20-3-5 | 0.875 | 15.833 | 18.095 |
| primes-15-20-2-3 | 0.756 | 1.254 | 1.659 |
| primes-15-20-3-1 | 0.596 | 0.903 | 1.515 |

Table 4.6: Comparison between costlessMSAC and MAC for some selected instances of the *primes* problem dom/ddeg: the wall clock time (in seconds) and the speed-up ratio.



(a) The effective SAC tests per level.

| Instance | costlessMSAC | MAC | speed-up ratio |
|---|---|---|---|
| haystacks-06 | 0.88 | 1.43 | 1,625 |
| val10-43 | 0.772 | 43.983 | 56.973 |

(b) The wall-clock time and the speed-up ratio.

Figure 4.11: Comparison between costlessMSAC and MAC for two instances of different speed-up ratio.

We have been disappointed by the inability of our approach, and in particular the singleton arc consistency, to reduce significantly the search tree for most of the experimented instances. This is why we tried to maintain a stronger consistency: weak-k-SAC. We conducted an experimentation over more than 360 different instances. Unfortunately, weak-k-SAC failed to prune an important search space during search, and thus, maintaining it during search does not seem to be effective in the resolution of the different instances. Table 4.7 describes the results obtained for some selected instances when establishing weak-k-SAC (presented in a decreasing order of speed-up ratio)

| Instance | (#nbVar, #nbVal) | #costlessMWeak-1-SAC | MAC | speed-up ratio |
|---|---|---|---|---|
| val7-12 | (20, 960) | 0.133 | 294.59 | 2215 |
| val16-27 | (53, 4,558) | 0.318 | 422.25 | 1328 |
| val45-50 | (128, 16,384) | 0.412 | 57.63 | 139.88 |
| val46-50 | (128, 16,384) | 0.473 | 46.65 | 98.63 |
| val47-50 | (128, 16,384) | 1.504 | 46.82 | 31.13 |
| val7-27 | (20, 1,720) | 0.127 | 2.831 | 22.29 |
| val24-41 | (75, 9,600) | 0.331 | 5.676 | 17.15 |
| val44-50 | (128, 16,384) | 4.783 | 55.24 | 11.55 |
| E-12 | (270, 81,000) | 25.80 | 181.17 | 7.02 |
| val18-39 | (64, 7,744) | 1.919 | 9.819 | 5.12 |
| val12-29 | (48, 4,800) | 0.319 | 0.959 | 3 |
| val35-43 | (120, 15,360) | 0.614 | 1.319 | 2.15 |
| lei450-15c-05 | (450, 2,250) | 1.444 | 1.769 | 1.22 |
| ash958GPIA-3 | (1,916 , 5,748) | 1.684 | 1.565 | 0.929 |
| F-10 | (270, 81,000) | 13.144 | 2.654 | 0.201 |

Table 4.7: The wall clock time (in seconds) for some selected instances with costlessMWeak-1-SAC and MAC (dom/ddeg) establishing weak-1-SAC.

## Conclusion

In this chapter, we introduced a new approach using a parallel architecture in order to enhance the classical solving process. Several workers establish a partial consistency (SAC/ weak-k-SAC) and send the discovered facts to the master in order to avoid useless search space. Contrary to our expectations, our approach seems to be less effective than MAC (the most used search algorithm). This could be due to the fact that the discovered SAC-inconsistent values are, in fact, easily removed by the master by propagation. This implies that the sent information is not as strong as expected to speed-up the master search. The choice of the variable ordering heuristic could be also the reason behind the failure of our approach.

# Chapter 5

# Conclusions and Future Work

The work presented can be divided into two main parts:

**Compressing Table Constraints** In the first part, we deal with the most used kind of constraints which are table constraints. Since table constraints offer expressiveness to non-expert users, its use is widespread in the industrial world. Over time, the represented data keeps growing which implies the use of compact data-structures to store them, in a first place, and optimized filtering algorithms to manage them during the solving process, in a second place. In the last decades, several techniques proposed compact ways to represent table constraints while using the minimal storage space, and mostly, not penalizing their use during the search. We present some techniques of the state-of-the-art in the first section of Chapter 2. In our turn, we propose two different methods of compressing table constraints. Both of them are based on frequent patterns search in order to avoid redundancy. However, the manner of defining pattern, the patterns-detecting process and the new compact representation differs significantly. In fact, in the first method $STR^c$, we use the "trie" data structure in order to find patterns which are defined as a sequence of values. After detecting the most frequent patterns, we build a patterns table in which the most frequent patterns are stored. The new form of the table constraint replaces all occurrences of the patterns by a reference to them in the patterns table. In this way, we could benefit from avoiding these redundancies at two levels. At the space complexity level, the use of references enables us to reduce the space required to represent table constraints avoiding repeating the same sequences of values. At the time complexity level, using references to patterns enables us to avoid redundant validity checks. Rather than testing the validity of the same sequence several times in a pass of the table constraints, the test is done only one time and simply re-used for every occurrence. In order to manage the new compressed form of table constraint, we use an adapted variant of STR taking into consideration the use of patterns. The experimental results shows that our approach achieves a high compression, but compared to STR variants it competes only with $STR1$. The choice of the manner of defining patterns are the reasons behind the non-competitiveness of our results. In fact, since in our approach $STR^c$ we consider that a pattern is a sequence of values, we are not able to use a filtering algorithm based on the optimized variants of STR ($STR2$ and $STR3$). We take into account this drawback in our second approach *STR-slice*, where we defined patterns in another way (as pairs of a *variable* and a *value*). The frequent patterns detection is done, this time, using a data-mining algorithm. The new form of table constraint is, in fact, composed of "slices" of sub-tables. The original constraint table is fragmented based on the frequent patterns. Each "slice" is, then composed of a frequent pattern and the remaining part of the table where the pattern, originally, appears. The experimental results shows, this time, more competitive results where *STR-slice* supplants $STR2$ and, sometimes $STR3$ even if it is penalized by the compression process.

Generally speaking, each of the proposed approaches has its specificity and the experimental results depends mostly on the constraints representation. The main issue that all the proposed approaches of the-state-of-the-art have to find an answer is how to find a compromise between achieving a high compression ratio and filtering as effective as possible the new compact data structure.

**Future Work** First of all, we think that the tuning of the parameters used for guiding compression should be automatized (possibly, employing some machine learning techniques). $STR^c$ and *STR-slice* could then benefit from a better compression which depends on the nature of class of problem instance. Second, we believe that, in the rising context of big data, new constraint problems should emerge rapidly where constraints could be of (very) large arity and involve very large tables. *STR-slice* could advantageously handle such "huge" constraints, especially if we consider that slicing could be conducted recursively on the sub-tables which is another perspective of this thesis work. Finally, we think that the concept of sliced table constraints is interesting on its own for modeling, as certain forms of conditionality can be represented in a simple and natural way, directly with sliced table constraints.

**Using Singleton Arc Consistency in parallel to improve search** The second part is dedicated to another way to optimize CSP solving which is the use of a parallel architecture. The advancements of the parallel computing field has encouraged the CSP community, among others, to benefit from the computers power in order to solver larger problems. Several approaches are proposed in which the problem is distributed over workers or a copy of the problem is used for each of them. Some methods of the state-of-the-art of parallel computing and also parallel CSP solving are introduced in the second section of Chapter 2. All these methods tried to speed-up the solving time. We propose a method in

Chapter 4 where we use a parallel architecture in order to enhance the solving process by establishing parallel consistencies. In fact, different workers sent to their master the result of establishing partial consistencies as new discovered facts. The master, in its turns tries to benefit from them by removing the values leading to a failure. Contrary to our expectation, the work made by the workers does not imply a significant improvement of the solving process. In fact, an in-depth analysis shows that even in a sequential resolution enforcing such strong consistencies for free cannot imply an important reduction in the search space which makes the results obtained through our approach understandable.

**Future Work** A natural perspective of our work is to incorporate other forms of consistencies that could impact larger the neighborhood, and thus, reduce significantly the search space. We may consider "wise" messages that include many forms of consistencies which can together form a form of explanation that avoid a master to use a given path or given sequence of assignments. Moreover, we could investigate the impact of heuristics on the effectiveness of the received messages. Choosing a variable of the neighborhood may imply more deductions.

# Bibliography

[Agrawal et al., 1993] Agrawal, R., Imieli'nski, T., and Swami, A. (1993). Mining association rules between sets of items in large databases. In *Proceedings of SIGMOD'93*, pages 207–216.

[Agrawal and Srikant, 1994] Agrawal, R. and Srikant, R. (1994). Fast algorithms for mining association rules in large databases. In *Proceedings of ICVLDB'94*, pages 487–499.

[Amadini et al., 2015] Amadini, R., Gabbrielli, M., and Mauro, J. (2015). Sunny-cp: a sequential cp portfolio solver. In *Proceedings of SAC'15*, pages 1861–1867.

[Amdahl, 1967] Amdahl, G. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of ACM'67*, pages 483–485.

[Appel and Haken, 1977] Appel, K. and Haken, W. (1977). Every planar map is four colorable. Part I: Discharging. *Illinois Journal of Mathematics*, 21(3):429–490.

[Bartak and Erben, 2004] Bartak, R. and Erben, R. (2004). A new algorithm for singleton arc consistency. In *Proceedings of FLAIRS'04*, pages 257–262.

[Benson et al., 1992] Benson, J., Brent, W., and Freuder, E. (1992). Interchangeability preprocessing can improve forward checking search. In *Proceedings of ECAI'92*, pages 28–30.

[Bessiere, 1994] Bessiere, C. (1994). Arc consistency and arc consistency again. *Artificial Intelligence*, 65:179–190.

[Bessiere, 2006] Bessiere, C. (2006). Constraint propagation. In *Handbook of Constraint Programming*, chapter 3, pages 29–83. Elsevier.

[Bessiere and Debruyne, 2004] Bessiere, C. and Debruyne, R. (2004). Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of CP'04*, pages 17–27.

[Bessiere and Debruyne, 2005] Bessiere, C. and Debruyne, R. (2005). Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of IJCAI'05*, pages 54–59.

[Bessiere et al., 1999] Bessiere, C., Freuder, E., and Régin, J. (1999). Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107:125–148.

[Bessiere and Régin, 1996] Bessiere, C. and Régin, J. (1996). MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings of CP'96*, pages 61–75.

[Bessiere and Régin, 2001] Bessiere, C. and Régin, J. (2001). Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI'01*, pages 309–315.

[Bessiere and van Hentenryck, 2003] Bessiere, C. and van Hentenryck, P. (2003). To be or not to be ... a global constraint. In *Proceedings of CP'03*, pages 789–794.

[Bordeaux et al., 2009] Bordeaux, L., Hamadi, Y., and Samulowitz, H. (2009). Experiments with massively parallel constraint solving. In *Proceedings of IJCAI'09*, pages 443–448.

[Bornemann, 2003] Bornemann, F. (2003). Primes is in p, une avancée accessible à "l'homme ordinaire". *Gazette des Mathématiciens*, (98):14–30.

[Boussemart et al., 2004] Boussemart, F., Hemery, F., Lecoutre, C., and Sais, L. (2004). Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150.

[Bryant, 1986] Bryant, R. (1986). Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691.

[Cheng and Yap, 2006] Cheng, K. and Yap, R. (2006). Maintaining generalized arc consistency on ad-hoc n-ary Boolean constraints. In *Proceedings of ECAI'06*, pages 78–82.

[Cheng and Yap, 2010] Cheng, K. and Yap, R. (2010). An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2):265–304.

[Chu et al., 2009] Chu, G., Schulte, C., and Stuckey, P. (2009). Confidence-based work stealing in parallel constraint programming. In *Proceedings of CP'09*, pages 226–241.

[Chu et al., 2008] Chu, G., Stuckey, P., and Harwood, A. (2008). PMiniSAT: A Parallelization of MiniSAT 2.0. Technical report.

[Colton and Miguel, 2001] Colton, S. and Miguel, I. (2001). Constraint generation via automated theory formation. In *Proceedings of CP'01*, pages 575–579. Springer.

[Dasygenis and Stergiou, 2014] Dasygenis, M. and Stergiou, K. (2014). Building portfolios for parallel constraint solving by varying the local consistency applied. In *Proceedings of ICTAI'14*, pages 717–724.

[de la Banda et al., 2006] de la Banda, M. G., Marriott, K., Rafeh, R., and Wallace, M. (2006). The modelling language Zinc. In *Proceedings of CP'06*, pages 700–705.

[Dean and Ghemawat, 2008] Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

[Debruyne and Bessiere, 1997] Debruyne, R. and Bessiere, C. (1997). Some practical filtering techniques for the constraint satisfaction problem. In *Proceedings of IJCAI'97*, pages 412–417.

[Dijkstra, 2001] Dijkstra, E. (2001). Solution of a problem in concurrent programming control. In *Pioneers and Their Contributions to Software Engineering*, pages 289–294. Springer.

[Domshlak et al., 2005] Domshlak, R. B. C., Fernández, C., Gomes, C., Krishnamachari, B., Selman, B., and Valls, M. (2005). Sensor networks and distributed csp: communication, computation and complexity. *Artificial Intelligence*, 161(1):117–147.

[Dunstan, 1991] Dunstan, N. (1991). Semaphores for fair scheduling monitor conditions. *ACM SIGOPS Operating Systems Review*, 25(3):27–31.

[Fahle et al., 2001] Fahle, T., Schamberger, S., and Sellman, M. (2001). Symmetry breaking. In *Proceedings of CP'01*, pages 93–107.

[Focacci and Milano, 2001] Focacci, F. and Milano, M. (2001). Global cut framework for removing symmetries. In *Proceedings of CP'01*, pages 77–92.

[Fredkin, 1960] Fredkin, E. (1960). Trie memory. *Communications of the ACM*, 3(9):490–499.

[Freuder, 1978] Freuder, E. (1978). Synthesizing constraint expressions. *Communication of the ACM*, 21(11):958–965.

[Frisch et al., 2007] Frisch, A., Grum, M., Jefferson, C., Hernandez, B. M., and Miguel, I. (2007). The design of ESSENCE: A constraint language for specifying combinatorial problems. In *Proceedings of IJCAI'07*, pages 80–87.

[Frisch et al., 2002] Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., and Walsh, T. (2002). Global constraints for lexicographic orderings. In *Proceedings of CP'02*, pages 93–108.

[Gaschnig, 1979] Gaschnig, J. (1979). Performance measurement and analysis of search algorithms. Technical Report CMU-CS-79-124, Carnegie Mellon.

[Gent et al., 2006] Gent, I., Jefferson, C., and Miguel, I. (2006). Minion: A fast, scalable constraint solver. In *Proceedings of ECAI'06*, pages 98–102.

[Gent et al., 2011] Gent, I., Jefferson, C., Miguel, I., Moore, N., Nightingale, P., Prosser, P., and Unsworth, C. (2011). A preliminary review of literature on parallel constraint solving. In *Proceedings of PMCS'11*.

[Gent et al., 2007] Gent, I., Jefferson, C., Miguel, I., and Nightingale, P. (2007). Data structures for generalised arc consistency for extensional constraints. In *Proceedings of AAAI'07*, pages 191–197.

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

[Gharbi et al., 2014] Gharbi, N., Hemery, F., Lecoutre, C., and Roussel, O. (2014). Sliced table constraints: Combining compression and tabular reduction. In *Proceedings of CPAIOR'14*, pages 120–135.

[Ginsberg, 1993] Ginsberg, M. (1993). Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46.

[Gomes and Selman, 2001] Gomes, C. and Selman, B. (2001). Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62.

[Granas and Dugundji, 2003] Granas, A. and Dugundji, J. (2003). *Fixed Point Theory*. Springer, 1 edition.

[Gustafson, 1988] Gustafson, J. (1988). Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533.

[Hamadi, 1999] Hamadi, Y. (1999). Optimal distributed arc-consistency. In *Proceedings of CP'99*, pages 219–233.

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

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

[Haralick and Elliott, 1980] Haralick, R. and Elliott, G. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313.

[Henz et al., 1993] Henz, M., Smolka, G., and Würtz, J. (1993). Oz-a programming language for multi-agent systems. In *Proceedings of IJCAI'93*, pages 404–409.

[Herken, 1995] Herken, R. (1995). *The universal Turing machine: a half-century survey*. Springer-Verlag New York, Inc.

[Herlihy and Wing, 1987] Herlihy, M. P. and Wing, J. M. (1987). Axioms for concurrent objects. In *Proceedings of SIGACT-SIGPLAN'87*, pages 13–26.

[Hoare, 1974] Hoare, C. A. R. (1974). Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557.

[Hubbe and Freuder, 1992] Hubbe, P. and Freuder, E. (1992). An efficient cross product representation of the constraint satisfaction problem search space. In *Proceedings of AAAI'92*, pages 421–427.

[Jabbour et al., 2013a] Jabbour, S., Sais, L., and Salhi, Y. (2013a). A mining-based compression approach for constraint satisfaction problems. Technical Report arXiv:1305.3321, CoRR.

[Jabbour et al., 2013b] Jabbour, S., Sais, L., Salhi, Y., and Uno, T. (2013b). Mining-based compression approach of propositional formulae. In *Proceedings of CIKM'13*, pages 289–298.

[Katsirelos and Bacchus, 2005] Katsirelos, G. and Bacchus, F. (2005). Generalized nogoods in CSPs. In *Proceedings of AAAI'05*, pages 390–396.

[Katsirelos and Walsh, 2007] Katsirelos, G. and Walsh, T. (2007). A compression algorithm for large arity extensional constraints. In *Proceedings of CP'07*, pages 379–393.

[Kotthoff and Moore, 2010] Kotthoff, L. and Moore, N. (2010). Distributed solving through model splitting.

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

[Lecoutre and Cardon, 2005] Lecoutre, C. and Cardon, S. (2005). A greedy approach to establish singleton arc consistency. In *Proceedings of IJCAI'05*, pages 199–204.

[Lecoutre and Hemery, 2007] Lecoutre, C. and Hemery, F. (2007). A study of residual supports in arc consistency. In *Proceedings of IJCAI'07*, pages 125–130.

[Lecoutre et al., 2012] Lecoutre, C., Likitvivatanavong, C., and Yap, R. (2012). A path-optimal GAC algorithm for table constraints. In *Proceedings of ECAI'12*, pages 510–515.

[Lecoutre and Prosser, 2006] Lecoutre, C. and Prosser, P. (2006). Maintaining singleton arc consistency. In *Proceedings of CPAI'06*, pages 47–61.

[Lecoutre et al., 2007] Lecoutre, C., Sais, L., Tabary, S., and Vidal, V. (2007). Transposition Tables for Constraint Satisfaction. In *Proceedings of AAAI'07*, pages 243–248.

[Lecoutre and Szymanek, 2006] Lecoutre, C. and Szymanek, R. (2006). Generalized arc consistency for positive table constraints. In *Proceedings of CP'06*, pages 284–298.

[Lecoutre and Tabary, 2007] Lecoutre, C. and Tabary, S. (2007). Abscon 109: a generic CSP solver. In *Proceedings of the 2006 CSP solver competition*, pages 55–63.

[Li et al., 2009] Li, T., Baumberger, D., and Hahn, S. (2009). Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *Proceedings of SIGPLAN'09*, pages 65–74. ACM.

[Mackworth, 1977] Mackworth, A. (1977). Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118.

[Maheswaran et al., 2004] Maheswaran, R., Tambe, M., Bowring, E., Pearce, J., and Varakantham, P. (2004). Taking dcop to the real world: Efficient complete solutions for distributed multi-event scheduling. In *Proceedings of AAMAS'04*, pages 310–317.

[Mairy et al., 2015] Mairy, J., Deville, Y., and Lecoutre, C. (2015). The smart table constraint. In *Proceedings of CPAIOR'15*, pages 271–287.

[Michel and Hentenryck, 2012] Michel, L. and Hentenryck, P. V. (2012). Activity-based search for black-box constraint programming solvers. In *Proceedings of CPAIOR'12*, pages 228–243.

[Mohr and Henderson, 1986] Mohr, R. and Henderson, T. (1986). Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233.

[Mohr and Masini, 1988] Mohr, R. and Masini, G. (1988). Good old discrete relaxation. In *Proceedings of ECAI'88*, pages 651–656.

[Montanari, 1974] Montanari, U. (1974). Network of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7:95–132.

[Nguyen and Deville, 1998] Nguyen, T. and Deville, Y. (1998). A distributed arc-consistency algorithm. *Science of Computer Programming*, 30(1–2):227 – 250.

[Nightingale et al., 2011] Nightingale, P., Gent, I., Jefferson, C., and Miguel, I. (2011). Exploiting short supports for generalised arc consistency for arbitrary constraints. In *Proceedings of IJCAI'11*, pages 623–628.

[Nightingale et al., 2013] Nightingale, P., Gent, I., Jefferson, C., and Miguel, I. (2013). Short and long supports for constraint propagation. *Journal of Artificial Intelligence Research*, 46:1–45.

[Nyström et al., 2003] Nyström, D., Tešanović, A., and Hansson, C. N. J. (2003). The COMET database management system. Technical report, Mälardalen Real-Time Research Centre, Mälardalen University.

[O'Mahony et al., 2008] O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., and O'Sullivan, B. (2008). Using case-based reasoning in an algorithm portfolio for constraint solving. In *Proceedings of ICAIG'08*, pages 53–62.

[Perez and Régin, 2014] Perez, G. and Régin, J. (2014). Improving gac-4 for table and mdd constraints. In *Proceedings of CP'14*, pages 606–621.

[Pesant, 2001] Pesant, G. (2001). A filtering algorithm for the stretch constraint. In *Proceedings of CP'01*, pages 183–195.

[Pesant, 2004] Pesant, G. (2004). A regular language membership constraint for finite sequences of variables. In *Proceedings of CP'04*, pages 482–495.

[Pesant et al., 2012] Pesant, G., Quimper, C.-G., and Zanarini, A. (2012). Counting-based search: Branching heuristics for constraint satisfaction problems. *Journal of Artificial Intelligence Research*, 43:173–210.

[Pramod and Vyas, 2010] Pramod, S. and Vyas, O. (2010). Article: Survey on frequent itemset mining algorithms. *International Journal of Computer Applications*, 1(15):86–91.

[Prosser, 1993] Prosser, P. (1993). Hybrid algorithms for the constraint satisfaction problems. *Computational Intelligence*, 9(3):268–299.

[Prosser et al., 1992] Prosser, P., Conway, C., and Muller, C. (1992). A constraint maintenance system for the distributed resource allocation problem. *Intelligent Systems Engineering*, 1(1):76–83.

[Rao and Kumar, 1988] Rao, V. and Kumar, V. (1988). Superlinear speedup in parallel state-space search. In *Foundations of Software Technology and Theoretical Computer Science*, pages 161–174. Springer.

[Refalo, 2004] Refalo, P. (2004). Impact-based search strategies for constraint programming. In *Proceedings of CP'04*, pages 557–571.

[Régin, 1994] Régin, J. (1994). A filtering algorithm for constraints of difference in CSPs. In *Proceedings of AAAI'94*, pages 362–367.

[Régin, 2011a] Régin, J. (2011a). Global constraints: a survey. In *Hybrid Optimization*, chapter 2, pages 63–134. Springer.

[Régin, 2011b] Régin, J. (2011b). Improving the expressiveness of table constraints. In *Proceedings of ModRef'11*.

[Régin et al., 2013] Régin, J., Rezgui, M., and Malapert, A. (2013). Embarrassingly parallel search. In *Proceedings of CP'13*, pages 596–610.

[Régin et al., 2014] Régin, J., Rezgui, M., and Malapert, A. (2014). Improvement of the embarrassingly parallel search for data centers. In *Principles and Practice of Constraint Programming*, pages 622–635. Springer.

[Rochart et al., 2006] Rochart, G., Cambazard, H., Laburthe, F., Jussien, N., and Benoist, T. (2006). Choco: a java library for constraint satisfaction problems (csp), constraint programming (cp) and explanation-based constraint solving (e-cp).

[Roussel and Lecoutre, 2009] Roussel, O. and Lecoutre, C. (2009). XML representation of constraint networks: Format XCSP 2.1. Technical Report arXiv:0902.2362, CoRR.

[Sabin and Freuder, 1994] Sabin, D. and Freuder, E. (1994). Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of CP'94*, pages 10–20.

[Schubert et al., 2008] Schubert, T., Lewis, M., and Becker, B. (2008). Pamiraxt: Parallel sat solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:203–222.

[Schulte, 2000] Schulte, C. (2000). Parallel search made simple. In *University of Singapore*, pages 41–57.

[Schulte et al., 2006] Schulte, C., Tack, G., and Lagerkvist, M. (2006). Gecode: A generic constraint development environment.

[Simonis and O'Sullivan, 2008] Simonis, H. and O'Sullivan, B. (2008). Search strategies for rectangle packing. In *Proceedings of CP'08*, pages 52–66. Springer.

[Sleep, 1981] Sleep, F. B. M. (1981). Executing functional programs on a virtual tree of processors. In *Proceedings of FPCA'81*, pages 187–194.

[Stergiou, 2008] Stergiou, K. (2008). Heuristics for dynamically adapting propagation. In *Proceedings of ECAI'08*, pages 485–489.

[Stuckey et al., 2010] Stuckey, P., Becket, R., and Fischer, J. (2010). Philosophy of the minizinc challenge. *Constraints*, 15(3):307–316.

[Tiedemann et al., 2007] Tiedemann, P., Andersen, H. R., and Pagh, R. (2007). Generic global constraints based on mdds.

[Tseitin, 1983] Tseitin, G. (1983). On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer.

[Ullmann, 1977] Ullmann, J. (1977). A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words. *Computer Journal*, 20(2):141–147.

[Ullmann, 2007] Ullmann, J. (2007). Partition search for non-binary constraint satisfaction. *Information Science*, 177:3639–3678.

[van Dongen, 2006] van Dongen, M. (2006). Beyond singleton arc consistency. In *Proceedings of ECAI'06*, pages 163–167.

[van Hentenryck, 1999] van Hentenryck, P. (1999). *The OPL Optimization Programming Language*. The MIT Press.

[Wallace, 2005] Wallace, R. (2005). Heuristic policy analysis and efficiency assessment in constraint satisfaction search. In *Proceedings of CPAI'05*, pages 79–91.

[Witten and Frank, 2005] Witten, I. and Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., second edition.

[Wu et al., 2008] Wu, X., Kumar, V., Quinlan, J. R., Ghosh, J., Yang, Q., Motoda, H., McLachlan, G., Ng, A., Liu, B., Yu, P., Zhou, Z.-H., Steinbach, M., Hand, D., and Steinberg, D. (2008). Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37.

[Xia and Yap, 2013] Xia, W. and Yap, R. (2013). Optimizing STR algorithms with tuple compression. In *Proceedings of CP'13*, pages 724–732.

[Xie and Davenport, 2010] Xie, F. and Davenport, A. (2010). Massively parallel constraint programming for supercomputers: Challenges and initial results. In *Proceedings of CPAIOR'10*, pages 334–338. Springer.

[Yokoo et al., 1998] Yokoo, M., Durfee, E., Ishida, T., and Kuwabara, K. (1998). The distributed constraint satisfaction problem: Formalization and algorithms. *Knowledge and Data Engineering, IEEE Transactions on*, 10(5):673–685.

[Yokoo et al., 1992] Yokoo, M., Ishida, T., Durfee, E., and Kuwabara, K. (1992). Distributed constraint satisfaction for formalizing distributed problem solving. In *Proceedings of ICDCS'92*, pages 614–621. IEEE.

[Zhang and Yap, 2001] Zhang, Y. and Yap, R. (2001). Making AC3 an optimal algorithm. In *Proceedings of IJCAI'01*, pages 316–321.