UMONS
Université de Mons

Faculté
des Sciences

Mémoire

Data Repairing

Réparation de contraintes

Écrit par : Maxime Van Herzeele
Année académique : 2017-2018
Directeur de mémoire : Jef Wijsen
Section : MAB2 Sciences Informatiques

Faculté des Sciences • University of Mons • Place du Parc 20 • B-7000 Mons

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

De nombreuses institutions et entreprises collectent, stockent et utilisent de grandes quantités d'informations. Le traitement de ces données permettent à ces entreprises d'effectuer des statistiques ou de fournir un service à l'un de leurs clients. Mais dans les bases de données, nous pouvons retrouver des données *erronées*. Leur présence peut être due à une erreur lors de l'encodage ou lors de la collecte des informations(appareils imprécis, mauvaises méthodes de collecte, personne qui ment ou se trompe,...). Ces erreurs nuisent au traitement des données, biaisant les statistiques et perturbant les services qui en ont besoin. Afin d'éviter ce problème, les données devraient respecter des *contraintes d'intégrité*. Une contrainte d'intégrité est une règle appliquée aux données. Si une donnée ne respecte pas au moins une de ses contraintes alors cette donnée doit être considérée comme erronée.

Malheureusement, il se peut que ces contraintes soient elles aussi erronées. A cause de ses erreurs sur les contraintes, celles-ci peuvent échouer dans la différenciation des données réellement erronées et des données correctes. Il se peut que certaines données soient identifiées comme étant des violations des contraintes (données erronées) malgré qu'elles ne le devraient pas. Il existe aussi des contraintes qui n'arrivent pas à détecter des violations. Les erreurs à la fois sur les données et sur les contraintes sont un problème pour quiconque souhaite utiliser la base de données

Le terme *Data repairing* ou réparation de données signifie réparer les données mais aussi les contraintes d'intégrité. Il serait naïf de penser que l'on puisse supprimer chaque ligne qui possède au moins une donnée erronée. La perte d'information serait importante parce qu'une telle pratique demanderait d'effacer une ligne complète de la table et ce malgré qu'il n'y ait qu'une seule erreur dans la ligne. Par exemple, si nous avons une base de données des employés d'une entreprise et la contrainte "Le salaire d'un employé augmente chaque année". Cette contrainte n'est pas respectée à la table 1.1. En effet, si nous prenons l'année 2016, nous voyons que monsieur Dupont gagne moins d'argent qu'en 2015 et 2014. Si nous supprimons l'année 2016, nous perdons l'information concernant le

	Nome	Année	Salaire (€)	Grade	Bureau
t_2	Dupont	2013	24.000	ouvrier	1B14
t_3	Dupont	2014	38.000	chef de service	2B12
t_4	Dupont	2015	42.000	chef de service	2B18
t_5	Dupont	2016	37.000	chef de service	2B18
t_6	Dupont	2017	44.000	chef de service	2B18
t_7	Dupont	2018	28.000	ouvrier	1B14
t_8	Ana	2011	26.000	ouvrier	1B08
...					

TABLE 1.1 –

bureau que monsieur Dupont occupait.

En outre, les contraintes d'intégrité peuvent aussi être erronées ce qui veut dire que l'on pourrait supprimer une ligne ne contenant que des données correctes. Reprenons la contrainte "Le salaire d'un employé augmente chaque année". Cette contrainte n'est pas respectée à la table 1.1 car en 2017 Dupont gagnait 44.000€ et en 2018, il gagnait 28.000€ de salaire annuel. Nous avons l'une des deux lignes de la table qui contient une donnée erronée. Mais c'est sans compter sur le fait que monsieur Dupont a perdu son grade de chef. Du coup la contrainte est peut-être erronée et la contrainte "Le salaire d'un employé augmente chaque année à condition de garder son grade" est correcte. Dans ce cas ci, l'erreur ne vient pas des données mais de la contrainte. Pour cette raison, nous avons besoin de techniques afin de réparer à la fois les données et les contraintes, ce sans perdre trop d'informations tout en évitant d'échouer dans la détection d'erreurs dans les données.

Dans cette thèse de mémoire, nous allons analyser le *modèle de réparation θ-tolérant* comme il a été introduit dans un article scientifique de Shaoxu SONG, Han ZHU et Jianmin WANG[4]. Nous allons faire varier nos contraintes et chacune de nos variations aura un coût que nous devrons déterminer. Le coût de toutes ces variations ne devra pas excéder θ. Le nouvel ensemble de contraintes sera utilisé pour la réparation des données.

Dans un premier temps, nous allons introduire le concept de *denial constraint*, une forme de contraintes d'intégrité qui va nous aider à définir et comprendre le concept du modèle de réparation θ-tolérant. Nous allons également introduire quelques bases de données que nous utiliserons pour illustrer les différentes notions que nous allons aborder. Ensuite, nous allons présenter des techniques pour faire varier nos denial constraint en respectant le modèle de réparation θ-tolérant. Puis, puisque nous resterons critique par rapport au papier scientifique de référence, nous expliquerons en quoi notre démarche est différente et pourquoi nous avons choisi de modifier certaines approches et certains concepts. Enfin, nous terminerons avec une implémentation du modèle θ-tolérant et analyserons ses performances avec quelques exemples.

Chapitre 2

Les contraintes d'intégrité

Dans ce chapitre, nous allons rappeler quelques notions bien connues mais nous allons également introduire de nouveaux concepts. Dans un premier temps, nous allons présenter quelques bases de données que nous utiliserons en tant qu'exemple pour expliquer et illustrer de nombreuses propriétés et définitions. Ces bases de données suivent le modèle relationnel qui a été introduit par E.F. Codd [2]. Ensuite nous allons travailler sur les contraintes d'intégrité et nous allons introduire un nouveau type de contraintes appelées *denial constraint*. Nous allons expliquer plusieurs caractéristiques et propriétés de ces contraintes et expliquer pourquoi nous n'utilisons pas une forme plus conventionnelle de contrainte, comme par exemple les dépendances fonctionnelles.

2.1 Base de données

Dans cette section, nous allons présenter des bases de données que nous allons utiliser comme exemple dans cette thèse de mémoire. Nous utiliserons ces bases de données pour illustrer le modèle de réparation de données θ-tolérant ainsi que d'autres notions que nous définirons.

La première base de données est tirée de l'article principal utilisé dans la bibliographie de cette thèse [4]. Elle représente une table de personnes comprenant le nom de ces personnes, leur date de naissance, leurs numéros de téléphone, leurs revenus, le montant de taxe que ces personnes paient ainsi que l'année à laquelle toutes ces données ont été encodées.

	Nom	Anniversaire	NumTel	Année	Revenu	Taxe
t1	Ayres	8-8-1984	322-573	2007	21k	0
t2	Ayres	5-1-1960	***-389	2007	22k	0
t3	Ayres	5-1-1960	564-389	2007	22k	0
t4	Stanley	13-8-1987	868-701	2007	23k	3k
t5	Stanley	31-7-1983	***-198	2007	24k	0
t6	Stanley	31-7-1983	930-198	2008	24k	0
t7	Dustin	2-12-1985	179-924	2008	25k	0
t8	Dustin	5-9-1980	***-870	2008	100k	21k
t9	Dustin	5-9-1980	824-870	2009	100k	21k
t10	Dustin	9-4-1984	387-215	2009	150k	40k

TABLE 2.1 – Base de données de l'article principal [4]

.

La seconde base de données que nous allons utiliser est inspirée d'une expérience personnelle. Lors d'un stage en entreprise, j'ai pu travailler sur un projet lié à une base de donnée contenant des données erronées. Ces données ne pouvant pas être utilisées en dehors de l'entreprise, nous utiliserons une base de données reprenant l'idée générale. C'est une table appelée "Personne" contenant différentes informations basiques sur des personnes en Belgique [1] .

— **NISS :** Le numéro national de la personne. Un numéro national est unique. En règle général, un NISS est formé de la manière suivante : [1]
  — Il commence avec la date de naissance de la personne dans un format YY-MM-DD. Des exceptions existent pour les étrangers (c'est à dire des personne n'ayant pas la nationalité belge) mais nous n'allons pas considérer ces cas. En effet ces cas peuvent être difficiles à comprendre et ne sont aucunement intéressants pour la suite.
  — Le nombre composé du septième, huitième et neuvième chiffres est pair pour les hommes et impair pour les femmes
  — Le nombre composé des deux derniers chiffres est le resulat de $n \mod 97$ avec n le nombre formé des 9 premiers chiffres
— **Nom :** Nom de famille de la personne.
— **Prénom :** Prénom de la personne.
— **Nai_Date :** Date de naissance de la personne dans le format DD-MM-YYYY.
— **Dec_Date :** Date de décès de la personne dans le format DD-MM-YYYY.
— **Etat_Civil :** État civil courant de la personne, celui ci doit être parmi les suivants : (célibataire, marié, divorcé, décédé, veuf)

1. Les données sont fictives

— **Ville** : La ville où la personne vit.
— **Code_Post** : Le code postal de la ville.
— **Salaire** : Le salaire perçu par la personne en une année.
— **Taxe** : Le montant de taxe payé par la personne en une année.
— **Enfant** : Le nombre d'enfants que la personne a à charge.

	Niss	Nom	Prénom	Nai_Date	Dec_Date	Etat_Civil	Ville	Code_Post	Salaire	Taxe	Enfant
t1	14050250845	Dupont	Jean	14-05-1902	18-05-1962	décédé	Ath	7822	25k	4k	2
t2	08042910402	Brel	Jacques	08-04-1929	09-10-1978	décédé	Schaerbeek	1030	100k	8k	1
t3	45060710204	Merckx	Eddy	07-06-1945	null	décédé	Schaerbeek	1030	125k	9k	2

TABLE 2.2 – La table Personne

.

2.2 Contraintes sur les bases de données

Les bases de données devraient n'accepter que des valeurs qui respectent certaines normes et règles. Ce serait un problème si on pouvait ajouter n'importe quelle valeur à chaque colonne d'une base de données. Pour éviter ce problème, nous avons recours à des règles sur les bases de données. Ces règles sont appelées *contraintes d'intégrité* et fonctionnent de la manière suivante : Si une relation i.e un ensemble de tuples respecte toutes les conditions de ces contraintes alors les données sont acceptables. Si la relation ne respecte pas toutes les conditions alors au moins un tuple de la relation contient au moins une valeur erronée.

Le modèle relationnel des bases de données introduit la notion de *dépendance fonctionnelle* :

Définition 1. Une **dépendance fonctionnelle (DF)** est une expression $X \rightarrow Y$ avec $X, Y \subseteq sort(R)$ et où $sort(R) = \{A_1, A_2, ..., A_n\}$ signifient que pour chaque ensemble de tuples où les attributs de X correspondent, on a les attributs de Y qui correspondent aussi.

En d'autres mots, la contrainte $X \rightarrow Y$ signifie que pour une valeur spécifique de X, il n'y a au plus une valeur possible pour Y. Si la DF est respectée sur la relation R, nous pouvons dire que R satisfait la DF. Prenons quelques exemple sur la table 2.2 :

1. *Un NISS identifie une personne* : En d'autre mot, pour une valeur spécifique du NISS, il n'y a qu'une seule valeur possible pour tout les autres attributs de la table. Cela peut se décrire par la DF suivante : $NISS \rightarrow Nom, Prenom, Nai\_Date, Dec\_Date, Etat\_Civil, Ville, Code\_Post, Salaire, Taxe, Enfant$
2. *Deux personnes avec le même code postal vivent dans la même ville.* : Pour une valeur spécifique de $Code\_Post$ dans notre table il n'y a qu'une valeur possible de $Ville$. Par exemple si la valeur de $Code\_Post$ d'une personne est '7822', la seule valeur possible pour l'attribut $Ville$ est 'Ath'. La dépendance fonctionnelle dans ce cas est $Code\_Post \rightarrow Ville$.

Définition 2. Si pour chaque paire de tuples de la relation R, la DF τ est respectée, nous disons que la relation R *satisfait* τ. Cela ce note $R \models \tau$.

Évidement, certaines bases de données ne contiennent pas qu'une seule contrainte mais plusieurs. Il est important qu'elles soient toutes respectées. Ce qui nous conduit à la définition suivante :

Définition 3. Soit un ensemble Σ de DF sur la relation R. On dit que la relation R satisfait Σ noté $R \models \Sigma$ si pour chaque DF $\tau \in \Sigma$, on a $R \models \tau$

Malheureusement, les dépendances fonctionnelles sont limitées en terme de puissance. En effet, il existe de nombreuses contraintes que nous ne pouvons pas exprimer avec une DF. Par exemple, si nous souhaitons exprimer le fait que *'La date de naissance d'une personne*

doit être antérieure à sa date de décès', nous avons besoin de comparer la Nai_Date et la Dec_Date de la personne et de s'assurer que la date de décès ne soit antérieure à la date de naissance. Les dépendances fonctionnelles ne permettent pas d'utiliser des opérateurs de comparaison, il est donc nécessaire d'exprimer les contraintes d'une autre façon. Pour ce faire, nous allons introduire un nouveau type de contrainte qui répondra bien à nos besoins : les *denial constraints*.

2.3 Les Denial constraints

Dans cette section, nous allons définir ce qu'est une denial constraint. Nous allons aussi expliquer son utilisation dans les bases de données et nous allons également lister et expliquer plusieurs propriétés que peuvent avoir ces contraintes. Commençons d'abord par définir la denial constraint

Définition 4. Considérons un ensemble S fini d'*attribut*. Une *denial constraint (DC)* sur l'ensemble S est une fonction partielle de S vers l'ensemble des parties(aussi appelé ensemble puissance) de $\{<, =, >\}$ noté OP. Nous utiliserons la lettre grecque φ pour représenter une DC. Chaque élément de OP se nomme *opérateur*.

Définition 5. Soit $(\mathbf{dom}, \leq)$ un domaine totalement ordonné contenant au moins deux éléments distincts. Un *tuple sur* S est une fonction totale de S à $\mathbf{dom}$. Une *relation sur S* est un ensemble fini de tuples sur S.

Par définition l'ensemble puissance d'un ensemble S noté $\mathcal{P}(S)$ est l'ensemble de tous les sous-ensembles de S. Cela inclut l'ensemble S lui même mais aussi l'ensemble vide $\emptyset$. Par exemple OP (l'ensemble des parties de $\{<, =, >\}$) est $\mathcal{P}(OP)$ = $\{\emptyset, \{<\}, \{=\}, \{>\}, \{<, =\}, \{=, >\}, \{<, >\}, \{<, =, >\}\}$. Nous utiliserons la lettre grecque ϕ ou θ pour représenter un opérateur. Il existe différentes abréviations pour les éléments de OP, ceux-ci étant répertoriés dans la table 2.3. Nous avons eu besoin d'introduire 2 nouveaux opérateur $\top$ et $\bot$, chacun étant l'abréviation pour l'ensemble $\{<, =, >\}$ et $\emptyset$ respectivement. Nous les définissons comme tel :

Définition 6. $\forall a, b \in \mathbf{dom}$,nous avons $a \bot b$ est toujours faux et $a \top b$ est toujours vrai.

Expliquons maintenant la sémantique qui se cache derrière la denial constraint.

Définition 7. On dit qu'une relation I sur S *satisfait* la DC φ, noté $I \models \varphi$ si il **n'existe pas** deux tuples $s, t \in I$ tel que pour chaque attribut A dans le domaine de φ, nous avons $s(A)\theta t(A)$ avec $\theta = \varphi(A)$

Notons que les tuples s et t ne doivent pas forcément être distinct. La relation I peut être vide. Une relation vide satisfait toutes les DCs.

θ	Abréviation	$\bar{\theta}$	$\hat{\theta}$
$\emptyset$	$\bot$	$\top$	$\bot$
$\{<\}$	$<$	$\geq$	$>$
$\{=\}$	$=$	$\neq$	$=$
$\{>\}$	$>$	$\leq$	$<$
$\{<,=\}$	$\leq$	$>$	$\geq$
$\{<,>\}$	$\neq$	$=$	$\neq$
$\{>,=\}$	$\geq$	$<$	$\leq$
$\{<,=,>\}$	$\top$	$\bot$	$\top$

TABLE 2.3 – Element de OP, l'ensemble des parties de $\{<,=,>\}$

Exemple 1. Prenons un exemple sur la table 2.1, nous avons $S = \{Nom, Anniversaire, NumTel, Annee, Revenu, Taxe\}$ Une DC pour S est $\varphi = \{(Nom, =), (Anniversaire, =), (NumTel, \neq), (Annee, \top), (Revenu, \top), (Taxe, \top)\}$. Celle-ci est satisfaite par la relation I si il n'existe pas deux tuples $s, t \in I$ tel que $s(Nom) = t(Nom) \wedge s(Anniversaire) = t(Anniversaire) \wedge s(NumTel) \neq t(NumTel) \wedge s(Annee) \top y(Annee) \wedge s(Revenu) \top t(Revenu) \wedge s(Taxe) \top t(Taxe)$.

Lorsqu'une relation I sur S ne satisfait pas une DC φ, on dit que I *viole* φ que l'on note $I \not\models \varphi$. Chaque élément de φ est appelé un prédicat. Pour chaque prédicat (A, θ), on appelle A l'attribut du prédicat, et θ l'opérateur du prédicat. Soit I une relation sur S et φ une DC. Dès lors on peut dire que $I \models \varphi$ si pour tout $s, t \in I$ au moins un des prédicats est faux pour $\{s, t\}$ i.e pour au moins un prédicat (A, θ_A) on a $s.A\theta_A t.A$ qui est faux. Si un prédicat P a pour opérateur $\top$ alors P sera toujours vrai pour tout $t, s \in I$. Dès lors à l'avenir, nous ne noterons plus les prédicats ayant $\top$ pour opérateur par facilité syntaxique. L'exemple précédent s'écrira désormais $\varphi = \{(Nom, =), (Anniversaire, =, ?(NumTel, \neq)\}$. Si un prédicat a pour opérateur $\bot$, il sera toujours faux. Dès lors $I \models \varphi$. Notons que la relation vide est la seule relation qui satisfait la DC $\varphi = \{(A_1, \top), (A_2, \top), ...(A_n, \top)\} \equiv \{\}$.

Si nous prenons l'instance I comme étant la table 2.1, nous avons $I \not\models \varphi$. En effet prenons $s = t_2$ et $t = t_3$ nous avons bien $t_2(Nom) = t_3(Nom) \wedge t_2(Anniversaire) = t_2(Anniversaire) \wedge y_2(NumTel) \neq t_3(NumTel)$. On dit que $\{t_2, t_3\}$ *viole* la contrainte φ

Pour chaque opérateur dans OP nous pouvons définir son inverse et sa réciproque. Les valeurs de l'inverse et de la réciproque de chaque élément de OP se trouve également à la table 2.3.

Définition 8. Soit θ un élément de OP. L'*inverse* de ϕ noté $\overline{\theta}$ est égal à $\{<,=,>\} \setminus \theta$.

Définition 9. Soit $\rho := \{(<,>), (=,=), (>,<)\}$ une permutation de $\{<,=,>\}$. Pour tout $\theta \in OP$, nous pouvons définir la *réciproque* de θ comme étant $\hat{\theta} := \{\rho(e) | e \in \theta\}$

Notons que pour une DC φ sur S, nous pouvons définir $\hat{\varphi}$ une DC S tel que $\hat{\varphi}$ et φ ont le même domaine et pour tout $A \in S$, si $\varphi(A) = \theta$ alors $\hat{\varphi} = \hat{\theta}$. Notons également que nous avons $\hat{\hat{\varphi}} = \varphi$.

Une DC peut être *sur-simplifiée* ce qui veut dire qu'une donnée correcte peut être considérée comme une violation. Prenons un exemple sur la table 2.1 avec la denial constraint suivante :

$$\varphi_2 = (Nom, =)(NumTel, \neq)$$

La contrainte est satisfaite s'il n'existe pas deux tuples avec le même nom mais des numéros de téléphone différents. On ne souhaite pas imposer une telle contrainte, car il est possible que deux personnes différentes aient le même noms et des numéros de téléphone différents. Dans notre relation, le nom seul ne suffit pas à identifier si deux personne sont identiques. Prenons par exemples t_1 et t_2, ils ne satisfont pas φ_2. Si l'on regarde de plus près, on peut facilement comprendre qu'il s'agit de deux personnes différentes. Ces deux personnes n'ont pas le même age i.e elles ont une date d'$Anniversaire$ différent. Si nous souhaitons améliorer la précision de la contrainte et éviter que $\{t_1, t_2\}$ soit considéré comme une violation, nous avons besoin de regarder l'attribut $Anniversaire$. Une meilleure DC serait :

$$\varphi_2' = (Nom, =), (Anniversaire, =), (NumTel, \neq)$$

Une DC peut être également *sur-raffiné* ce qui entraine qu'une donnée erronée peut ne pas être détectée par la DC.. Prenons un exemple sur la table 2.1 avec la denial constraint suivante :

$$\varphi_2' = (Nom, =), (Anniversaire, =), (NumTel, \neq), (Annee, =)$$

Dans ce cas, l'information $Annee$ n'est pas utile pour distinguer deux personnes différentes. Dans la table, l'attribut année correspond à l'année où les autres attributs ont été encodés. Une même personne peut être encodée deux fois à deux années différentes. Avec cette DC, on ne reconnait pas $\{t_5, t_8\}$ comme étant une violation.

2.4 Quelques définitions et propriétés sur les DC

Dans cette sous-section, nous allons définir quelques notions et propriétés sur les DC qui nous serviront dans les chapitres qui suivront.

2.4.1 Satisfiabilité

Définition 10. Soit φ DC sur S. On dit que φ est *satisfiable* si elle peut être satisfaite par une relation non vide sur S.

Si φ n'est pas satisfiable, nous dirons qu'elle est *insatisfiable*

Il est intéressant de savoir à l'avance si une une denial constraint est satisfiable ou pas. Le lemme suivant nous permet de détecter les DC qui ne sont pas satisfiables.

Lemme 1. Soit φ une denial constraint sur S, alors φ est satisfiable si et seulement si il existe un prédicat $P = (A, \theta) \in \varphi$ tel que θ ne contient pas $=$, i.e $\theta_i \notin \{\{=\}, \{<, =\}, \{=, >\}, \{<, =, >\}, \}$

Démonstration.
$\boxed{\Longrightarrow}$ Supposons que pour tout $P = (A, \theta) \in \varphi$, θ contient $=$. Alors pour chaque tuple $s \in I$ avec I relation sur S, pour chaque $P \in \varphi$ on a $s(A)\theta s(A)$. Il s'ensuit que toute relation non vide ne satisfait pas φ
$\boxed{\Longleftarrow}$ Supposons $B \in S$ tel que B ne contient pas $=$. Alors pour chaque tuple s sur S, nous avons que $s(B)\ \theta s(B)$ avec $\theta = \varphi(B)$ faux. Il s'ensuit que n'importe quelle relation avec exactement un tuple satisfait φ.

□

2.4.2 Implication logique

Définition 11. Soit φ_1, φ_2 deux DC sur S. On dit que φ_1 *implique (logiquement)* φ_2, que l'on note $\varphi_1 \models \varphi_2$, si pour chaque relation I sur S, si $I \models \varphi_2$ alors on a $I \models \varphi_1$. On dira aussi que φ_2 est *plus faible* que φ_1 ou bien que φ_1 est *plus fort* que φ_2

Exemple 2. Soit $S = \{A, B\}$. Soit $\varphi_1 = \{(A, \leq), (B, \neq)\}$ et $\varphi_2 = \{(A, <), (B, >)\}$. Alors φ_1 implique φ_2. En effet, soit I une relation qui satisfait φ_1. Alors pour tout tuples $s, t \in I$, on a $s(A) > t(A)$ ou bien $s(B) = t(B)$ (ou éventuellement les deux en même temps). Il s'ensuit que pour tout tuples $s, t \in I$, on a $s(A) \geq t(A)$ ou bien $s(B) \leq t(B)$ (ou les deux en même temps). Dès lors, I ne contient pas deux tuples s, t tel que $s(A) < t(A)$ et $s(B) > t(B)$. On a donc I qui satisfait φ_2. D'un autre côté, φ_2 n'implique pas φ_1. En effet, considérons la relation I suivante.

I	A	B
	1	2
	1	3

Dès lors, nous avons $I \models \varphi_2$, mais $I \not\models \varphi_1$.

Lemme 2. Soit φ_1 et φ_2 deux denial constraints sur S. Si $\varphi_2(A) \subseteq \varphi_1(A)$ pour tout $A \in S$, alors $\varphi_1 \models \varphi_2$.

Démonstration. Supposons que $\varphi_2(A) \subseteq \varphi_1(A)$ pour tout $A \in S$. Soit I une relation sur S tel que $I \models \varphi_1$. Nous avons besoin de démontrer que $I \models \varphi_2$. Soit $s, t \in I$. Puisque $I \models \varphi_1$, nous pouvons supposer l'existence d'un $A \in S$ tel que $s(A)\ \theta\ t(A)$ est faux, avec $\theta = \varphi(A)$. Puisque $\varphi_2(A) \subseteq \varphi_1(A)$, alors nous aurons $s(A)\ \theta'\ t(A)$ est faux, avec $\theta' = \varphi_2(A)$. □

Exemple 3. $\{(A,<),(B,\leq)\} \models \{(A,<),(B,<)\}$. car $\{<\} \subseteq \{<,=\} \equiv \leq$. La DC de gauche signifie que deux tuples $s,t \in I$ satisfont la DC si la contrainte "si $s(A) < t(A)$ alors $s(B) > t(B)$" est satisfaite. La DC de droite signifie que deux tuples $s,t \in I$ satisfont la DC si la contrainte "$sis(A) < t(A)$ alors $s(B) >= t(B)$ est satisfaite.

La réciproque d'une DC est une implication logique par le Lemme suivant :

Lemme 3. Pour toute DC φ sur S, nous avons $\varphi \models \hat{\varphi}$

Démonstration. Soit une DC φ sur S. Soit $D \subseteq S$ le domaine de φ. Pour tout $A \in D$, nous pouvons définir $\theta_A := \varphi(A)$. Soit une relation I sur S tel que $I \models \varphi$. Supposons par l'absurde que $I \not\models \hat{\varphi}$. Alors, nous pouvons supposer que pour tout tuples s,t tel que pour tout $A \in D$, nous avons $s(A)\ \hat{\theta_A}\ t(A)$. Il est trivial de voir que pour tout A dans le domaine de φ, nous avons $t(A)\ \theta_A\ s(A)$, et par conséquent nous avons $I \not\models \varphi$, ce qui est une contradiction. Nous pouvons conclure par contradiction que $I \models \hat{\varphi}$ □

Exemple 4. $\{(A,=),(B,>)\} \models \{(A,=),(B,<)\}$. Notons que nous avons également $\{(A,=),(B,<)\} \models \{(A,=),(B,>)\}$

En effet, par le lemme 3 nous avons que $\varphi \models \hat{\varphi}$. Mais nous avons aussi $\hat{\varphi} \models \varphi$ car $\hat{\hat{\varphi}} = \varphi$.

2.4.2.1 Implication logique d'ensemble de DC

Définition 12. Soit Σ un ensemble de DCs sur S. Soit I une relation sur S. On écrit $I \models \Sigma$ si $I \models \varphi'$ pour chaque $\varphi' \in \Sigma$.

Soit φ une DC sur S. Nous écrivons $\Sigma \models \varphi$ si pour chaque relation I sur S, nous avons $I \models \Sigma$ implique $I \models \varphi$. Nous allons montrer grâce à ça qu'il est possible de remplacer plusieurs DCs par une seule. En effet si un sous ensemble Σ' de Σ est tel que $\Sigma' \models \varphi$, si $\Sigma \models I$ alors $(\Sigma \backslash \Sigma') \cup \varphi \models I$. Parcourons plusieurs exemples :

Exemple 5. Soit $S = \{A,B,C\}$. Soit

$$\begin{aligned} \varphi_1 &= \{(A,<),(B,\geq)\} \\ \varphi_2 &= \{(A,>),(C,\leq)\} \\ \varphi &= \{(A,\neq),(B,\geq),(C,\leq)\} \end{aligned}$$

Nous allons montrer $\{\varphi_1,\varphi_2\} \models \varphi$. Soit une relation I sur S telle que $I \models \varphi_1$ (*) et $I \models \varphi_2$ (**). Supposons par l'absurde que $I \not\models \varphi$. Alors, nous pouvons supposer qu'il existe deux tuple $s,t \in I$ tels que nous avons $s(A) \neq t(A)$ et $s(B) \geq t(B)$ et $s(C) \leq t(C)$. Puisque $s(A) \neq t(A)$ et $(\mathbf{dom},\leq)$ est un ordre total, deux cas peuvent se présenter :

Cas $s(A) < t(A)$**.** Puisque $s(B) \geq t(B)$, nous avons $I \not\models \varphi_1$, ce qui est contradictoire avec (*).

Cas $s(A) > t(A)$**.** Puisque $s(C) \leq t(C)$, nous avons $I \not\models \varphi_2$, ce qui est contradictoire avec (**).

Nous pouvons donc conclure que nous avons $I \models \varphi$.

Exemple 6. Soit $S = \{A, B\}$. Soit

$$\begin{aligned} \varphi_1 &= \{(A, <), (B, \geq)\} \\ \varphi_2 &= \{(A, >), (B, \leq)\} \\ \varphi &= \{(A, \neq), (B, =)\} \end{aligned}$$

Nous allons montrer que $\{\varphi_1, \varphi_2\} \models \varphi$. Soit une relation I sur S telle que nous avons $I \models \varphi_1$ (*) et $I \models \varphi_2$ (**). Supposons par l'absurde que $I \not\models \varphi$. Dès lors nous pouvons dire qu'il existe deux tuples $s, t \in I$ tels que $s(A) \neq t(A)$ et $s(B) = t(B)$. Puisque $s(A) \neq t(A)$ et $(\mathbf{dom}, \leq)$ est un ordre total, deux cas de figure peuvent se produire :

Cas $s(A) < t(A)$**.** Puisque $s(B) \geq t(B)$, nous avons$I \not\models \varphi_1$, ce qui est une contradiction avec (*).

Case $s(A) > t(A)$**.** Puisque $s(B) \leq t(B)$, nous avons $I \not\models \varphi_2$, ce qui est une contradiction avec (**).

We conclude by contradiction that $I \models \varphi$.

Les exemples précédents, nous suggère qu'il existe le Lemme suivant :

Lemme 4. Soit $A \in S$ et soit $\Sigma = \{\varphi_1, \varphi_2, ..., \varphi_n\}$ un ensemble de n DCs sur S et φ une DC sur S tel que :

1. $\varphi(A) = \varphi_1(A) \cup \varphi_2(A) \cup ... \cup \varphi_n(A)$;
2. pour tout $B \in S \backslash \{A\}$, $\varphi(B) = \varphi_1(B) \cap \varphi_2(B) \cap ... \cap \varphi_n(B)$

alors nous avons $\{\varphi_1, \varphi_2, ..., \varphi_n\} \models \varphi$

Démonstration.
Soit I une relation sur S tel que $I \models \varphi_1$,$I \models \varphi_2$,...,$I \models \varphi_n$. Supposons que $I \not\models \varphi$ et que les conditions 1 et 2 du Lemme sont respecté. Soit $A \in S$ et s, t deux tuples de I. Alors nous pouvons dire par la définition de $I \not\models \varphi$ que nous avons :

— $s.A\theta_A t.A$ avec $\theta_A = \varphi(A)$

— pour tout $B \in S \backslash \{A\}$ $s.A\theta_B t.A$ avec $\theta_B = \varphi_B$

Puisque nous avons la première condition du Lemme, nous avons que $\varphi(A) = \varphi_1(A) \cup \varphi_2(A) \cup ... \cup \varphi_n(A)$ et donc nous devons avoir $s.A\varphi_i(A)t.A$ avec $\varphi_i(A)$ un sous-ensemble de $\{<, =, >\}$ pour tout $\varphi_i \in \Sigma = \{\varphi_1, \varphi_2, ..., \varphi_n\}$

Puisque nous avons la seconde condition du Lemme, nous avons que pour tout $B \in S \backslash \{A\}$, on a $\varphi(B) = \varphi_1(B) \cap \varphi_2(B) \cap ... \cap \varphi_n(B)$ et donc nous avons $s.A\varphi_i(B)t.A$ avec $\varphi_i(B)$ un sous-ensemble de $\{<, =, >\}$ pour tout $\varphi_i \in \Sigma = \{\varphi_1, \varphi_2, ..., \varphi_n\}$

n cas peuvent apparaitre :

Cas $s(A)\ \varphi_1(A)t(A)$**.** Puisque $s(B)\ \varphi_1(B)t(B)$ pour tout $B \in S \setminus \{A\}$, nous avons $I \not\models \varphi_1$, une contradiction.

...

Cas $s(A)\ \varphi_i(A)t(A)$**.** Puisque $s(B)\ \varphi_i(B)t(B)$ pour tout $B \in S \setminus \{A\}$, nous avons $I \not\models \varphi_i$, une contradiction.

Dès lors par ces contradictions nous avons $I \models \varphi$ □

Par le lemme 4, nous pouvons simplifier nos ensemble de DC. En effet si nous avons un ensemble de DC $\Sigma = \{\varphi_1, \varphi_2, ..., \varphi_n\}$ et $\Sigma' = \{\varphi\}$ tel qu'ils respectent le lemme 4 alors nous pouvons remplacer Σ par Σ'.

Notons que l'exemple 5 est un cas particulier de l'exemple6. En effet, dans l'exemple 5 nous pouvons écrire $\varphi_1 = \{(A, <), (B, \geq), (C, \top)\}$ et $\varphi_2 = \{(A, <), (B, \top), (C, \leq)\}$.

Corollaire 1. Soit φ_1, φ_2 deux DC sur S. Soit $A \in S$ tel que $\varphi_1(A) \cup \varphi_2(A) = \top$. Soit φ une DC sur $S \backslash \{A\}$ tel que pour tout $B \in S \backslash \{A\}$, nous avons $\varphi(B) = \varphi_1(B) \cap \varphi_2(B)$. Alors nous avons $\{\varphi_1, \varphi_2\} \models \varphi$

2.4.3 Trivialité

Une DC peut être inutile et toujours vraie. De telles DC ne devraient pas être présentes dans la base de données puisqu'elles ne détecteront jamais aucune violation. Dans ce cas, on dira que la DC est *triviale*.

Définition 13. Une DC φ est dite *triviale* si pour tout I sur S, on a $I \models \varphi$

Lemme 5. Soit φ une DC sur S. alors φ est triviale si et seulement si nous avons $\theta = \perp$ pour un prédicat $P = (A, \theta) \in \varphi$.

Démonstration.
$\boxed{\Longrightarrow}$ Supposons que pour chaque prédicat $P = (A, \theta) \in \varphi$ avec $\theta \neq \perp$. Soit s, t deux tuples tel que pour chaque $P = (B, \theta)$, nous avons $s(B)\ \theta\ t(B)$ vrai. Puisque $\theta \neq \perp$ et que **dom** contient au moins deux éléments, s, t peuvent être construit. Dès lors $\{s, t\}$ ne satisfont pas φ puisque φ n'est pas triviale.
$\boxed{\Longleftarrow}$ Supposons qu'il existe un prédicat $P = (A, \theta) \in \varphi$ tel que $\theta = \perp$. Puisque $s(A) \perp t(A)$ est faux pour tout tuples s, t sur S, aucune relation ne peut contenir deux tuples s, t tel que $s(A) \perp t(A)$ est vrai. □

Chapitre 3

Data Repairing

Les erreurs sont fréquentes dans les bases de données et ces anomalies nuisent à la fiabilité de certaines applications qui les utilisent. Il existe des méthodes dont le but est de détecter ces erreurs mais ces méthodes ne les réparent pas. A la place, les applications pourront filtrer les données et ignorer les erreurs détectées mais malgré ce filtrage, les applications peuvent toujours être non fiables [3]. Au lieu de simplement détecter les erreurs et les filtrer, il est préférable de réparer les données erronées.

Dans le chapitre précédent, nous avons vu comment détecter des erreurs au moyen de denial constraints. Nous avons aussi discuté brièvement de la sur-simplification ou du sur-raffinement de ces DC. Nous allons maintenant aborder la réparation des données. Nous aborderons la réparation des DCs dans la prochain chapitre.

Le but d'une réparation de données est de trouver une nouvelle relation I' qui est une modification d'une relation I de S. Dans I' il n'y a pas de données erronées. Cela nous amène à définir ce qu'est une base de données sans erreurs. Une base de données sans erreurs est une base de données dont toutes les contraintes sont satisfaites, c'est à dire :

Définition 14. Soit Σ un ensemble de DC sur S. Soit une relation I sur S. On dit que I *satisfait* Σ noté $I \models \Sigma$ si pour chaque DC φ avec $\varphi \in \Sigma$, nous avons $I \models \varphi$

Définition 15. Soit Σ un ensemble de DC sur S. Soit une relation I sur S. Une *réparation de* I est une fonction f de domaine I qui attribue à chaque tuple t de I un nouveau tuple $f(t)$. L'ensemble d'arrivée de f est noté $I' = f(I)$. On a donc $I' \models \Sigma$

Donc, lorsque nous parlons de réparation de données, nous cherchons à trouver $f(I) = I'$ avec I la relation à réparer tel que $I' \models \Sigma$ c'est à dire une nouvelle relation où toutes les violations dans l'ensemble de contraintes Σ sont éliminées. Nous allons considérer que lors d'une réparation, nous ne supprimons pas de tuples mais que nous les modifions. En effet, chaque ligne de la table est susceptible de contenir des précieuses informations. Chaque ligne supprimée diminuerait la quantité d'informations de la base de données.

Dans de nombreux cas, seules quelques colonnes de la table contiennent quelques erreurs et les autres colonnes ne présentent aucun problème. Nous n'ajoutons pas de tuples à la relation. Lorsqu'un tuple t est réparé cela peut conduire à avoir deux tuples identiques.

Exemple 7. Soit la relation I suivante :

$$I \left| \begin{array}{cc} A & B \\ \hline a & b \\ a & c \end{array} \right. \qquad\qquad I' \left| \begin{array}{cc} A & B \\ \hline a & b \\ a & b \end{array} \right.$$

et la DC $\varphi = \{(A, =), (B, \neq)\}$. Nous avons $I' \models \varphi$ et $I \not\models \varphi$. $I' = f(I)$ est une réparation de I, le tuple (a, c) a été modifié en (a, b)

Notons ici la présence de deux lignes identiques dans l'instance I'. Ce n'est pas une pratique courante dans les bases de données, nous sommes donc dans ce cas ci autorisé à supprimer une des lignes ce qui nous donne :

$$I' \left| \begin{array}{cc} A & B \\ \hline a & b \end{array} \right.$$

La réparation de données suit le principe du changement minimum : La nouvelle relation I' doit minimiser le coût de réparation de données défini comme étant :

Définition 16. Soit I une relation sur S et $f(I)$ une fonction réparation de I, alors le coût de réparation est le suivant :

$$\Delta(I, f(I)) = \sum_{t \in I, A \in S} w(t.A).dist(t.A, f(t).A)$$

où :

— $dist(t.A, f(t).A)$ est la distance entre la valeur $t.A$ et sa réparation $f(t).A$.

— $w(t.A)$ est un poids sur la valeur $t.A$.

Dans le coût, nous avons un poids $w(t.A)$ pour un attribut $A \in S$ et un tuple $t \in I$. Ce poids correspond à la confiance que l'on accorde en la valeur de t pour l'attribut A. On peut grâce à cette valeur influencer la réparation de données pour privilégier la réparation d'une valeur plutôt qu'une autre. Pour pouvoir assigner un poids $w(t.A)$, il faut avoir une bonne connaissance du contexte de la base de données d'origine. Il est courant d'avoir la même valeur pour $w(t.A)$ et $w(s.A)$ avec pour tout $s, t \in I$ car en général on a connaissance de la confiance pour un attribut en particulier pas pour chaque valeur du tuple. On remplace donc $w(t.A)$ par $w(A)$dans notre formule. Par exemple, pour la table 2.2, nous pouvons supposer qu'une valeur pour l'attribut *Enfant* est plus susceptible d'être précise que la valeur pour l'attribut *Salaire* ou *Taxe*. Dès lors $w(Enfant)$ sera plus grand que $w(Salaire)$ et $w(Taxe)$. Lorsqu'on manque de connaissances sur la base de données, on fixe le même poids à chaque attribut. Nous supprimons donc le poids $w(t.A)$ dans notre

formule.

Le coût de réparation peut être le nombre de valeurs de I que l'on a changées si nous décidons que :

$$dist(t.A, f(t).A) = \begin{cases} 1 \; si \; t.A \neq f(t).A \; (La \; valeur \; a \; changé) \\ 0 \; sinon \; (aucun \; changement \; n'est \; fait) \end{cases}$$

Nous pouvons aussi décider que la distance est égale à la différence entre les deux valeurs dans le cas d'un attribut numérique. Pour un attribut de type chaine de caractère, nous pouvons utiliser la distance d'édition [1].

Pour réparer une donnée, nous devons remplacer sa valeur par une autre plus adéquate. Mais quelle valeur choisir ? Dans l'article de référence principal [4], si la valeur $t.A$ est erronée, les auteurs essayent de trouver une valeur pour $f(t).A$ qui soit dans le domaine actif $\mathbf{dom}(A, I)$ et qui respecte les contraintes. Le domaine actif $\mathbf{dom}(A, I)$ est l'ensemble de toues les valeurs de A dans I. Si ce n'est pas possible ils attribuent une *variable fraiche* fv à $t.A$.

Définition 17. Une *variable fraiche* fv est une valeur qui ne satisfait aucun prédicat c'est à dire : soit φ une DC sur S et I une relation sur S. Une variable fraiche fv pour $A \in S$ est une valeur tel que $fv \notin \mathbf{dom}(A, I)$ le prédicat $(A, \theta_A) \in \varphi$ est toujours faux quelque soit l'opérateur θ_A donc pour tout tuple s, t avec $t \in I$, si $t.A = fv$ alors $s, t \models \varphi$.

L'ajout des variables fraiches a un impact très important sur la théorie vue lors du chapitre précédent. Premièrement nous savons que pour tout tuples $s, t \in I$, le prédicat $s.A\theta t.A$ est toujours vrai si $\theta = \top$. Si $s.A$ ou $t.A$ est une variable fraiche le prédicat devient faux malgré que $\theta = \top$. Ceci nous force à redéfinir plus précisément l'opérateur $\top$ lorsque des variables fraiches étendent le domaine $\mathbf{dom}$:

Définition 18. Soit I une relation sur S, pour tout tuples $s, t \in I$, si $s.A, t.A \in \mathbf{dom}(A, I)$ pour $A \in S$, alors $s.A \top t.A$ est toujours vrai.

Autre impact important est que nous avons $fv = fv$ faux ainsi que $fv \neq fv$ faux. De manière générale, $fv\theta fv$ et $fv\overline{\theta}fv$ sont tous les deux faux. Ce comportement ne se retrouve pas avec des valeurs de $\mathbf{dom}$. En effet par la définition de $\overline{\theta}$, pour tout tuples $s, t \in I$ et un attribut $A \in S$, si $s.A, t.A \in \mathbf{dom}(A, I)$, $s.A\theta t.A$ et $s.A\overline{\theta}t.A$ ne peuvent être vraie ou faux en même temps.

Nous allons procéder de manière un peu différente par rapport à l'article de référence [4]. Nous allons automatiquement attribuer une nouvelle valeur fv pour chaque donnée

1. Le nombre minimum d'opération nécessaire pour transformer la chaine de caractère initiale en la chaine de caractère cible

erronée. Par la suite, nous allons essayer de trouver une valeur adéquate pour remplacer la valeur fraiche fv si possible. Une des motivations derrière ce changement vient du fait qu'après une réparation, certaines valeurs du domaine n'y figurent plus (ces valeurs étaient erronées). De plus il n'est pas pertinent de croire que si une valeur du domaine peut réparer une valeur erronée, alors elle est forcément la bonne valeur. Une valeur en dehors du domaine peut tout aussi bien convenir.

Si nous attribuons la variable fraiche $fv_{t.A}$ pour l'attribut A d'un tuple t et la valeur fraiche $fv_{s.A}$ pour l'attribut A d'un tuple s, ces deux variables peuvent être différentes ou identiques et ce pour tout $s, t \in I$. Par défaut, nous considérons dans un premier temps que toutes les variables fraiches sont différentes.

Exemple 8. Prenons un exemple pour relation I la table 2.1. Supposons que notre denial constraint est la suivante :

$$\varphi = \{(Revenu, >), (Taxe, \leq)\}$$

nous pouvons interpréter cette relation comme étant : si $t.Revenu > s.Revenu$ alors $t.Taxe > s.Taxe$. En d'autres termes, on suppose par cette contrainte que si une première personne perçoit un revenu annuel plus élevé qu'une seconde personne, alors la première personne doit payer un montant de taxe annuelle plus élevé. Nous avons $\{t_2, t_1\} \not\models \varphi$ parce que $t_2.Revenu > t_1.Revenu$ et $t_2.Taxe \leq t_1.Taxe$. Nous avons également $\{t_3, t_1\} \not\models \varphi, \{t_5, t_1\} \not\models \varphi, ...$ Toutes les violations de φ peuvent être trouvé à la figure 3.1. Une réparation I' de I pourrait être la table 3.1.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
Taxe	0	fv_1	fv_2	3k	fv_3	fv_4	fv_5	21k	21k	40k

TABLE 3.1 – Exemple de réparation I' pour l'attribut Taxe de φ.

Nous avons choisi fv_1 comme variable fraiche pour t_2. Bien que ce soit une variable fraiche, nous savons plusieurs choses à son propos. En effet nous savons les choses suivantes :

1. $I(t_1.Taxe) = 0$ donc $I'(t_2.Taxe) > 0$, i.e $fv_1 > 0$ car $I(t_1.Revenu) < I(t_2.Revenu)$
2. $I(t_4.Taxe) = 3$ donc $I'(t_2.Taxe) < 3k$, i.e $fv_1 < 3k$ car $I(t_2.Revenu) < I(t_4.Revenu)$

Nous avons assigné une variable fraiche fv_1 comme valeur pour $t_2.Taxe$ mais nous savons quand même que $0 < fv_1 < 3k$ i.e $fv_1 \in]0, 3k[$ grâce à 1 et 2. Nous pouvons utiliser la même logique pour connaitre les valeurs possible pour les autres variables fraiches. Nous pouvons dire que $fv_2 = fv_1$, $3k < fv_3 < 21k$, $fv_4 = fv_3$ et $fv_3 < fv_5 < 21k$.

/	t_β									
t_α	/	1	2	3	4	5	6	7	8	9	10
	1	/	V	V	V	V	V	V	V	V	V
	2	F	/	V	V	V	V	V	V	V	V
	3	F	V	/	V	V	V	V	V	V	V
	4	V	V	V	/	V	V	V	V	V	V
	5	F	F	F	F	/	V	V	V	V	V
	6	F	F	F	F	V	/	V	V	V	V
	7	F	F	F	F	F	F	/	V	V	V
	8	V	V	V	V	V	V	V	/	V	V
	9	V	V	V	V	V	V	V	V	/	V
	10	V	V	V	V	V	V	V	V	V	V

FIGURE 3.1 – Toutes les violations pour φ

Maintenant que nous avons identifié les valeurs erronées ainsi qu'un ensemble de valeurs pour remplacer les variables fraiches, il reste à savoir quelle valeur finale on peut prendre. Pour cela nous avons plusieurs solutions possibles.

— Prendre une valeur de $\mathbf{dom}(A, I)$. C'est la solution envisagée dans le principal article de référence que nous utilisons. Il ne sera pas toujours possible de prendre une valeur dans le domaine, par exemple, si on prend I la relation de la table 2.1 il n'y a pas de valeur dans le $dom(Taxe, I) = \{0, 3k, 21k, 40k\}$ qui puisse satisfaire la condition fv_1. Dès lors, dans le cas où aucune valeur de dom(A) n'est attribuable à la variable fraiche, celle-ci est conservée dans la base de données. Cette pratique n'est pas la plus logique puisque même si nous avions une valeur dans $\mathbf{dom}(Taxe, I)$ qui puisse satisfaire fv_1, il y a plusieurs valeurs en dehors de $\mathbf{dom}(Taxe, I)$ qui sont tout aussi correctes.

— Prendre une valeur aléatoire mais respectant les conditions sur la variable fraiche. C'est une très mauvaise idée puisque nous avons une chance de s'éloigner de la vraie valeur. Cela peut impacter énormément l'ajout de tuples dans la relation après que la réparation soit effectuée. Ces nouveaux tuples malgré qu'ils soient corrects pourraient être perçus comme contenant des données erronées.

— Garder les variables fraiches dans la base de données tout en conservant les informations que l'on connait à propos de celles-ci. Si nous n'avons qu'une seule valeur possible alors nous privilégions cette valeur à fv. Cette solution est celle qui respectera le mieux l'intégrité des données. Le seul problème qu'apporte cette solution est que de nombreuses applications ne pourront plus fonctionner correctement avec des variables fraiches. De nombreux SGBD ne permettent pas de stocker ces variables. Les informations que l'on connait sur elles peuvent changer au fur et à mesure que

la relation se remplit.

Exemple 9. Nous pouvons calculer le coût de réparation pour la relation de la table 2.1 en considérant les distances suivantes :

$$\forall a, b \in dom(A, I)\ avec\ a \neq b, fv \notin \mathbf{dom}(A, I). \begin{cases} dist(a, a) = 0 \\ dist(a, b) = 1 \\ dist(a, fv) = 1.5 \\ dist(fv, fv) = 1.5 \\ dist(fv, b) = 1 \end{cases}$$

Lorsqu'on ne change pas la valeur, la distance est bien évidement égale à zéro. $dist(a, fv)$ soit être supérieure à $dist(a, b)$ pour privilégier les valeurs aux variables fraiches. $dist(fv, fv)$ représente le fait qu'on avait déjà une variable fraiche venant d'une réparation antérieure, et qu'on garde une variable fraiche. $dist(fv, b)$ représente le changement d'une variable fraiche venant d'une réparation antérieure par une valeur précise. Cela peut arriver lorsque l'on possède de plus amples informations sur la valeur fraiche,par exemple grâce à de nouveaux tuples dans la relation. Nous avons besoin que $dist(fv, fv) > dist(fv, b)$ pour favoriser la correction par une vraie valeur quand c'est possible. Dans l'exemple précédent, avec les valeurs susmentionnées, nous pouvons calculer un coût de réparation $\Delta(I, I')$ =$5 * dist(a, a) + 5 * dist(a, fv)$ = 7, 5.

3.0.0.1 Graphe de conflits

Maintenant, nous allons introduire le *graphe de conflits* qui peut représenter les violations dans une relation I sur S. Dans un premiers temps, nous avons besoins de trouver l'ensemble de toutes les violations et ensuite obtenir nos deux bornes grâce à cet ensemble. On définition l'*ensemble des violations* comme étant : [4]

Définition 19. L'*ensemble des violations* pour la DC φ $viol(I, \varphi) = \{(s, t)|(s, t) \not\models \varphi$ avec $s, t \in I\}$ est un ensemble de couple de tuples qui viole φ.L'ensemble de violation de Σ est $viol(I, \Sigma) = \cup_{\varphi \in \Sigma} viol(I, \varphi)$.

Capturer les violations $viol(I, \Sigma)$ permet de connaitre les tuples dont aucune des valeurs doit être modifié et les tuples dont au moins une valeur est suceptible d'être modifié. Soit une relation I sur S, soit un tuple $t \in I$ et φ une DC sur S, si pour tout tuple $s \in I$, $(s, t) \notin viol(I, \varphi)$ et $(t, s) \notin viol(I, \varphi)$ alors aucune valeur de t ne doit être modifiée.

Lorsqu'un tuple t est suspecté, toutes ses valeurs ne sont pas suspectes. Les valeurs que nous sommes susceptible de changer sont celles impliquées par une DC. On définit les cellules impliquées comme étant :

Définition 20. Soit une relation I sur S et une DC φ sur S. Soit deux tuples $s, t \in I$. Les *cellules de s et t impliquées dans la DC φ* se notent $cell(I, \varphi)$ et valent $cell(s, t, \varphi) = \{s.A, t.A|A \in Set(A, \theta_A) \in \varphi\, avec\, \theta_A = \varphi(A) \neq \top\}$

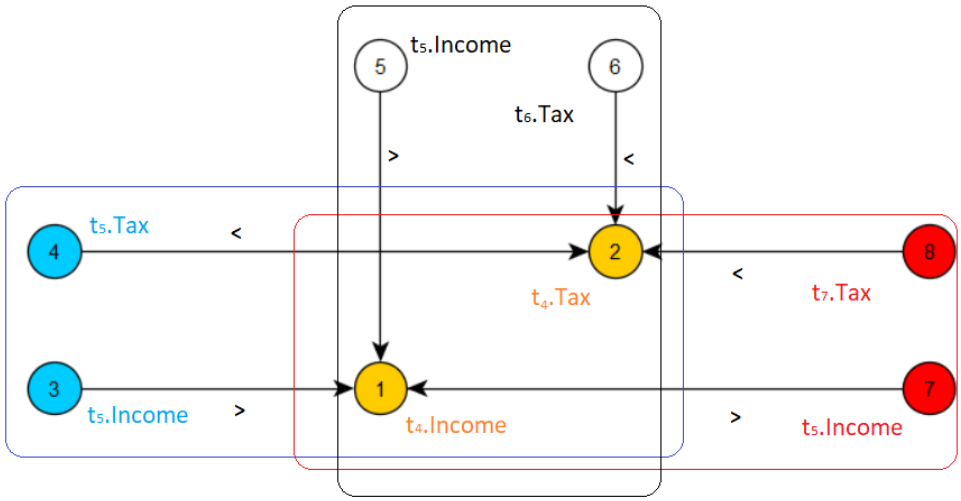

FIGURE 3.2 – Graphe de conflits pour φ

Si s, t sont deux tuples de I tel que (s, t) ou (t, s) est une violation alors seul les éléments de $cell(s, t, \varphi)$ doivent être éventuellement réparer. Chaque élément de $cell(s, t, \varphi)$ ne doit pas forcément être réparé. Parfois la réparation d'un seul de ses éléments suffit à réparer la violations.

On peut également définir le *degré d'une DC* φ qui se défini comme étant la norme des cellules impliquées de φ. Pour tout $s, t \in I$: $Deg(\varphi) = |cell(s, t, \varphi)|$. Nous définissons également le degré de Σ noté $Deg(\Sigma)$ comme étant :

$$Deg(\Sigma) = \sum_{\varphi \in \Sigma} |cell(s, t, \varphi)|$$

Nous allons maintenant pouvoir construire notre *graphe de conflit* de (Σ, I) formé de la manière suivante :

— Chaque nœud est un couple (t, A) avec $t \in I$. Pour une question de lisibilité on ne représentera que les tuples t qui apparaissent dans une violation c'est à dire qu'il existe un $s \in I$ tel que $(s, t) \in viol(I, \Sigma)$ ou $(t, s) \in viol(I, \Sigma)$ et $t.A \in cell(s, t, \varphi)$ avec $\varphi \in \Sigma$.

— Il y a un arc de (s, A) vers (t, A) si $(s, t \in viol(I, \Sigma)$ et $s.A, t.A \in cell(s, t, \varphi)$ avec $\varphi \in \Sigma$.

— Si un graphe de conflit ne contient aucune arrête, alors la relation n'a pas de violation.

Exemple 10. Prenons un exemple sur la table 2.1 ainsi qu'une DC que nous avons déjà utilisé auparavant :

$$\varphi' = \{(Revenu, >), (Taxe, <)\})$$

Pour notre relation, l'ensemble des violations est (voir figure 4.1) :

$$viol(I, \varphi') = \{(t_5, t_4), (t_6, t_4), (t_7, t_4)\}$$

Dans notre graphe, $(t_5, t_4) \in viol(I, \varphi')$ est représenté par $cell(t_5, t_4; \varphi')$ qui est égale à $\{t_5.Revenu, t_4.Revenu, t_5.Taxe, t_4.Taxe\}$. Nous voulons éliminer les violations, ce qui se traduit par éliminer les arcs du graphe.

Le but d'une réparation est de se débarrasser de tous les arcs du graphe de conflit. Si il n'y a plus d'arc, il n'y a plus de violations, et donc nous avons une relation réparée. Pour éliminer une violation d'un graphe de conflit, i.e pour éliminer un arc, il faut modifier la valeur associée à un nœud de ce graphe. Par exemple si nous avons un arc de (t, A) à (s, A), si nous modifions correctement la valeur $t.A$ ou $s.A$, l'arc est supprimé.

Le coût minimum pour réparer (t, A) est $dist(t.A, a)$ avec $a \in \mathbf{dom}(I, A)$. En effet, remplacer $t.A$ par une variable fraiche coute plus cher. Nous allons essayer de réparer le moins de cellule possible, i.e modifier le moins de nœuds possibles tout en supprimant toutes les arrêtes. Celz correspond à trouver la *couverture minimale par sommet*. Le problème de la couverture minimale par sommet, aussi communément appelé "Vertex Cover" consiste à trouver un ensemble minimum de sommet tel que l'on couvre toutes les arrêtes.

Propriété 1. Soit $\mathbb{V}(G)$ le minimum Vertex Cover du graphe G correspond à (Σ, I), on a que son poids vaut :

$$||\mathbb{V}^*(G)|| = \sum_{t.A \in \mathbb{V}(G)} min_a dist(t.A, a)$$

et nous savons que $\Delta(I, f(I)) \geq ||V * (G)||$ pour toute fonction de réparation f.

En effet, corriger tout les éléments de $V * (G)$ consiste à supprimer toutes les arrêtes, c'est à dire retirer toutes les violations. C'est donc la réparation la moins chère !

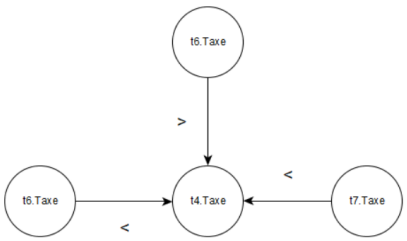

FIGURE 3.3 – Graphe de conflits pour l'attribut Taxe pour φ

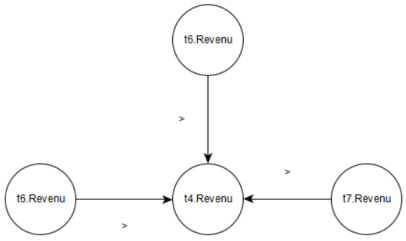

FIGURE 3.4 – Graphe de conflits pour l'attribut Revenu pour φ

Notons que réparer une valeur $t.A$ de la relation supprime toutes les arrêtes du graphe de conflit qui relient (t, A) à un autre nœud (s, A). Mais les arcs reliant (t, B) à un autre

nœud (s, B) avec $B \in S \backslash \{A\}$ peuvent aussi être supprimés. En effet si $t.A$ est modifié, il est possible que t ne soit plus parmi les violations de I. Dès lors tout arc reliant (t, B) à un autre nœud ne doit plus être inclut dans le graphe de conflit. Pour illustrer cela prenons le graphe de conflit de la figure 3.2 et séparons le en deux (figure 3.4 et 3.3). Si nous réparons $t5.Revenu$, l'arc $t_5.Revenu$ à $t_4.Revenu$ est supprimé. Mais puisque nous réparons $t_5.Revenu$, $(t_5, t_4) \notin viol(I, \varphi')$, nous pouvons supprimer l'arc de $t_5.Taxe$ à $t_4.Taxe$ aussi.

Le problème du minimum Vertex Cover est connu comme étant un problème de la classe NP-Complet. Nous avons donc besoin de trouver une approximation de cette couverture. Dans l'article de référence[4], les auteurs proposent de considérer un facteur d'approximation f de $\mathbb{V}^*(G)$, i.e $\frac{||\mathbb{V}(G)||}{||\mathbb{V}*(G)||} \leq f$ avec $\mathbb{V}(G)$ une approximation de $\mathbb{V}^*(G)$ et f le degré maximum des sommets du graphe de conflit. Nous connaissons donc f, c'est le degré de Σ! Dès lors au vu de ce résultat et de la propriété 1, nous pouvons dire que :

$$||\mathbb{V}(G)|| \leq f||\mathbb{V}^*(G) \leq Deg(\Sigma)\Delta(I, f(I)$$

Dès lors nous pouvons définir la borne inférieure à notre coût de réparation :

Définition 21. Soit Σ un ensemble de DC sur S et I une relation sur S, le coût réparation maximum, i.e la borne inférieure pour le coût est de : $\delta_l(\Sigma, I) = \frac{||\mathbb{V}(G)||}{Deg(\Sigma)}$

Trouvons la borne supérieure. Pour se débarrasser des violations la solution la plus simple est de modifier tous les sommet de la couverture du graphe de conflit par une variable fraiche. Ce faisant, on a le coût de variation le plus élevé possible. En effet, $dist(a, fv)$ avec $a \in \mathbf{dom}(A, I)$ et fv variable fraiche est la distance la plus grande. Nous avons donc ici la borne supérieure de réparation, c'est à dire la réparation la plus couteuse pour I avec l'ensemble Σ. On définit cette borne supérieure comme étant :

Définition 22. Soit Σ un ensemble de DC sur S et I une relation sur S, le coût réparation maximum, i.e la borne supérieure pour le coût est de : $\delta_u(\Sigma, I) = \sum_{t.A \in \mathbb{V}(G)} dist(I(t.A), fv)$

Pour un ensemble de DCs Σ et une relation I sur S ainsi qu'une fonction de réparation f pour I se basant sur Σ, nous savons que $\delta_l(\Sigma, I) \leq \Delta(I, f(I) \leq \delta_u(\Sigma, I)$

Si nous revenons à notre example et que nous supposons que nous avons les distances suivantes :

$$\forall a \in dom(A)\ avec\ a \neq b. \begin{cases} dist(a, a) = 0 \\ dist(a, b) = 1 \\ dist(a, fv) = 1.5 \\ dist(fv, fv) = 1.5 \\ dist(fv, b) = 1 \end{cases}$$

Donc si chaque arc a un poids de 1 (=$dist(a, b)$) et si nous posons $\mathbb{V}(G)$ = $\{t_4.Tax\}$ nous avons $||\mathbb{V}(G)||$ = 1. Nous avons aussi $Deg(\Sigma)$ = 4, donc en utilisant les formules pour le calcul des bornes supérieure et inférieure : $\delta_l(\Sigma, I)$=$\frac{||\mathbb{V}(G)||}{Deg((\Sigma)}$ = $\frac{1}{4}$ = 0.25 and $\delta_u(\Sigma, I)$ = $\sum_{t.A \in \mathbb{V}(G)} dist(I(t.A), fv)$ = $dist(a, fv)$=1.1 .

3.1 Réparation au coût minimal

Grâce au graphe de conflit, nous connaissons maintenant les valeurs $t.A$ qui doivent être modifiée pour respecter l'ensemble de DC Σ et réparer la relation I Mais pour l'instant, nous n'avons pas encore vu comment effectuer la réparation. Nous avons seulement expliqué que chaque mauvaise donnée est remplacée par une variable fraiche qui est ensuite remplacée par une vraie valeur si possible. Dans cette section, nous allons nous concentrer sur la réparation de données en minimisant le coût et en se basant sur Σ'. Pour ce faire, nous allons devoir nous assurer que nous ne créons pas de nouvelle violation après avoir corrigé une donnée. Par exemple, si nous choisissons de changer la valeur de $t_5.Taxe$ à $22k$, nous réglons la violation sur (t_5, t_4) que nous avions avec la denial constraint $\varphi = \{(Revenu, >)(Taxe, <)\}$. En changeant cette valeur de cette manière, on crée une nouvelle violation (t_8, t_5).

Il est important de rappeler que trouver une réparation de coût minimum est un problème NP-difficile. Pour ces problèmes, il est important de trouver une approximation. Pour la suite de cette section, nous allons noter $\mathbb{C}$ les cellules sélectionnées dans $\mathbb{V}(G)$.

3.1.1 Identification des suspects

Trouver la réparation de coût minimal à partir d'un ensemble Σ' de DC est un problème NP-complet [4], nous avons donc besoin d'un algorithme d'approximation pour trouver les solutions en un temps polynomial. Nous allons commencer par travailler avec la couverture $\mathbb{V}(G) = \mathbb{C}$ et nous assurer qu'aucune nouvelle valeur n'introduit de nouvelle violations. Nous savons que les variables fraiches n'introduisent aucune violation puisque par définition une variable fraiche rend tout prédicat faux et donc la DC est vraie. Mais puisqu'on tente de trouver une valeur à ces variables, nous devons détecter quels tuples sont les plus susceptibles d'introduire de nouvelles violations après une correction des cellules de $\mathbb{C}$.

Définition 23. L'ensemble des *suspects de* φ noté $susp(\mathbb{C}, \varphi)$ est un ensemble de couple de tuples qui satisfont tous les prédicats de φ qui n'implique pas de cellules impliquées de $\mathbb{C}$, i.e les tuples qui satisfont la *condition de suspicion :*

$$sc(s, t, \varphi) = \{s.A\theta t.A | s, t \in I; P : (A, \theta) \in \varphi; s.A, t.A \notin \mathbb{C}\}$$

Soit deux tuples $s, t \in I$ on peut utiliser le graphe à la figure 3.5 pour représenter les attributs et le lien avec les suspects et les cellules impliquées. Les cercles représentent les éléments de $\mathbb{C}$ i.e les éléments qui vont changer et les carrés représentent les éléments qui ne sont pas de $\mathbb{C}$ i.e des éléments qui ne doivent pas changer. Les flèches représentent les opérateurs de nos prédicats. Par exemple la flèche allant de 1 à 5 représente l'opérateur Θ_A du prédicat (A, Θ_A). Si toutes les flèches noires de φ sont satisfaites alors nous avons un suspect.

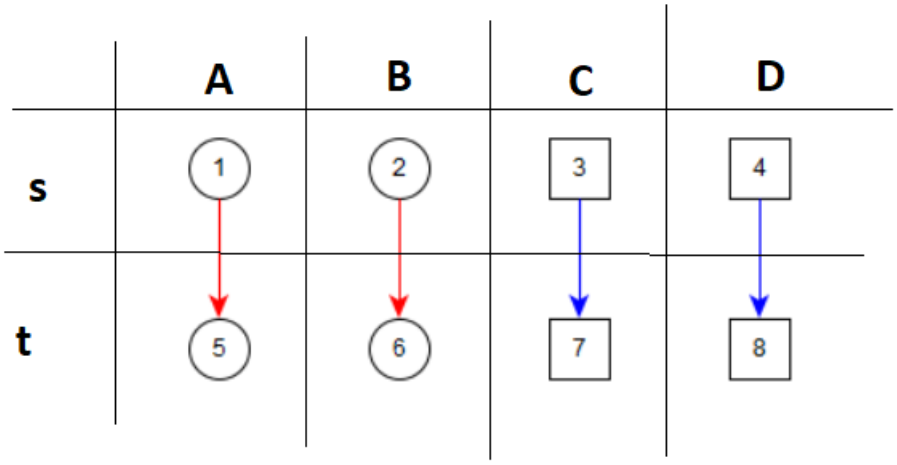

FIGURE 3.5 – s, t sont deux suspect

Le lemme suivant nous permet d'affirmer qu'avoir la liste des suspects d'une DC contient toutes les violations de cette même DC [4] :

Lemme 6. Soit I une relation sur S , Σ un ensemble de DC ,φ une DC de Σ et $\mathbb{C}$ une approximation de la couverture de poids minimum pour un graphe de conflit G correspondant à Σ, I nous avons toujours $viol(I, \varphi) \subseteq susp(\mathbb{C}, \varphi)$

Démonstration. Soit I une instance sur S , deux tuples $s, t \in I$ et φ une DC d'un ensemble de contrainte Σ pour I.
Supposons que nous avons $(s, t) \in viol(I, \varphi)$ alors tous les prédicats de φ sont vrais pour (s, t) i.e $(s, t) \models \varphi$ et donc ils satisfont la condition de suspicion □

Exemple 11. Reprenons l'exemple sur $\varphi' = \{(Revenu, >), (Taxe, >)\}$ avec comme instance I le tout en relation avec le graphe de conflits à la figure 3.2. Nous allons uniquement changer $t_4.Taxe$ comme nous l'avons fait à la table 3.1. Donc nous prenons $\mathbb{C}$ = $\{t_4.Taxe\}$ and $susp(\mathbb{C}, \varphi') = \{(t_4, t_1), (t_4, t_2), (t_4, t_3), (t_5, t_4), (t_6, t_4), (t_7, t_4), (t_8, t_4), (t_9, t_4), (t_{10}, t_4)\}$ Regardons en détails (t_4, t_1) à la figure 3.6. Nous avons $t_4.Taxe \in \mathbb{C}$ représenté par un cercle et $t_1.Taxe, t_4.Revenu, t_1.Revenu \notin \mathbb{C}$ et donc représenté par un carré. Le prédicat $t_4.Taxe > t_1.Taxe$ a une flèche rouge car il contient un cercle i.e une valeur de Taxe qui doit être changée et son opérateur $>$ est remplacé par $<$ (pour représenté la situation actuelle) et $t_4.Revenu > t1.Revenu$ ne contient pas de cercle donc sa flèche est noire i.e aucune valeur à changer.

Nous avons comme condition de suspicion sur (t_4, t_1) : $sc(t_4, t_1, \varphi) = \{t_4.Revenu > t_1.Revenu\}$. En se référant à la table 2.1, nous avons $t_4.Revenu > t_1.Revenu$ ce qui implique que tous les prédicats avec une flèche bleue, c'est à dire ceux impliquant une valeur dans $\mathbb{C}$

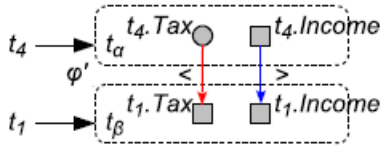

FIGURE 3.6 – Les conditions de suspicions sont représentées par des flèches bleues et le contexte de réparation est représenté par des flèches rouges (avec l'opérateur inverse)

sont satisfaits. Nous avons donc bien (t_4, t_1) suspect , donc une réparation sur $t_4.Taxe$ peut introduire de nouvelles violations avec $t_1.Taxe$, i.e $f(t_4.Taxe) < t_1.Taxe$ avec f fonction de réparation pour la relation I.

3.1.1.1 Contexte de réparation

Pour chaque couple de tuples $(s, t) \in susp(\mathbb{C}, \varphi)$ nous pouvons définir un *contexte de réparation* de la DC φ qui se note $rc(s, t\varphi)$. Ce contexte de réparation nous permettra d'assurer que les réparations ne vont pas introduire de nouvelles violations. Le contexte de réparation d'une DC φ est défini comme étant :

Définition 24. Soit I une relation sur S, soit φ une DC sur I, soit $s, t \in I$, et f une fonction de réparation pour I le contexte de réparation de φ est noté $rc(s, t, \varphi)$ et vaut :

$$\begin{aligned} rc(s,t,\varphi) = &\{t.A\overline{\theta}f(s.A) | (A,\theta) \in \varphi, s, t \in I\} \cup \\ &\{f(t.A)\overline{\theta}s.A | (A,\theta) \in \varphi, s, t \in I\} \cup \\ &\{f(t.A)\overline{\theta}f(s.A) | (A,\theta) \in \varphi, s, t \in I\} \end{aligned}$$

Si nos reprenons notre figure 3.5 ou 3.6, les arcs rouges représentent le contexte de réparation.

Propriété 2. Soit I une relation sur S et Σ un ensemble de DC pour I. Toute relation I' issue de la réparation de I i.e $I' = f(I)$ est *valide* si I' satisfait tous les contextes de réparations.

Exemple 12. (suite) Pour le couple de suspect $(t_4, t_1) \in susp(\mathbb{C}, \varphi)$ nous avons le contexte de réparation $rc(t_4, t_1, \varphi) = \{t_4.Taxe \geq t_1.Taxe\}$ que l'on obtient en prenant l'opposé de l'opérateur du prédicat $(Taxe, <)$ (voir définition) ce qui est représenté par une flèche rouge dans la figure 3.5. En considérant toutes les paires de tuples de $susp(\mathbb{C}, \varphi)$, nous obtenons l'entièreté du contexte de réparation de φ visible graphiquement à la figure 3.7. Par les contextes de réparations de φ : $f(t_4.Taxe) > t_1.Taxe = 0$ et $f(t_4.Taxe) \leq t_5.Taxe = 0$ par transitivité , nous pouvons affirmer avec certitude que la variable fraiche fv que nous avions assigné vaut $fv = 0$.

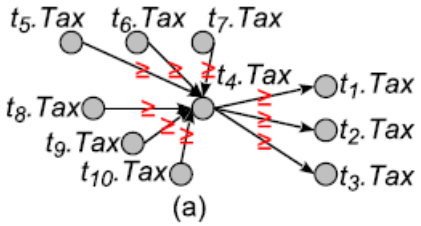

FIGURE 3.7 – Contexte de réparation

Nous avons défini le contexte de réparation pour une DC φ. Nous pouvons maintenant définir le contexte de réparation pour un ensemble Σ de DC :

Définition 25.

$$rc(\mathbb{C}, \Sigma) = \bigcup_{(s,t) \in susp(\mathbb{C},\varphi), \varphi \in \Sigma} rc(s, t, \varphi)$$

Nous pouvons exprimer le problème de réparation d'une relation I comme étant un problème de minimisation à résoudre :

$\min \sum_{t.\in\mathbb{C}} dist(t.A, f(t.A))$
Sous la contrainte $rc(\mathbb{C}, \Sigma)$

Il est possible d'utiliser la programmation linéaire pour résoudre ce problème d'optimisation pour les valeurs numériques et une "value frequency map" pour les chaines de caractères [4].

Nous pouvons décomposer $\mathbb{C}$ en plusieurs sous ensemble $\mathbb{C}_1, \mathbb{C}_2, ..., \mathbb{C}_n$ tel que $\forall s, t \in I$, il n'existe pas de $f(t.A)\overline{\Theta_A}f(s.A) \in rc(s, t, \varphi)$ pour $(s, t) \in susp(\mathbb{C}, \varphi)$ avec $\varphi \in \Sigma$ et Σ un ensemble de DC pour la relation I sur S. Nous pouvons dire que $rc(\mathbb{C}, \Sigma) = rc(\mathbb{C}_1, \Sigma) \cup ... \cup rc(\mathbb{C}_n, \Sigma)$ et donc nous pouvons résoudre notre problème en résolvant un problème de minimisation pour chaque $rc(\mathbb{C}_k, \Sigma)$ individuellement. En particulier nous pouvons résoudre le problème tuple par tuple.

Maintenant nous pouvons décrire un algorithme de réparation de donnée :

Algorithme 1 : Data_repairing(Σ,I,$\mathbb{C}$) :

Entrée : Un ensemble de DC Σ, une relation I à réparer et l'ensemble $\mathbb{C}$ des éléments à réparer.

Sortie : Une réparation de I

Initialiser $f(I) = I$
Initialiser $rc(\mathbb{C}, \Sigma)$
pour *chaque* $t \in C$ **faire**
    Résoudre $rc(\{t\}, \Sigma)$ par un solveur **si** $rc(\{t\}, \Sigma)$ *peut être résolu* **alors**
        $f(t)$ = solution du solveur
    $t.A$ = l'élément apparaissant le plus de fois dans $rc(\{t\}, \Sigma)$. $f(t.A) = fv$
return $f(I)$

Chapitre 4

Variation de contraintes

4.1 Variation sur les denial constraints

Nous avons vu précédemment qu'une denial constraint peut être sur-raffinée, échouant donc dans la détection d'erreurs. Une DC peut aussi être sur-simplifiée ce qui conduit à considérer des données correctes comme étant erronées. Parce que les contraintes peuvent être imprécises et inexactes, nous avons besoin de les corriger. En modifiant ces contraintes, nous pouvons obtenir une meilleure réparation.

Exemple 13. Par exemple prenons la DC suivante :

$$\varphi = \{(Revenu, >), (Taxe, \leq)\}$$

Nous pouvons traduire la DC φ de la manière suivante : pour tout tuples $s, t \in I$ si $t.Revenu > s.Revenu$ alors $t.Taxe > s.Taxe$. Cette DC exprime le fait que si une personne reçoit un revenu plus élevé qu'une autre personne, alors elle doit payer un montant de taxe strictement plus élevé. Nous allons modifier φ pour obtenir une nouvelle denial constraint φ' en changeant le prédicat $(Taxe, \leq)$ en $(Taxe, <)$, ce qui peut se traduire comme étant : pour tout tuples $s, t \in I$ si $t.Revenu > s.Revenu$ alors $t.Taxe > s.Taxe$. Dès lors, maintenant φ' exprime le fait que si une personne reçoit un revenu plus élevé qu'une autre personne, alors cette personne doit payer un montant de taxe plus élevé ou équivalent. Cela permet d'exempter de taxe plusieurs personnes ayant un Revenu faible, mais différent les uns des autres.

$$\varphi' = \{(Revenu, >), (Taxe, <)\}$$

Avec cette nouvelle contrainte, nous avons moins de violations détectées. Toutes les violations peuvent être trouvées à la figure 4.1. Les modifications que nous avons faites dans la table 3.1 sont :

/	t_β									
t_α	/	1	2	3	4	5	6	7	8	9	10
	1	/	V	V	V	V	V	V	V	V	V
	2	V	/	V	V	V	V	V	V	V	V
	3	V	V	/	V	V	V	V	V	V	V
	4	V	V	V	/	V	V	V	V	V	V
	5	V	V	V	F	/	V	V	V	V	V
	6	V	V	V	F	V	/	V	V	V	V
	7	V	V	V	F	V	V	/	V	V	V
	8	V	V	V	V	V	V	V	/	V	V
	9	V	V	V	V	V	V	V	V	/	V
	10	V	V	V	V	V	V	V	V	V	V

FIGURE 4.1 – All the violation for φ'

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
Taxe	0	0	0	fv_1	0	0	0	21k	21k	40k

TABLE 4.1 – Example of repair with Tax

.

Contrairement à la réparation de φ qui proposait une correction avec cinq variables fraiches (voir table 3.1), nous avons ici qu'une seule variable fraiche. De plus, nous savons certaines choses à propos de cette variable fraiche :

1. $t_1.Taxe = 0$ donc $f(t_4).Taxe \geq 0$ car $t_1.Revenu < t_4.Revenu$
2. $t_5.Taxe = 0$ donc $f(t_4).Taxe \leq 0$ car $t_4.Revenu < It_5.Revenu$

Nous avons donc $0 \leq fv_1 \leq 0$, dès lors la seule valeur possible pour remplacer fv_1 est $fv_1 = 0$. Nous avons donc un coût de réparation de $\Delta(I, I') = 1$ en reprenant les valeurs de distance de l'exemple 9.

Nous voyons donc que la modification de la contrainte conduit à un coût de réparation plus faible. Mais modifier une contrainte doit aussi avoir un coût. En effet , si on ne donne pas de coût à la modification d'une contrainte, il suffirait de sur-raffiner toutes nos contraintes pour détecter moins de violations et donc faire moins de modifications. Nous allons aussi considérer que nous n'ajoutons pas DC ni n'en retirons de Σ pour obtenir Σ' et donc $|\Sigma| = |\Sigma'|$.

Nous devons désormais définir une marche à suivre concernant les variations de DC. Nous devons trouver un moyen de définir un coût de variation. Considérons le treillis tel qu'illustrer à la figure 4.2 pour un attribut A de S. Chaque nœud du treillis représente un

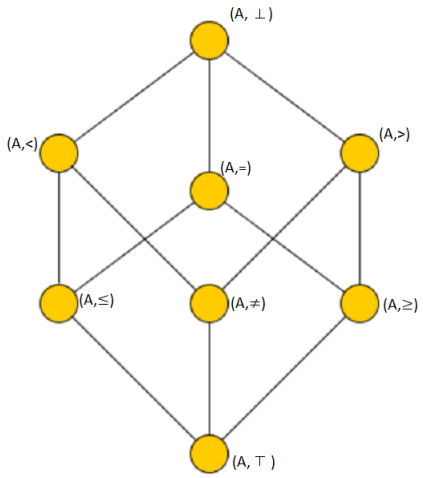

FIGURE 4.2 – Treillis pour un prédicat d'attribut A

prédicat différent pour A, donc A associé à un opérateur, un des éléments de l'ensemble puissance de OP. Descendre dans une branche du treillis consiste à ajouter un élément de OP à l'opérateur et donc à rendre la DC plus forte. Par exemple passer de $(A, <)$ à $(A, \leq)$ consiste à ajouter l'opérateur $=$ à $\{<\}$. Le noeud le plus bas est $(A, \top)$ et la DC sur $S = \{A\}$ qui n'est même pas satisfiable. Lorsque l'on monte dans le treillis, on retire un élément de OP à l'opérateur. Par exemple, passer de $(A, \geq)$ à $(A, =)$ consiste à retirer l'opérateur $>$ à $\{>, =\}$. Le nœud le plus haut dans le treillis est $(A, \perp)$ qui est trivial.

Nous devons modifier nos DC de telle manière qu'une contrainte sur-raffinée ou sur-simplifiée ne le soit plus. Si une DC est sur-raffinée, nous devons monter dans le treillis. En effet , plus on monte dans le treillis, plus le prédicat sur l'attribut A sera fort, i.e la DC sera fort. A l'inverse, si une DC est sur-simplifiée ; nous devons descendre dans le treillis. En effet plus nous descendons dans le treillis, plus le prédicat sur l'attribut A sera faible, i.e la DC sera plus faible. Plus nous changeons de prédicats, plus le *coût de variation des contraintes* sera important. Le *coût de variation des contraintes* se définit comme tel :

Définition 26. Soit Σ un ensemble de DC sur un ensemble S. Pour une variante Σ' de Σ, la fonction de calcul du *coût de variations des contraintes* de Σ est :

$$\Theta(\Sigma, \Sigma') = \sum_{\varphi in \Sigma} edit(\varphi, \varphi')$$

avec φ' une variante de φ et $edit(\varphi, \varphi')$ est le coût pour changer φ en φ'.

La fonction $edit(\varphi, \varphi')$ qui indique le coût pour changer φ en φ' est défini comme étant :

Définition 27.

$$edit(\varphi, \varphi') = \sum_{A \in S} path(\varphi(A), \varphi'(A))$$

avec $path(\varphi(A), \varphi'(A))$ le coût du chemin emprunté dans le treillis.

Nous avons représenter un treillis pour un seul prédicat. Si nous souhaitons représenté le treillis pour deux prédicats, nous avons besoins de $8^2 = 64$ nœud ce qui n'est visuellement pas pratique. Nous avons choisis la solution de représenté deux treillis, un pour chacun des attributs et la variation consiste à se déplacer dans les deux treillis à la fois et de sommer le coût des deux chemins (voir formule de la définition 27). Les deux approches donnent le même résultat.

Nous souhaitons que notre coût de variations des contraintes ne soit pas trop élevé. Pour ce faire, nous avons besoin que ce coût n'excède pas une valeur θ. Plus tard lorsque nous présenterons l'algorithme de réparation, nous devrons choisir un paramètre θ pour limiter le coût de variation.

Si nous considérons notre treillis comme un graphe, il reste maintenant à trouver le poids de chacun des arcs de ce graphe.

4.1.1 Parcours dans le treillis

Maintenant que nous avons défini le coût de variation des contraintes, regardons le coût pour se déplacer dans le treillis. Soit φ une DC sur S et φ' une variante de φ. Soit A un attribut sur S. Le cout pour passer de $\varphi(A)$ à φ', i.e la valeur de $path(\varphi(A), \varphi'(A)$ est le coût du plus court chemin du treillis. Le treillis doit être vu comme un graphe pouvant être parcouru dans les deux sens. Pour un attribut $A \in S$, le poids de chaque arc est de $c(A)$, $c(A)$ étant une valeur symbolisant la confiance qu'on accorde à la valeur de A. Si on n'a aucune connaissance sur la base de données, $c(A) = c(B)$, pour tout $A, B \in S$.

Exemple 14. Soit $\varphi = \{(Taxe, \leq), (Revenu, <)\}$ et $\varphi' = \{(Taxe, >)(Revenu, <)\}$ et c(Taxe) = c(Revenu) =1.

Pour passer de $(Taxe, \leq)$ à $(Taxe, >)$, un chemin possible est $(Taxe, \leq) \rightarrow (Taxe, <) \rightarrow (Taxe, \perp) \rightarrow (Taxe, >)$ et ce chemin à un coût de 3.

Par observation, on peut facilement se rendre compte que le plus court chemin d'un treillis est de longueur 3 maximum. Dans les sous-sections qui suivent, nous verrons qu'il sera important de limiter nos candidats, ce qui modifiera éventuellement le poids de certains arcs.

Nous avons vu dans le chapitre précédent un lemme disant que $\varphi \models \hat{\varphi}$ et que nous avons aussi $\hat{\varphi} \models \varphi$ pour φ une DC sur S. Pourtant pour modifier φ en $\hat{\varphi}$, nous avons un coût non nul. Ce n'est pas un problème, la réparation d'une relation I sera la même peut importe qu'on ait φ ou $\hat{\varphi}$ dans l'ensemble des DC. Dès lors le coût de réparation sera le

même, et puisque l'on cherche à minimiser le coût de réparation aucun des deux ensemble ne sera privilégié par rapport à l'autre.

4.1.2 Limitation des candidats

Dans notre treillis, nous avons 8 nœuds. Pour une DC sur un ensemble S de taille 1, nous avons 8 différentes variantes de DC. Pour un ensemble S de taille n nous avons 8^n variations. Ce nombre est très important et envisager toutes les variations de contraintes serait coûteux en terme de complexité. Si le calcul d'une variation se fait en $O(1)$ alors considérer toutes les variantes possible est en $O(8^n)$. Si l'ensemble Σ de contraintes est de taille l, puisqu'il faut considérer toutes les variantes pour chaque DC $\varphi \in \Sigma$, la complexité serait de $O((8^n)^l)$ Nous avons donc besoin de limiter le nombre de variations à considérer. Pour ce faire, nous allons utiliser des propriétés sur les DC pour limiter le nombre de contraintes à considérer.

Nous savons déjà par le chapitre précédent que les contraintes triviales sont inutiles. En effet, une contrainte triviale sera toujours vraie et donc considérera tous les tuples comme étant corrects. Le Lemme 5 nous permet de rejeter toutes les DC dont au moins un des prédicats est de la forme $(A, \perp)$.

Nous avons également besoin que la denial constraint soit satisfiable. Si une DC n'est pas satisfiable alors elle possède uniquement des prédicats avec l'opérateur $\top$ ou $=$. Nous pouvons donc rejeter toutes les DC ne possédant aucun opérateur parmi $\{\perp, \neq, <, >\}$

Nous souhaitons aussi rendre nos DC plus faible. Plus une DC est faible, moins il y aura de violations dans la relation. Puisque nous cherchons à minimiser le coût de réparation, nous devons minimiser le nombre de violations détectées. Dès lors nous ne chercherons plus que les DCs plus faible que celle dont on cherche les variations. Grâce à cette ça, nous savons maintenant qu'il est inutile de considérer toutes les insertions [1] de prédicats possibles. De manière plus générale, si nous considérons une variation d'un prédicat, on ne peut le changer par un prédicat plus bas dans le treillis. Notons que nous pouvons toujours descendre dans le treillis à condition de ne pas descendre plus de fois qu'on n'est monté. Par exemple, pour $A \in S$, si nous avons le prédicat $(A, \{=\})$ alors nous pouvons descendre en $(A, \{=, <\})$ puis monter en $(A, \{<\})$.

Illustrons le fait qu'une DC plus faible demande moins de réparation avec un exemple.

Exemple 15. Nous allons prendre deux DC sur la relation de la table 2.1 :

$$\varphi_1 : \{(Nom, =), (Revenu, =), (Numtel, \neq)\}$$

$$\varphi_2 : \{(Nom, =), (Revenu, \leq), (Numtel, \neq)\}$$

1. Par insertion de prédicats, comprenons ici que nous passons de $\varphi(A) = \top$ à $\varphi(A) \neq \top$

Nous pouvons appliquer le Lemme 2, en effet nous avons que $\{=\} \subseteq \{<, =\}$. Nous avons donc $\varphi_1 \models \varphi_2$. Pour passer de φ_1 à φ_2, nous avons monté dans le treillis de $Revenu$. Dans ce scénario, nous pouvons calculer un coût de réparation pour l'attribut $NumTel$ de 7 pour la DC φ_2 et la DC φ aura un coût de réparation de $Numtel$ de valeur 3.

	Nom	NumTel	Revenu
t1	Ayres	564-389	22k
t2	Ayres	564-389	22k
t3	Ayres	564-389	22k
t4	Stanley	930-198	24k
t5	Stanley	930-198	24k
t6	Stanley	930-198	24k
t7	Dustin	824-870	100k
t8	Dustin	824-870	100k
t9	Dustin	824-870	100k
t10	Dustin	824-870	100k

TABLE 4.2 – Correction avec φ_2 : 7 changement nécessaire dans la colonne $NumTel$

.

	Nom	NumTel	Revenu
t1	Ayres	322-573	21k
t2	Ayres	564-389	22k
t3	Ayres	564-389	22k
t4	Stanley	868-701	23k
t5	Stanley	930-198	24k
t6	Stanley	930-198	24k
t7	Dustin	179-924	25k
t8	Dustin	824-870	100k
t9	Dustin	824-870	100k
t10	Dustin	387-215	150k

TABLE 4.3 – Correction avec φ_1 : seulement 3 changements sont nécessaires.

.

Nous voyons que la DC raffinée a un coût de réparation plus faible. Ce n'est pas une coincidence puisque le Lemme suivant existe : [4]

Lemme 7. Soit Σ un ensemble de DC sur S et I une relation sur S. Soit 2 variantes de Σ noté Σ_1, Σ_2 et Σ_2 est un raffinement de Σ. Si $\Sigma \models \Sigma_1$ et $\Sigma_1 \models \Sigma_2$ alors $\Delta(I, I_1) \geq \Delta(I, I_2)$ avec I_1 la relation résultante de la réparation avec Σ_1 et I_2 la relation résultante de la réparation avec Σ_2.

Comme conséquence à ce Lemme, tous les ensembles de denial constraint Σ qui ne sont pas plus faible ne donneront pas la meilleure réparation.

Nous avons dit précédemment que nous avions $8^{|S|}$ variantes possibles pour une DC φ sur S. Nous avons éliminé certains cas, le résumé de ces cas sont les suivants :

— Nous pouvons éliminer toutes les variantes triviales i.e les variantes contenant un opérateur $\perp$ c'est à dire $2^{|S|} - 1$ [2].

— Nous pouvons éliminer toutes les variantes non-satisfiables i.e les DCs qui ne contiennent pas d'opérateurs parmi $\{\neq, <, >\}$

— Les DCs qui ne sont pas plus faibles que la DC de base.

— Les DCs qui ont un coût de variations supérieur à θ.

Ils reste beaucoup de cas valide, et calculer le coût de réparation $\Delta(I, f(I))$ pour une relation I est coûteux. Nous allons devoir continuer à limiter nos cas.

2. $|S| \geq 1$ sinon le problème de trouver des variantes de DCs n'aurait pas de sens.

4.1.3 Limitation par bornes

Nous avons vu précédemment que pour une seule DC, nous avions $8^{|S|}$ variantes possibles (voir treillis à la figure 4.2). Nous avons donné des pistes pour limiter le nombre de candidats. Pour continuer dans cette lancée, nous allons déterminer deux bornes entre lesquelles le coût de la réparation la moins coûteuse se trouve. Soit I une relation sur S, Σ et Σ' deux ensembles de DC sur S, f une fonction de réparation pour I telle que $f(I) \models \Sigma$ et g une fonction de réparation pour I telle que $f(I) \models \Sigma'$. En général, le calcul du coût $\Delta(I, f(I))$ peut avoir une complexité élévée. Dans certains cas, il est possible d'obtenir, par un calcul simple, une borne inférieure δ_l et une borne supérieure δ_u de ce coût. Si la borne inférieure pour $\Delta(I, f(I))$ est plus grand que le coût supérieur de $\Delta(I, g(I))$, alors on peut privilégier g à f. Nous avons donc la propriété suivante :

Propriété 3. Soit deux ensemble de denial constraints Σ_1 et Σ_2 pour une relation I sur R, f une fonction de réparation pour I telle que $f(I) \models \Sigma_1$ et g une fonction de réparation pour I telle que $f(I) \models \Sigma_2$. Si $\delta_u(\Sigma_1, I) < \delta_l(\Sigma_2, I)$ alors $\Delta(I, g(I)) > \Delta(I, f(I))$

Démonstration.

Soit I une relation sur S et Σ un ensemble de DC.

Soit Σ_1, Σ_2 deux ensemble de DC, variantes de Σ.

Soit I_1, I_2 deux relations résultantes de la réparation de I avec les ensembles de DC Σ_1, Σ_2 respectivement.

Par hypothèse nous avons $\delta_l(\Sigma_1, I) < \Delta(I, I_1) < \delta_u(\Sigma_1, I)$ et $\delta_l(\Sigma_2, I) < \Delta(I, I_2) < \delta_u(\Sigma_2, I)$ et $\delta_u(\Sigma_1, I) < \delta_l(\Sigma_2, I)$.

Dès lors nous avons $\delta_l(\Sigma_1, I) < \Delta(I, I_1) < \delta_u(\Sigma_1, I) < \delta_l(\Sigma_2, I) < \Delta(I, I_2) < \delta_u(\Sigma_2, I)$.

Alors par transitivité $\Delta(I, I_1) < \Delta(I, I_2)$. I_2 est une réparation plus couteuse que I_1 i.e Σ_1 est un meilleur candidat que Σ_2. □

Cela veut dire que si la pire borne (borne supérieure) de la réparation de Σ_1 est moins bonne que la meilleure borne (borne inférieure) de la réparation de Σ_2 alors Σ_1 sera un meilleur ensemble DC pour la réparation que Σ_2 et donc Σ_2 peut être abandonnée. C'est intéressant pour l'élaboration d'un algorithme cherchant à trouver le meilleur ensemble de contraintes. En effet, si le calcul de ces bornes est moins couteux que le calcul du cout de réparation $\Delta(I, I')$ alors on peut réduire la complexité de cet algorithme.

Nous connaissons déjà le calcul de ces deux bornes avec les Définitions : 21 et 22 vu au chapitre précédent. Les calculs sont :

$$\delta_u(\Sigma, I) = \sum_{t.A \in \mathbb{V}(G)} dist(I(t.A), fv)$$

$$\delta_l(\Sigma, I) = \frac{||\mathbb{V}(G)||}{Deg(\Sigma)}$$

Nous pouvons donc gagner du temps de calcul en utilisant les deux bornes dans notre algorithme !

4.1.4 Algorithme de variations de DCs

Nous pouvons déjà proposer l'algorithme suivant qui nous permet d'obtenir toutes les variations d'une DC φ

Algorithme 2 : get_variation(φ) :

Entrée : Une DC φ dont on veut la variation
Sortie : L'ensemble de toutes les variations de φ respectant les conditions énoncés précédemment

```
1 Prendre un attribut A quelconque de S.
2 variations = get_recursive_variations(A,∅,φ)
3 pour chaque var dans variations faire
4     si var est trivial ou var n'est pas satisfiable ou var n'est pas plus faible que φ alors
5         Supprimer var de variations
6 return variations
```

Algorithme 3 : get_recursive_variation(A,att_pris,φ) :

Entrée : Un attribut A, att_pris l'ensemble des attributs déjà utilisé dans les récursions précédentes, une DC φ dont on veut la variation
Sortie : Toutes les variations de φ en modifiant les prédicats contenant A

```
1 Initialiser Σ vide
2 si |att_pris| < |S| alors
3     pour chaque prédicat pred plus faible que (A, φ(A)) faire
4         Former une nouvelle DC φ' en remplaçant (A, φ(A)) par pred
5         Ajouter φ' à Σ
6         Prendre un attribut B de S\att_pris
7         Ajouter les DC retournées par get_recursive_variation(B,att_pris∪{A},φ)
8 return Σ
```

L'algorithme 2 retourne toutes les variations d'une DC φ donnée en entrée. Les variations sont donnés par la ligne 2 en appelant l'algorithme 3, un algorithme récursif. Nous trions chaque variations dans les lignes 3 à 5, suivant les différentes limitations que nous avons évoquées dans les sections précédentes.

L'algorithme 3 est un algorithme récursif qui donne toutes les variations possibles pour une DC φ en paramètre. On crée des DCs φ' à partir de φ en changeant juste le prédicat sur l'attribut A , un attribut en paramètre. On obtient donc une variation de φ et on l'ajoute à Σ. Pour chacune de ces variations on fait un appel récursif sur φ' en prenant un attribut qui n'a pas encore utilisé, c'est à dire un attribut de S qui n'est pas dans att_pris. On

ajoute également A dans les attributs pris. La complexité en temps dans le pire des cas de l'algorithme est en $O(8^n)$. Le pire cas est la DC φ = {} c'est à dire que tout les opérateurs sont $\top$ et donc l'entièreté des treillis sont parcourus pour chaque attribut de S.

4.1.5 Coût dans le treillis

Nous pouvons donc modifier une DC φ en modifiant ses prédicats. Nous avons vu précédemment que les opérateurs peuvent être représenter dans un treillis et modifier l'opérateur consiste à se déplacer dans le treillis. Le coût dépend donc du nombre d'arcs parcourus dans le treillis. Nous verrons dans cette sous-section quel est le poids pour chacun de ces arcs.

Il est possible de représenter le treillis comme à la figure 4.3. Chaque couleur représente un type de changement. Par exemple chaque arc orange représente un changement depuis l'opérateur $\{\} \equiv \perp$ vers un élément de OP de taille 1. Discutons maintenant du poids que l'on peut donner à chacun de ces arcs.

Définition 28. Soit I une relation sur S. Soit $\mathbb{T}$ un treillis pour un attribut $A \in S$. On note $w(a, b)$ le poids d'un arc de $\mathbb{T}$ depuis un élément de OP de taille a vers un élément de OP de taille b.

Soit φ la DC dont nous voulons calculer les variations. Nous avons vu précédemment que nous n'allions considérer que les DCs qui sont plus faibles que φ. Dès lors nous n'avons besoin de définir que les poids $w(a, b)$ avec $a > b$. En effet on ne descend jamais dans le treillis.

Nous avons vu précédemment qu'une DC qui contient l'opérateur $\perp$ est triviale (voir Lemma 5). Si Nous avons une DC non triviale, nous voulons éviter de la rendre triviale. En effet si on rend une DC triviale, on a certes un coût de réparation qui diminue mais on a directement une DC sur-raffinée. Il ne faut donc pas transformer un opérateur en $\perp$. Les arcs allant vers $\perp$, i.e les arcs rouges dans la figure 4.3, ne doivent jamais être empruntés. Pour cela, le poids de ces arcs sera $w(1, 0) = +\infty$. Notons que si la DC φ que nous voulons faire varier contient un opérateur $\perp$, quelque soit la variation que l'on fait, on obtiendra une DC qui qui est équivalente. Il n'y a donc aucun besoin de chercher une variation et on peut donc l'ignorer.

Il nous reste à définir deux poids montant : $w(3, 2)$ et $w(2, 1)$. A priori, il n'y a aucune raison de mettre un poid particulier sur ces transitions. Nous poserons que $0 < w(3, 2) = w(2, 1) = \alpha$ avec α un poids que nous définirons en fonction de θ ($\alpha < \theta$). θ est une valeur que nous devrons définir dans l'algorithme θ-tolérant.

Les poids des arcs descendant seront mis à ∞ afin d'éviter qu'on emprunte ces arcs car nous voulons rendre nos DC plus faible, pas plus forte !

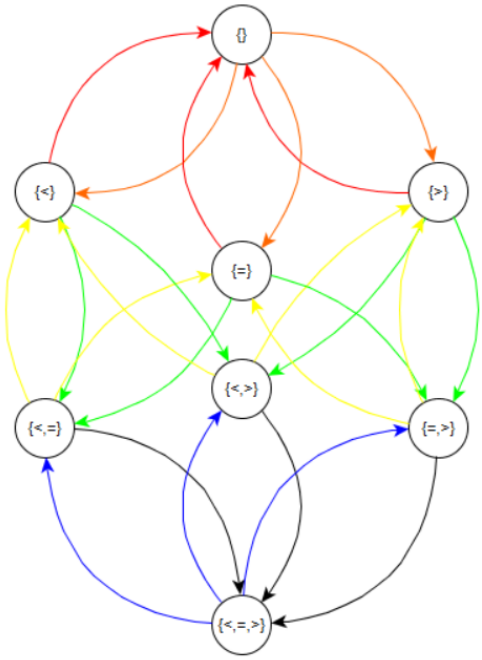

FIGURE 4.3 – Chaque types de transition aura son propre poids.

Dans notre implémentation, nous allons préférer la modification d'un prédicat avec un opérateur parmi $\{\leq, \neq, \geq\}$ vers un opérateur $\{<, =, >\}$ plutôt que la modification d'un prédicat avec l'opérateur $\top$ vers $\{\leq, \neq, \geq\}$. En effet, si un prédicat à un opérateur qui n'est pas $\top$, c'est qu'on a considérer que ce prédicat était important. Nous considérons qu'il n'a pas été choisi au hasard et donc $0 < w(2,1) = \alpha$ et $w(3,2) = 2\alpha$.

4.1.6 Variation d'un ensemble de contraintes

Nous avons jusqu'à présent abordé le sujet de la variation d'une seule contrainte. Pour faire varier un ensemble de contraintes Σ, il suffit d'appliquer le même raisonnement à au moins une des DCs de Σ. Mais nous devons statuer si on peut ajouter des DC ou même en supprimer. Nous déciderons d'aller dans le même sens que l'article sur lequel nous nous sommes initialement basés [1] et donc nous n'ajouterons pas de DC ni n'en retirerons. Retirer une DC diminue le nombre d'erreurs détectées (l'ensemble des DC est sur-raffiné)et on peut supposer que chaque DC avait un sens lors de sa création (mais peut évidement être imprécis). L'ajout de DC est bien plus compliqué a mettre en oeuvre et on peut rapidement obtenir des DC qui n'ont pas de sens et qui détecteraient des erreurs qui ne le sont pas (l'ensemble de DC est sur-simplifié).

Nous avons décidé de ne pas supprimer des DCs dans un ensemble de DC Σ, mais

suivant la manière dont nous modifions Σ, nous pouvons avoir deux DCs équivalent ce qui équivaut à une suppressions. Si un ensemble contient au moins deux DC et qu'on modifie une des contraintes jusqu'à ce qu'elle soit identique à une autre, alors ça revient à faire une suppression.

Exemple 16. Soit $S = \{A, B, C\}$ Prenons $\Sigma_1 = \{\varphi_1 = \{(A, <), (B, \neq)\}\varphi_2 = \{(A, <), (C, \neq)\}\}$ et la variation $\Sigma_2 = \{\{(A, <), (B, \neq), \}, \{(A, <), (B, \neq)\}\}$ nous voyons que la variation contient deux fois la même DC, et donc on peut réécrire $\Sigma_2 = \{\{(A, <), (B, \neq)\}\}$ ce qui équivaut à une suppression.

ou encore :

Exemple 17. Reprenons $S = \{A, B, C\}$ et $\Sigma' = \{\varphi_1 = \{(A, <), (B, \neq)\}\varphi_2 = \{(A, <), (C, \neq)\}\}$ et la variation $\Sigma' = \{\varphi'_1 = \{(A, <), (B, \neq), (C, \neq)\}\varphi'_2 = \{(A, <), (B, \neq), (C, \neq)\}\}$, les deux DC ont été modifiés mais nous avons le même ensemble !

4.2 θ-tolerant model

Dans cette section nous allons enfin aborder le *modèle de réparation θ-tolérant*. θ représente ici un seul de variation sur l'ensemble des contraintes Σ, nous ne voulons donc pas une variante de contraintes dont le coût serait plus grand que θ : $\Theta(I, f(I)) \leq \theta$ (voir défintion 26 pour la définition du coût de variations des contraintes). L'idée est d'éviter le sur-raffinement et donc d'éviter de laisser certaines données erronées. Afin d'éviter la sur-simplification, nous utilisons le principe du changement minimum. Nous avons donc besoin d'une relation réparée I' de la relation originale I et minimisant le coût de réparation $\Delta(I, I')$.

Trouver la meilleure réparation suivant le modèle θ-tolérant, i.e trouver la réparation de coût minimum est un problème de la classe NP-difficile. La classe de problèmes NP-difficile est une classe dont les problèmes sont au moins aussi difficile que les problèmes les plus difficiles de la classe NP [3]. Nous devons donc retenir ici que ce n'est pas possible de résoudre la réparation θ-tolérante en un temps polynomial à moins que P=NP [4]. L'approche naïve est de récupérer toutes les variantes Σ' de Σ et de ensuite calculer $\Theta(I, I')$ avec I' la relation résultante de la réparation suivant les DC de Σ'. Si $\Theta(I, I') \leq \theta$, on calcule le coût de réparation. Ensuite on compare le coût de la réparation I' pour chaque Σ' et on garde la réparation la moins coûteuse. Cette méthode est évidement très haute en complexité.

3. Les problèmes NP sont des problèmes dont une solution peut être vérifiée comme étant bonne en un temps polynomial

4. La question de savoir si P=NP est une conjecture dans le domaine de l'informatique théorique.

Nous avons vu qu'on remplaçait chaque donnée erronée par une variable fraiche fv puis nous essayons de remplacer certaines de ces variables fraiches par des vraies valeurs. Plus une réparation privilégies les vraies valeurs aux valeurs fraiches, plus elle aura de chance de minimiser le coût de réparation.

Maintenant, considérons $\mathbb{D}$ comme étant l'ensemble de toutes les variations Σ' de Σ. Considérons que ces variantes sont limitées par θ, donc nous avons $\Theta(\Sigma, \Sigma') \leq \theta$. L'algorithme 4 retourne la meilleure relation I_{min} de l'ensemble des contraintes Σ_{min}. L'algorithme est simple : pour chaque Σ_i , si la borne inférieure est plus petite que la borne supérieure (voir la propriétés 3), nous mettons à jour la valeur de δ_{min} car une meilleure réparation I_i a été trouvée.

Algorithme 4 : θ-TolerantRepair($\mathbb{D}, \Sigma, I$)

Entréé : Relation I, ensemble de DC Σ, ensemble $\mathbb{D}$ de variantes de DC bornées par θ
Sortie : Une relation réparée I_{min}

```
1 δ_min = δ_u(Σ, I)
2 pour chaque variante de contrainte Σ_i ∈ 𝔻 faire
3     si δ_l(Σ_i, I) ≤ δ_min alors
4         I_i = DATAREPAIR(Σ_i, I, 𝕍(G_i))
5         si Δ(I, I_i) ≤ δ_min alors
6             δ_min = Δ(I, I_i)
7             I_min = I_i
8 return I_min
```

Exemple 18. Pour prendre un exemple, imaginons que nous avons un $\theta = \frac{?}{?}$ et un ensemble de variations de contraintes $\mathbb{D} = \{\Sigma_1, \Sigma_2\}$ avec comme premier ensemble $\Sigma_1 = \{\varphi'\}$ et comme second ensemble $\Sigma_2 = \{\varphi''\}$ avec :

$$\varphi' = \{(Revenu, >), (Taxe, <)\}$$

$$\varphi'' = \{(Revenu, >), (Taxe, =)\}$$

Nous avons déjà fait le graphe de conflit pour la DC φ' que l'on retrouve à la figure 3.2 et on sait également que $\delta_u(\Sigma_1, I) = 1.1$

Pour $\Sigma 2$ nous obtenons le graphe de conflit de la figure 4.4 (et les violations se retrouvent à la figure 4.5) avec $\mathbb{V}(G_2) = \{t_2.Tax, t_3.Tax, t_5.Tax, t_6.Tax, t_7.Tax\}$. Nous $Deg(\Sigma_2) = Deg(\varphi')$ [5], donc nous avons $\delta_l(\Sigma_2, I)$= $\frac{6}{2}$ = 1,5. Rappelons tout de même que $\delta_u(\Sigma_1, I)$ =1.1,

5. Même raisonnement que précédement, nous avons 4 cellules impliquées.

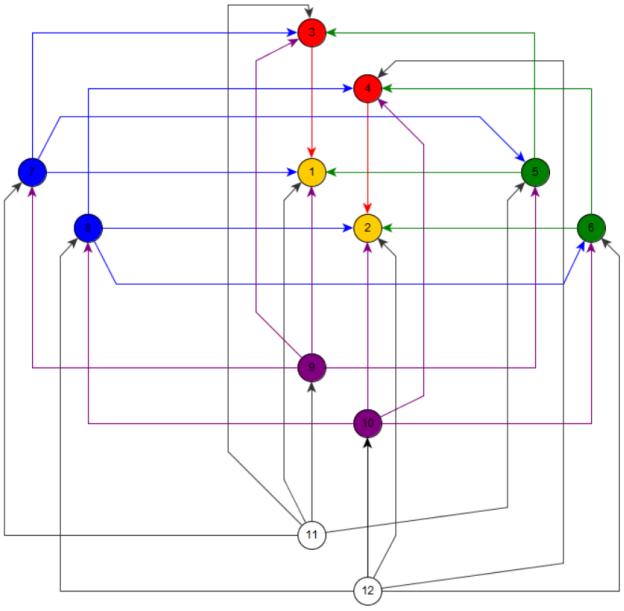

FIGURE 4.4 – Graphe de conflit pour Σ_2 avec :
nombre pair pour les Taxe, nombre impair pour le Revenu.
t_1 en jaune, t_2 en rouge, t_3 en vert,
t_4 en bleu, t_5 en violet et t_6 en blanc.

donc nous avons $\delta_u(\Sigma_1, I) < \delta_l(\Sigma_2, I)$ ce qui signifie que nous pouvons ignorer $\Sigma 2$ et donc ne pas appeler la fonction DATAREPAIR pour Σ_2. Nous parlerons de la fonction DATA-REPAIR plus tard.

/	t_β									
t_α	/	1	2	3	4	5	6	7	8	9	10
	1	/	V	V	V	V	V	V	V	V	V
	2	F	/	V	V	V	V	V	V	V	V
	3	F	V	/	V	V	V	V	V	V	V
	4	V	V	V	/	V	V	V	V	V	V
	5	F	F	F	V	/	V	V	V	V	V
	6	F	F	F	V	V	/	V	V	V	V
	7	F	F	F	V	F	F	/	V	V	V
	8	V	V	V	V	V	V	V	/	V	V
	9	V	V	V	V	V	V	V	V	/	V
	10	V	V	V	V	V	V	V	V	V	V

FIGURE 4.5 – Toutes les violations pour φ''

Parlons maintenant de la complexité de l'algorithme. Disons que l est le nombre de contraintes impliquées dans Σ dès lors nous pouvons dire que la construction du graphe de conflits G_i pour chaque $\Sigma_i \in \mathbb{D}$ est en $O(|I|^l)$. L'algorithme de réparation de données a une complexité en temps de $O(|I|^l)$ et l'algorithme 4 a une complexité en temps de $O(|I|^l|\mathbb{D}|)$.

Chapitre 5

Différence par rapport à l'article de base

Dans ce chapitre, nous parlerons des différences entre l'article de référence sur le modèle θ-tolérant et ce mémoire. Certaines notions et définitions ont été revues, et le traitement théorique a été amélioré. Certaines erreurs ont été corrigées notamment des erreurs concernant les variations de contraintes et coût de réparation de la base de données.

Pour la rédaction de ce mémoire, nous nous sommes inspiré de deux articles. Le premier est un article sur le modèle θ-tolérant, le second est un article qui porte sur les denial constraints. Ils utilisent une approche différente et une définition différente de la notre. Dans les deux articles [4, 5] ils définissent une DC comme étant :

Définition 29. Considérons un schéma de relation R avec comme attributs $att(R)$. Soit l'espace de prédicat $\mathbb{P}$ qui est un ensemble de prédicat P de la forme $v_1 \phi v_2$ ou $v_1 \phi c$ avec $v_1, v_2 \in t_x.A,\ x \in \{\alpha, \beta\}$, $t_\alpha, t_\beta \in R$, $A \in attr(R)$, c une constante et $\phi \in \{=, <, >, \leq, \geq, \neq\}$ est un opérateur. Une *denial constraint* (DC)

$$\varphi : t_\alpha, t_\beta, ... \in R, \neg(P_1 \wedge P_2 \wedge ... \wedge P_m)$$

signifie que pour tout tuples t_α, t_β dans R, tous les prédicats $P_i \in pred(\varphi)$, i = 1,....,m, ne devraient pas être tous vrais en même temps.

Une DC peut donc être vue comme une conjonction de prédicats et l'un de ses prédicats doit être faux afin que la DC soit vraie, i.e si pour deux tuples chaque prédicat est vrai, alors il y a au moins une donnée erronée dans l'un des deux tuples.

La définition bien qu'étant différente d'un point syntaxique représente toutes les deux la même chose. Cette définition-ci permet en plus de comparer un attribut à une constante. Bien qu'il puisse être intéressant d'avoir des denial contraints qui fixe un salaire minimum par exemple $t.Revenu > 10k$ ou exprime le fait qu'un revenu ou une taxe ne peut être négative $(t.Revenu > 0) \wedge (t.Taxe > 0)$ cela crée plusieurs problèmes.

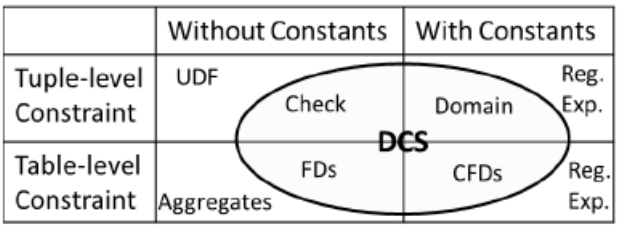

FIGURE 5.1 – La denial constraint tel que définie dans les articles de référence peut exprimer plusieurs types d'autres contraintes [5]

Dans notre définition, nous avons un nombre de prédicats maximum qui est la norme de S. Ici nous ne sommes pas limités et ce à cause des constantes. Et cela peut générer des problèmes lorsque l'on tente de changer une DC lors de la réparation. En effet, si notre DC est de la forme $\varphi = \neg(t.A > 10)$ doit-on considérer toutes les variations possibles ($\varphi' = \neg(t.A > 11)$,$\varphi'' = \neg(t.A > 12)$) ? Et malgré que cela puisse être un problème, ce n'est jamais abordé dans l'article [4].

Concernant les variations de contraintes, pour les auteurs de l'article, modifier un prédicat consiste à le supprimer puis le remplacer par un autre différent. Ce concept comporte un problème : toutes les modifications ne devraient pas être traités de la même façon. Par exemple si notre prédicat est de $(A, <)$ il devrait être moins couteux de le modifier en $(A, \leq)$ que de le modifier en $(A, \geq)$. En effet, pour modifier $\{<\}$ en $\leq \equiv \{<, =\}$, il suffit d'ajouter l'opérateur $=$. Pour modifier $\{<\}$ en $\geq \equiv \{>, =\}$ il faut ajouter $=$ mais aussi ajouter $>$, et retirer $<$. Notre système de treillis reflète mieux le fait que la seconde modification est plus grande et donc plus couteuse.

Pour ce qui est de la réparation de donnée, l'article de référence propose l'utilisation des variables fraiches fv sans discuter des conséquence que l'introduction de telles variables peut avoir sur la théorie. En effet, à aucun moment il ne mentionne le fait que $fv \neq a$ et $fv = a$ sont tous les deux faux.

Chapitre 6

Implémentation

Nous avons décidé d'implémenter l'algorithme θ-tolérant en utilisant les différents concepts vus lors des chapitres précédents pour tester son efficacité. Dans ce chapitre, nous expliquerons comment nous avons implémenté les différentes notions. Nous ne donnerons pas toutes les explications du code car le code joint avec ce mémoire comporte des commentaires suffisamment explicites. Nous détaillerons les idées générales, les choix d'implémentations tels que le langage choisi, le format des bases de données, ... Nous terminerons par une discussion à propos de l'efficacité de l'algorithme, du choix de la valeur de θ , etc.

6.1 Choix de langage et format de base de données

Le langage de programmation que nous avons choisi est le Python et le développement de l'outil s'est fait avec *Jupyter Notebook*, une application web et open-source qui permet de créer et partager du code , des équations, etc. Cet outil est utilisé entre autres pour la visualisation de données, le machine learning, des simulations, transformation et nettoyage de données, etc. Il supporte plus de 40 langages de programmation incluant entre autre le Python, le Scala et R.

Le langage choisi, le python a l'avantage d'être plus facilement lisible. Puisque le but de l'implémentation est de regarder si l'algorithme est performant et fonctionne bien, un code lisible est nécessaire. Le débug est plus facile que d'autres langages comme le C++ et le Java. Puisqu'une interface utilisateur graphique n'est pas nécessaire, le python se présentait comme étant une très bonne solution.

La première chose dont nous avons besoin est de bases de données. Le choix a été fait de les stocker dans des fichiers $.txt$. L'avantage de ce format est que ne nous sommes soumis à aucune contrainte particulière d'un SGBD, nous pourrons donc stocker des variables fraiches, des entiers et des chaines de caractères comme bon nous semble. Le but n'étant pas d'utiliser l'algorithme développé pour faire des réparations concrètes et réelles, nous n'avions donc aucunement besoin de choisir un autre format. Les fichiers textes doivent être formatés de la façon suivante :

— La première ligne contient le nom des attributs séparés chacun d'un espace.
— La seconde ligne contient le type de variable parmi str , int et $float$.
— Chaque ligne qui suit contient un tuple de la base de données. Chaque attribut est séparé avec un espace.

Parmi les bases de données , nous retrouvons bien évidement la relation de la table 2.1.

6.2 Code

6.2.1 Classe Predicate et DC

Le premier choix fut de faire deux classes pour représenter les prédicats et les denial constraint.

6.2.1.1 Classe Predicate

La classe prédicat représente donc un prédicat comme nous l'avons défini, il y a donc un attribut et un opérateur pour ce prédicat. Parmi les opérateurs, nous ne retrouvons que les 8 opérateurs suivants : $<, >, =, \neq, \geq, \leq, \top$ et $\perp$.

Nous avons plusieurs fonctions pour manipuler les prédicats. Parmi ces fonctions nous retrouvons :

— La fonction is_true($row1$,$row2$) permet de savoir si deux tuples $row1$ et $row2$ respecte le prédicat.
— La fonction get_weaker() permet d'obtenir un prédicat plus faible, i.e une variation du prédicat en montant dans le treillis.
— La fonction get_weaker_not_empty() permet d'obtenir un prédicat plus faible mais en excluant l'opérateur $\perp$
— ...

6.2.1.2 Classe DC

La classe DC est une collection(liste python) de prédicats. Une DC est initialisée avec un opérateur $\top$ pour tout les prédicats.

Nous avons plusieurs fonctions pour manipuler les DCs. Parmi ces fonctions nous retrouvons :

— La fonction get_violations(data) permettant de savoir quelles sont les couples de tuples de $data$ qui viole la DC.
— La fonction implies(φ) : permet de savoir si la DC courante est implique la DC φ
— La fonction is_trivial() permettant de savoir si une DC est triviale ou pas.
— La fonction get_varation qui permet d'avoir toutes les variations plus faible que la DC courante. Cette fonction est implémentée comme nos algorithme 2 et 3.

6.2.2 Réparation

Chapitre 7

Conclusion

TODO

Bibliographie

[1] Description des données du registre national et du registre bcss. https://www.ksz-bcss.fgov.be/sites/default/files/assets/services_et_support/cbss_manual_fr.pdf. accessed : 2018-02-15.

[2] ics relational database model. http://databasemanagement.wikia.com/wiki/Relational_Database_Model. Accessed : 2018-02-13.

[3] Zhang Aoqian, Song Shaoxu, Wang Jianmin, and S. Yu Philip. Time series data cleaning : From anomaly detection to anomaly repairing. *PVLDB*, 10(10) :1046–1057, 2017.

[4] Song Shaoxu, Zhu Han, and Wang Jianmin. Constraint-variance tolerant data repairing. In *SIGMOD Conference*, pages 877–892. ACM, 2016.

[5] Chu Xu, F. Ilyas Ihab, and Papotti Paolo. Discovering denial constraints. *PVLDB*, 6(13) :1498–1509, 2013.