Università degli Studi di Salerno

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

Intelligenza Artificiale

UN ALGORITMO PER L'INFERENZA DI DIPENDENZE FUNZIONALI RILASSATE

Professori

Prof. Vincenzo Deufemia

Prof. Giuseppe Polese

Studenti

Raffaele Ceruso

Giovanni Leo

Anno Accademico 2017-2018

Indice

Elenco delle tabelle

CAPITOLO 1

Introduzione

1.1 Incipit

Nella progettazione di una base di dati ci sono aspetti essenziali da prendere in considerazione per assicurare un servizio quanto più efficiente possibile. Uno di questi servizi è certamente la *qualità dei dati*, una base di dati con questa caratteristica farà si che le inconsistenze tra i dati siano il minor numero possibile. Negli ultimi anni la crescita delle reti ha portato ad un aumento considerevole del flusso di dati rendendo la *data quality* una materia estremamente interessante vista la cospicua presenza di dati "sporchi" proveniente da fonti differenti. Per ridurre questo tipo di anomalie è impensabile tentare di eliminare le *inconsistenze* manualmente, una procedura di questo tipo può essere facilmente incline ad errori sopratutto con la quantità di dati precedentemente citata. In questo lavoro ci vengono incontro le *Dipendenze funzionali*, utilizzate ampiamente per stabilire vincoli di integrità tra i dati e ridurre anomalie e inconsistenze all'interno della nostra base di dati. La grande mole di dati, però, ha reso necessario un riadattamento delle dipendenze funzionali rendendole in grado di catturare inconsistenze più ampie nei dati. Le *Dipendenze funzionali rilassate o approssimate* **(RFD)**

sono da considerarsi come una naturale evoluzione o generalizzazione delle *dipendenze funzionali canoniche*. Questo nuovo strumento ci permette di adattare le semplici dipendenze funzionali a diversi contesti applicativi, infatti, le RFD possono applicarsi anche solo ad una porzione di database. Il concetto più importante introdotto dalle RFD è quello della *similarità*. Nelle dipendenze funzionali classiche esisteva soltanto il concetto di uguaglianza tra dati, nelle RFD espandiamo questo concetto ad una similarità, questo ci permetterà di coprire una quantità di dati maggiore. Tuttavia le RFD possono fornire vantaggi solo se possono essere scoperte automaticamente dai dati. Il lavoro di tesi si è basato su questo ultimo concetto di ottenere le RFD in seguito ad una procedura automatizzata. Durante le varie fasi di studio si è pensato ed implementato un algoritmo che permette, attraverso tre fasi intermedie, la scoperta di RFD di un dataset dato come input. Le tre fasi di questo algoritmo sono: *Feasibility, Minimality, Generation*. Per questo algoritmo, particolare attenzione è stata posta sull'efficienza, oltre che sull'efficacia, studiando un'implementazione basata sul multithreading e predisponendola ad eventuale adattamento parallelo.

1.2 Nozioni Preliminari

Prima di esporre le RFD è necessario introdurre alcuni concetti preliminari.

1.2.1 Schema di relazione

Uno schema di relazione è costituito da un simbolo R, detto nome della relazione, e da un insieme di attributi $X = \{A_1, A_2, ..., A_n\}$, di solito indicato con $R(X)$. A ciascun attributo $A \in X$ e associato un dominio $dom(A)$. Uno schema di base di dati è un insieme di schemi di relazione con nomi diversi:

$$R = \{R_1(X_1), R_2(X_2), \ldots, R_n(X_n)\}.$$

Una relazione su uno schema $R(X)$ è un insieme r di tuple su X. Per ogni istanza $r \in R(X)$, per ogni tupla $t \in r$ e per ogni attributo $A \in X$, $t[A]$ rappresenta la proiezione di A su t. In modo analogo, dato un insieme di attributi $Y \subseteq X$, $t[Y]$ rappresenta la proiezione di Y su t.[1]

Matricola	Cognome	Nome	Data di nascita
123456	Rossi	Maria	25/11/1991
654321	Neri	Anna	23/04/1992
456321	Verdi	Fabio	12/02/1992

Tabella 1.1: Esempio di schema di relazione

1.2.2 Dipendenze funzionali canoniche

Una *dipendenza funzionale*, abbreviata in FD, è un vincolo di integrità semantico per il modello relazionale che descrive i legami di tipo funzionale tra gli attributi di una relazione.

Data una relazione r su uno schema $R(X)$ e due sottoinsiemi di attributi non vuoti Y e Z di X, diremo che esiste su r una dipendenza funzionale tra Y e Z, se, per ogni coppia di tuple t_1 e t_2 di r aventi gli stessi valori sugli attributi Y, risulta che t_1 e t_2 hanno gli stessi valori sugli attributi Z:

$$\forall t_1, t_2 \in r, t_1[Y] = t_2[Y] \implies t_1[Z] = t_2[Z] \tag{1.1}$$

Una dipendenza funzionale tra gli attributi Y e Z viene indicata con la notazione $Y \to Z$ e viene associata ad uno schema.

Se l'insieme Z è composto da attributi $A_1, A_2, \ldots, A_k$, allora una relazione soddisfa $Y \to Z$ se e solo se essa soddisfa tutte le k dipendenze $Y \to A_1$, $Y \to A_2$,..., $Y \to A_k$. Di conseguenza, quando opportuno, possiamo assumere che le dipendenze abbiano la forma $Y \to A$, con A singolo attributo.

Una relazione funzionale è *non banale* se A non compare tra gli attributi di Y.

Data una chiave K di una relazione r, si può facilmente notare che esiste una dipendenza funzionale tra K ed ogni altro attributo dello schema di r. Quindi una dipendenza funzionale $Y \to Z$ su uno schema $R(X)$ degenera nel vincolo di chiave se l'unione di Y e Z è pari a X. In tal caso Y è superchiave per lo schema $R(X)$.

Con la notazione $\langle R(X), F\rangle$ indicheremo uno schema $R(X)$ su cui è definito un insieme di dipendenze funzionali F. Un'istanza r di $R(X)$ viene detta *istanza legale* di $\langle R(X), F\rangle$ se soddisfa tutte le dipendenze funzionali in F. Infine, data una relazione funzionale $Y \to Z$, se ogni istanza legale r di $\langle R(X), F\rangle$ soddisfa anche $Y \to Z$, allora diremo che F *implica logicamente* $Y \to Z$, indicato come $F \models Y \to Z$.

1.2.3 Dipendenze funzionali rilassate

In alcuni casi per risolvere dei problemi in alcuni di domini di applicazioni, come l'identificazione di inconsistenze tra i dati, o la rilevazione di relazioni semantiche fra i dati, è necessario rilassare la definizione di dipendenza funzionale, introducendo delle approssimazioni nel confronto dei dati. Invece di effettuare dei controlli di uguaglianza, si utilizzano dei controlli di similarità. Inoltre spesso si potrebbe desiderare che una certa dipendenza valga solo su un sottoinsieme di tuple che su tutte. Per questo motivo sono nate delle dipendenze funzionali che rilassano alcuni dei vincoli delle FD,

prendono il nome di Dipendenze Funzionali Rilassate o Approssimate [1]. Esistono differenti tipi di RFD, ciascuna di esse rilassa uno o più vincoli delle FD, si possono dividere in due macro aree:

1. Confronto di attributi: La funzione di uguaglianza delle FD canoniche viene sostituita da una funzione di similarità, ciò implica che l'AFD deve descrivere una soglia di rilassamento per ogni attributo.

2. Estensione: Permette che il vincolo non sia valido su tutte le tuple, ma solo su di un sottoinsieme di esse.

Le RFD sono utilizzate in attività di: data cleaning, record matching e di rilassamento delle query. Le definizione formale di una RFD è la seguente:

Teorema 1 *Sia R uno schema relazionale definito su di un insieme di attributi finito, e sia $R = (A_1, A_2, ..., A_k)$ uno schema relazionale definito su R. Una RFD φ su R viene rappresentata come:*

$$D_c : (X)_{\Phi_1} \xrightarrow{\Psi(X,Y) \leq \epsilon} (Y)_{\Phi_2}$$

dove:

- $\mathbb{D}_c = \{(t) \in dom(R) | (\bigwedge_{i=1}^{k} c_i(t[A_i]))$, *dove* $c = (c_1, ..., c_k)$ *con* c_i *è un predicato sul* $dom(A_i)$, *utilizzato per filtrare le tuple a cui* φ *va applicata;*

- $X, Y \subseteq attr(R)$ *tali che* $X \cap Y = 0$*;*

- Φ_1*(*Φ_2 *rispettivamente) è un insieme di vincoli* $\phi[X](\phi[Y])$ *definito sull'attributo* X *e* (Y *rispettivamente). Per qualsiasi coppia di tuple* $(t_1, t_2) \in \mathbb{D}_c$ *il vincolo*

[1] RFD abbreviazione di Relaxed Functional Dependency.

$\phi[X](\phi[Y]$ *rispettivamente*) *restituisce vero se la similarità fra* t_1 *e* t_2 *sugli attributi* X *e* (Y *rispettivamente*) *concordano con i vincoli specificati da* $\phi[X](\phi[Y]$ *rispettivamente*);

- Ψ : *rappresenta una misura di copertura su* $\mathbb{D}_c$ *e indica il numero di tuple che violano o soddisfano* φ;
- ϵ *è la soglia che indica il limite superiore o inferiore per il risultato della misura di copertura;*

Nel lavoro di tesi vengono trattate solo le RFD che rilassano il vincolo di uguaglianza. Data RFD $X \rightarrow Y$ essa vale su una relazione r se e solo se la distanza fra due tuple t_1 e t_2, i cui valori sui singoli attributi A_i non superano una certa soglia β_i, è inferiore ad una certa soglia a_A su ogni attributo $A \in X$, allora la distanza fra t_1 e t_2 su ogni attributo $B \in Y$ è minore di una certa soglia a_B.
La struttura delle RFD utilizzate è la seguente:

$$attr_1(\leq soglia_1), \ldots, attr_n(\leq soglia_n) \rightarrow RHS$$

Gli attributi che si trovano a sinistra della freccia costituiscono la parte LHS[2], l'attributo che invece si trova dopo la freccia costituisce l'RHS[3]. È importante focalizzare l'attenzione su questo concetto in quanto le dipendenze funzionali hanno un verso, ed è quello indicato dalla freccia. Qualsiasi operazione effettuata con le RFD deve sempre tener conto del verso, le RFD non forniscono conoscenza nel verso opposto.

[2]Left Hand Side o lato sinistro.
[3]Right Hand Side o lato destro.

Questa non è una proprietà riguardante solo le RFD, bensì riguarda qualsiasi tipo di dipendenza funzionale. Ad esempio consideriamo la relazione in questa tabella:

Impiegato	Stipendio	Progetto	Bilancio	Funzione
Rossi	20000	Sito web	2000	tecnico
Verdi	35000	App Mobile	15000	progettista
Verdi	35000	Server	15000	progettista
Neri	55000	Server	15000	direttore
Neri	55000	App Mobile	15000	consulente
Neri	55000	Sito web	2000	consulente
Mori	48000	Sito web	15000	direttore
Mori	48000	Server	15000	progettista
Bianchi	48000	Server	15000	progettista
Bianchi	48000	App Mobile	15000	direttore

Tabella 1.2: Esempio di Relazione con anomalie

Si può osservare che lo stipendio di ciascun impiegato è unico, quindi in ogni tupla in cui compare lo stesso impiegato verrà riportato lo stesso stipendio. Possiamo dire che esiste una Dipendenza Funzionale: $Impiegato \rightarrow Stipendio$. Si può fare lo stesso discorso tra gli attributi Progetto e Bilancio, quindi anche qui abbiamo una dipendenza funzionale $Progetto \rightarrow Bilancio$. Non si può dire che di conseguenza vale anche il verso opposto:

$$Impiegato \rightarrow Stipendio \neq Stipendio \rightarrow Impiegato$$

Infatti percepiscono 48000 di stipendio sia Mori che Bianchi.[1]

1.2.4 Scoperta di RFD

Data una relazione r, la scoperta di una RFD è il problema di trovare un *minimal cover set* di RFD che si verificano per r. Questo problema rende ancor più complesso il problema della scoperta delle dipendenze dei dati visto l'ampio spazio di ricerca dei

possibili vincoli di similarità. Dunque è necessario trovare algoritmi efficienti in grado di estrarre RFD con vincoli di similarità significativi.

Se i vincoli di similarità e le soglie sono noti per ogni attributo del dataset, scoprire le RFD si riduce a trovare tutte le possibili dipendenze che soddisfano la seguente regola:

Lemma 1 *Le partizioni di tuple che sono simili sugli attributi contenuti nel lato sinistro o LHS della dipendenza, devono corrispondere a quelle che sono simili nel lato destro o RHS.*

Questo problema è simile a trovare le FD, dove bisogna trovare le partizioni di tuple che condividono lo stesso valore sull'RHS quando esse condividono lo stesso valore sull'LHS. Il problema viene reso più semplice dal fatto che, nel caso della scoperta delle FD, tali partizioni sono disgiunte, cosa che però non vale nelle RFD in quanto uno stesso valore può essere simile a valori differenti. Ciò impedisce quindi di sfruttare gli algoritmi utilizzati nella scoperta delle FD, nella scoperta delle RFD

1.2.5 Dominanza

Nel corso del nostro lavoro, abbiamo applicato alcuni risultati dell'intelligenza artificiale al campo della discovery delle *Dipendenze Funzionali Rilassate*. In particolare, cercando di individuare le dipendenze funzionali rilassate ci siamo serviti di un importante risultato nella succitata materia: la dominanza stretta *(strict dominance)*.

Teorema 2 *Dato un vettore di attributi* $\mathbf{X} = X_1, X_2, \ldots, X_n$*, siano* $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ *e* $\mathbf{y} = (y_1, y_2, \ldots, y_n)$ *due vettori di assegnamenti definiti sugli attributi di* $\mathbf{X}$*, dove l'i-esimo elemento* x_i *o* y_i *può essere sia un valore numerico sia un valore discreto con un assunto ordinamento su tali valori. Diremo che* $\mathbf{x}$ *domina strettamente (o*

deterministicamente) $\mathbf{y}$ *se e solo se*

$$y_i \leq x_i \quad i = 1, 2, \ldots, n,$$

ovvero

$$\mathbf{y} - \mathbf{x} \leq \mathbf{0}$$

1.3 Studi preliminari

Prima di iniziare lo sviluppo dell'algoritmo per la scoperta di RFD si è reso necessario uno studio approfondito di un algoritmo precedentemente sviluppato come progetto del corso di Intelligenza Artificiale [2]. Tale algoritmo è stato sviluppato in Python, pertanto, abbiamo effettuato uno studio del linguaggio precedentemente citato. Oltre le principali caratteristiche di questo linguaggio, è stato fatto uno studio anche delle librerie utilizzate all'interno del progetto:

- ***Pandas***: È una libreria che include delle strutture dati e tool di analisi facili da usare e fortemente ottimizzate.
- ***Numpy***: È un package dedicato all'elaborazione scientifica sul linguaggio Python.

Una volta concluso questo tipo di studio si è cominciato a pensare allo sviluppo dell'algoritmo in un ambiente differente. La scelta è ricaduta su *Java*, tale scelta è dovuta, oltre che alla già piena conoscenza del team di questo linguaggio, alla potenza e versatilità che questo linguaggio offre, oltre che al gran numero di framework presenti per la gestione di parallelizzazione e concorrenza, essendo quest'ultimo un aspetto molto importante per l'efficienza dell'algoritmo. Le librerie esterne studiate ed utilizzate

saranno ben approfondite nel capitolo 4 (*Implementazione*) di questo elaborato. Le sopracitate librerie esterne utilizzate sono:

- ***MapDB***: Un open source database engine basato su java con prestazioni simili alle collection presenti nel package **java.util**.
- ***Joinery Dataframe***: Struttura dati simile al dataframe presente in *Pandas* di Python.

All'infuori delle conoscenze legate ai linguaggi di programmazione, è stato necessario leggere e studiare vari documenti legati al mondo delle dipendenze funzionali.

CAPITOLO 2

Stato dell'Arte

Esistono svariati metodi per scoprire le RFD data una determinata soglia ϵ, un esempio è il metodo *top-down*.

I metodi di discovery *top-down* effettuano una generazione di possibili FD livello per livello e controllano se queste si verificano. L'algoritmo inizia generando un grafo di attributi, con una struttura a lattice, dove vengono considerati tutti i possibili sottoinsiemi di attributi. Dato uno schema relazionale $R = (A_1, A_2, \cdots, A_n)$, il livello 0 del lattice non contiene nessun attributo, il livello 1 contiene tutti i singleton dei singoli attributi dello schema relazionale R, il livello due tutte le possibili coppie di attributi in R fino ad arrivare all'ultimo livello, l'n-esimo, che contiene un unico insieme con tutti gli attributi di R al suo interno. Ogni sottoinsieme contenuto nel lattice rappresenta un candidato per una possibile FD.

Generato il lattice, l'algoritmo parte dal livello 0 fino ad arrivare all'ultimo, e per ogni livello verifica, per tutti i possibili sottoinsiemi $X \in L_r$[1], l'esistenza di possibili dipendenze funzionali. Nello specifico, per ogni attributo $A \in X$ si cerca di verificare se la FD $X \setminus \{A\} \rightarrow A$ vale. Per ridurre il tempo di esecuzione esponenziale, assieme

[1] livello r-esimo

alla verifica avviene una potatura del grafo sfruttando la scoperta di nuove FD.
Inoltre negli ultimi anni c'è stata una proliferazione delle RFDs di cui solo alcune di loro erano dotate dell'algoritmo per la scoperta dai dati. Mostriamo adesso alcune di esse[3].

2.1 AFD Discovery

Una *dipendenza funzionale approssimata*(AFD) è una canonica FD che deve essere soddisfatta da 'più' tuple, piuttosto che 'tutte', di una relazione r. In altre parole, una AFD permette a una piccolissima porzione di tuple di r di violarla. Diversi approcci sono stati proposti per calcolare il grado di soddisfacibilità di una AFD. Gli approcci principali sono basati su una piccola porzione di tuple $s \subset r$ per decidere se una AFD esiste su r. Come conseguenza, le AFDs che esistono su s possono anche esistere su r, con una data probabilità. Alcuni metodi sfruttano la misurazione dell'errore della super chiave per determinare la soddisfacibilità approssima delle AFDs.

2.2 MD Discovery

Matching dependencies(MDs) sono delle RFD proposte recentemente per l'object identification. Sono definite in termini di predicati di similarità per adeguarsi agli errori e a differenti rappresentazioni di dati in sorgenti inaffidabili. Infatti è stato proposto un algoritmo che ha a che fare con la valutazione dell' utilità delle MDs in una data istanza di un database e la determinazione del pattern di similitudine delle MDs. L'utilità è misurata considerando la convenienza e il sostegno delle MDs, mentre le soglie sono

determinate in base alla distribuzione statistica dei dati. Inoltre sono state introdotte delle strategie di Pruning per filtrare i pattern con un basso sostegno.

2.3 DD Discovery

Differential dependencies(DDs) sono delle RFD che specificano vincoli sulla differenza dei valori degli attributi invece delle corrispondenze esatte delle FD canoniche. Il discovery delle DDs eredita la complessità esponenziale dal problema del discovery delle FD.

Un algoritmo per il discovery delle DDs si basa sugli algoritmi di riduzione, il quale una volta fissate le funzioni di differenza per l' RHS per ogni attributo della relazione r, l'insieme delle funzione di differenza per gli LHS ridotti viene cercato per formare le DDs. Le strategie di pruning sono state proposte per migliorare le performance del discovery.

Un algoritmo alternativo riduce lo spazio di ricerca per mezzo di limiti superiori alle soglie di distanza per gli intervalli di LHS specificati dall' utente. Un ulteriore proposta per il DD discovery è un algoritmo che estrae un minimal cover di DDs, basato su regole di associazione. In particolare l'algoritmo estrae una classe di regole di associazione non ridondanti le quali verranno trasformate in DDs.

Infine è stato proposto un algoritmo per ottenere delle soglie adatte per una data DD. In particolare data una istanza di un database e una DD su di esso, l'algoritmo determina le soglie di distanza per la DD al fine di massimizzare la sua utilità.

CAPITOLO 3

Algoritmo

In questo capitolo saranno mostrati i passi da effettuare per ottenere, partendo da un dataset rappresentante una relazione, una lista di dipendenze funzionali rilassate. La sequenza di passi che l'algoritmo affronterà sono:

- ***Feasibility***
- ***Minimality***
- ***Generation***

Per fare in modo che la prima fase(*Feasibility*) abbia inizio, ci dobbiamo creare la matrice delle distanze, che, insieme ad alcune informazioni aggiuntive verranno date in input alla suddetta fase. In seguito si passerà alla fase di *Minimality* la quale fornirà l'input per l'ultima fase ovvero, *Generation*. In questo capitolo verrà descritta nello specifico la fase di *Generation* che è oggetto di questo lavoro di tesi.

3.1 Matrice delle distanze

Il primo passo consiste nel calcolo delle distanze tra ogni coppia di tuple del dataset. Questo passaggio viene fatto utilizzando diverse funzioni di distanza, a seconda del tipo di RFD che si vuole ricavare.

3.1.1 Funzione di distanza

Date due tuple t_i e t_j tali che $t_i, t_j \in \mathbf{r}$, risulta necessario definire un metodo per capire quanto queste siano distanti tra di loro. Definiamo quindi una funzione

$$\mathbf{d} : \mathbf{r} \to \mathbb{R}^{k+1}$$

tale che, date due tuple $t_i, t_j \in \mathbf{r}$,

$$\mathbf{d}(t_i, t_j) = [(d_1, d_2, ..., d_{k+1}) \mid d_p = d(t_i[p], t_j[p])],$$

dove $t_i[p]$ rappresenta il p-esimo elemento della tupla t_i e $d(x, y)$ è una funzione di distanza tra gli elementi x e y che cambia in base al tipo di elemento.
Una funzione $\mathbf{d}(t_i, t_j)$ può quindi essere ibrida, ossia composta da diverse distanze su attributi di tipi diversi: per confrontare tuple composte da valori misti (e.g. numerici, stringhe e date) è necessario utilizzare diverse distanze. Quindi, una generica funzione distanza $\mathbf{d}$ operante su due tuple, è composta al suo interno da altre funzioni di distanza d_t, anche diverse tra di loro, operanti su singoli attributi.
Durante la progettazione del nostro algoritmo abbiamo considerato le seguenti funzioni distanza, in base ai diversi tipi di elemento che potevano far parte di una tupla:

3.1.1.0.1 Distanza euclidea mono-dimensionale tale funzione viene utilizzata quando x e y sono entrambi tipi numerici:

$$d_{eucl}(x, y) = \sqrt{(x - y)^2} = |x - y|. \tag{3.1}$$

3.1.1.0.2 Distanza di Levenshtein è una metrica per misurare la differenza fra due stringhe (o sequenze in generale). Tale distanza indica il minimo numero di caratteri singoli che è necessario modificare (e.g. inserimenti, cancellazioni, sostituzioni) per far sì che le due stringhe siano uguali. È quindi definita come:

$$d_{lev}(x, y) = lev_{x,y}(|x|, |y|) \tag{3.2}$$

dove

$$lev_{a,b}(i, j) = \begin{cases} \max(i, j) & \text{se } min(i, j) = 0 \\ \min \begin{cases} lev_{a,b}(i - 1, j) + 1 \\ lev_{a,b}(i, j - 1) + 1 \\ lev_{a,b}(i - 1, j - 1) + 1_{(a_i \neq b_j)} \end{cases} & \text{altrimenti} \end{cases}$$

dove $1_{(a_i \neq b_j)}$ è una *funzione indicatrice*, uguale a 0 se $a_i = b_j$ e uguale a 1 altrimenti, e $lev_{a,b}(i, j)$ è la distanza di Levenshtein tra i primi i caratteri di a e i primi j caratteri di b;

3.1.1.0.3 Distanza tra date quando gli attributi da confrontare sono di tipo data (a prescindere dal formato di questa), viene utilzzata una semplice funzione di distanza

che permette di calcolare i giorni trascorsi dalla data meno recente a quella più recente.

3.1.2 Calcolo della matrice delle distanze

Dal confronto tra tutte le coppie di tuple della relazione ricaviamo una tabella delle distanze DT tale che $|DT| = \binom{n}{2} = \frac{n(n-1)}{2}$.

La tabella delle distanze (supponendo senza perdere di generalità di utilizzare l'attributo X_{k+1} come attributo RHS) ricavata è quindi:

	RHS	X_1	X_2	$\dots$	X_k
1	d_1^{RHS}	d_{11}	d_{12}	$\dots$	d_{1k}
2	d_2^{RHS}	d_{21}	d_{22}	$\dots$	d_{2k}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
m	d_m^{RHS}	d_{m1}	d_{m2}	$\dots$	d_{mk}

dove $m = \frac{n(n-1)}{2}$ e ogni riga rappresenta il vettore distanza ottenuto confrontando due diverse tuple facenti parte della relazione, ad esempio, date due tuple $t_r = (x_1, x_2, \dots, x_{k+1})$ e $t_l = (y_1, y_2, \dots, y_{k+1})$ tali che $t_r \in \mathbf{r}$ e $t_l \in \mathbf{r}$, calcoliamo la distanza tra queste due tuple e poniamo il vettore

$$d(t_r, t_l) = [d(x_1, y_1), d(x_2, y_2), \dots, d(x_{k+1}, y_{k+1})]$$

in $DT[i]$. Il generico d_{ij} rappresenta la distanza sull'attributo X_i calcolato su una coppia di tuple.

La colonna

$$DT[\cdot]['RHS'] = d_1^{RHS} d_2^{RHS} : d_m^{RHS}$$

rappresenta la colonna delle distanze tra tutte le n tuple rispetto all'attributo RHS.

Dopo aver calcolato la tabella DT, occorre ordinarla utilizzando come pivot la colonna $DT[\cdot]['RHS']$, ottenendo il seguente data frame:

	RHS	X_1	X_2	$\ldots$	X_k
1	d_i^{RHS}	d_{i1}	d_{i2}	$\ldots$	d_{ik}
2	d_j^{RHS}	d_{j1}	d_{j2}	$\ldots$	d_{jk}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
m	d_r^{RHS}	d_{r1}	d_{r2}	$\ldots$	d_{rk}

dove $d_i^{RHS} \leq d_j^{RHS} \leq \ldots \leq d_r^{RHS}$.

A questo punto occorre dividere e raggruppare tutte le righe del data frame: le righe aventi lo stesso valore($ClusterID$) sulla colonna $DT[\cdot]['RHS']$ saranno raggruppate in sezioni di data frame disgiunte. Questo data frame finale può finalmente essere utilizzato dalle successive fasi dell'algoritmo.

3.2 Feasibility

Una volta creato il dataframe contenente la matrice delle distanze, siamo pronti per l'avvio del *Feasibility test*. Questa fase prende in input la precedentemente citata matrice con un determinato campo che sarà il nostro *RHS*.

Prima di espletare il da farsi, diamo una prima occhiata allo pseudocodice:

Algorithm 1 Feasibility Test

Input: Matrice delle distanze **D** ordinata di taglia n
Output: Insieme C

```
1: clusterID ← getClusterID(n)+1;
2: actualC ← 0;
3: dominates ← false;
4: for each t_p ∈ D in ordine decrescente do
5:     if getClusterID(t_p) != clusterID then
6:         clusterID ← getClusterID(t_p);
7:         if actualC != 0 then
8:             addSet(C,actualC)                    ▷ inserimento actualC in C
9:         end if
10:    end if
11:    if getClusterID(t_p) == 0 then
12:        break;          ▷ processo stoppato quando si raggiunge il cluster 0
13:    end if
14:    if actualC == 0 then
15:        addTuple (actualC, t_p);                 ▷ Per il primo pattern
16:    else
17:        dominates ← false;
18:        for each t'_p ∈ actualC do
19:            if !dominance (t_p, t'_p) then
20:                if dominance (t'_p, t_p) then
21:                    remove (actualC,t'_p);       ▷ t'_p non è pattern minimale
22:                end if
23:            else
24:                dominates ← true;
25:                break;
26:            end if
27:        end for
28:        if !dominates then addTuple(actualC, t_p);
29:        end if
30:    end if
31: end for
32: return C;
```

Come detto nella fase di creazione della matrice delle distanze, tale matrice è stata ordinata per il valore dell'*RHS* e successivamente sono stati raggruppati i pattern con valori uguali di tale campo(*Cluster*)[1]. Lo scopo di questa fase è l'ottenimento di insiemi contenenti pattern che superino il test di Feasibility, tali insiemi saranno costruiti uno

[1]Per questo algoritmo ci riferiremo a ClusterN come il gruppo di pattern aventi N come valore di RHS

per ogni valore differente di RHS. Come si evince dallo pseudocodice l'analisi ha inizio con il clusterN con N di valore massimo. Per questo caso l'insieme viene inizializzato con l'ultima riga trovata, essendo l'insieme ancora vuoto.

Analizzando l'esempio di dataset mostrato nella sezione precedente **??**, per *RHS=ShoeSize* avremo la seguente situaizione iniziale:

$$C_3 : < 4, 5 >$$

L'analisi per questo cluster proseguirà con le altre righe, esse saranno inserite nell'insieme in costruzione corrente solo se non dominano quelle già presenti. La verifica della dominanza(concetto espletato nell'introduzione), però, è effettuata non solo dalla riga da inserire verso quelle già presenti, tale controllo viene effettuato anche dai pattern esistenti verso la riga che vuole "entrare" nell'insieme. Se il risultato della dominanza rileva che un pattern nell'insieme *domina* la riga in entrata, allora questo viene rimosso. Il duplice controllo ci garantirà che in uno stesso cluster non ci siano pattern che dominano altri. Una volta finita l'analisi di un cluster, l'algoritmo avrà l'insieme corrispondente e sarà pronto ad analizzare quello successivo. Per effettuare questo cambio, l'insieme corrispondente verrà inizializzato con i pattern facente parte dell'insieme ottenuto in precedenza. Questa analisi verrà conclusa o con la fine dei cluster disponibili oppure quando si giunge ad un valore di *RHS uguale a 0*.

Facendo riferimento sempre allo stesso esempio **??**, per *RHS=ShoeSize* avremo uno stato di terminazione come questo:

$$C_3 : < 4, 5 >$$

$$C_2 : < 4, 6 >, < 3, 5 >, < 2, 5 >, < 0, 5 >, < 1, 4 >$$

$$C_1 < 1, 5 >, < 0, 6 >, < 2, 6 >, < 0, 4 >, < 1, 3 >, < 1, 2 >, < 0, 1 >$$

Infine, avremo ottenuto un insieme di pattern per ogni cluster, tali insiemi sono definiti **insiemiC**. A questo punto, con il nostro output, potremo procedere verso le fasi di *Minimality e Generation*.

3.3 Minimality e Generation

Le fasi di *Minimality* e *Generation* sono le ultime due fasi dell'algoritmo. La fase di *Minimality* prendendo come input gli *insiemiC* dalla fase di *Feasibility* genererà sottoinsiemi di pattern minimali. L'output di tale fase sarà input della fase di *Generation* la quale ricaverà, da questi sottoinsiemi, le **RFD**.

3.3.1 Nozioni preliminari

Dalla fase di *Feasibility* si otterranno per ogni RHS un certo numero di insiemi C_i (con $i = 1, ..., m$) dove ogni C_i sarà composto da un numero di pattern P_j (con $j = 1, ..., h$) sull'insieme di attributi (A_1,...,A_n).

3.3.2 Minimality

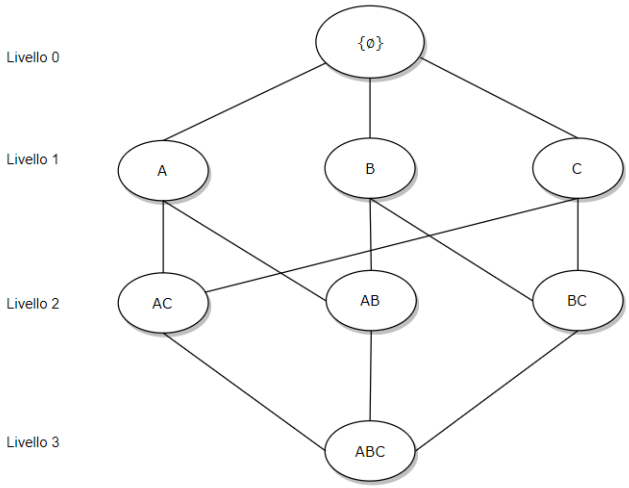
L'obiettivo della fase di *Minimality* è quello di determinare per ogni insieme C_i, generato come output dalla fase di *Feasibility*, quei pattern che siano ammissibili e minimali. Diremo che un sotto-pattern S_k, definito come un insieme di attributi per i quali esistono pattern minimali, è ammissibile se quest'ultimo non domina rispetto a tutti gli altri nell'insieme C_i. Invece un sotto-pattern S_k è minimale se esiste almeno un sotto-pattern di S_k che non è ammissibile. Inizialmente l'algoritmo per ogni pattern P_j calcola le differenze tra i valori di distanza di P_j e i valori di distanza dei restanti pattern P_y (con y =1, 2, ..., h e $y \neq j$). Ad esempio, considerando il caso in

cui il $pattern_j$ sia composto dai valori 6,4,2 rispettivamente per gli attributi A,B,C e $pattern_1 = 3, 2, 5$ fino ad arrivare a $pattern_h = 1, 3, 2$, avremo una situazione simile.

	A	B	C
T_{j1}	3	2	-3
.	.	.	.
.	.	.	.
T_{jh}	5	1	-0

Tabella 3.1: Esempio di matrice per le distanze tra P_j e P_y in C_i

Una volta calcolati tali valori e salvati in una matrice $M_{(h,n)}$ l'algoritmo, a partire da quest'ultima, esegue la ricerca dei sotto-pattern minimi seguendo (virtualmente) una struttura denominata *Lattice*. Dato un generico insieme S di n elementi {A,B,C}, il *Lattice* è una struttura dati ad albero che ha come radice l'insieme vuoto e come suoi figli i singoli attributi A,B,C. Iterativamente ad ogni livello i dell'albero, si avranno tutte le possibili combinazioni di n elementi di classe i.



Per ognuno dei nodi generati l'algoritmo effettua la verifica di minimalità. Definiamo S_k ottimo se valgono le seguenti due proprietà:

Con $|S_k| = 1$:

• $\forall$ valore di $t_{jy} \in T_{jy} \mid t_{jy} \leq 0$.

Con $|S_k| > 1$:

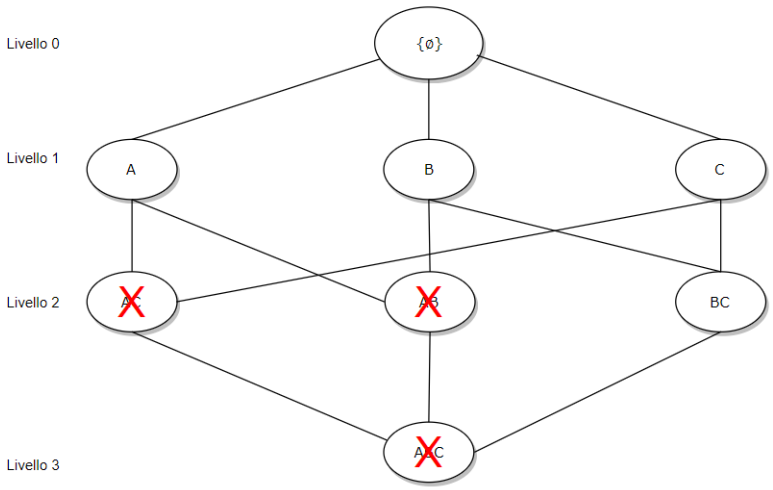
• $\exists$ un valore di $t_{jy} \in T_{jy} \mid t_{jy} < 0$, oppure $\forall\ t_{jy} \in T_{jy}$, $t_{jy} = 0$.

Affinché però si eviti di considerare super-set ammissibili di set minimali oppure, casi in cui nessuna combinazione del sotto-pattern S_k considerato riconduce ad un minimale, applicheremo la tecnica del *Pruning*. Il *Pruning* è una tecnica che riduce le dimensioni degli alberi decisionali rimuovendo sezioni che non riconducono all'ottimo. Definiti tali concetti l'algoritmo, partendo dal livello 1, per ogni nodo dell'albero verifica due condizioni:

1) Tutti i valori di S_k sono positivi.

2) Tutti i valori di S_k sono negativi.

Se si verifica uno solo dei due casi poc'anzi citati l'algoritmo elimina dal *Lattice* tutti gli archi tra il nodo rappresentante S_k e i suoi figli, riducendo così di molto i tempi computazionali. Ritornando all'esempio precedente, con sotto-pattern $S_k = \{A\}$ composto da tutti valori negativi, l'algoritmo elimina gli archi uscenti sui nodi $\{AC\}$, $\{AB\}$, e di conseguenza anche $\{ABC\}$: infine, avendo S_k superato il test di ottimalità, tale sotto-pattern risulta essere minimale.



L'output del *Minimality* sarà dunque, per ogni pattern P_j, i sotto-pattern minimali S_k per quel pattern. Di seguito vengono riportati i pseudocodice dei metodi di MinimalityTest, validate&PrunePatterns e generateSuperseAttribute.

Algorithm 2 generateSuperSetAttributes(Set **LevelAttributes**, Set[] ***columnSet**, Set[] ***columnEqualSet**,)

```
1: Set  newLevelAttributes
2: Set[] newColumnSet
3: Set[] newColumnEqualSet
4: int codIndex = 0
5: column combination newJ
6: for each column combination j in columnSet/columnEqualSet do
7:     for each column combination j' in columnSet/columnEqualSet
8:         starting from j + 1 do
9:         if S thenAMPREFIX(columnSet[j],columnSet[j'])
10:            newJ = GETSUPERSET(columnSet[j],columnSet[j'])
11:            newLevelAttributes.ADD(newJ)
12:            newColumnSet[ccIndex]=columnSet[j].GETELEMENTS()
13:            U columnSet[j'].GETELEMENTS()
14:            newColumnEqualSet[ccIndex]=columnEqualSet[j].GETELEMENTS()
15:            U columnEqualSet[j'].GETELEMENTS()
16:            ccIndex = ccIndex + 1
17:        end if
18:    end for
19: end for
20: columnSet = newColumnSet
21: columnEqualSet = newColumnEqualSet
22: return newLevelAttributes
```

Algorithm 3 Validate&PrunePatterns(Set ***LevelAttributes**, Set ***results**, Set[] **columnSet**, Set[] **columnEqualSet**, int **totElements**)

```
1: for each column combination j in columnSet/columnEqualSet do
2:     if columnSet[j].SIZE() = totElements then
3:         results.ADD(columnSet[j])
4:         LevelAttributes.REMOVE(columnSet[j])
5:     else if columnSet[j].SIZE() + columnEqualSet[j].SIZE() = totElements then
6:         results.ADD(columnSet[j])
7:     else if columnSet[j].SIZE() + columnEqualSet[j].SIZE() = 0 then
8:         results.REMOVE(columnSet[j])
9:     end if
10: end for
```

Algorithm 4 Minimality Test(Set **C**, int **numColumn**, int **numRows**)

```
1: int[][][] diffPatterns = null
2: int Level = 1
3: Set LevelAttributes =
4: Set results = null
5: Set[] columnSet = null
6: Set[] columnEqualSet = null
7: for each Set C_i in C in descending order do
8:     results = null
9:     for each pattern tuple t_p in C_i do
10:        diffPatterns = CREATEDIFFERENCEPATTERNS(t_p)
11:        //fase di inizializzazione
12:        for each row tuple t_j in diffPatterns do
13:            for each column j of t_j do
14:                levelAttributes.ADD(j)
15:            end for
16:            for each value e in column of t_j do
17:                e = diffPatterns.GETELEMENT(j)
18:                if thene = 0
19:                    columnEqualSet[j].ADD(e)
20:                else if e < 0
21:                    columnSet[j].ADD(e)
22:                end if
23:            end for
24:        end for
25:        //fine inizializzazione
26:        //fase di ricerca via lattice
27:        while level<= numColumn do
28:            VALIDATE&PRUNEPATTERNS(*LevelAttributes,*results,
29:            columnSet,columnEqualSet,numRows)
30:            Level = Level + 1
31:            LevelAttributes =
32:            if LevelAttributes.SIZE() = 0 then
33:                break
34:            end if
35:        end while
36:        //fase di ricerca via lattice
37:    end for
38:    GENERATED(results) //fase descritta nella tesi Leo
39: end for
```

CAPITOLO 4

Implementazione

4.1 Tecnologie utilizzate

Al fine di testare l'algoritmo progettato, abbiamo sviluppato una applicazione scritta in Java. La scelta nell'utilizzare Java, come detto negli studi preliminari, è nata dalla necessità di ottenere maggiori prestazioni, dalla potenza del linguaggio e dal gran numero di librerie e framework utilizzabili al nostro scopo. Nella seguente sezione analizzeremo quelle che sono le librerie utilizzate per la realizzazione del seguente progetto.

4.1.1 Executor Service

In Java ci sono servizi molto evoluti denominati **Executor Service**, un framework che semplifica l'esecuzione di task in maniera asincrona. Generalmente parlando l'*ExecutorService* fornisce automaticamente un pool di thread e un API per assegnare task ad esso. La maniera più semplice per creare un *ExecutorService* è utilizzare uno dei metodi della classe **Executors**. Tra i differenti metodi che essa offre sono stati utilizzati

- **Executors.newFixedThreadPool(1)** per l'esecuzione dell'algoritmo in sequenziale.

- **Executors.newCachedThreadPool()** per l'esecuzuine dell'algoritmo in parallelo.

ExecutorService può eseguire dei task di tipo **Runnable**[1] e **Callable<T>**[2] attraverso il metodo **submit()** il quale restituisce un risultato di tipo
Future<T> se viene fatto il submit di un task di tipo **Callable<T>** oppure un risultato del tipo **Future<?>** se viene fatto il submit di un task di tipo **Runnable**. L'oggetto di tipo **Future** è una interfaccia che fornisce un metodo **get()**, il quale permette di ottenere il risultato del task, ed esso è bloccante se il risultato non è ancora pronto altrimenti ritorna subito. Per questa tecnologia si può fare riferimento alla documentazione ufficiale: `https://goo.gl/xTEGZa`

4.1.1.1 CachedThreadPool

Crea un pool di thread espandibile. Nuovi thread vengono creati quando è necessario ed i thread precedentemente creati vengono riutilizzati quando sono disponibili. I thread in Idle vengono mantenuti nel pool per un certo periodo di tempo, questo è il meccanismo più standard per accomodare i picchi di carico. Inoltre nel momento in cui si ha carico abbastanza prevedibile la **CachedThreadPoolSize**[3] rimane fissa.

4.1.1.2 FixedThreadPool

Crea un pool di thread a dimensione fissa. Questo pool garantisce che non ci siano più di un certo numero di thread concorrenti. Se viene sottomesso un task e tutti i thread

[1] Non restituiscono nessun oggetto
[2] Restituiscono un oggetto di tipo T
[3] Il numero di thread presenti nel pool.

sono in esecuzione, quest'ultimo aspetterà che qualche thread diventi disponibile per essere eseguito.

4.1.2 MapDB

MapDB è un open-source (Apache 2.0 licensed), embedded Java database engine e collection framework. Fornisce Maps, Sets, Lists, Queues, Bitmaps. MapDB è probabilmente il database Java più veloce, con prestazioni paragonabili alle collezioni java.util. Fornisce inoltre funzionalità avanzate come transazioni ACID, istantanee, backup incrementali e molto altro. Per questa tecnologia si può fare riferimento alla documentazione ufficiale: `https://jankotek.gitbooks.io/mapdb/content/`

4.1.3 FastUtil

Fornisce una implementazione più efficiente delle pricipali strutture dati di java. Si può fare riferimento a tale libreria sulla documentazione ufficiale: `http://fastutil.di.unimi.it/`.

4.1.4 Joinery Dataframe

Il DataFrame di joinery è una struttura dati simile a dataframe presente nella libreria *pandas* del linguaggio Python. Si può fare riferimento a tale libreria sulla documentazione ufficiale: `https://cardillo.github.io/joinery/`.

4.2 Struttura del progetto

Tutto il progetto è strutturato in 3 *package(Domino,Task,Utility)*. Nel corso della descrizione del codice, assumeremo che il lettore possieda già basi del linguaggio di

programmazione *Java*. Per cui, sarà omessa la differenziazione tra metodi pubblici, metodi privati e metodi statici delle singole classi.

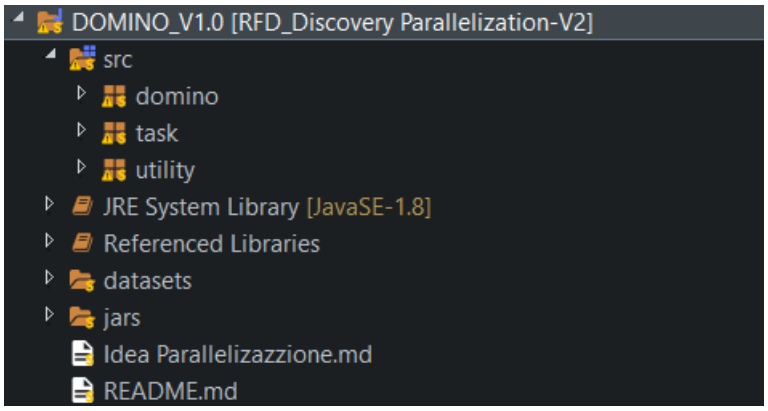


Figura 4.1: Struttura del Progetto

4.2.1 Package Domino

Il contenuto di questo pacchetto è costituito dalle principali classi che compongono la logica dell'algoritmo. Particolare enfasi è stata sottoposta alla classe **Domino.java**.

Figura 4.2: Package Domino

Quest'ultima ha il compito di parallelizzare la fase di **Feasibility** e **Minimality**. In particolare per ogni candidato **RHS** viene associato all'executor un **Feasibility-Task**. Una volta ottenuti gli **insiemiC**, ognuno di essi verrà associato al rispettivo **MinimalityTask**. Ottenuto l'output dai task si procede alla ricombinazione di essi all'interno della **listaCC**.

```
ObjectArrayList<Future<ObjectArrayList<ObjectArrayList<String>>>> result =
        new ObjectArrayList<>();
ObjectArrayList<ObjectArrayList<String>> insiemiC = new ObjectArrayList<>();;

for(int i=0;i<colNumber;i++){

    visitaVettore(vettoreColonna,i,map,debugCode);
    result.add(exec.submit(new FeasabilityTask(i, colNumber, map, vettoreColonna,thresholds[i])));

}

ObjectArrayList<Future<ObjectArrayList<RFDMap>>> resultMinmality = new ObjectArrayList<>();
for(int i=0;i<colNumber;i++){

    insiemiC = result.get(i).get();

    for(int k=0; k<insiemiC.size(); k++) {
        ObjectArrayList<String> insieme = insiemiC.get(k);
        System.out.println("Cardinalità "+k+"-esimo insieme: "+insieme.size());
        for(int k2=0; k2<insieme.size(); k2++)
            System.out.println(insieme.get(k2));
    }
    System.out.print(".");
    resultMinmality.add(exec.submit(new MinimalityTask(insiemiC, colNumber, i, colNames, thresholds, debugCode)));
}

for(int i=0;i<colNumber;i++){
    listaCC.addRFDS(resultMinmality.get(i).get());
}
```

Figura 4.3: Codice Executor

4.2.2 Package Task

Il contenuto di questo pacchetto è costituito dalle classi implementate per l'utilizzo dei task.

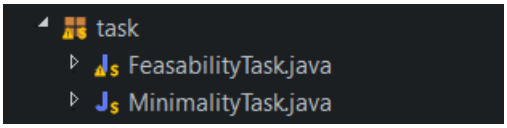


Figura 4.4: Package Task

4.2.2.1 FeasibilityTask

```java
package task;
import java.util.ArrayList;

public class FeasabilityTask implements Callable<ObjectArrayList<ObjectArrayList<String>>> {

    int rhs;
    int numCols;
    SortedTableMap<Long,Object[]> map;
    ArrayList<ArrayList<Long>> vettoreColonna;
    double maxThr;


    public FeasabilityTask(int rhs,int numCols, SortedTableMap<Long,Object[]> map,
            ArrayList<ArrayList<Long>> vettoreColonna, double maxThr) {
        this.rhs = rhs;
        this.numCols = numCols;
        this.map = map;
        this.vettoreColonna = vettoreColonna;
        this.maxThr = maxThr;

    }


    @Override
    public ObjectArrayList<ObjectArrayList<String>> call() throws Exception {
        return FeasibilityNew.feasibilityTestNew2(rhs, numCols, map, vettoreColonna, maxThr);
    }

}
```

Figura 4.5: Feasibility Task

4.2.2.2 MinimalityTask

```java
package task;

import java.util.concurrent.Callable;

public class MinimalityTask implements Callable<ObjectArrayList<RFDMap>> {
    private ObjectArrayList<ObjectArrayList<String>>  insiemiC;
    int colNumber;
    String[] colNames;
    int i;
    double[] thresholds;
    boolean debugCode;

    public  MinimalityTask(ObjectArrayList<ObjectArrayList<String>> insiemiC,int colNumber,
            int i, String[] colNames,
            double[] thresholds, boolean debugCode) {

        this.insiemiC = insiemiC;
        this.colNumber = colNumber;
        this.i = i;
        this.colNames = colNames;
        this.thresholds = thresholds;
        this.debugCode = debugCode;
    }

    @Override
    public ObjectArrayList<RFDMap> call() throws Exception {

        return  MinimalityAndGenerationRFD.startMinimalityAndGeneration2(insiemiC, colNumber,
                i, colNames, thresholds,debugCode);
    }

}
```

Figura 4.6: Minimality Task

CAPITOLO 5

Sperimentazione

5.1 Test

Tutti i test sono stati eseguiti su una macchina con sistema operativo windows 10, un processore Intel Core i7 4750HQ a 2.0GHz e con 12Gb di RAM DDR3, e settando la maxThreshold a 5, con Approx spuntato e senza nessun parametro.
Per ogni dataset utilizzato abbiamo testato l'algoritmo tramite la classe **Domino.java** descritto nel capitolo di implementazione, ricavando così i tempi impiegati su ciascun dataset, provando ogni possibile colonna come RHS e le restanti colonne come LHS.

Mostreremo quelli che sono i test ritenuti rilevanti:

- Test in sequenziale con **FixedThreadPoo**;
- Test in parallelo con **CachedTreadPool**;

5.1.1 Dataset utilizzati

Oltre ad una serie di dataset creati appositamente per verificare la correttezza di alcune operazioni, abbiamo prelevato una serie di dataset dal sito dell'Information Systems Group dell'Hasso-Plattner-Institut [4]: un un gruppo di ricerca della suddetta Università tedesca che si occupa, tra le altre cose, di progettare algoritmi dedicati alla ricerca delle dipendenze funzionali. Su tale sito, oltre a poter consultare gli algoritmi sviluppati, è possibile accedere a tutti i dataset sui quali tali algoritmi sono stati testati corredati a varie informazioni (i.e. fonte, numeri di attributi, numero di righe, dipendenze funzionali trovate, dipendenze funzionali ordinate trovate ecc).

Dataset	**Colonne**	**Righe**	**Size [KB]**
Foodstamp.csv	4	150	3
Emissions.csv	4	8088	479
Vocab.csv	4	21638	530
Iris.csv	5	150	5
Car.csv	7	1728	51
Chess.csv	7	28056	519
Breast-Cancer.csv	11	699	20
Bridges.csv	13	108	6
Echocardiogram.csv	13	132	6

Tabella 5.1: Dataset utilizzati

5.1.2 Tempi

In questa fase di testing mostreremo i risultati ottenuti lavorando in sequenziale e parallelo. Nella seguente tabella sono mostrati i risultati ottenuti dai test sui dataset considerati, considerando come unità di tempo i minuti.

Dataset	Seq. Threshol 5	Par. Threshol 5	Seq. Approx	Par. Approx	Seq. Nessun Parametro	Par. Nessun Parametro
Foodstamp.csv	0,02033333	0,02053333	0,02068333	0,02041667	0,02168333	0,02133333
Emissions.csv	0,46033333	0,45738333	0,49261667	0,52401667	0,86556667	1,1295
Vocab.csv	4,42098333	4,25195	4,34378333	4,33988333	4,28013333	5,37476667
Iris.csv	0,03138333	0,0314	0,01945	0,01951667	0,01986667	0,01845
Car.csv	0,1217333	0,11505	0,07536667	0,09918333	0,09873333	0,07543333
Chess.csv	38,80671667	25,53381667	8,68918333	8,38323333	8,5204	8,45833333
Breast-Cancer.csv	31,90678333	25,53381667	1,4976	0,87755	1,3184167	0,87385
Bridges.csv	6,29508333	4,15786667	5,88285	5,73366667	16,93371667	13,1462667
Echocardiogram.csv	12,6269167	9,50878333	4,94633333	4,92563333	6,78765	4,92665

Tabella 5.2: Tempi

Di Seguito mostriamo il grafico relativo alla 5.2

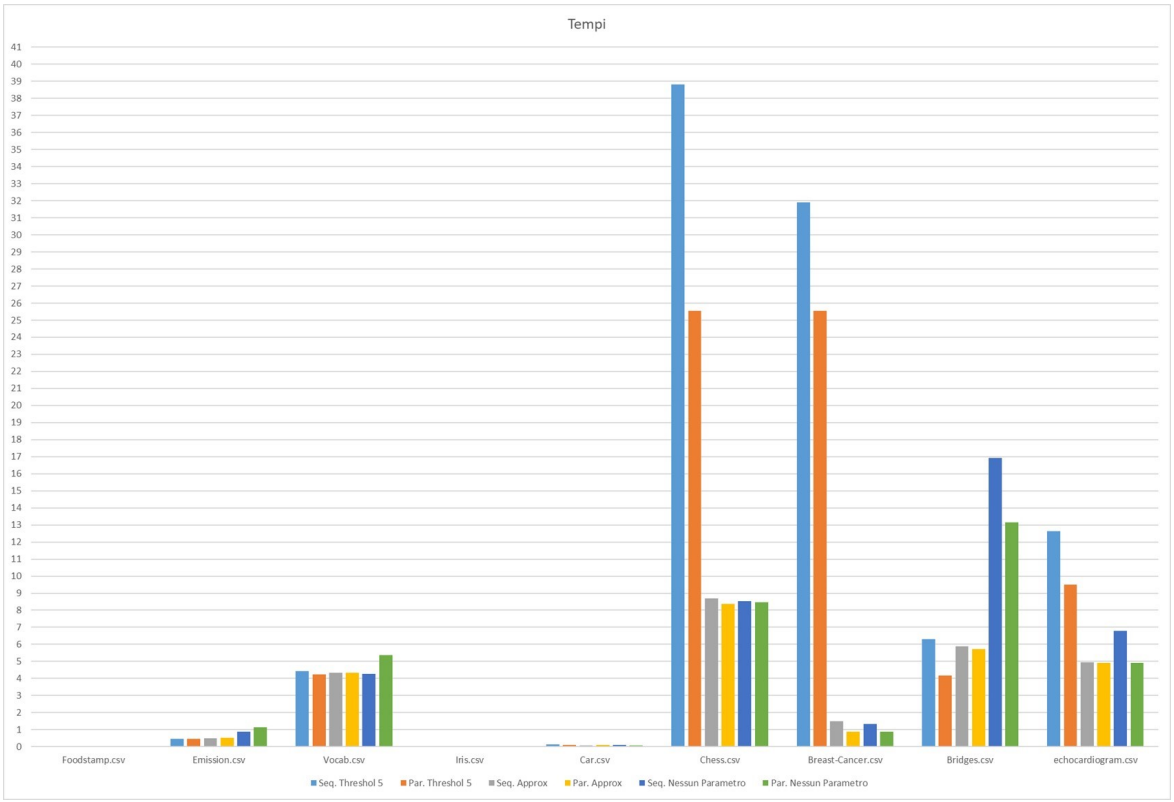


Figura 5.1: Grafico Tempi

5.1.3 Miglioramenti

In questa fase mostreremo i miglioramenti ottenuti passando dal sequenziale al parallelo. Nella seguente tabella sono mostrati i risultati ottenuti dai test sui dataset considerati in forma percentuale.

Dataset	Percentuale miglioramento Threshold 5	Percentuale miglioramento Approx	Percentuale miglioramento Nessun Parametro
Foodstamp.csv	-0,983606719	1,289250812	1,614143215
Emissions.csv	0,640839976	-6,37412453	-30,49254773
Vocab.csv	3,823432874	0,089783484	-25,57474863
Iris.csv	-0,053117372	-0,34277635	7,130888065
Car.csv	5,490116509	-31,60105123	23,59892045
Chess.csv	34,20258434	3,52104436567	0,728447843
Breast-Cancer.csv	19,97370463	41,40291132	33,71974126
Bridges.csv	33,9505698	2,535902326	22,3663242
Echocardiogram.csv	24,69433706	0,418491812	27,41744197

Tabella 5.3: Miglioramenti

Di Seguito mostriamo il grafico relativo alla 5.3

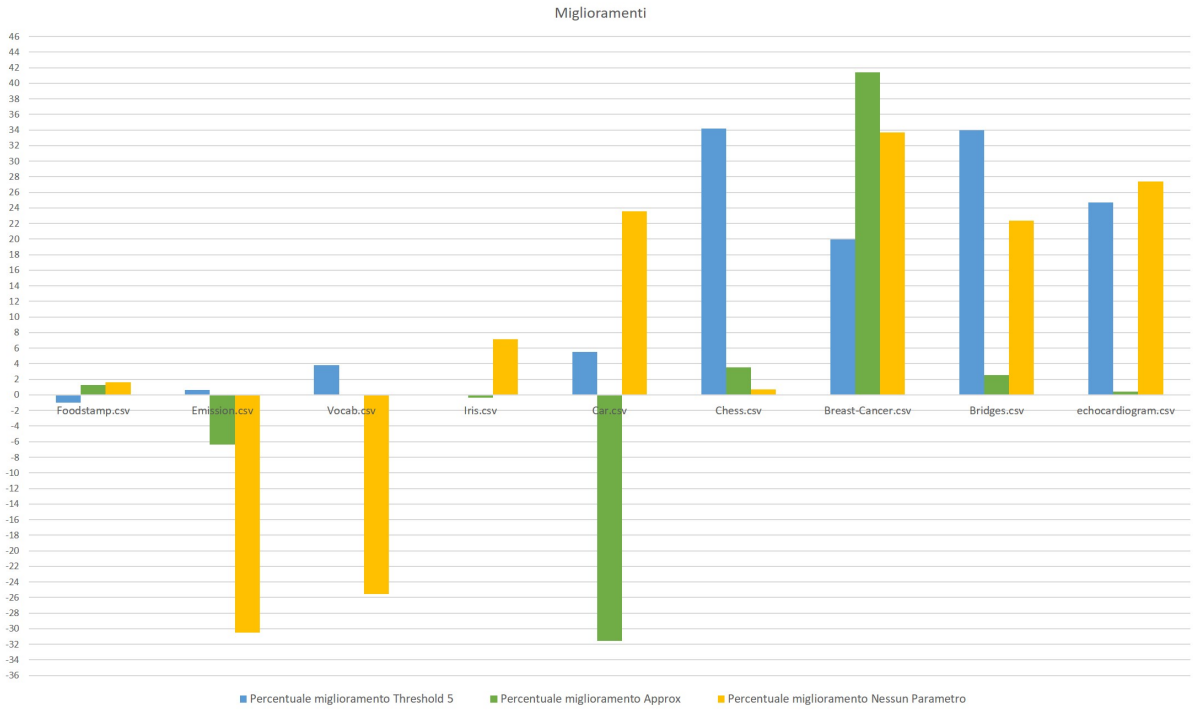


Figura 5.2: Grafico Miglioramenti

5.2 Conclusioni

Dopo aver completato la descrizione del progetto sviluppato è necessario fare qualche osservazione: dalla tabella 5.3 possiamo notare che alcuni dataset presentano un peggioramento in termini di prestazioni. Questi risultati sono dovuti dal fatto che l'ammontare di computazione di alcuni dataset era minore dei tempi di sincronizzazione e ricombinazione dei thread generati. Invece altri dataset presentano un notevole miglioramento questo dovuto alla migliore distribuzione del carico sui singoli thread.

5.3 Lavori Futuri

Durante la fase di testing, si è notato che la parte che prendeva più tempo era la parte di MinimalityAndGeneration; su di essa è stata effettuata una parallelizzazione a grana grossa ma, per ottenere prestazioni migliori, è possibile eseguire una parallelizzazione a grana fine.

Bibliografia

[1] P. Atzeni, S. Ceri, P. Fraternali, S. Paraboschi, and R. Torlone, *Basi di dati. Modelli e linguaggi di interrogazione.* McGraw Hill, 2013.

[2] A. Altamura, D. Di Pasquale, and M. Tomeo, "Relaxed functional dependencies discovery," *Unisa*, 2017.

[3] L. Caruccio, V. Deufemia, and G. Polese, "On the discovery of relaxed functional dependencies," 2017.

[4] I. S. Group, "Hasso-plattner-institut: Information-systeme."