ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

**SCUOLA DI SCIENZE**

**Corso di Laurea Magistrale in Informatica**

# Modelli e linguaggi per grafi multi-livello

**Tesi di Laurea in Basi di Dati**

**Relatore:**
**Chiar.mo Prof.**
**Danilo Montesi**

**Presentata da:**
**Miro Mannino**

**Sessione II**
**Anno Accademico 2012-13**

**Sommario**

I grafi sono molto utilizzati per la rappresentazione di dati, sopratutto in quelle aree dove l'informazione sull'interconnettività e la topologia dei dati è importante tanto quanto i dati stessi, se non addirittura di più. Ogni area di applicazione ha delle proprie necessità, sia in termini del modello che rappresenta i dati, sia in termini del linguaggio capace di fornire la necessaria espressività per poter fare interrogazione e trasformazione dei dati. È sempre più frequente che si richieda di analizzare dati provenienti da diversi sistemi, oppure che si richieda di analizzare caratteristiche dello stesso sistema osservandolo a granularità differenti, in tempi differenti oppure considerando relazioni differenti. Il nostro scopo è stato quindi quello di creare un modello, che riesca a rappresentare in maniera semplice ed efficace i dati, in tutte queste situazioni. Entrando più nei dettagli, il modello permette non solo di analizzare la singola rete, ma di analizzare più reti, relazionandole tra loro. Il nostro scopo si è anche esteso nel definire un'algebra, che, tramite ai suoi operatori, permette di compiere delle interrogazioni su questo modello. La definizione del modello e degli operatori sono stati maggiormente guidati dal caso di studio dei social network, non tralasciando comunque di rimanere generali per fare altri tipi di analisi. In seguito abbiamo approfondito lo studio degli operatori, individuando delle proprietà utili per fare delle ottimizzazioni, ragionando sui dettagli implementativi, e fornendo degli algoritmi di alto livello. Per rendere più concreta la definizione del modello e degli operatori, in modo da non lasciare spazio ad ambiguità, è stata fatta anche un'implementazione, e in questo elaborato ne forniremo la descrizione.

# INDICE

# CAPITOLO 1

# INTRODUZIONE

## 1.1 MOTIVAZIONI E SCOPI

I grafi sono molto utilizzati per la rappresentazione di dati, sopratutto in quelle aree dove l'informazione sull'interconnettività e la topologia dei dati è importante tanto quanto i dati stessi, se non addirittura di più.

Le applicazioni che hanno motivato lo studio dei grafi per la rappresentazione dei dati sono moltissime. Negli anni '80 troviamo molto presenti i sistemi ipertestuali [1]. Successivamente, negli anni '90 troviamo i dati semi-strutturati [2, 3] e i database ad oggetti [4]. Nell'ultimo decennio invece troviamo i social network [5, 6] ed il semantic web [7, 8, 9, 10].

Ogni area di applicazione, come quelle appena citate, ha delle proprie necessità, sia in termini del modello che rappresenta i dati, sia in termini del linguaggio capace di fornire la necessaria espressività per poter fare interrogazione e trasformazione dei dati.

In questa tesi vorremmo focalizzarci sullo studio di un'area che in questo momento sta crescendo molto, di pari passo con la crescita dei social network e del semantic

web. È sempre più frequente che gli studi scientifici e le applicazioni commerciali richiedano di analizzare dati provenienti da diversi sistemi, oppure richiedano di analizzare caratteristiche dello stesso sistema osservandolo a granularità differenti, in tempi differenti, oppure considerando relazioni differenti. Il nostro scopo è quindi quello di creare un modello che riesca a rappresentare in maniera semplice ed efficace i dati in tutte queste situazioni.

Per far ciò, come prima cosa, ricercheremo e descriveremo i vari lavori che affrontano questi temi. In secondo momento invece creeremo un modello che non solo ci permetta di analizzare la singola rete, ma che ci permetta di analizzare più reti, correlandole tra loro. Queste relazioni possono essere gerarchiche [11, 12], per semplificare reti troppo complesse; oppure temporali [11], permettendo l'analisi dell'evoluzione di una rete; oppure possono riguardare la stessa rete, con gli stessi nodi, ma curando connessioni di tipo differente [11, 13, 14, 15, 11]; ed infine possono essere tra reti totalmente differenti, che riguardano dati aventi la stessa natura [16] (e.g. Facebook e Twitter) o di natura differente (e.g. rete urbana e rete di comunicazione telefonica). Il modello che creeremo sarà pertanto abbastanza generale da rappresentare tutti gli scenari descritti negli altri lavori presi in esame, verificando ciò con opportune dimostrazioni di riduzione.

Ogni modello ben fatto fornisce anche un linguaggio che permetta l'accesso ai dati e consente la loro manipolazione. Definiremo pertanto un'algebra che, tramite i suoi operatori, permetta di compiere delle interrogazioni utili nelle situazioni elencate. In particolar modo cercheremo di tener sempre presente la risoluzione di problemi legati ai social network, in modo da avvicinarci di più al nostro caso di studio. Nonostante ciò non trascureremo mai la generalità, garantendo il loro utilizzo anche in circostanze differenti. In seguito faremo anche una rassegna dei più importanti lavori che propongono algebre per dati strutturati a grafo, cercando di discutere le differenze, i pregi e i difetti.

Successivamente approfondiremo lo studio degli operatori. Individueremo alcune proprietà, comprendendo meglio gli operatori in sé e come questi siano tra loro re-

lazionati. Ragioneremo sui dettagli implementativi, fornendo degli algoritmi di alto livello. Infine individueremo alcune ottimizzazioni, necessari come primo passo per rendere più efficienti gli operatori, sia in termini di tempo che di spazio.

Per rendere più concreta la definizione del modello e degli operatori, in modo da non lasciare spazio ad ambiguità, li realizzeremo con un reale linguaggio di programmazione (i.e. Java). In questo modo potremo verificare la correttezza concettuale del modello e degli operatori, la correttezza degli algoritmi illustrati, la correttezza del modo con cui gli operatori possono essere utilizzati (in particolar modo l'interazione tra il modello e gli operatori e l'interazione tra gli operatori stessi), e infine testare alcune delle ottimizzazioni proposte.

## 1.2 Analisi multi-livello

L'importanza di analizzare reti disposte su più livelli, la definizione di modelli, di linguaggi, di nuovi metodi e di algoritmi, sono già stati affrontati da alcuni autori. Possiamo citare Mucha, Peter et al. che si focalizza sulla ricerca di community structure [11]; oppure Magnani e Rossi che definiscono un modello dei dati per queste reti multistrato prendendo come caso di studio l'analisi di reti sociali [16], estendendo le misure delle SNA (social network analysis) per adattarle a queste reti [16, 15].

Queste tipologie di reti vengono intese in vari modi, ad ognuna spesso viene attribuito un proprio nome. Abbiamo però distinto due approcci generali: Multislice Network [11] e Multilayer Network [16]. La prima mette in relazione parti differenti della stessa rete, focalizzandosi sulla sua evoluzione nel tempo, sulla sua granularità o sul tipo di connessioni; la seconda invece mette in relazione reti totalmente differenti cercando di accoppiarle in maniera opportuna.

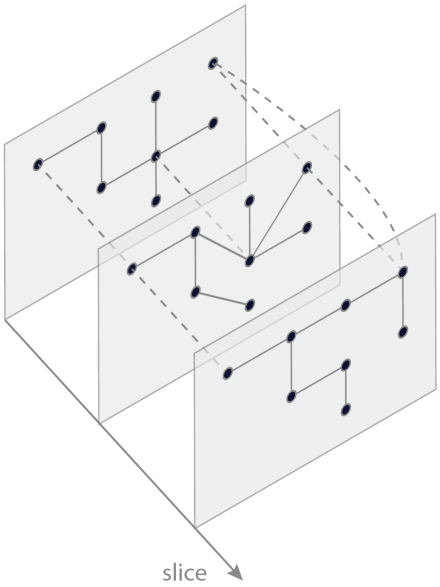



**Figura 1.2.1:** Multislice Network

### 1.2.1 MULTISLICE NETWORK

Introdotte in maniera informale nel lavoro di Mucha, Peter et al. [11], vengono costruite accoppiando più matrici di adiacenza, dove ognuna di queste descrive una slice. Ognuna di queste può assumere diversi significati, può avere nodi ed archi diversi rispetto alle altre, e può avere dei nodi accoppiati con quelli di un'altra. Nonostante gli autori non esplicitino molti dettagli, durante la descrizione di queste tipologie di reti, cercheremo anche di fare delle considerazioni per delineare le principali proprietà di ognuna di esse, come l'importanza dell'ordine tra le slice, oppure il tipo di relazioni che si possono formare tra le varie slice.

#### 1.2.1.1 TIME-DEPENDENT NETWORK

In questo caso si focalizza l'attenzione su una rete particolare, cercando di analizzare la sua evoluzione, in altre parole la sua variazione nel tempo. Ogni slice assume

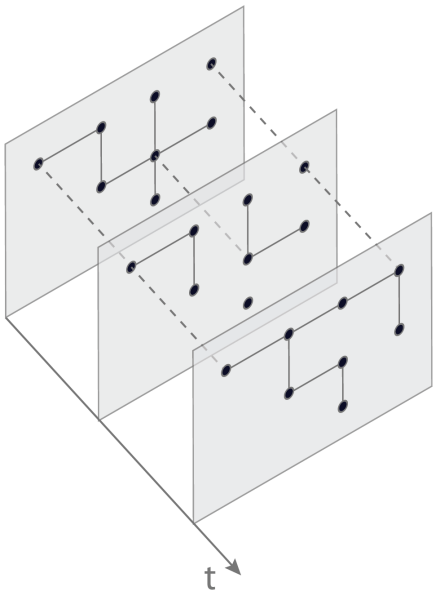



**Figura 1.2.2:** Time-Dependent Network

quindi un significato temporale, una fotografia della rete ad un certo istante, il tempo è pertanto discreto. L'ordine tra di essi di conseguenza è rilevante, e può essere definito etichettando opportunamente ogni slice (e definendo un ordine totale tra le etichette) oppure ordinando le slice stesse. Gli accoppiamenti tra le varie slice, visto il loro ordine totale, coinvolgono prevalentemente una slice di un certo istante temporale con quella dell'istante successivo.

#### 1.2.1.2 MULTISCALE NETWORK (O CLUSTERED GRAPH)

In questo caso si focalizza l'attenzione sulla risoluzione (o granularità) della rete, ad esempio per rilevare le community della stessa rete attraverso scale differenti.

L'ordine tra le varie slice continua ad essere importante, rispecchiando quello dei diversi livelli di granularità. Più alto è il livello di dettaglio e più bassa è la granularità (e viceversa). Però, a differenza dei Time-Dependent Network, ogni accoppiamento tra una slice ed un'altra non è una relazione one-to-one, ma one-to-many. Un nodo

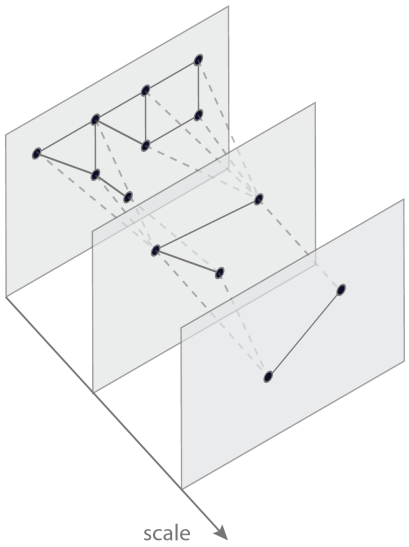



**Figura 1.2.3:** Multiscale Network

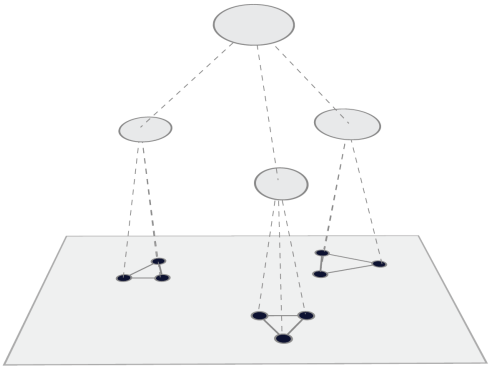


**Figura 1.2.4:** Clustered Graph

di una particolare slice può essere infatti il raggruppamento di più nodi della sua slice precedente (o sottostante), che identifica un livello di dettaglio maggiore.

Tra i vari lavori che parlano di questo argomento, si trovano anche delle descrizioni formali per questo tipo di reti. Un esempio è quello dei Clustered Graph [12], che costituiscono un'interpretazione molto vicina ai Multiscale Network.

**Definizione 1.2.1.** Un *Clustered Graph* $C = (G, T)$ è un grafo non diretto $G$ e un albero con radice $T$. Le foglie di $T$ sono esattamente i vertici di $G$. Ogni nodo $v$ di $T$ invece rappresenta un cluster $V(v)$ di vertici di $G$ dove $V(v) \subset G$. L'albero $T$, inoltre descrive una relazione di inclusione tra i vari cluster. [12]

In realtà esistono alcune differenze tra i due: in un Clustered Graph, a causa della sua struttura ad albero, è necessario esprimere tutti i livelli di dettaglio, dalle foglie fino alla radice. Quest'ultima è formata da un unico nodo, mentre l'insieme delle foglie corrisponde ai nodi del grafo stesso. Nei Multiscale Network, invece, ciò non è necessario: la slice di granularità massima potrebbe avere molti nodi. Nonostante questo le differenze sono trascurabili, visto che in alcuni casi un Multiscale Network non è altro che un Clustered Graph, dove viene utilizzata solo parte dell'albero.

#### 1.2.1.3 MULTIPLEX NETWORK (O MULTIDIMENSIONAL NETWORK)

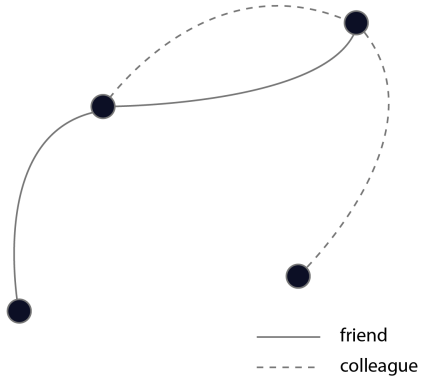



**Figura 1.2.5:** Multidimensional Network

Questa tipologia di reti sono invece abbastanza legate ai multigrafi [13], abbiamo infatti diverse tipologie di connessioni tra i vari nodi. Vengono citate in diversi lavori e chiamate anche in maniera differente: Multidimensional Network [14], Multirelationship [15] o Multiplex Network [11].

Nel mondo reale le reti sono spesso multidimensionali, c'è bisogno di distinguere le varie tipologie di connessioni, oppure di guardare le interazioni attraverso differenti prospettive. Le dimensioni tra i vari nodi potrebbero far parte dei dati stessi (i.e. esplicite) oppure devono essere definite dagli analisti ed estrapolate (i.e. implicite) per permettere di analizzare qualità interessanti delle interazioni. Esempi di reti del primo tipo sono quelle dei social network, dove le varie interazioni rappresentano la diffusione delle informazioni: scambio di email, scambio di messaggi istantanei, e così via. Esempi invece del secondo tipo sono quelle dei social network con altre caratteristiche. In Flickr, ad esempio, nonostante tra due utenti ci siano dimensioni esplicite (e.g. amicizia, collaborazione), esistono altrettante dimensioni implicite interessanti, come ad esempio l'insieme delle loro foto preferite. [14]

Dalla prospettiva dei Multislice Network, ogni slice ha il compito di rappresentare una dimensione particolare, dove ciò che cambia sono le connessioni tra i nodi del grafo. In questo caso l'ordine non influisce, e si tende ad accoppiare ogni slice con tutte le altre. Da un'altra prospettiva [14] invece, si utilizzano i multigrafi per modellare queste reti.

**Definizione 1.2.2.** Una *multidimensional network* è denotata dalla tripla $G = (V, E, L)$ dove: $V$ è l'insieme dei nodi, $L$ è l'insieme delle etichette e $E$ è l'insieme degli archi etichettati (insiemi di triple $(u, v, d)$ dove $u$ e $v$ sono i nodi coinvolti e $d$ invece è l'etichetta). Ogni etichetta identifica una dimensione della rete. Un arco infatti appartiene alla dimensione $d$ solamente se è etichettato con $d$. Un nodo appartiene ad una certa dimensione $d$ solamente se esiste un arco etichettato con $d$ che ha quel nodo in uno dei suoi due estremi.

### 1.2.2 MULTILAYER NETWORK

Vengono introdotte nel lavoro di Magnani e Rossi [16]. Queste rappresentano reti differenti interconnesse tra di loro, rispecchiando la reale organizzazione dei social network che si trovano on-line. Ogni layer rappresenta una singola rete, ad esempio Facebook o LinkedIn. Queste, nonostante la loro eterogeneità, hanno in comune diversi elementi. Gli autori descrivono infatti la presenza di un soggetto in Internet, da un punto di vista metaforico, come un segmento che attraversa i molteplici piani (layer) di questa architettura, ognuno di questi rappresenta una diversa rete nella quale il soggetto è connesso. Due utenti $a$ e $b$ potrebbero quindi essere connessi in una certa rete (e.g. LinkedIn), ma non esserlo in un'altra (e.g. Facebook), esprimendo il fatto che hanno un certo tipo di rapporto (e.g. colleghi), ma non un altro (e.g. amicizia). Analizzando la risultante rete interconnessa nella sua interezza, attraversando reti diverse, è sicuramente molto più efficace rispetto ad analizzare le varie reti prese singolarmente (e.g. si pensi ai vari fenomeni come la diffusione di informazioni).

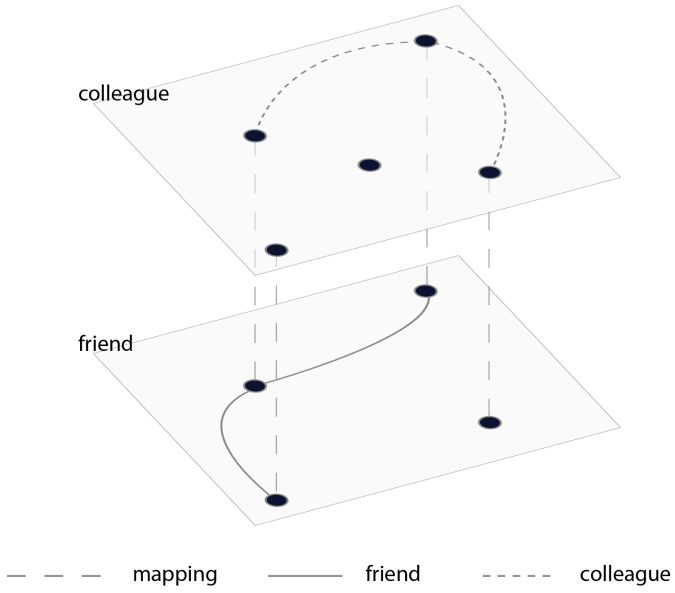



**Figura 1.2.6:** Multilayer Network

Il legame che connette i vari layer viene definito con degli opportuni accoppiamenti, chiamati *Node Mapping*. Questi possono essere definiti partendo da delle caratteristiche esplicite della rete, come ad esempio lo stesso utente che si registra su più social network utilizzando la stessa email. Inoltre, possono anche essere definiti partendo da delle caratteristiche che è necessario estrapolare, come ad esempio una comune locazione geografica di un Ristorante presente sia su Facebook che su Foursquare ma che è stato registrato utilizzando differenti identificativi. Infine, tali accoppiamenti possono essere delle relazioni 1:N, oltre che delle relazioni 1:1, per modellare i casi in cui un soggetto in una rete rappresenti più soggetti di un'altra rete, come ad esempio il caso in cui una testata giornalistica su un social network rappresenti l'aggregazione di più giornalisti presenti su un altro social network.

Viene definito un modello per questo tipo di reti, chiamato ML Model (Multi Layer Model) [16].

**Definizione 1.2.3.** Un *Network Layer* è un grafo pesato $(V, w)$ dove $V$ è l'insieme di vertici e $w : (V \times V) \rightarrow [0, 1]$ è la funzione che stabilisce il peso tra due vertici.

Per avere un esempio dell'ultima definizione, notiamo come nella Figura 1.2.6 abbiamo due layer: friend e colleague. Questi layer, come abbiamo detto, possono essere accoppiati.

**Definizione 1.2.4.** Un *Node Mapping* da un Network Layer $L_1 = (V_1, w_1)$ ed un Network Layer $L_2 = (V_2, w_2)$ è una funzione $m : V_1 \times V_2 \rightarrow [0,1]$. Per ogni $u \in V_1$, l'insieme $C(u) = \{v \in V_2 \ : \ m(u,v) > 0\}$ è l'insieme dei vertici $V_2$ che corrispondono ad $u$.

Riprendendo nuovamente la Figura 1.2.6 possiamo osservare visivamente il mapping tra i nodi dei due livelli. In questo esempio però, non ci sono casi in cui un nodo viene mappato in più nodi di un altro livello, ma con questa definizione è possibile anche fare queste associazioni one-to-many.

**Definizione 1.2.5.** Un *Multi Layer Network* è una tupla $MLN = (L_1, L_2, ..., L_n, IM)$ dove $L_i = (V_i, w_i)$ (con $i \in \{1, 2, ..., n\}$) sono Network Layer. IM (i.e. Identity Mapping) è una matrice $n \times n$ di Node Mapping, con $IM_{i,j} : V_i \times V_j \rightarrow [0,1]$.

# CAPITOLO 2

# MULTI LEVEL DATA MODEL

Questo modello dei dati, che abbiamo chiamato Multi Level Data Model, dei dati nasce per riuscire ad avere una rappresentazione unica di modelli che abbiamo precedentemente mostrato: Time-Dependent Network, Multiscale Network (o Clustered Graph), Multiplex Network (o Multidimensional Network) e Multilayer Network. Ognuno di questi nasce per soddisfare uno scopo ben preciso, guidato da un caso di studio a volte troppo stringente, che limita di molto l'espressività del modello stesso. Il nostro scopo è stato quindi quello di ricercare una definizione che fosse più generale, e che potesse non solo rappresentare egregiamente gli scenari studiati da tali modelli, ma anche rappresentare il nostro scenario tipico dove abbiamo diversi social network correlati da una qualche relazione.

A livello dei singoli grafi, che compongono ogni livello, si è prima cercata una definizione che potesse rappresentarli in maniera generale e semplice. Per fare ciò sono stati presi in esame diversi modelli dei dati di grafi, partendo da quelli più conosciuti che troviamo spesso nella teoria dei grafi [13, 17], a quelli meno generali che più si intercalano ad applicazioni specifiche. In particolare sono stati studiati e presi in esame lavori quali: GOOD [4, 18], GDM [19], GOAL [20], Gram [21], GraphDB [22], Hypernode [23], Oracle Spatial [24]. Da aggiungere a quest'ultima lista altri articoli

[25, 26, 27] e libri [28, 13, 29] che sono stati utili per avere ulteriori spunti e che hanno favorito lo studio dei fondamenti necessari per trattare questi argomenti.

Le definizioni sono state scritte sempre con un'ottica vicina a quella di una reale implementazione, pensando sempre a quale potesse essere il modo (in maniera concettuale) più minimale e efficiente. La definizione dei singoli livelli è stata pensata per essere generale, non limitandola ad un dominio di applicazione ben definito, come succede per alcuni dei lavori sopra citati. La correlazione tra i livelli invece trova nel Multilayer Network il suo modello più simile, nonostante le notevoli differenze che comunque ci sono tra i due e che commenteremo successivamente.

Una volta definito il modello, mostreremo come questo sia correlato alle reti che abbiamo precedentemente mostrato: Time-Dependent Network, Multiscale Network (o Clustered Graph), Multiplex Network (o Multidimensional Network) e Multilayer Network. Con ciò dimostreremo come sia possibile, data una qualsiasi di queste, creare un Multi Level Network semanticamente identico, ovvero che abbia un modo corretto per rappresentare gli stessi dati e le stesse relazioni. In questo confronto cercheremo anche di commentare le differenze, cercando di capire i pregi ed i difetti del modello che presenteremo, giustificandone quindi le scelte progettuali.

## 2.1 Casi di studio

In Figura 2.1.1 abbiamo un esempio con due livelli $A$ e $B$. Tali livelli possono avere molteplici significati quali: la rappresentazione di due social network differenti (e.g. Facebook e Twitter), oppure la rappresentazione del medesimo social network in tempi differenti, oppure la rappresentazione di una rete sociale a granularità di dettaglio differenti (come succede dei Multiscale Network), ed infine la rappresentazione di una medesima rete sociale dove vengono mostrate relazioni differenti (come accade invece nei Multidimensional Network).

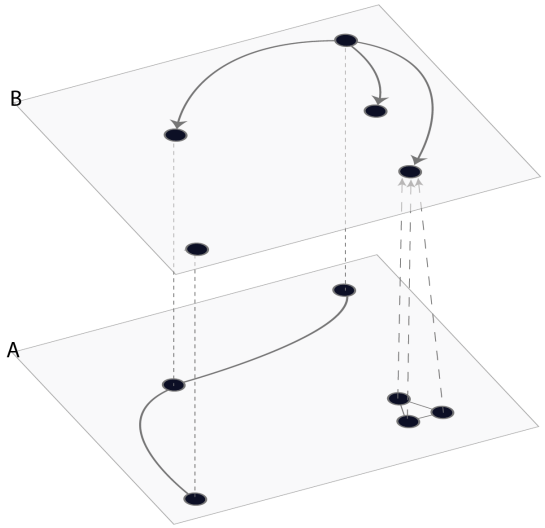



**Figura 2.1.1:** Multilevel Network

Il primo è collegato con il secondo attraverso diverse relazioni: una prima relazione, che nell'esempio è raffigurata da linee con tratteggiature corte, rappresenta una corrispondenza one-to-one; una seconda invece, è raffigurata da linee con tratteggiature più lunghe, ed archi unidirezionali, rappresenta invece una relazione one-to-many. Anche le relazioni possono assumere significati differenti: la prima può essere una relazione che ci mostra come un nodo con una stessa identità (e.g. uno stesso individuo) può essere presente in più livelli (e.g. più social network); la seconda invece potrebbe essere usato per fondere i cluster in singoli nodi, similmente a come abbiamo visto per i Multiscale Network. Un'altra interpretazione per questo esempio è quello dove il social network $B$ rappresenta un blog, dove gli articoli vengono prodotti da più autori che corrispondono a differenti persone che possono essere identificate nel social network di $A$.

Focalizzandoci sull'interpretazione a noi preferita, che ci ha guidato nella stesura delle definizioni del modello, dove ogni livello rappresenta un differente social network, possiamo osservare come in ognuno di essi le relazioni tra i nodi differisca. Possiamo ad esempio pensare che due persone potrebbero avere una relazione di

amicizia in un social network, ma non in un altro. Oppure che le relazioni sono di natura completamente differente (e.g. una di amicizia e l'altra una relazione geografica di vicinanza). Da ciò possiamo subito intuire come possiamo essere interessati a cercare di capire se due entità siano collegate, anche passando attraverso molteplici social network differenti. In questo momento infatti i social network rappresentano dei mondi chiusi, che non si interfacciano con gli altri social network: possiamo sapere se $x$ è connesso con $y$ attraverso una catena di relazioni del social network $A$, ma non possiamo sapere se questi sono connessi attraverso una catena del social network $A$, che parte da $x$ e finisce in $z$ e poi attraverso un'ulteriore catena in un altro social network $B$, che parte da $z$ e finisce in $y$.

Quello che cercheremo di fare è quindi considerare questi casi di studio, che possono essere utili quando si fanno delle analisi sui social network, e cercheremo di costruire opportunamente un modello dei dati che possa rappresentarle in maniera semplice ed efficace. Inoltre, cercheremo anche di dare delle definizioni che siano generali abbastanza da poter rappresentare i modelli che abbiamo descritto precedentemente, e che costituiscono in qualche modo i primi passi verso la modellazione di questi scenari.

## 2.2 Definizione

### 2.2.1 Network Level

**Definizione 2.2.1.** Un *Network Level* è un grafo diretto pesato definito dalla quadrupla $(N, A, v, d)$ dove N è l'insieme dei nodi, $A \subseteq N \times N$ è l'insieme degli archi, $v : A \rightarrow Dom_A$ è una funzione che associa un valore ad ogni arco e $d : N \rightarrow Dom_N$ è una funzione che associa un valore ad ogni nodo.

Ogni Network Level rappresenta una rete che, rispettivamente agli altri lavori che abbiamo citato, può rappresentare una fotografia di una rete ad un certo istante

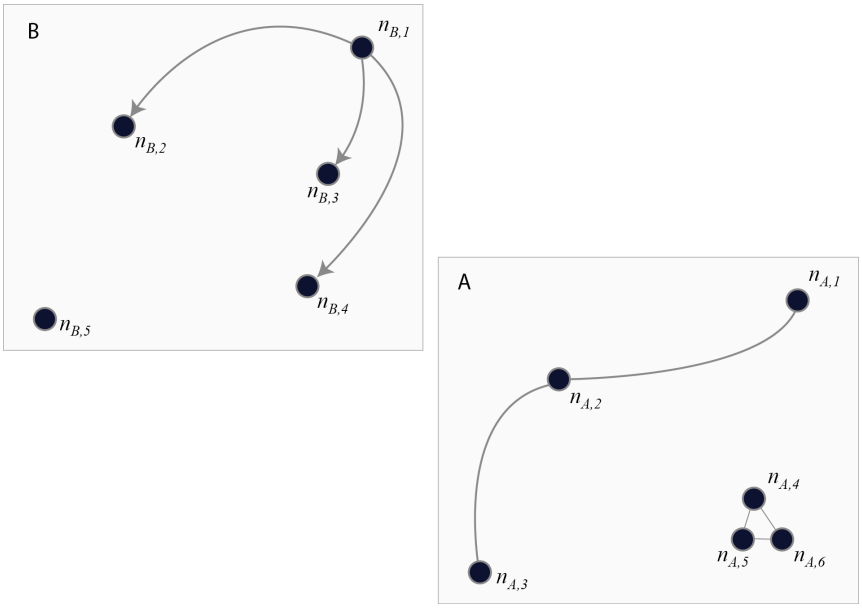



**Figura 2.2.1:** I livelli, visti singolarmente, del Multilevel Network in fig. 2.1.1

temporale, un certo livello di granularità della stessa rete, un multigrafo mostrando solamente gli archi di un certo tipo, o la rete di un determinato social network.

Il grafo è stato definito come diretto per essere generale, le relazioni tra i nodi infatti non sono sempre bidirezionali, in alcune situazioni (e.g. Twitter), possono esistere relazioni (e.g. following relationship) unidirezionali. Inoltre, ogni arco può esprimere un valore; nel caso più conosciuto, $Dom_A$ potrebbe essere l'insieme $[0,1]$ che esprime una probabilità o la forza della relazione. Allo stesso modo, anche i nodi possono esprimere un valore, che in futuro chiameremo *dato del nodo*. Quest'ultimo potrebbe rispecchiare fedelmente i dati della base di dati, oppure potrebbe contenere informazioni utili solamente ad analisi di qualunque tipo: si pensi, ad esempio, ad un grafo dove ogni nodo identifica una foto, ed alla relativa memorizzazione del ranking.

**Esempio 2.2.1.** Seguendo l'esempio in Figura 2.1.1, identificati con $A$ e $B$, abbiamo due Network Level, ognuno con il proprio insieme di nodi e con le proprie connessioni. Per riuscire ad essere più esplicativi, in Figura 2.2.1, raffigurato i due livel-

li, prendendoli singolarmente, ed abbiamo attribuito un identificativo ad ogni nodo (e.g. $n_{A,1}$, $n_{B,2}$, ecc.). E' buona norma notare di non creare confusione tra gli identificativi che si attribuiscono ai nodi (e.g. $n_{A,1}$) e le etichette (o dati) che vengono anche queste associate ai nodi; infatti, mentre due nodi distinti non possono avere stesso identificativo, possono invece avere stesse etichette. Questi identificativi sono utili per fare esempi, ma sono anche oggetti che possono concretamente essere utilizzati (e.g. puntatore in memoria al nodo). Prendendo in esame il livello B, possiamo dire che esso sia la quadrupla $(N_B, A_B, v_B, d_B)$ dove $N_B = \{n_{B,1}, n_{B,2}, n_{B,3}, n_{B,4}, n_{B,5}\}$ mentre $A_A = \{(n_{B,1}, n_{B,2}), (n_{B,1}, n_{B,3}), (n_{B,1}, n_{B,4})\}$. Invece, per $v_B$ e $d_B$ possiamo avere delle funzioni opportune, con opportuni domini (rispettivamente $Dom_A$ e $Dom_N$), che determinano il valore di ogni arco ed il dato di ogni nodo.

Avremmo potuto definire gli archi in maniera più semplice (vedi la definizione 1.2.3), utilizzando un'unica funzione $weight : (N \times N) \rightarrow [0,1]$, lasciando intendere la presenza di archi tra i nodi $n_1 \in N$ ed $n_2 \in N$ solamente con $weight(n_1, n_2) > 0$. In quest'ultimo modo però, avremmo avuto delle difficoltà con tutti quei domini che non hanno modo di esprimere l'inesistenza di una relazione; costringendoci a modificare ogni volta tali domini introducendo elementi extra-domino, oppure a modificare la funzione $w$ facendola diventare parziale.

### 2.2.2 Multi Level Network

**Definizione 2.2.2.** Un *Multi Level Network* è una coppia $(L, C_l)$ dove $L = (l_1, l_2, ..., l_n)$ identifica i Network Level di cui è composto, mentre $C_l = (c_1, c_2, ..., c_n)$ identifica i loro accoppiamenti, chiamati Levels Coupling (def. 2.2.3).

**Esempio 2.2.2.** Riprendendo l'esempio in Figura 2.1.1: $L = (A, B)$ è una tupla di due elementi dove il primo rappresenta il livello che abbiamo identificato con $A$, il secondo invece quello che abbiamo identificato con $B$. $C_l$, invece, è composto dai due accoppiamenti (chiamiamoli $c_1$ e $c_2$) che abbiamo raffigurato: il primo con i tratteggi più corti, ed un secondo accoppiamento con tratteggi più lunghi.

I Network Level vengono ordinati secondo l'ordine definito dalla stessa tupla L; in questo modo possiamo modellare, ad esempio, tutti quei casi in cui ai livelli viene attribuita un'informazione temporale, come ad esempio nei Time-Dependent Networks [11].

Nella definizione, $L$ non è stato definito come un'insieme poiché in realtà potremmo avere la necessità di avere più livelli con gli stessi nodi ed archi; si noti che si sarebbe potuto utilizzare un multiinsieme, ed imporre la definizione di un ordine totale tra i suoi elementi, a discapito però della semplicità della definizione stessa. Nonostante l'ordine tra gli accoppiamenti non sia importante, anche nel caso di $C_l$ valgono le stesse considerazioni appena fatte, poiché anche in questo caso è possibile che due accoppiamenti, anche se uguali, esprimano concetti differenti.

### 2.2.3 Levels Coupling

**Definizione 2.2.3.** Un *Levels Coupling* è una tupla $C_l = (c_1, c_2, ..., c_n)$, ogni elemento $c = (s, d, M, w)$ $(c \in C_l)$ indica un accoppiamento (chiamato *Couple*) tra due Network Level. $M$ è un insieme di coppie $(a, b)$ (con $a \in N_s$ e $b \in N_d$) che relazionano i nodi del Network Level di partenza $s = (N_s, A_s, v_s, d_s)$ con quello di arrivo $d = (N_d, A_d, v_d, d_d)$. La funzione $w : M \rightarrow Dom_W$ stabilisce il valore attribuito ad ogni relazione tra questi due livelli.

**Esempio 2.2.3.** Continuando gli esempi precedenti, i Couple $c_1$ e $c_2$ sono rispettivamente la tupla $(A, B, M_1, w_1)$ e $(A, B, M_2, w_2)$, dove, osservando la Figura 2.1.1, il primo indica l'accoppiamento rappresentato dalle linee con tratteggiature corte, mentre il secondo rappresentato dalle linee con tratteggiature lunghe. I livelli coinvolti sono $A$ e $B$ per tutti e due; ciò che cambia sono i nodi coinvolti (i.e. $M_1$ ed $M_2$), e verosimilmente anche i valori di ogni arco (i.e. le funzioni $w_1$ e $w_2$). Riprendendo il Network Level di esempio, dove la Figura 2.1.1 esplicita le connessioni tra i due livelli, e dove la Figura 2.2.1 attribuisce degli identificativi ad ogni nodo, possiamo anche esplicitare i nodi coinvolti dei due accoppiamenti: $M_1 = \{(n_{B,5}, n_{A,3}), (n_{A,3}, n_{B,5}),$

$(n_{B,2}, n_{A,2}), (n_{A,2}, n_{B,2}), (n_{B,1}, n_{A,1}), (n_{A,1}, n_{B,1})\}$ ed $M_2 = \{(n_{A,4}, n_{B,4}), (n_{A,5}, n_{B,4}), (n_{A,6}, n_{B,4})\}$.

Da notare che, con la precedente definizione, possiamo avere due accoppiamenti $c_i = (s_i, d_i, M_i, w_i)$ e $c_j = (s_j, d_j, M_j, w_j)$ (con $i, j \in \{1, 2, ..., n\}$) tra gli stessi Network Level $s$ e $d$. Infatti, potremmo aver bisogno che il primo esprima un concetto, mentre il secondo ne esprima un altro anche non correlato con il primo, avendo, ad esempio, stessi nodi coinvolti (i.e. $M_i = M_j$) ma diversi valori delle relazioni (i.e. $w_i \neq w_j$).

Questa definizione è molto generale, permettendo di esprimere relazioni one-to-one, one-to-many o many-to-many.

## 2.3 Espressività

In questa sezione mostreremo come sia possibile rappresentare gli altri lavori che abbiamo citato utilizzando la definizione di Multilevel Network. In questo modo ci convinceremo che questo formalismo è abbastanza generale da poter modellare tutte le situazioni di cui abbiamo discusso.

### 2.3.1 Multislice Network

#### 2.3.1.1 Time-Dependent Network

Nei Time-Dependent Network ogni slice ha un significato temporale, per prima cosa è quindi importante avere un ordine ben definito tra le varie slice. Nelle definizioni che abbiamo dato precedentemente quando abbiamo introdotto i Multilevel Network siamo stati attenti nel definire un ordine tra i vari livelli. Nella definizione 2.2.2 notiamo infatti che $L$ è stato definito come una tupla che, come sappiamo, hanno un ordine ben preciso, in modo da poter distinguere la posizione di ogni suo elemento.

Detto ciò, è del tutto evidente di come sia possibile ordinare i vari livelli in modo da rispecchiare l'ordine temporale.

Un altro tipo di considerazione potrebbe essere quella delle slice duplicate; è possibile infatti che, nella costruzione di una base di dati per l'analisi di un grafo che cambia nel corso del tempo, per questioni di ottimizzazione, si decida di non avere slice al tempo $i$ che siano identiche a quelle del tempo $i+1$; ma ciò non esclude comunque il fatto di avere slice duplicate, per esempio a distanze maggiori. E' proprio anche per questo motivo che, nella definizione 2.2.2, è stato scelto di utilizzare una tupla, poiché permette di distinguere bene i vari elementi, anche se questi sono duplicati, dando ad ognuno di questi una collocazione temporale ben definita.

Introducendo i Time-Dependent Network, abbiamo visto come gli accoppiamenti vengono prevalentemente fatti tra una slice e la sua successiva. E' possibile verificare che questo è permesso dalle Multilevel Network, osservando la definizione 2.2.3.

Possiamo riassumere, in maniera più formale, con questa semplice dimostrazione:

**Teorema 2.3.1.** *Dato un Time-dependent Network, possiamo creare un Multilevel Network che modelli le stesse informazioni.*

*Dimostrazione.* Dato un Time-dependent Network $T$, possiamo creare un Multilevel Network $N = (L, C_l)$, con $L = (t_0, t_1, ..., t_n)$ dove il livello $t_i$ ($\forall\, i \in \{0, 1, ..., n\}$) rappresenta la slice i-esima di $T$. Il Level Coupling, identificato da $C_l$ ha esattamente n accoppiamenti $(c_0, c_1, ..., c_n)$, dove $c_i = (t_i, t_{i+i}, M_i, w_i)$. $M_i$ è l'insieme di coppie $(s, d)$ con $s \in t_i$ e $d \in t_{i+1}$ dove in $T$ abbiamo $s$ correlato a $d$. Il peso di ogni accoppiamento tra nodi, essendo ininfluente in questo caso, può essere definito come $w_i : M_i \rightarrow \{1\}$ dove $\forall (a, b) \in M_i,\ w_i((a, b)) = 1$. □

#### 2.3.1.2 MULTISCALE NETWORK (O CLUSTERED GRAPH)

Nei Multiscale Network ogni slice rappresenta una certa granularità del grafo; anche in questo caso, come abbiamo già detto, è importante avere un ordine ben definito. Valgono perciò le stesse considerazioni fatte nel paragrafo precedente che riguardava i Time-dependent Network.

In questo caso è difficile che ci siano delle slice duplicate; ognuna di esse ha nodi ed archi differenti. Ricordiamo infatti che ogni slice ha dei nodi che sono raggruppamenti dei nodi della slice sottostante.

Attenendoci alla definizione di Clustered Graph, possiamo anche in questo caso dare un'evidenza in maniera più formale:

**Teorema 2.3.2.** *Dato un Clustered Graph, possiamo creare un Multilevel Network che modelli le stesse informazioni.*

*Dimostrazione.* Dato un Clustered Graph $C = (G, T)$ con $G = (V_G, E_G)$, possiamo creare un Multilevel Network $(L, C_l)$, con $L = (l_0, l_1, ..., l_n)$ dove il livello $l_i = (N_i, A_i, v_i, d_i)$ $(\forall i \in \{0, 1, ..., n\})$ rappresenta il livello i-esimo dell'albero $T$. Per fare ciò, definiamo con $TL_i$ l'insieme dei nodi presenti in $T$ al livello i-esimo. I livelli $l_i$ (con $i \in \{0, 1, ..., n\}$) sono costruiti in modo tale che $n \in N_i \iff n \in TL_i$, dove il livello n-esimo è quello che ha un livello di dettaglio massimo (i.e. $l_n = V_G$). Il Level Coupling, identificato da $C_l$ ha esattamente n accoppiamenti $(c_0, c_1, ..., c_n)$, dove $c_i = (l_i, l_{i+i}, M_i, w_i)$; $M_i$ è l'insieme di coppie $(s, d)$ con $s \in l_i$ e $d \in l_{i+1}$ dove in $C$ abbiamo $d \in V(s)$. Il peso di ogni accoppiamento tra nodi, essendo ininfluente in questo caso, può essere definito come $w_i : M_i \to \{1\}$ dove $\forall (a, b) \in M_i$, $w_i((a, b)) = 1$. □

#### 2.3.1.3 MULTIPLEX NETWORK (O MULTIDIMENSIONAL NETWORK)

Sappiamo che un multigrafo è un grafo dove è possibile avere due archi differenti che collegano gli stessi nodi. Nella definizione di Multidimensional Network tale concetto di multigrafo viene meglio espresso. Questo è rappresentato dalla tripla $(V, E, L)$

dove l'insieme $L$ rappresenta difatti l'insieme dei possibili livelli necessari alla sua rappresentazione. Inoltre, ricordiamo che un arco appartiene alla dimensione $d$ solamente se è etichettato con $d$, e che un nodo appartiene ad una certa dimensione $d$ solamente se esiste un arco etichettato con $d$ che ha quel nodo in uno dei suoi due estremi. Tale visione è quindi più slegata dalle definizioni che spesso si incontrano sui multigrafi [13] e più legata invece al nostro concetto di grafo multi-livello, dove ogni livello è un semplice grafo correlato in qualche modo agli altri presenti negli altri livelli.

Il nostro modello, anche in questo caso è abbastanza generale per poter rappresentare questa tipologia di strutture, e cercheremo adesso di fornire una piccola dimostrazione in tal senso.

**Teorema 2.3.3.** *Dato un Multidimensional Network, possiamo creare un Multilevel Network che modelli le stesse informazioni.*

*Dimostrazione.* Dato un Multidimensional Network $G = (V, E, D)$, dove $V$ è l'insieme dei nodi, $E$ è l'insieme degli archi e $D$ invece è l'insieme delle etichette $\{e_1, e_2, ..., e_n\}$. Possiamo creare un Multilevel Network $(L, C_l)$, con $L = (l_0, l_1, ..., l_n)$, dove il livello $l_i = (N_i, A_i, v_i, d_i)$ $(\forall\, i \in \{0, 1, ..., n\})$ rappresenta la dimensione i-esima (i.e. $e_i$) del Multidimensional Network $G$. Per far ciò, $N_i$ contiene tutti i nodi della dimensione $e_i$, ovvero $N_i = \{v : \exists\ a = (x_1, x_2, e_i) \in E \wedge (x_1 = v \vee x_2 = v)\}$. L'insieme degli archi del livello $l_i$ contiene pertanto solamente una parte degli archi in $D$, in particolare: $A_i = \{(u, v) : \exists (u, v, e_i) \in D\}$. Il Level Coupling, identificato da $C_l$ accoppia ogni livello con gli altri, poiché non ci sono gli stessi vincoli che abbiamo visto nel Multiscale Network o nel Time-Dependent Network. Il peso di ogni accoppiamento tra nodi, come anche il peso degli archi che collegano i vari nodi ed i valori associati ad ogni nodo, possono essere fissati arbitrariamente poiché poiché sono degli elementi che non vengono rappresentati nel Multidimensional Network. □

### 2.3.2 MULTILAYER NETWORK

Il Multilayer Network, tra quelli che abbiamo visto, è il modello che più si avvicina alla nostra definizione di Multi Level Network. Esso comunque non permette di essere generale e allo stesso modo, poiché è nato principalmente per risolvere un problema specifico, a differenza di quelli che sono i nostri obbiettivi.

DIFFERENZE NEL MAPPING DEI LIVELLI Il Multilayer Network infatti si basa sul concetto di Node Mapping, similmente a ciò che viene fatto nel Multi Level Network con il Level Coupling. Il primo è però meno generale, e ciò deriva da due motivazioni principali.

- Un Node Mapping tra il Network Layer $L_1 = (V_1, w_1)$ ed il Network Layer $L_2 = (V_2, w_2)$ è una funzione $m : V_1 \times V_2 \rightarrow [0,1]$. Il nodo $u \in V_1$ si considera correlato a $v \in V_2$ soltanto se $m(u,v) > 0$. Al contrario, un accoppiamento tra il Network Level $L_1 = (N_1, A_1, v_1, d_1)$ e il Network Level $L_2 = (N_2, A_2, v_2, d_2)$, è una quadrupla $c = (L_1, L_2, M, w)$ dove $M$ è un insieme di coppie $(a,b)$, con $a \in N_1$ e $b \in N_2$, e dove $w : M \rightarrow Dom_W$ stabilisce il valore attribuito ad ogni relazione tra questi due livelli. Il Node Mapping, nei Multilayer Network, è stato pensato per modellare il fatto che un nodo in un particolare social network potesse replicare la diffusione di un'informazione in altri social network secondo una certa probabilità $p \in [0,1]$. Nel caso dei Multi Level Network questa è solamente una soltanto delle possibili applicazioni: potrebbe esserci un accoppiamento tra $L_1$ ed $L_2$ che assume lo stesso ruolo del Multilayer Network, ed un altro accoppiamento che invece mostra come le relazioni che coinvolgono nodi di social network differenti (si pensi ad un social network come Youtube che mostra le informazioni basandosi sulle relazioni di amicizia di un altro social network come Facebook), oppure che mostra relazioni sullo stesso social network (come accade nei Multidimensional Network, nei Multiscale Network oppure nei Time-Dependent Network).

- Nel Multilayer Network gli accoppiamenti tra i vari Layer sono definiti da una matrice $n \times n$ di Node Mapping. Ciò esclude la presenza di più accoppiamenti tra due Layer differenti, che potrebbero invece essere utili per rappresentare situazioni differenti. Al contrario, nel Multi Level Network, i Levels Coupling sono definiti come una tupla $C_l = (c_1, c_2, ..., c_n)$ di accoppiamenti, che potrebbero benissimo anche coinvolgere gli stessi livelli più volte.

- Nel Multilayer Network, il fatto che gli accoppiamenti sia definito con una matrice implica che ogni nodo di ogni livello è potenzialmente correlato con qualsiasi altro nodo di qualunque altro livello, e l'unico modo per sapere se due livelli $i$ e $j$ non sono correlati è quello di verificare che la funzione $\forall a \in V_i, b \in V_j \,.\, IM_{i,j}(a,b) = 0$, sicuramente più oneroso dell'approccio che abbiamo adottato quando abbiamo definito i Levels Coupling dei Multi Level Network.

DIFFERENZE NEI SINGOLI LIVELLI Possiamo osservare come i Multi Level Network siano più generali anche a livello del singolo livello. Possiamo osservare per prima cosa come, nei Multilayer Network, i singoli nodi non prevedono di avere associato un valore, mentre invece in un Network Level la funzione $d : N \to Dom_N$ può associare un valore ad ogni nodo.

Inoltre, la costruzione degli archi è onerosa in una reale implementazione: un Network Layer $L = (V, w)$, dove $V$ è l'insieme dei nodi, prevede che la funzione $w : V \times V \to [0,1]$ stabilisca il collegamento tra i nodi, lasciando intendere che i nodi $a$ e $b$ non sono collegati da un arco nel caso in cui $w(a,b) = 0$. Un Network Level $L = (N, A, v, d)$ invece ha l'insieme $A \subseteq N \times N$ che definisce gli archi del grafo, e l'etichettatura degli archi, definita dalla funzione $v : A \to Dom_A$, richiede quindi meno informazioni da memorizzare, specialmente nei casi in cui il grafo non è interamente connesso. Ciò non è soltanto utile in termini di informazioni da memorizzare, ma ciò si ripercuote anche sul tempo di esecuzione delle operazioni, che hanno bisogno di scoprire se due nodi $a$ e $b$ sono connessi, controllando ogni volta che la funzione $w(a,b)$ sia maggiore di zero.

Infine, l'etichettatura degli archi dei Network Level possono avere associato un valore arbitrario, non soltanto limitato ai valori contenuti nell'intervallo $[0,1]$, e dove possiamo utilizzare tutto il range dei valori (si pensi ad esempio che, con $Dom_A = [0,1]$, l'arco tra i nodi $a$ e $b$, con associato il valore 0, non significa che i due nodi siano sconnessi).

RIDUZIONE Adesso vedremo come sia possibile, dato un Multilayer Network, avere una rappresentazione semanticamente identica nel nostro modello che abbiamo chiamato Multi Level Network.

**Teorema 2.3.4.** *Dato un Multilayer Network, possiamo creare un Multilevel Network che modelli le stesse informazioni.*

*Dimostrazione.* Dato un Multilayer Network $MLN = (L_1, L_2, ..., L_n, IM)$ dove $L_i = (V_i, w_i)$ (con $i \in \{1,2,...,n\}$) sono Network Layer, e dove IM è una matrice $n \times n$ di Node Mapping, con $IM_{i,j} : V_i \times V_j \rightarrow [0,1]$. Possiamo costruire un Multi Level Network $M = (L, C_l)$ dove $L = (l_1, l_2, ..., l_n)$ e dove $C_l = (c_{1,1}, c_{1,2}, ..., c_{n,n})$ identifica i loro accoppiamenti.

In particolare, il Network Level $l_i = (N_i, A_i, v_i, d_i)$ (con $i \in \{1,2,...,n\}$) rappresenta il Network Layer $L_i$, dove abbiamo $N_i = V_i$ e dove $A_i = \{(a,b) : a,b \in V_i \wedge w(a,b) > 0\}$. Inoltre, la funzione $v_i$ è definita in modo che l'output sia nell'intervallo $[0,1]$ (i.e $Dom_v = [0,1]$), e che $\forall (a,b) \in A_i \ . \ v_i((a,b)) = w(a,b)$. Infine possiamo ignorare la composizione della funzione $d_i$, in quanto i Multilayer Network non prevedono di associare dei valori ai singoli nodi.

Gli accoppiamenti $c_{i,j}$ (con $i,j \in \{1,2,...n\}$) sono invece definiti in modo che rispecchino quelli della matrice. In particolare, $c_{i,j} = (l_i, l_j, M_{i,j}, w_{i,j})$, dove la correlazione dei nodi viene rispettata mediante l'insieme $M_{i,j} = \{(a,b) : a \in l_i \wedge b \in l_j \wedge IM_{i,j}(a,b) > 0\}$, e dove il valore delle singole correlazioni viene anch'essa rispettata definendo $\forall (a,b) \in M_{i,j} \ . \ w_{i,j}((a,b)) = IM_{i,j}(a,b)$. Per far ciò la funzione $w_{i,j}$

(con $i, j \in \{1, 2, ...n\}$) è definita in modo che l'output sia nell'intervallo $[0, 1]$, ovvero $Dom_W = [0, 1]$. □

# CAPITOLO 3

# ALGEBRA

## 3.1 FONDAMENTI

Una volta costruito il modello viene naturale pensare di definire degli operatori opportuni per poter manipolare queste reti, che abbiamo chiamato Multi Level Network. Data la natura e lo scopo del nostro studio ci concentreremo non tanto nella creazione di operatori che costruiscano tali reti, ma piuttosto cercheremo di definire degli operatori che possano estrapolare dei dati. Questo perché è ragionevole pensare che le reti siano già esistenti e siano continuamente manipolate dagli stessi servizi (si pensi ad esempio a Facebook), mentre ciò che vogliamo fare è analizzare tali reti, con opportuni operatori per l'estrazione dei dati, relazionando anche reti diverse provenienti da servizi differenti, utilizzando il concetto già visto di Multi Level Network. Tali operatori comunque non escludono il fatto di poter essere utilizzati per la creazione di nuovi contenuti a partire da quelli già esistenti, come difatti vedremo con l'operatore di sintesi.

Nel nostro studio abbiamo visto che molti modelli che sono stati definiti sono molto semplici, e spesso si basano su tecnologie preesistenti basate sul modello relazionale, come ad esempio accade in Oracle Spatial. Allo stesso modo abbiamo potuto notare

come anche i linguaggi siano a volte elementari, proprio per garantire di avere un linguaggio molto espressivo, ma che infine sono lontani dal determinare come possano essere utilizzati negli scopi reali. Il nostro approccio è leggermente differente: vorremmo definire degli operatori, il più elementari possibile, che, anche combinati tra loro, forniscano una risposta a quelle che riteniamo essere le più comuni interrogazioni su queste tipologie di reti. Abbiamo riscontrato come sempre di più negli ultimi anni questo approccio si sia diffuso, con articoli che sempre meno parlano del potere espressivo da un punto di vista troppo teorico ed al contrario invece considerano come tali linguaggi possano essere applicati in contesti reali.

Nelle nostre definizioni ci siamo sforzati molto nel definire degli operatori con la stessa filosofia del modello relazionale [30], cercando di comprendere come ognuno di questi potesse essere interpretato sui grafi e come potesse poi essere combinato con gli altri operatori in maniera consistente per formulare query complesse. Siamo partiti quindi dal definire da un punto di vista insiemistico ciò che poteva essere un risultato di una query, dando una struttura che ricorda molto le tuple del modello relazionale.

Come accade nell'Oracle Spatial la definizione di un grafo, su un database relazionale, può essere fatta mediante l'uso di due tabelle: una che memorizza i nodi del grafo ed un'altra invece che ne memorizza gli archi. È intuibile come trovare un percorso all'interno di un grafo memorizzato in questo modo sia abbastanza oneroso. Infatti ciò costringe ad effettuare $n$ join per costruire un percorso di lunghezza $n$. Tale problema di ricorsione è noto, ed è molto spesso considerato nelle definizioni dei linguaggi di interrogazione sui grafi. Dal nostro punto di vista abbiamo risolto tale problema definendo un nostro concetto di path, ed un nostro concetto di pattern su tali path, utilizzando un approccio che ricorda molto le espressioni regolari. In tal modo è possibile ricercare tutti i percorsi che hanno una particolare struttura all'interno del grafo, e di conseguenza costruire su di esso gli altri operatori, con una filosofia molto vicina al modello relazionale, che sarà chiara una volta che li avremo mostrati. Abbiamo riscontrato in seguito come l'utilizzo dei path, e di pattern su tali

path (utilizzando le espressioni regolari), sia comunque già stato adottato in diversi lavori [21, 31, 32], anche se in maniera molto differente.

Vedere un recente interesse per l'utilizzo dei path, e di espressioni regolari come pattern, è stato sicuramente soddisfacente ed ha ancora di più confermato il lavoro svolto e motivato un interesse nel suo proseguimento. Infatti si è potuto notare come ci siano altri approcci [33, 34] che si concentrano sul creare operatori di selezione che utilizzano dei grafi come pattern, ricercando poi tutti i sotto-grafi ad esso isomorfi, un problema che sappiamo essere NP-completo [35, 36]. Riteniamo comunque tale approccio sicuramente più generale, in quanto ci consente di esprimere, ad esempio, la volontà di cercare dei sotto-grafi che abbiano dei cicli, cosa che utilizzando i path non è invece possibile fare; d'altra parte però un simile approccio è computazionalmente molto più pesante, ed utilizzabile quasi esclusivamente su database molto piccoli. Ciò è stato anche uno dei motivi che ci ha permesso di intuire l'utilizzo dei pattern sui path, che altro non è che una visione semplificata della ricerca di sotto-grafi isomorfi.

Per definire la nostra algebra daremo per prima cosa una definizione di path e di insieme di path. In seguito definiremo un operatore di proiezione $\pi$, in grado di estrapolare un sotto-path di un path singolo o eseguire dei tagli verticali su insiemi di path, similmente a quanto accade nell'algebra relazionale con gli insiemi di tuple. Successivamente definiremo anche un semplice operatore di concatenazione, che permette la concatenazione di path. La definizione dei path sarà utile per poi definire l'operatore di selezione che ha come risultato un insieme di path che hanno una struttura simile, ovvero che sono conformi ad un determinato path pattern. Per fare ciò definiremo cosa sia un path pattern, descrivendo ricorsivamente la sua sintassi, e successivamente definiremo anche una relazione di soddisfacibilità, ovvero definiremo quando un path $p$ è strutturalmente conforme ad un path pattern $P$ (indicato con $p \vDash P$). Tale semantica verrà per chiarezza definita in due modi differenti, entrambi dimostrati essere equivalenti: nel primo caso definiremo ricorsivamente

quando $p \vDash P$, mentre nel secondo caso definiremo l'insieme dei path che soddisfa un determinato path pattern P (ovvero $\{p : p \vDash P\}$).

L'operatore di selezione restituisce insiemi di path, che hanno delle informazioni aggiuntive rispetto ad un sotto-grafo, ma nonostante ciò, può essere utile creare dei sotto-grafi a partire da questi insiemi di path (si pensi al sotto-grafo dei soli nodi utili per andare da un nodo $a$ ad un nodo $b$). Per questo motivo definiremo in seguito anche un operatore, che chiameremo operatore di sintesi $\eta$, in grado di generare un sotto-grafo a partire dal risultato dell'operatore di selezione. Tale operatore di sintesi contribuisce anche a creare un'algebra chiusa: si pensi ad esempio all'uso annidato dell'operatore di selezione.

Per realizzare alcune query complesse nei contesti reali, definiremo anche un operatore di aggregazione $\alpha$, reso possibile soprattutto grazie al fatto che l'operatore di selezione ritorni insiemi di path, piuttosto che direttamente sotto-grafi.

Infine, per riuscire ad estrapolare informazioni da più livelli, definiremo anche un operatore di Join $\bowtie$, che ha il compito di fondere due livelli secondo certi criteri.

Dopo aver dato la definizione di tutti questi operatori illustreremo anche le differenze principali con alcuni lavori correlati di linguaggi su grafi che si possono trovare in letteratura, e che utilizzano concetti simili.

## 3.2 PATH

Con il termine *path* viene comunemente indicata una sequenza ordinata di nodi $v_1, v_2, ..., v_n$ che forma un "cammino" all'interno di un grafo. Questa sequenza di nodi caratterizza un cammino univoco, poiché unici sono gli archi che collegano i vari nodi contenuti nella stessa. Da notare che in letteratura possono essere trovate anche altre definizioni che, ad esempio, raffigurano i path come una sequenza di archi, o come una sequenza alternata di archi e nodi; queste definizioni alternative non vengono prese in esame, ed in seguito, quando parleremo di path, ci riferiremo esclusivamente alla seguente definizione.

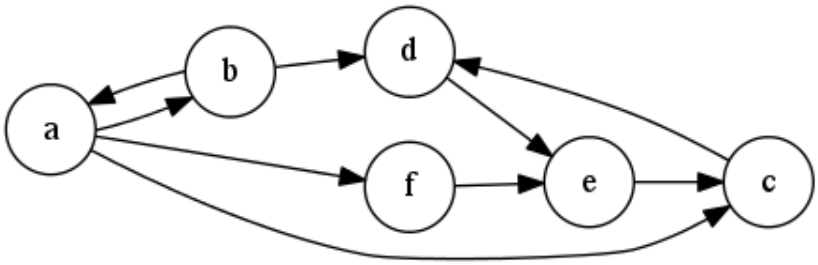



**Figura 3.2.1:** Un Network Level $G_g = (N_g, A_g, v_g, d_g)$ che rappresenta una piccola rete sociale di Twitter: i nodi rappresentano i singoli utenti, mentre gli archi rappresentano le relazioni unidirezionali di *Following*. Ogni singolo nodo viene identificato con una lettera, si noti ancora una volta che queste non rappresentano il loro dato, ma i nomi con la quale li identificheremo. Per riferirci ai loro dati si utilizzano le funzioni $d$ e $v$, che rispettivamente restituiscono il dato di un nodo ed il dato di un arco.

**Definizione 3.2.1.** Sia $G = (N, A, v, d)$ un Network Level. Un *path* di $G$ è una tupla $P = (p_1, p_2, ..., p_n)$ (con $n \in \mathbb{N}$), dove ogni nodo della tupla è anche un nodo di $G$ (i.e. $\forall i \in \{1, 2, ..., n\}\ \ p_i \in N$) ed anche dove ogni coppia formata da nodi adiacenti sia un arco di $G$ (i.e. $\forall i \in \{1, 2, ..., n-1\}\ \ (p_i, p_{i+1}) \in A$). Inoltre, tutti i nodi devono essere distinti (i.e. $\forall i \in \{1, 2, ..., n\}\ \nexists j \in \{1, 2, ..., i-1, i+1, ..., n\}$ t.c. $p_i = p_j$).

**Esempio 3.2.1.** Guardando la Figura 3.2.1, possiamo notare che nel Network Level $G_g = (N_g, A_g, v_g, d_g)$, esistono molteplici path, ad esempio $(a, b)$, $(a, c, d)$. Possiamo verificare che queste siano effettivamente dei path verificando per la prima che $(a, b) \in A_g$, mentre per la seconda che $(a, c) \in A_g \wedge (c, d) \in A_g$. La tupla $(d, z)$ non è un path, poiché $z \notin N_g$, come neppure la tupla $(f, e, d)$ poiché $(e, d) \notin A_g$.

### 3.2.1 Lunghezza

La prima nozione basilare legata ai path è la loro lunghezza. Definiamo pertanto la lunghezza come il numero di archi che compongono il path stesso. Ciò vuol dire che ogni path $P = (p_1, p_2, ..., p_n)$, composta di $n$ nodi, ha lunghezza $n-1$. In futuro ci riferiremo alla lunghezza di un path utilizzando la funzione $len$, che presa un path

ritorna la sua lunghezza. Si noti che nel caso particolare di un path di un solo elemento, ad esempio il singoletto $(s_1)$, abbiamo lunghezza 0. Nel caso invece di un path vuoto, la lunghezza è $-1$, indicata anche come lunghezza nulla.

**Esempio 3.2.2.** Anche in questo caso possiamo fare qualche esempio per essere più esplicativi, nonostante il concetto si abbastanza semplice e noto. Guardando la Figura 3.2.1, possiamo analizzare il path $P = (a, b, d)$ che passa per i nodi $a$, $b$ e $d$; essa ha lunghezza 2, poiché due sono gli archi che formano il path. Utilizzando la funzione $len$, scriveremo che $len(P) = 2$.

Quando si parla di grafi pesati (i.e. network), la lunghezza dei cammini viene a volte definita come la somma dei pesi degli archi coinvolti. Nel nostro caso, la nostra definizione di lunghezza non viene influenzata dal peso degli archi, e dal loro tipo in genere. Ciononostante non si esclude comunque la possibilità, per il lettore, di definire altre funzioni che esibiscano comportamenti differenti, in base alle proprie esigenze.

### 3.2.2 Path e Simple Path

In letteratura si fa spesso una distinzione netta tra quelli che vengono chiamati *path* e quelli che vengono chiamati *simple path*. Con il primo viene solitamente indicato un cammino all'interno di un grafo, escludendo il vincolo di non poter avere nodi ripetuti. Con il secondo viene comunemente indicato un path, come quello della definizione 3.2.1, che invece non può avere nodi ripetuti. Intenderemo implicitamente, come succede anche per altri autori [29], che ogni path sia una simple path, riservando la parola "walk" alle path che possono avere nodi ripetuti.

Le motivazioni che ci hanno spinto a limitarci di considerare solamente i simple path saranno più chiare in seguito, quando parleremo dell'operatore di selezione. Infatti, per intuire subito il concetto che ripresenteremo in seguito, possiamo notare che guardando l'esempio 3.2.1, i possibili walk che hanno come nodo iniziale $a$ e come nodo finale $f$ sono infinite, per via, ad esempio, del ciclo tra i nodi $a$ e $b$.

## 3.3 Operatore di Concatenazione

Poniamo il caso di avere un determinato path $P = (p_1, p_2, ..., p_n)$, ed un'altro $P' = (p'_1, p'_2, ..., p'_{n'})$. A volte c'è la necessità di unire (o concatenare) questi due path per formarne uno nuovo, nel nostro caso vorremmo formare il path $(p_1, p_2, ..., p_n, p'_1, p'_2, ..., p'_{n'})$.

**Definizione 3.3.1.** Sia $G = (N, A, v, d)$ un Network Level, e siano $P = (p_1, p_2, ..., p_n)$ e $P' = (p'_1, p'_2, ..., p'_{n'})$ due path di $G$. Definiamo l'operazione di concatenazione tra due path nel seguente modo:

$$P.P' = \begin{cases} (p_1, ..., p_n, p'_1, ..., p'_{n'}) & \text{se } n \geq 0 \wedge n' \geq 0 \\ (p_1, ..., p_n) & \text{se } n \geq 0 \wedge P' = () \\ (p'_1, ..., p'_{n'}) & \text{se } P = () \wedge n' \geq 0 \\ () & \text{se } P = () \wedge P' = () \end{cases}$$

Si noti che la concatenazione di due path può formare path che non sono più esistenti. Il path vuoto invece rappresenta una sorta di elemento neutro della congiunzione.

**Esempio 3.3.1.** Guardando la Figura 3.2.1, possiamo avere i path $P_1 = (a, b, d)$, $P_2 = (e, c)$ e $P_3 = (f, e)$. Notiamo che mentre $P_1.P_2$ è un path di $G$, al contrario $P_1.P_3$ non lo è, poiché $(d, f) \notin A$.

## 3.4 Operatore di Proiezione

Dopo aver discusso di come è definito un path, e quindi anche di come può essere creato, dobbiamo anche definire come questo può essere distrutto.

Per prima cosa l'operatore di proiezione deve poter lavorare con un singolo path, permettendoci di estrapolare un determinato suo elemento, oppure di estrapolare un determinato suo sotto-path. In questo senso possiamo osservare il seguente esempio:

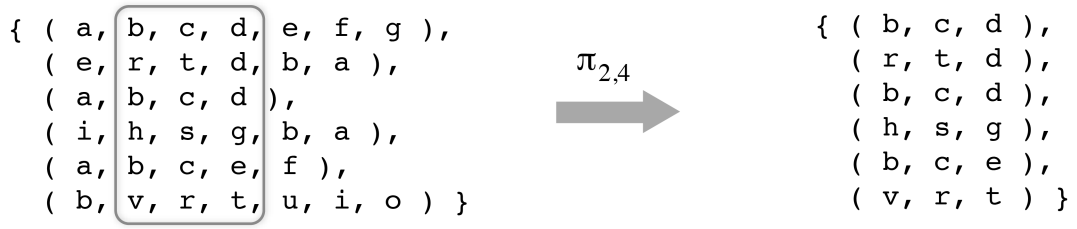



**Figura 3.4.1:** A sinistra un esempio di un insieme di path, anche di lunghezza differente, dove sono state evidenziate tutti i path che vorremmo estrapolare, facendo appunto un taglio verticale. Utilizzando l'operatore di proiezione si estraggono, per ogni path, i nodi che vanno dal secondo al quarto. Il risultato pertanto è rappresentato nella parte di destra.

**Esempio 3.4.1.** Considerando la Figura 3.2.1, prendiamo il path $P = (a, f, e, c, d)$. Con il nostro operatore di proiezione vorremmo prendere un suo elemento in particolare, per esempio l'ultimo (i.e. $d$). Inoltre, vorremmo anche poter prendere una parte del path, per esempio dal terzo all'ultimo nodo (i.e. $(e, c, d)$). Da notare che non vorremmo invece che il nostro operatore di proiezione ci permettesse di selezionare sottoparti del path che potrebbero non esistere, come per esempio il sotto-path $(a, c, d)$.

Inoltre, vorremmo anche che il nostro operatore di proiezione possa lavorare con insiemi di path, effettuando un "taglio verticale" a tali insiemi. Per chiarificare le motivazioni di questa seconda osservazione, possiamo osservare l'esempio successivo:

**Esempio 3.4.2.** Nella Figura 3.2.1, costruiamo un insieme di path $P$, dove ogni path ha lunghezza 1, ed ha come nodo iniziale $a$. Partendo da tale insieme $P = \{(a, b), (a, f), (a, c)\}$, vorremmo ad esempio determinare tutti i vicini di $a$, prendendo solamente i nodi che sono in seconda posizione.

Per rendere più chiaro invece il concetto che comunemente viene chiamato taglio verticale, possiamo pensare a ciò che accade con l'operatore di proiezione dell'algebra relazionale. Richiamiamo questo concetto guardando l'esempio in tabella 3.4.1.

| Nome | Età | Peso |
|---|---|---|
| Mario | 45 | 68 |
| Erica | 53 | 60 |
| Giorgio | 25 | 72 |

| Nome | Peso |
|---|---|
| Mario | 68 |
| Erica | 60 |
| Giorgio | 72 |

**Tabella 3.4.1:** Un esempio grafico della proiezione nell'algebra relazionale. A sinistra abbiamo la relazione $R$ che contiene gli attributi *Nome*, *Età* e *Peso*; a destra la proiezione $\pi_{Nome,\ Peso}(R)$.

Come abbiamo già osservato nell'esempio 3.4.2, a differenza dell'algebra relazionale, non vogliamo effettuare un taglio verticale prendendo solamente alcune colonne della relazione, ad esempio in maniera sparsa come in tabella 3.4.1, poiché questo potrebbe generare path che in realtà non esistono. Per fare ciò, dobbiamo limitare la selezione degli elementi all'interno del path, facendo in modo che sia possibile selezionare solamente parti contigue. Inoltre, osservando che ogni path può avere una propria lunghezza $n$, consideriamo naturale la selezione di elementi in una posizione $i > n+1$, in questo caso il valore di ritorno sarà semplicemente l'insieme vuoto.

**Definizione 3.4.1.** Sia $G = (N, A, v, d)$ un Network Level e sia $Path_G$ l'insieme di tutte le possibili path di $G$. Si definisce l'operatore di proiezione $\pi_{s,e}(P)$, dove $s : Path_G \to \mathbb{N}_{>0}$, $e : Path_G \to \mathbb{N}_{>0}$, ed anche che $\forall p \in Path_G.\ s(p) \leq e(p)$, nel seguente modo:

$$\pi_{s,e}(P) = \begin{cases} \bigcup_{p \in P} \pi_{s,e}(p) & \text{se } P \text{ è un insieme di path} \\ (n_{s(P)}, n_{s(P)+1}, ..., n_{min(e(P),k)}) & \text{se } P = (n_1, n_2 ..., n_k) \text{ è un path e } s(P) \leq k \\ \emptyset & \text{se } P = (n_1, n_2, ..., n_k) \text{ è un path e } s(P) > k \end{cases}$$

Possiamo subito discutere delle funzioni $s$ e $e$, che potrebbero essere la cosa più difficile da comprendere. Queste sono semplicemente degli indici, rispettivamente di inizio e di fine, e che possono dipendere dal path che si sta analizzando. Possiamo quindi avere delle funzioni costanti come $s(p) = 1$ ed $e(p) = 2$, che sono come degli

indici che selezionano sempre il sotto-path formato dal primo e secondo nodo, oppure possono essere funzioni che dipendono dal path che si sta analizzando, come $s(p) = 2$ ed $e(p) = len(p) + 1$, che rimuovono da ogni path il primo elemento.

Per non sovraccaricare la scrittura, specificheremo sempre negli indici il corpo della funzione, lasciando intendere che questo prenda in input un path $p$. Ad esempio con $\pi_{1,2}(P)$, intendiamo $\pi_{s,e}(P)$ dove $s(p) = 1$ e $e(p) = 2$. Allo stesso modo, per fare un altro esempio, con $\pi_{2,len(p)}(P)$, intendiamo $\pi_{s,e}(P)$ dove $s(p) = 2$ e $e(p) = len(p)$.

**Esempio 3.4.3.** Riprendiamo l'esempio 3.4.1, dove abbiamo il path $P = (a, f, e, c, d)$. Per selezionare il primo elemento possiamo utilizzare l'operatore di proiezione in questo modo:

$$\pi_{1,1}(P) = (a)$$

Per selezionare invece l'ultimo elemento possiamo invece utilizzarlo in quest'altro modo:

$$\pi_{(len(p)+1),(len(p)+1)}(P) = (d)$$

Per selezionare invece il sotto-path che va dal terzo all'ultimo nodo possiamo invece utilizzarlo assegnando ai due argomenti della proiezione diversi valori:

$$\pi_{2,(len(p)+1)}(P) = (e, c, d)$$

**Esempio 3.4.4.** Riprendiamo l'esempio 3.4.2, dove abbiamo invece un insieme di path $P = \{(a, b), (a, f), (a, c)\}$. Per riuscire ad avere tutti i vicini in questo caso dobbiamo sfruttare la natura polimorfa di questo operatore:

$$\pi_{(len(p)+1),(len(p)+1)}(P) = \{(b), (f), (c)\}$$

Su questo operatore, che viene utilizzato di frequente, possiamo introdurre alcuni abusi di notazione che verranno utilizzati in futuro dove non ci sarà nessun pericolo

di creare confusioni, e che renderanno più semplice la comprensione. Pertanto intenderemo con $\pi_s$ l'operatore di proiezione $\pi_{s,e}$ dove $s = e$. Riprendendo l'esempio 3.4.3, prendendo il path $P$, il risultato di $\pi_{(len(p)+1)}(P)$ sarà pertanto il path $(d)$ che è composto dal solo nodo $d$. Anche in quest'ultimo caso, laddove non si creino confusioni, ci riferiremo a $\pi_i(p)$, con $i \in \mathbb{N}$, come il nodo i-esimo del path $p$, piuttosto che il path composto solamente dal medesimo elemento. Riprendendo l'esempio 3.4.4, il risultato della proiezione, composto da tanti path di lunghezza 0, lo potremmo intendere invece come un insieme di nodi, che rappresentano i vicini del nodo $a$. Viene fatta invece meno confusione per quanto riguarda gli archi, visto che, avendo ad esempio un path $p = (v_1, v_2, ..., v_n)$, è indifferente riferirsi all'arco tra il nodo $v_i$ e $v_{i+1}$ nei seguenti modi: $(v_i, v_{i+1})$, oppure $(\pi_i(p), \pi_{i+1}(p))$, oppure $\pi_{i,i+1}(p)$.

## 3.5 Operatore di Selezione

I path, come abbiamo visto, costituiscono uno dei fondamenti della nostra algebra. Per riuscire a recuperare questi path all'interno di un grafo, e quindi poter definire concretamente un operatore di selezione, è necessario però introdurre un concetto di pattern. In tal modo possiamo possiamo chiedere ad un operatore (i.e. di selezione) di collezionare tutti i path che abbiano una determinata struttura descritta da un pattern. Ovviamente la scelta effettuata dall'operatore di selezione non può dipendere solamente dalla struttura dei path, ma c'è bisogno anche di guardare al loro contenuto; per questo motivo l'operatore di selezione ha bisogno anche di una formula booleana, che determini se un determinato path, che sia conforme ad una determinata struttura, abbia anche un determinato contenuto.

Procederemo quindi con l'introdurre i path pattern, definendoli e specificando quali siano i path che vengono descritti da un determinato path pattern. In seguito definiremo anche l'operatore di selezione, utilizzando le definizioni che riguardanti i path pattern. Successivamente daremo anche qualche esempio per riuscire a vedere le cose da un punto di vista pratico.

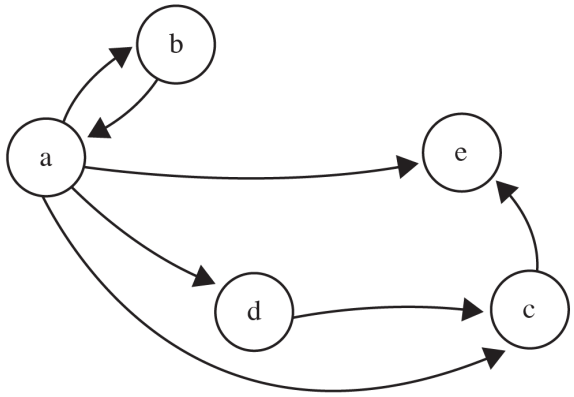



**Figura 3.5.1:** Un Network Level $G_g = (N_g, A_g, v_g, d_g)$ che rappresenta una piccola rete sociale di Twitter: i nodi rappresentano i singoli utenti, mentre gli archi rappresentano le relazioni unidirezionali di *Following*.

### 3.5.1 PATH PATTERN

Un pattern è molto simile ad un path, quest'ultimo infatti può essere visto come un pattern, anche se troppo preciso poiché identifica in modo univoco un solo path presente nel grafo. Dobbiamo quindi rilassare il vincolo che obbliga tutte le coppie di nodi presenti nel path ad essere effettivamente degli archi contenuti nel grafo. In questo modo possiamo anche rappresentare dei path che in realtà non esistono.

Per fare un esempio di quest'ultimo concetto possiamo riferirci alla Figura 3.5.1, dove non possiamo avere un path $(b, c, e)$, poiché non esiste nessun arco che colleghi il nodo $b$ con il nodo $c$. D'altra parte possiamo invece costruire un pattern che descriva un simile path, semplicemente l'operatore di selezione non riuscirà a trovare nessun risultato.

Ma c'è un altro concetto altrettanto importante: dobbiamo avere bisogno di esprimere nodi e path generici. Introdurremo infatti degli elementi particolari, che identificheremo con i simboli "%", "?" e "$*$". Questi identificano rispettivamente nodi generici che devono obbligatoriamente essere presenti, nodi generici che possono non

essere presenti e path generici che possono essere vuote. In questo modo, saremo in grado di identificare non solo un cammino ben preciso, ma anche uno generico.

Questo tipo di pattern ricorda molto le espressioni regolari, e sicuramente non siamo lontani dal dire che le due cose non siano affatto diverse nonostante il diverso domino di applicazione. Procederemo adesso con una definizione ricorsiva della sintassi di un path pattern.

**Definizione 3.5.1.** Sia $G = (N, A, v, d)$ un Network Level. Un *path pattern* di $G$ è definito ricorsivamente nel seguente modo:

| | | |
|---|---|---|
| $\emptyset$ | è un path pattern. | (*empty set*) |
| $\epsilon$ | è un path pattern. | (*empty path*) |
| $n$ | con $n \in N$, è un path pattern. | (*graph node*) |
| % | è un path pattern. | (*generic node*) |
| ? | è un path pattern. | (*optional generic node*) |
| $*$ | è un path pattern. | (*generic path*) |

Inoltre, se $P_1$ e $P_2$ sono dei path pattern:

| | | |
|---|---|---|
| $P_1 \rightarrow P_2$ | è un path pattern. | (*concatenazione*) |
| $P_1 \mid P_2$ | è un path pattern. | (*alternazione*) |

Dopo aver definito la sintassi dei path pattern, possiamo adesso definire quale sia il loro significato, definendone la semantica. Per fare questo dobbiamo descrivere quali sono i *path che soddisfano un determinato path pattern*, ovvero, cosa vuol dire che un certo path sia strutturalmente conforme ad un path pattern. Scriveremo pertanto che $p \models P$, per indicare che un path $p$ soddisfa il path pattern $P$.

**Definizione 3.5.2.** Sia $G = (N, A, v, d)$ un Network Level. Definiamo la relazione di soddisfacibilità $\vDash$ ricorsivamente nel seguente modo:

*Caso base:*

Sia $P$ un path pattern di $G$, e $p$ un path di $G$.

- Se $P = \emptyset$ allora $\nexists p \vDash P$
  ovvero, $\emptyset$ può essere visto come un pattern che esprime il falso.
- Se $P = \epsilon$ allora $p \vDash P \iff p = ()$
  ovvero, $\epsilon$ è un pattern che identifica il path vuoto.
- Se $P = n$ con $n \in N$, allora $p \vDash P \iff p = (n)$
  ovvero, $n$ è un pattern che identifica in modo preciso un nodo $n$ presente in $G$, più precisamente il path $p$ di lunghezza 0 formato dal solo nodo $n$.
- Se $P = \%$ allora $p \vDash P \iff p = (p_1) \land p_1 \in N$
  ovvero, % è un pattern che identifica un generico nodo di $G$, più precisamente un path $p$ di lunghezza 0 formato da un solo nodo $p_1$ presente in $G$.
- Se $P = ?$ allora $p \vDash P \iff (p = (p_1) \land p_1 \in N) \lor p = ()$
  ovvero, ? è un pattern che identifica un generico nodo di $G$, più precisamente il path $p$ che può essere: di lunghezza 0 formato da un solo nodo $p_1$, oppure un path vuoto.
- Se $P = *$ allora $p \vDash P \iff p$ è un path di $G$
  ovvero, $*$ è un pattern che identifica un generico path di $G$. Si noti che questo può anche essere vuoto, ovvero avere lunghezza nulla (i.e. $len(p) = -1$).

*Caso ricorsivo:*

Siano $P$ e $P'$ path pattern di $G$, ed anche $p = (p_1, p_2, ..., p_n)$ e $p' = (p'_1, p'_2, ..., p'_{n'})$ path di $G$.

- Se $p \vDash P \vee p \vDash P'$, allora $p \vDash P|P'$
  ovvero, $P|P'$ è un pattern che rappresenta un'alternazione, per soddisfare l'intero path pattern è necessario che una soltanto delle due alternative venga soddisfatta.

- Se $p \vDash P \wedge p' \vDash P'$ allora

  – Se $n \geq 0 \wedge n' \geq 0 \wedge p.p'$ è un path di $G$ allora $p.p' \vDash P \rightarrow P'$

  – Se $p = () \vee p' = ()$ allora $p.p' \vDash P \rightarrow P'$

  ovvero, $P \rightarrow P'$ è un pattern che rappresenta una concatenazione, l'unione dei due path è soddisfatta dal path pattern se esiste un arco che le congiunge.

Analogamente, per essere più chiari possiamo anche definire la relazione di soddisfacibilità in un'altra prospettiva, indicando con $M(P)$, dove $P$ è un path pattern, come l'insieme di tutti i path che soddisfano il path pattern $P$.

**Definizione 3.5.3.** Sia $G = (N, A, v, d)$ un Network Level, ed anche $P$ un path pattern di $G$. Definiamo $M(P)$ ricorsivamente nel seguente modo:

*Caso base:*

- Se $P = \emptyset$ allora $M(P) = \emptyset$
- Se $P = \epsilon$ allora $M(P) = \{ () \}$
- Se $P = n$ con $n \in N$, allora $M(P) = \{ (n) \}$
- Se $P = \%$ allora $M(P) = \{(n) : n \in N\}$
- Se $P = ?$ allora $M(P) = \{(n) : n \in N\} \cup \{ () \}$
- Se $P = *$ allora $M(P) = \{p : p \text{ è un path di } G\}$

*Caso ricorsivo:*

Siano $P_1$ e $P_2$ due path pattern di $G$.

- Se $P = P_1|P_2$ allora $M(P) = \{p : p \in M(P_1) \vee p \in M(P_2)\}$
- Se $P = P_1 \rightarrow P_2$ allora
  $M(P) = \{p.p' : p \in M(P_1) \wedge p' \in M(P_2) \wedge p.p'$ è un path di $G\}$

#### 3.5.1.1 ESEMPI

Per rendere le cose un po' più pratiche, riporteremo adesso qualche esempio, riconsultando la Figura 3.5.1.
Guardando gli elementi di base, possiamo fare alcuni esempi che fanno vedere come esprimere dei nodi particolari.

**Esempio 3.5.1.** Guardando la Figura 3.5.1, possiamo avere il path pattern $a$ che identifica unicamente il nodo che nella figura stessa è stato identificato con $a$. Con un path pattern % invece identifichiamo un generico nodo, ad esempio $a$ oppure $b$. Invece, con un path pattern ? identifichiamo gli stessi nodi di quello precedente, ma in più anche il path vuoto ( ). Con il path pattern $*$ identifichiamo invece un path generico all'interno di quel grafo, come ad esempio $(a,d,c)$ oppure anche $(a,d,c,e)$.

Questi elementi di base possono essere combinati, come abbiamo visto nelle definizioni, per costruire pattern che riconoscano strutture più complesse, e che abbiano una determinata forma.

**Esempio 3.5.2.** Guardando la Figura 3.5.1, possiamo avere il path pattern $a \rightarrow b$ che identifica l'arco tra $a$ e $b$. Quest'ultima è formata da due sotto-pattern $a$ e $b$, e la congiunzione tra queste identifica un path perché nel grafo esiste l'arco $(a,b)$. Inoltre, possiamo avere anche più sequenze di concatenazioni, come $a \rightarrow e \rightarrow d$ che identifica un path preciso che in realtà non esiste nel grafo. Oppure ancora $a \rightarrow \% \rightarrow c$ che identifica uno generico path di due archi che ha come punto iniziale $a$ e come punto finale $c$, in questo caso solamente $(a,d,c)$. Possiamo poi avere un path pattern $a \rightarrow ? \rightarrow c$, che come prima identifica $(a,d,c)$, ma anche $(a,c)$.

**Esempio 3.5.3.** Guardando la Figura 3.5.1, possiamo avere il path pattern $a|b$ che identifica il nodo $a$ come anche il $b$. Questa alternazione può essere usata anche in combinazione con altri path pattern più complessi, come ad esempio $(a \to b)|(d \to c)$ che identifica sia l'arco $(a,b)$ che l'arco $(d,c)$. Può inoltre essere utilizzato in quest'altro modo $a \to (d|b)$ per indicare l'arco $(a,d)$ come anche l'arco $(a,b)$.

**Esempio 3.5.4.** Guardando la Figura 3.5.1, possiamo avere il path pattern $\% \to \%$ che identifica un generico arco, ma non abbiamo modo di limitare questo ad sottoinsieme degli archi di $G$ senza esprimere delle condizioni sui singoli nodi. Con $a \to *$ identifichiamo tutti i path che partono dal nodo $a$, ma anche potremmo voler esprimere delle condizioni su questi path, come la lunghezza o il valore degli archi coinvolti.

Vedremo comunque in seguito che l'operatore di selezione ci darà la capacità di esprimere delle condizioni su questi path generici, come la lunghezza massima di un path, o il valore che deve avere un generico nodo, ed altro ancora.

#### 3.5.1.2 Model Checking

I due modi per definire i path che soddisfano un determinato path sono entrambe semplici e non differiscono molto tra loro. La relazione che intercorre tra i due è molto semplice, e può essere riassunta con la seguente relazione.

**Proposizione 3.5.1.** Sia $G = (N, A, \nu, d)$ un Network Level, e sia $P$ un path pattern di $G$. Vale che:

$$\forall\, p \text{ path di } G \; p \vDash P \iff p \in M(P)$$

La dimostrazione di tale proposizione può essere fatta per induzione, per ispezione delle definizioni 3.5.1 e 3.5.3. Tale dimostrazione viene comunque omessa, poiché abbastanza semplice.

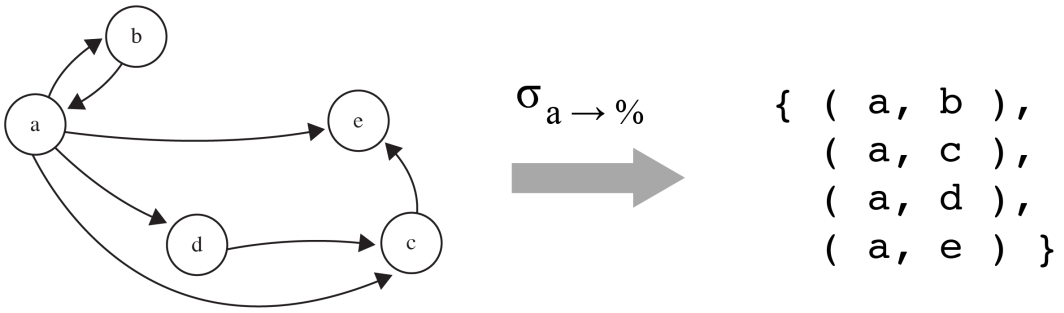



**Figura 3.5.2:** Sulla sinistra il network level $G_g = (N_g, A_g, v_g, d_g)$ della Figura 3.5.1 che rappresenta una piccola rete sociale di Twitter: i nodi rappresentano i singoli utenti, mentre gli archi rappresentano le relazioni unidirezionali di *Following*. Con l'operatore di selezione vorremmo estrapolare tutti i path conformi ad un determinato path pattern. In questo caso tutti i path di lunghezza 1, che cominciano dal nodo $a$ e che raggiungono un altro nodo.

Questa proposizione ci suggerisce un primo metodo per la verifica che un determinato path soddisfi un determinato path pattern. Questo metodo ricorda il model checking che viene fatto, ad esempio, nella logica proposizionale, dove per vedere che $A \vDash B$ si verifica che $M(A) \subseteq M(B)$, dove in questo caso $M(D)$ esprime l'insieme di tutti i mondi per cui la formula $F$ sia vera. Da questa piccola intuizione si può anche capire il perché sono stati utilizzati determinati simboli nelle le definizioni dei path pattern.

Comunque tale verifica può essere fatta ricorrendo alla costruzione di automi, proprio come viene fatto per le espressioni regolari, poiché è computazionalmente oneroso dover calcolare l'insieme di tutti i path che soddisfano un determinato pattern, solo per effettuare la verifica su un singolo path.

### 3.5.2 DEFINIZIONE

In questa sezione definiremo l'operatore di selezione, vedremo come utilizzerà i path pattern, ma anche come compenserà i loro limiti che abbiamo riscontrato preceden-

temente, utilizzando predicati che pongono delle condizioni sugli elementi generici dei path pattern.

**Definizione 3.5.4.** Sia $G = (N, A, \nu, d)$ un Network Level e sia $P_G$ un path pattern di $G$. Si definisce l'operatore di selezione $\sigma_{P_G,F(p)}(G)$, con $F(p)$ un predicato che determina se il path $p$, che soddisfa il path pattern $P_G$, fa parte del risultato della selezione. Pertanto:

$$\sigma_{P_G,F(p)}(G) = \{p : p \vDash P_G \wedge F(p)\}$$

Vediamo come questa definizione sia abbastanza semplice e concisa, poiché utilizza le definizioni sui path pattern. Per superare però i limiti, sul potere espressivo, che avrebbe avuto un semplice operatore di selezione che considerasse solamente i path pattern, viene utilizzato il predicato F(p). Questo predicato non fa altro che prendere un path $p$, che sappiamo essere conforme ad un determinato path pattern $P_G$ (i.e. $p \vDash P_G$), e determina se tale path ci interessa o meno.

**Esempio 3.5.5.** Consideriamo nuovamente la Figura 3.5.1, dove abbiamo il Network Level $G_g = (N_g, A_g, \nu_g, d_g)$. Osserviamo come il path pattern $a \rightarrow \%$ identifichi tutti gli archi uscenti dal nodo $a$, in particolare, utilizzando l'operatore di selezione abbiamo che

$$\sigma_{a\rightarrow\%}(G) = \{(a,b),(a,c),(a,d),(a,e)\}$$

Possiamo a questo punto avere l'insieme dei vicini utilizzando l'operatore di proiezione che abbiamo visto precedentemente:

$$\pi_2(\sigma_{a\rightarrow\%}(G)) = \{b,c,d,e\}$$

Da notare come in quest'ultimo esempio non abbiamo utilizzato il predicato $F(p)$, lasciando vuota la sua posizione, ed intendendo implicito un predicato che rispondesse sempre con un valore positivo. Quest'ultima convenzione verrà utilizzata in futuro per alleggerire la scrittura dei nostri esempi.

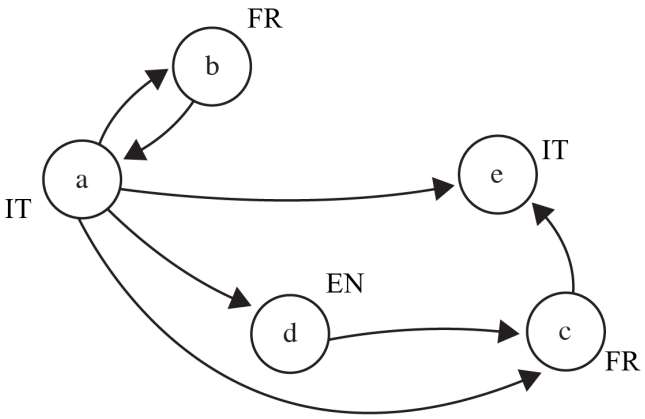



**Figura 3.5.3:** Il network level $G_g = (N_g, A_g, \nu_g, d_g)$ della Figura 3.5.1 che rappresenta una piccola rete sociale di Twitter: i nodi rappresentano i singoli utenti, mentre gli archi rappresentano le relazioni unidirezionali di *Following*. In questa figura però è stata popolata la funzione $d_g$, associando uno dato di nazionalità ad ogni nodo. Per ognuno di essi viene quindi rappresentato con il suo identificatore (e.g. $a$ e $b$) ed anche il suo dato (e.g. IT ed EN).

Cerchiamo adesso di fare degli esempi più elaborati, ponendo delle condizioni: supponiamo, come in Figura 3.5.3, che ad ogni nodo venga associato un dato. Vorremmo far dipendere la nostra selezione, e quindi la scelta dei vicini che abbiamo fatto nell'esempio 3.5.5, non solo dalla struttura (i.e. tutti i nodi vicini), ma anche dal contenuto di tali nodi.

**Esempio 3.5.6.** Guardando la Figura 3.5.3, vogliamo adesso prendere i vicini $a$, ma che non appartengono al suo stesso stato. Per fare questo, ancora una volta useremo l'operatore di selezione, in combinazione con quello di proiezione, ma verificheremo anche il loro contenuto popolando il predicato $F(p)$ dell'operatore di selezione.

$$\pi_2(\sigma_{a\to\%\,,\,\pi_1(p)\neq\pi_2(p)}(G)) = \{b, c, d\}$$

Come avevamo già osservato nell'operatore di proiezione, per il predicato $F(p)$, abbiamo scritto direttamente il corpo di tale predicato, lasciando intendere che questo accetti in input un path $p$. Ad esempio se volessimo selezionare tutti i path che han-

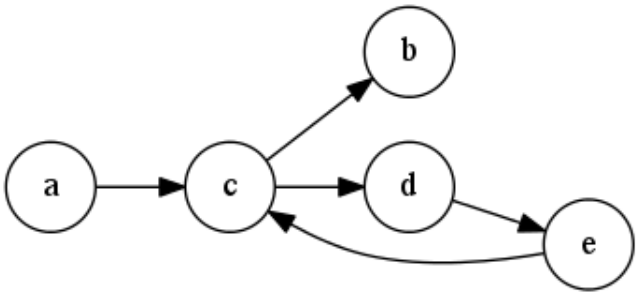



**Figura 3.5.4:** Un network level $G_g = (N_g, A_g, \nu_g, d_g)$ che rappresenta una rete di aereoporti, ogni arco rappresenta la possibilità di andare da un luogo ad un altro.

no una lunghezza minore di 2, con $\sigma_{*,len(p)<2}(P)$, intendiamo $\sigma_{*,F}(P)$ dove $F$ è il predicato $F(p) = len(p) < 2$.

### 3.5.3 DECIDIBILITÀ

Dopo aver introdotto l'operatore di selezione, sarà adesso più semplice riuscire a capire il perché abbiamo utilizzato i simple path. Per farlo ci concentriamo su un esempio particolare, guardando la Figura 3.5.4: supponiamo di voler trovare tutti i path che partono dal nodo $a$ e che terminano al nodo $b$.

$$\sigma_{a \to * \to b}(G)$$

In questo caso esiste solamente un simple path che parte dal nodo $a$ e che arriva al nodo $b$.

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

Così non è per i path che ammettono ripetizioni, poiché il ciclo nel grafo fa sì che questi diventino infiniti.

$$\{\,(a,c,b),(a,c,d,e,c,b),(a,c,d,e,c,d,e,c),(a,c,d,e,c,d,e,c,d,e,c),\ldots\,\}$$

Si è pertanto deciso di limitare la selezione in questo modo poiché, un algoritmo che implementa l'operatore di selezione, *deve* poter generare il proprio output. Inoltre, per la quasi totalità delle analisi, non serve considerare cammini che passano più volte per gli stessi sotto-cammini poiché ciò creerebbe informazioni ridondanti.

## 3.6 Operatore di Sintesi

Nell'esempio 3.5.5, quando trovavamo l'insieme dei vicini di un determinato nodo, abbiamo visto in maniera pratica l'utilità di avere degli operatori di selezione che diano come risultato insiemi di path, piuttosto che sotto-grafi. In questo modo riusciamo ad avere più informazioni, utilizzando ad esempio l'operatore di proiezione che lavora su insiemi di path.

### 3.6.1 Motivazioni

Se da una parte abbiamo più informazioni, dall'altra abbiamo più dati. Se prendiamo come esempio il risultato di una selezione, abbiamo anche molti elementi ridondanti. Per fare un esempio, il risultato di una selezione che restituisce l'insieme dei path $P = \{(a,b,c),(a,b,e)\}$, altro non è che il grafo composto da quei nodi e connessi rispecchiando gli archi descritti dalla struttura stessa dei path. Notiamo però che, in $P$, i nodi $a$ e $b$ vengono ripetuti per due volte. Pertanto, a discapito di una perdita di informazioni che comunque possono essere successivamente ricalcolate, potrebbe essere utile eliminare queste ridondanze, e formare dei sotto-grafi a partire da insiemi di path. Ritornando nell'esempio che abbiamo appena fatto vorremmo trasformare l'insieme $P$ nel grafo rappresentato in Figura 3.6.1.

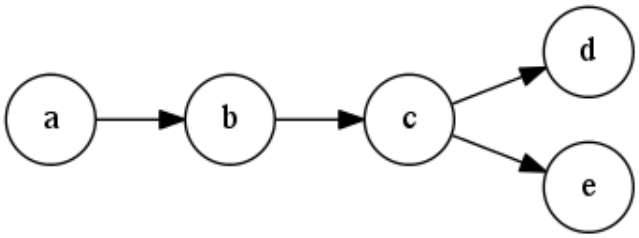



**Figura 3.6.1:** Sintesi dell'insieme di path $\{(a,b,c),(a,b,e)\}$.

Il risultato di un'operazione di selezione può essere quindi vista come un risultato intermedio, nel caso in cui ci interessi direttamente un determinato sotto-grafo. Inoltre, trasformare un insieme di path in un grafo può essere utile nel caso volessimo utilizzare tale insieme nuovamente all'interno di un operatore di selezione. Ciò è molto importante, soprattutto per riuscire ad avere un set di operatori che sia chiuso.

Definiamo pertanto l'operatore di sintesi $\eta$ nel seguente modo.

**Definizione 3.6.1.** Sia $G = (N, A, v, d)$ un Network Level e sia $P = \{p_1, p_2, ..., p_k\}$ un insieme di path di $G$. Si definisce l'operatore di sintesi $\eta_G(P)$ nel seguente modo:

$$\eta_G(P) = (N|_P,\ A|_P,\ v|_{(A|_P)},\ d|_{(N|_P)})$$

Dove $N|_P = \{n\ :\ \exists i\ .\ n \in p_i\}$ è l'insieme di tutti i nodi che sono presenti nei path di $P$; e dove $A|_P$ è l'insieme di tutti gli archi che sono presenti nei path di $P$. Inoltre, le funzioni $v|_{(A|_P)}$ e $d|_{(N|_P)}$ sono le funzioni ristrette ai nuovi insiemi $A|_P$ e $N|_P$, rispettivamente.

### 3.6.2 Esempio

Per riuscire a capire bene questo operatore, soprattutto per quanto riguarda la restrizione delle funzioni $v$ e $d$, riprendiamo l'esempio 3.5.6, dove avevamo trovato tutti i vicini di un determinato nodo $a$ che avevano però un dato differente.

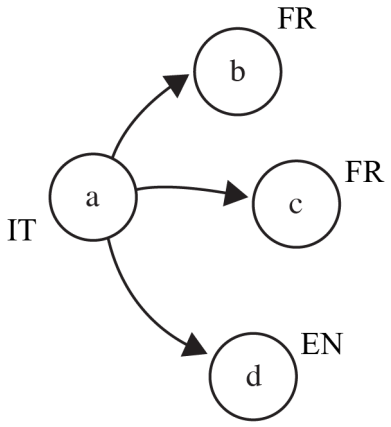



**Figura 3.6.2:** Il risultato dell'esempio 3.6.1, che prende il network level $G_g = (N_g, A_g, v_g, d_g)$ della Figura 3.5.3 e ricava il sotto-grafo che comprende il nodo $a$ ed i suoi vicini che sono però in uno stato differente.

**Esempio 3.6.1.** Guardando la Figura 3.5.3, vogliamo il sotto-grafo di $G_g$, composto da $a$ ed i suoi vicini che però non appartengono al suo stesso stato. Per fare questo, useremo l'operatore di selezione, in combinazione con quello di sintesi.

$$\eta_{G_g}(\sigma_{a\to\%,\,\pi_1(p)\neq\pi_2(p)}(G)) = \{(a,b),(a,c),(a,d)\}$$

Il Network Level risultante $G' = (N', A', v', d')$, rappresentato graficamente in Figura 3.6.2, dove $N' = \{a,b,c,d\}$, $A' = \{(a,b),(a,c),(a,d)\}$, e le funzioni $v_g = \{(a,\text{IT}),(b,\text{FR}),(c,\text{FR}),(d,\text{EN}),(e,\text{IT})\}$ e $d_g = \emptyset$ vengono ristrette considerando i due nuovi insiemi: $v' = \{(a,\text{IT}),(b,\text{FR}),(c,\text{FR}),(d,\text{EN})\}$ e $d' = d_g$.

## 3.7 Operatore di Aggregazione

### 3.7.1 Motivazioni

Come per l'algebra relazionale, abbiamo bisogno di un operatore che raggruppi i nostri dati, fornendo un unico dato di sintesi, a partire da molti altri. Questo è utile in

tutti quei casi in cui vogliamo ad esempio raccogliere delle statistiche, a partire da un insieme di dati; oppure in quei casi dove vogliamo unire dati identici unificando in qualche modo quelli differenti (e.g. media). Per riuscire a capire meglio il nostro contesto, partiremo da un esempio che per il lettore sarà più naturale, riferendoci ancora una volta all'algebra relazionale.

| Nome | Età | Peso |
|---|---|---|
| Mario | 45 | 68 |
| Erica | 53 | 60 |
| Marco | 53 | 70 |
| Giorgio | 25 | 72 |

| Età | Peso Medio |
|---|---|
| 45 | 68 |
| 53 | 65 |
| 25 | 72 |

**Tabella 3.7.1:** Un esempio grafico del raggruppamento nell'algebra relazionale. A sinistra abbiamo la relazione *R* che contiene gli attributi *Nome*, *Età* e *Peso*; a destra i pesi medi raggruppando per età.

Prendiamo la tabella 3.7.1, dove abbiamo una relazione *R* che contiene tutti i dati dei pazienti di un determinato ospedale, nel nostro esempio, per semplicità gli unici dati forniti sono quelli relativi al nome, età e peso. Adesso, per fare un esempio, vorremmo avere una panoramica dei pesi considerando le età. L'operatore di aggregazione, comunemente utilizzato nei DBMS basati sull'algebra relazionale, è proprio quello che viene utilizzato in questi casi. Nel nostro esempio basta raggruppare per età, e creare un nuovo attributo (i.e. Peso medio) riassuntivo, che ha come valori la media tra i pesi che hanno ugual età.

Nel nostro caso, parlando di grafi, queste operazioni diventano sicuramente più complesse. Il fatto stesso di creare un nuovo attributo, non è più un'operazione semplice: è necessario specificare dove deve essere inserito, se i valori siano inseriti in un arco o in un nodo. Per ovviare a queste difficoltà, ancora una volta, i path ci vengono in aiuto, poiché semplificano la struttura del grafo, fornendone una visione molto più simile a quella di una relazione dell'algebra relazionale.

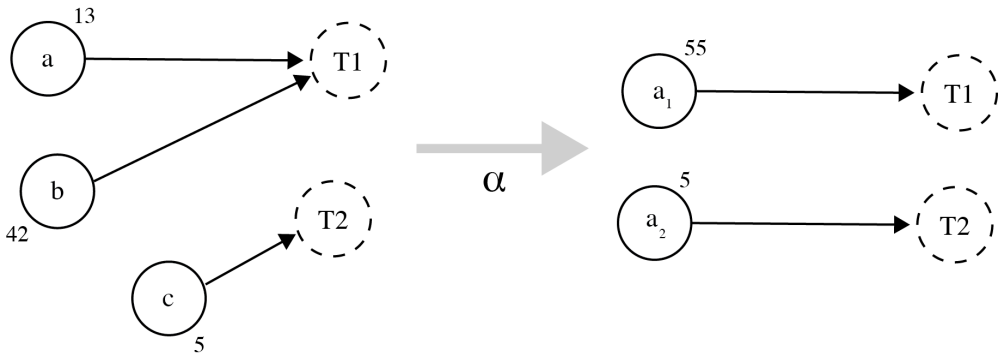



**Figura 3.7.1:** A sinistra un network level $G = (N, A, v, d)$ che rappresenta una piccola rete sociale. I nodi con il bordo continuo indicano gli utenti di tale rete, quelli con il bordo tratteggiato invece dei topic. Gli archi sono relazioni che indicano il fatto che un utente abbia menzionato almeno una volta un determinato topic nel corso del tempo. Ogni nodo ha associato come dato un numero, che indica la sua importanza all'interno del grafo: $d(a) = 13$, $d(b) = 42$ e $d(c) = 5$. A destra invece il risultato che vorremmo ottenere utilizzando l'operatore di aggregazione.

**Esempio 3.7.1.** Per cominciare, facciamo subito un piccolo esempio. Supponiamo di avere un Network Level come in Figura 3.7.1 che rappresenta un social network dove: da un lato abbiamo gli utenti, ognuno dei quali ha un certo peso (e.g. un peso che esprime la sua importanza nella rete); dall'altra invece una serie di topic che possono essere menzionati dagli utenti. Quello che vorremmo fare è cercare di estrapolare l'importanza di questi topic, ottenendo un semplice grafo dove ogni topic $T$ viene puntato da un nodo, e che quest'ultimo ha come dato la somma dei dati di tutti gli utenti che puntavano a $T$.

Per risolvere il problema presentato nel precedente esempio, cerchiamo per prima cosa di estrapolare i dati con l'operatore di selezione, selezionando tutte le coppie di nodi dove il primo punta al secondo con un arco, e dove il secondo è un topic. Per farlo, supponiamo di poter individuare la tipologia dei nodi guardando i loro dati (e.g. se $t$ è un topic allora $d(t).type = Topic$).

$$\sigma_{\%\to\%,\, d(\pi_1(p)).type=People \wedge d(\pi_2(p)).type=Topic}(G_g)$$

Il risultato di tale operazione è il seguente:

$$\{(a, T1), (b, T1), (c, T2)\}$$

Come vediamo, se consideriamo ogni path dell'insieme che è stato ritornato come fosse una tupla, l'operatore di selezione non è assai differente da quello dell'algebra relazionale. Anche in questo caso dobbiamo avere la possibilità di creare nuovi elementi come nuovi nodi, di poter creare nuovi archi, o anche di poter assegnare a questi nuovi valori.

### 3.7.2 DEFINIZIONE

Definiamo adesso l'operatore di aggregazione che, dopo aver parlato dell'esempio precedente, sarà più chiaro riuscirne a comprendere il significato.

**Definizione 3.7.1.** Sia $G = (N, A, v, d)$ un Network Level e sia $P = \{p_1, p_2, ..., p_k\}$ un insieme di path di $G$. Si definisce l'operatore di aggregazione $\alpha_{G;c;a}(P)$ nel seguente modo:

$$\alpha_{G;c;a}(P) = (N_H,\ A_H,\ v_H,\ d_H)$$

Dove $G$ serve all'operatore come riferimento al Network Level, in modo che possa recuperare i suoi dati; dove $c$ è una funzione che prende in input un path e restituisce un path; e dove $a$ è un insieme di assegnamenti. L'insieme $H$ è un insieme di path, generato a partire da $c$; che definisce come raggruppare, e come creare i nuovi nodi ed archi.

$$H = \{h : \exists p_i \in P \,.\, c(p) = h\}$$

Questo insieme può essere visto come un risultato intermedio necessario per costruire l'output, e che serve per poi poter generare il Network Level che costituisce il risultato finale. L'insieme di assegnamenti in $a$ invece popolerà le funzioni $v_H$ e $d_H$ onde stabilire quali valori assegnare a queste nuove informazioni.

L'insieme $N_H$ è l'insieme di tutti i nodi presenti in $H$, ovvero $N_H = \{n : \exists h \in H \,.\, n \in h\}$. L'insieme $A_H$ è l'insieme di tutti gli archi presenti in $H$, ovvero $A_H = \{(n_1, n_2) : \exists h \in H \,.\, (n_1, n_2) \in h\}$. Mentre, le funzioni $\nu_H$ e $d_H$ sono rispettivamente le funzioni $\nu$ e $d$, ristrette ai nuovi insiemi $A_H$ e $N_H$, che però hanno anche nuovi elementi, scelti a partire dagli assegnamenti di $a$.

### 3.7.3 DETTAGLI

Ma cerchiamo adesso di vedere i vari dettagli di questa definizione, in modo da capire meglio questo operatore che si riconosce essere abbastanza complesso.

Come già detto nella definizione, la funzione $c$ descrive come creare l'output e come raggruppare gli elementi. Essa, prendendo in input un path $p$ specificherà come crearne uno nuovo. Tutte queste faranno parte dell'insieme $H$, raggruppando gli elementi. Per la descrizione dell'output, la funzione $c$ può utilizzare alcuni nodi già presenti nell'insieme di input, ma può anche descriverne di nuovi.

UTILIZZARE GLI ELEMENTI GIÀ PRESENTI La funzione $c$ può prendere gli elementi già presenti all'interno dei path in esame utilizzando, ad esempio, l'operatore di proiezione per prendere i singoli nodi (o anche sotto-path) ed utilizzare la concatenazione per creare nuovi path. Questi potrebbero però non essere più path di $G$, ma saranno path del nuovo Network Level che costituisce il risultato.

Per fare un esempio, la funzione $c$ può essere definita per costruire dei path come formati solamente dal primo e dall'ultimo nodo di ogni path, specificando $c(p) = \pi_1(p) \,.\, \pi_{len(p)+1}(p)$.

Gli elementi già presenti che vengono menzionati nel corpo della funzione $c$ costituiscono gli elementi su cui viene fatto il raggruppamento. Ciò vuol dire che, definendo con $V(h)$ l'insieme dei nodi già presenti, ovvero $V(h) = \{n \;:\; n \in h \wedge n \in N\}$, vale che:

$$\nexists h_1, h_2 \text{ t.c. } h_1 \in H \wedge h_2 \in H \wedge V(h_1) = V(h_2)$$

Per chiarire questo concetto, che comunque è abbastanza naturale se si pensa ai classici DBMS basati sull'algebra relazionale, riprendiamo l'esempio precedente. In quel caso, essendo $c$ la funzione che descrive un output formato dal primo e dall'ultimo nodo dei path di input, allora il risultato intermedio costituito dall'insieme $H$ è fatto in modo da non avere due path che abbiano stesso nodo iniziale e stesso nodo finale, ovvero: i path di $H$ sono stati raggruppati per i nodi di inizio e di fine.

CREAZIONE DI NUOVI ELEMENTI Il corpo della funzione può esplicitare nuovi nodi, in modo che questi contengano il risultato di operazioni aggregate quali somme, medie, minimo, massimo e conteggio.

Per la creazione di questi nuovi nodi ci riferiremo, similmente a come abbiamo fatto con i path pattern, al simbolo % che specifica un generico nodo che deve essere creato. Questi nuovi nodi vengono creati in modo che $\forall h_1 \in H, h_2 \in H$ valga che:

$$V(h_1) = V(h_2) \implies h_1 = h_2$$

Ovvero i nuovi nodi non fanno in modo di creare due path che differiscano solamente per i nuovi nodi, poiché altrimenti si sarebbe violato il principio stesso del raggruppamento. Considerando anche la definizione di insieme che non ammette elementi duplicati, questo vincolo è in realtà già presente nel vincolo dato precedentemente, ma è comunque stato scritto per chiarificare meglio i concetti presentati.

Facendo un esempio, specificando $c(p) = \pi_1(p) \,.\, \% \,.\, \pi_{len(p)+1}(p)$, per prima cosa sappiamo che vengono raggruppati i nodi di inizio e di fine di ogni path; supponendo poi che l'insieme di input sia $\{(a,c,d),(a,e,f,d)\}$, l'insieme $H$ avrà un solo path $(a,b,d)$, dove $b$ è un nuovo nodo.

ASSEGNAMENTI I dati di questi nuovi nodi devono però opportunamente essere specificati, popolando la funzione $d_H$ e $v_H$. Questo viene fatto dall'insieme di assegnamenti in $a$.

Un assegnamento in $a$ è una coppia composta da due parti: $LV$ e $RV$; che scriveremo nella forma $LV = RV$. La prima è l'elemento che deve essere modificato, ovvero l'indice (o la coppia di indici) relativo (o relativa) al nodo (o all'arco) di cui vogliamo modificare il valore. La seconda invece è il valore che vogliamo assegnare, ad esempio utilizzando una funzione di aggregazione come media o somma. Importante è notare, per non creare confusioni, che gli indici si riferiscono all'output e non ai path di input. Se ad esempio abbiamo specificato nella funzione $c$ dei path lunghi 1, possiamo fare degli assegnamenti solamente al primo nodo, al secondo nodo e all'arco tra i due.

Questa seconda parte è una funzione che accetta un parametro $p$ che indica il path di input esaminato. Utilizzando quest'ultimo possiamo raccogliere i valori che vogliamo assegnare ai nodi (o agli archi), e che costituiranno l'output. Questa funzione di aggregazione prende in esame tutti i path $p$ che verranno in seguito raggruppati e rappresentati da un unico path in $H$, ovvero $\forall h \in H$ la funzione esaminerà ogni path dell'insieme $\{p : c(p) = h\}$. Parlando da un punto di vista più implementativo, la funzione di aggregazione prende un path alla volta, e non l'insieme appena discusso; è quindi importante definirle in modo che costruiscano il risultato per passi. Ciò è naturale se si pensa ad una sommatoria, ma non è altrettanto naturale nel caso si voglia fare una media, dove sarebbe necessario invece costruire il risultato a partire da una sommatoria dei valori, ed un conteggio di tutti gli elementi analizzati.

Per fare un esempio, se $c(p) = \pi_1(p) \,.\, \% \,.\, \pi_{len(p)+1}(p)$, con $2 = average(len(p))$ assegniamo al dato del secondo nodo di ogni path di output la lunghezza media dei path di input.

### 3.7.4 Esempi

Per riuscire bene a capire questo operatore adesso faremo alcuni esempi. Riprendiamo per prima cosa l'esempio 3.7.1.

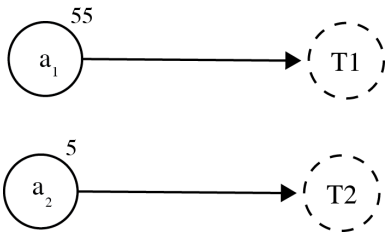



**Figura 3.7.2:** Il risultato dell'operazione di aggregazione dell'esempio 3.7.1. Tale Network Level $G_2$ è costituito dalla tupla $(N_2, A_2, \nu_2, d_2)$, dove ogni elemento è stato costruito come descritto nell'esempio stesso.

Dobbiamo raggruppare la seconda componente di ogni path, ovvero quella che identifica i topic. In seguito vogliamo creare un nuovo nodo per ogni topic, in modo che punti a quest'ultimo. Possiamo quindi definire $c = \{\% \,.\, \pi_2(p)\}$. Nell'esempio volevamo poi riuscire ad inserire come dato la somma dei dati degli utenti che puntavano allo stesso topic. Possiamo quindi definire $a = \{1 = Sum(\pi_1(p))\}$. Riassumendo, chiamando $P$ il risultato dell'operazione di selezione fatta nell'esempio 3.7.1:

$$\alpha_{\%\,.\,\pi_2(p)\,;\,1=Sum(\pi_1(p))}(P)$$

Quest'operazione avrà come risultato intermedio l'insieme $H = \{(a_1, T1), (a_2, T2)\}$, dove $a_1$ e $a_2$ sono nuovi nodi. Invece, per riuscire anche a vedere quali siano i dati di tutti questi nodi dobbiamo osservare il risultato per intero. Questo sarà il Network Level $G_2 = (N_2, A_2, \nu_2, d_2)$, con $N_2 = \{a_1, a_2, T1, T2\}$, $A_2 = \{(a_1, T_1), (a_2, T_2)\}$, $\nu_2 = \nu$ ed infine $d_2 = \{(a_1, 55), (a_2, 5), (T_1, d(T_1)), (T_2, d(T_2))\}$; dove abbiamo utilizzato anche le funzioni del grafo di input $G = (N, A, \nu, d)$ poiché alcuni dati (e.g. i nodi che identificano i topic) sono rimasti invariati. Possiamo quindi mostrare, in Figura 3.7.2, il risultato finale.

Adesso vogliamo anche raggruppare le informazioni su interi path. Quello che abbiamo è una serie di nodi $\{a, b, c, d\}$, e ci interessa sapere quanti archi bisogna attraversare per raggiungere dai nodi $a$ e $c$ i nodi $b$ e $d$. Vorremmo quindi raggruppare le informazioni dei path $(a, ..., b)$, $(a, ..., d)$, $(c, ..., d)$, ed infine $(c, ..., b)$. Per farlo voglia-

mo creare degli archi che colleghino direttamente gli estremi di tali path, assegnando ad ogni arco un valore che corrisponde alla lunghezza del path minimo.

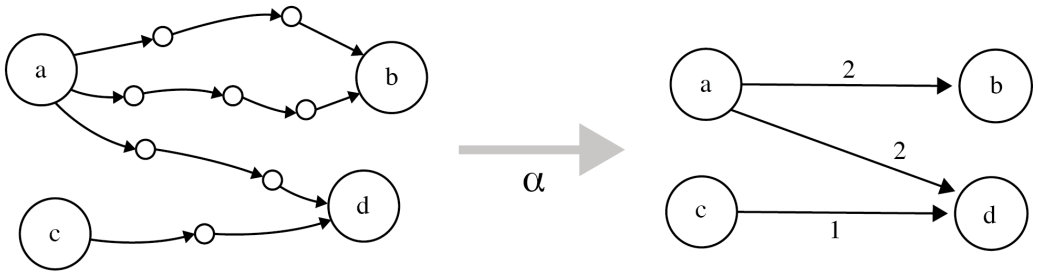



**Figura 3.7.3**

Per fare ciò prima di tutto assumiamo di ottenere con l'operatore di selezione i path che ci interessano:

$$P = \{(a, n_1, n_2, b), (a, n_3, n_4, n_5, b), (a, n_6, n_7, d), (c, n_8, d)\}$$

Utilizziamo quindi l'operatore di aggregazione per ottenere il risultato voluto in questo modo:

$$\alpha_{\pi_1(p)\,.\,\pi_{len(p)+1}(p)\,;\,(1,2)=Min(len(p))}(P)$$

## 3.8 Operatore di Join

### 3.8.1 Motivazioni

Come abbiamo visto dalle definizioni che abbiamo dato fin'ora, gli operatori operano solamente su uno specifico Network Level, ovvero compiono delle operazioni che potrebbero essere fatte anche su un classico grafo, a meno di ridefinizioni neanche troppo drastiche. Ciò che vogliamo fare adesso è invece definire un'operazione che

ne coinvolga più di uno, proponendo un'operazione binaria che sia in grado di fondere due Network Level, cercando di sfruttare le definizioni di accoppiamento che abbiamo dato nel capitolo precedente, e che definivano le relazioni che intercorrevano tra di essi.

Ancora una volta cerchiamo prima di tutto di cominciare con un esempio, in modo che sia chiaro fin da subito l'obiettivo e che quindi siano più chiare le definizioni che saranno riportate in seguito. Vogliamo fondere due Network Level differenti, sfruttando gli accoppiamenti che sono stati definiti tra di essi. Considerando solamente la fusione di due livelli, abbiamo una nozione abbastanza generale per considerare anche la fusione di $n$ livelli differenti.

**Esempio 3.8.1.** Supponiamo di avere due Network Level $A = (N_A, A_A, v_A, d_A)$ e $B = (N_B, A_B, v_B, d_B)$. Il primo ha solamente due nodi (i.e. $\{a, b\}$) mentre il secondo ne ha un altro in più (i.e. $\{a_2, b_2, c_2\}$). Gli accoppiamenti tra di essi sono definiti in modo da rispecchiare gli identificativi che gli abbiamo attribuito, ovvero $a$ è associato con $a_2$ e $b$ è associato con $b_2$. La Figura 3.8.1 che c'è alla sinistra, raffigura l'esempio che abbiamo appena descritto, mentre alla destra troviamo il risultato che vogliamo ottenere, ovvero la fusione di questi due Network Level in uno nuovo, rispecchiando gli accoppiamenti.

OSSERVAZIONI Come vediamo nella Figura 3.8.1, gli identificativi che abbiamo attribuito nella figura di destra sono puramente identificativi, in modo che sia chiaro quali siano stati i nodi che li hanno generati. Ciò significa che i nomi di tali nodi possono essere totalmente differenti.

Diventa invece differente la questione che riguarda i dati di tali nodi. Nel nodo che abbiamo identificato con $[a, a_2]$, bisogna decidere quale sia il dato finale. Quest'ultimo deve essere il risultato dei dati $d_A(a)$ e $d_B(a_2)$ fusi secondo una qualche funzione $f_d$. Allo stesso modo, se osserviamo l'arco $([a, a_2], [b, b_2])$, dobbiamo decidere come

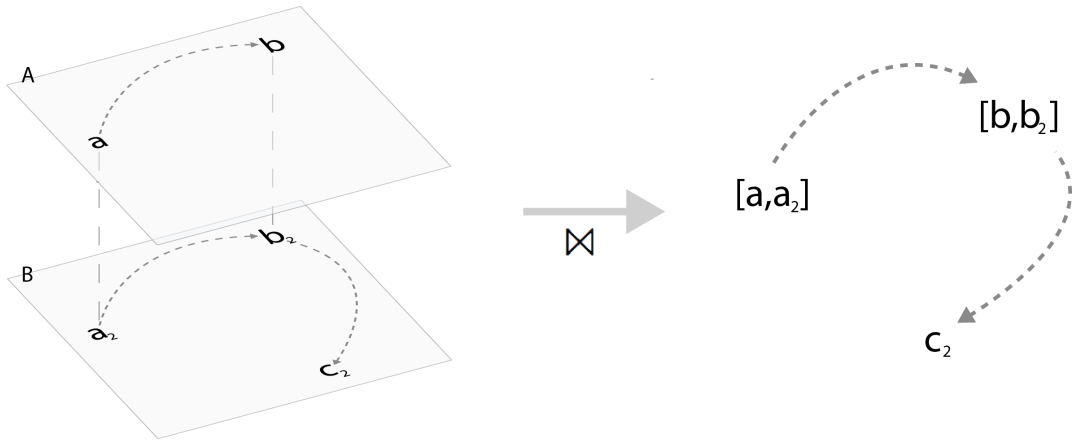



**Figura 3.8.1**

fondere i valori degli archi, analogamente a come abbiamo discusso per i nodi, utilizzando una qualche funzione $f_v$ che prende in considerazione i valori $v_A((a,b))$ e $v_B((a_2,b_2))$. Osservando invece il nodo $c_2$ non abbiamo questi problemi, e l'arco che congiunge $[b,b_2]$ con $c_2$ è unico; i dati del nodo, come anche il valore associato a tale arco, possono essere direttamente ricopiati così come sono.

POSSIBILI INTERPRETAZIONI Ma come possiamo definire $f_v$ ed $f_d$? Dipende sempre da cosa vogliamo analizzare. Se abbiamo tanti Network Level dove ognuno rappresenta un istante temporale di un social network, e dove gli archi tra i nodi rappresentano un messaggio inviato tra una persona ed un'altra, possiamo fondere tutti questi livelli per cercare di capire se una persona ha mai comunicato con un'altra. In questo modo ogni accoppiamento coinvolge nodi che sono in realtà la stessa persona, ma in momenti differenti; la funzione $f_d$ potrebbe essere quindi quella che, presi i dati di due nodi, fornisca come risultato sempre quello più recente, in modo che il risultato finale abbia, per ogni nodo, i loro dati più recenti. Allo stesso modo potremmo definire la funzione $f_v$. Un esempio potrebbe essere quello di diversi Network Level (come in Figura 3.8.2), dove il più basso rappresenta una rete di calcolatori, e dove gli archi rappresentano la latenza che intercorre tra due nodi differenti; gli altri livelli in-

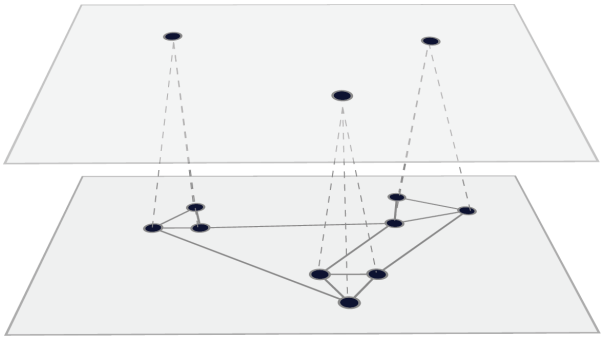


**Figura 3.8.2:** Esempio di un Multi Level Network, dove abbiamo il livello sottostante che rappresenta una rete di calcolatori, e quello soprastante che rappresenta la medesima rete ad un livello di granularità differente.

vece rappresentano la medesima rete, ma a livelli di granularità differenti. In questo caso potremmo voler fondere due livelli in modo che il livello soprastante riassuma le latenze di quello sottostante. Notando che un livello sovrastante ha associati molti nodi del livello sottostante, possiamo accorgerci subito che sono molti gli archi che coinvolgono gli stessi nodi. Ma volendo riassumere le latenze, possiamo definire $f_v$ come quella funzione che esegue sempre la media, oppure prende sempre il minimo tra i valori associati agli archi.

### 3.8.2 DEFINIZIONE

Definiamo adesso l'operatore di join $\bowtie$, definendolo seguendo le idee discusse precedentemente. In un secondo momento riprenderemo l'esempio 3.8.1, e parleremo di tutte le osservazioni e dei casi particolari.

**Definizione 3.8.1.** Siano $G_1 = (N_1, A_1, v_1, d_1)$ e $G_2 = (N_2, A_2, v_2, d_2)$ due Network Level. Sia anche $(G_1, G_2, M, w)$ un accoppiamento (i.e. Couple) tra $G_1$ e $G_2$. Si definisce l'operazione di join $G_1 \bowtie_{M, f_d, f_v} G_2$ come quella che genera un nuovo Network Level $G = (N, A, v, d)$ come descritto in seguito.

Definiamo $\forall n \in N_1$, l'insieme $C_M(n) = (n_1, n_2, ..., n_k)$ come l'insieme dei nodi di $G_2$ che sono associati ad $n$, secondo il mapping $M$ tra i nodi dei due livelli (notare quindi che $\forall v \in C_M(n) \implies v \in N_2$). Identifichiamo con $x_{C_M(n)}$ il generico nodo di $G$, che rappresenterà la fusione del nodo $n$ con tutti quelli appartenenti a $C_M(n)$, in altre parole $\forall n \in N_1$ abbiamo $x_{C_M(n)} \in N$.

Per quanto riguarda i dati di tali nodi invece, vengono definiti dalla funzione $f_d$ : $Dom_{N_1} \times Dom_{N_2} \rightarrow Dom_N$ nel seguente modo:

$$d(x_{C_M(n)}) = f_d(d_1(n),\ f_d(d_2(n_1),\ f_d(d_2(n_2),\ ...\ d_2(n_k)...))$$

Ovvero, la funzione $f_d$ fonde il dato di $n$ (che risiede in $G_1$, e per questo utilizziamo $d_1$) con i dati dei suoi nodi associati (ovvero $\{n_1, n_2, ..., n_k\}$ che risiedono in $G_2$, e per questo per loro utilizziamo $d_2$).

Per gli archi invece, presi due nodi di $G_1$ che chiamiamo $n$ e $n'$ (dove quindi $n \in N_1$ e $n' \in N_1$), considerando i corrispettivi insiemi di nodi associati $C_N(n) = \{n_1, n_2, ..., n_k\}$ e $C_N(n') = \{n'_1, n'_2, ..., n'_{k'}\}$, in $G$ viene inserito l'arco $(x_{C_M(n)}, x_{C_M(n')})$ se vale:

$$(n, n') \in A_1 \ \vee\ (\exists i, j\ .\ (n_i, n'_j) \in A_2)$$

Per quanto riguarda i valori associati a tali archi, definiamo prima di tutto l'insieme $A_E = \{(v_1, v_2)\ :\ v_1 \in C_M(n) \wedge v_2 \in C_M(n') \wedge (v_1, v_2) \in A_2\}$, che è l'insieme di tutti gli archi che, una volta fusi i nodi dei due livelli, devono essere a sua volta fusi in un unico arco. Nominiamo i suoi elementi per facilitare la lettura:

$$A_E = \{e_1, e_2, ..., e_z\}$$

Allora i dati associati agli archi del Network Level $G$ sono definiti dalla funzione $f_v$ : $Dom_{A_1} \times Dom_{A_2} \rightarrow Dom_A$ nel seguente modo:

$$v((x_{C_M(n)}, x_{C_M(n)})) = f_v(v_2(e_1),\ f_v(v_2(e_2),\ ...\ v_2(e_z)...))$$

e se inoltre, $(n, n') \in A_1$, modifichiamo tale valore in modo da includere anche il valore dell'arco in $G_1$:

$$v((x_{C_M(n)}, x_{C_M(n)})) = f_v(\, v((x_{C_M(n)}, x_{C_M(n)})),\ v_1((n, n'))\,)$$

### 3.8.3 ESEMPI

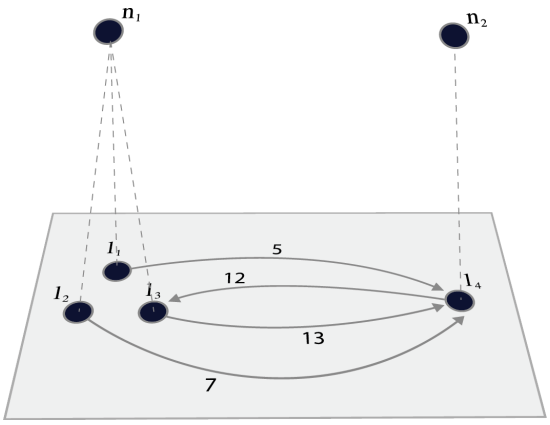
Riprendiamo adesso l'esempio 3.8.1, cercando di capire come possiamo utilizzare questo operatore, e qual'è il risultato atteso. Chiamiamo $C = (A, B, M, w)$ l'accoppiamento tra i due Network Level, dove $M = \{(a, a_2), (a_2, a), (b, b_2), (b_2, b)\}$, e dove non ci interessa come sia fatta la funzione $w$. Possiamo fare l'operazione di join nel seguente modo:

$$B \bowtie_{M, f_v, f_d} A$$

dove $f_d$ ed $f_v$ modificano rispettivamente i dati dei nodi, e dei valori associati agli archi in modo opportuno (ma è ininfluente dare ulteriori dettagli visto che nell'esempio 3.8.1 non parliamo di dati). Il risultato atteso è proprio il Network Level di destra, in Figura 3.8.1. Da notare che se avessimo invertito gli operandi avremmo avuto il medesimo risultato, ma senza il nodo $c_2$.

Cerchiamo però adesso di fare altri esempi, in modo da riuscire meglio a notare altri dettagli. Prima di tutto ci focalizziamo funzioni $f_d$ e $f_v$, poi cercheremo anche di osservare in che modo l'operatore non sia simmetrico.

**Esempio 3.8.2.** Supponiamo di avere due Network Level $A = (N_A, A_A, v_A, d_A)$ e $B = (N_B, A_B, v_B, d_B)$. Il livello $A$ ha due nodi, uno che rappresenta una nazione $n_1$, ed un altro che ne rappresenta un'altra $n_2$. Ogni nodo del livello $A$ è associato a più nodi del livello sottostante: $n_1$ è associato ai tre nodi $l_1$, $l_2$, $l_3$ poiché questi rappresentano dei luoghi geografici che appartengono alla nazione $n_1$; mentre $n_2$ è associato al solo nodo $l_4$ per lo stesso motivo.

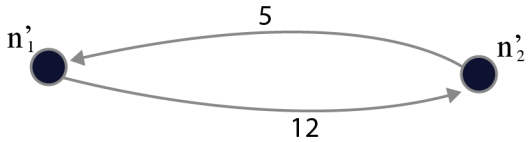



I nodi del livello $B$ sono poi collegati secondo degli archi che indicano i tempi di percorrenza tra i luoghi. Ciò che vogliamo ottenere è, a livello $A$, un riassunto del livello sottostante, ottenendo la percorrenza minima tra i due stati.

Per risolvere questo esempio, utilizziamo il join. Chiamiamo $C = (A, B, M, w)$ il Couple tra i due livelli che esprime le associazioni appena descritte, dove $M = \{(n_1, l_1), (l_1, n_1), (n_1, l_2), (l_2, n_1), (n_1, l_3), (l_3, n_1), (n_2, l_4), (l_4, n_2)\}$. Definiamo poi la funzione $f_d$ come quella che, presi i dati di due nodi, restituisce sempre quelli che risiedono nel livello $A$. Definiamo invece la funzione $f_v$ come quella che, presi i valori di due archi (numeri nel nostro caso), restituisce sempre quelli più piccoli. Utilizziamo quindi il join nel seguente modo:

$$A \bowtie_{M, f_d, f_v} B$$

Il valore ritornato dall'operazione è un Network Level, dove abbiamo solamente due nodi, uno che rappresenta $n_1$ e tutti i suoi nodi associati, con i dati di $n_1$ stesso; un altro invece che rappresenta $n_2$, con le stesse considerazioni.



## 3.9 LAVORI CORRELATI

In letteratura possono essere trovati diversi lavori che parlano di linguaggi per query su grafi. Tali linguaggi sono per lo più incentrati su database a grafi generici, che non hanno un particolare campo di applicazione. Al contrario, nonostante abbiamo cercato di definire la nostra algebra restando il più possibile generici, sappiamo che il nostro obbiettivo principale è quello di fare qualcosa che possa essere applicato ai social network, ed in particolare all'analisi di questi (piuttosto che alla loro creazione), comparando i diversi grafi (i.e. Network Level) che rappresentano reti differenti o prospettive differenti della stessa.

In questa sezione cercheremo di raccogliere i principali lavori che sono stati eseguiti in questo campo, comparandoli con l'algebra da noi proposta. Tali lavori sono quelli che hanno molto in comune con la nostra algebra. Una tabella cercherà poi di riassumere alcune differenze principali, elencando anche ulteriori lavori.

### 3.9.1 G

G [37] è un linguaggio per fare query su dati rappresentati come grafi etichettati. Questo è basato sulle espressioni regolari, che fanno in modo di formulare in maniera semplice query ricorsive. Gli autori comparano il potere espressivo di tale linguaggio con l'algebra relazionale, facendo vedere come sia possibile fare un mapping tra le relazioni e i grafi. In tal modo mostrano come sia possibile vedere tale linguaggio grafico per i grafi come un linguaggio per database basati sul concetto di relazione. Seguendo questa strada, non si propongono come un linguaggio che sostituisce quello comunemente utilizzato nei database relazionali, ma piuttosto come un linguaggio complementare, che permette di fare in maniera semplice query ricorsive.

Le query vengono formulate in maniera "grafica": una graphical query Q, su un grafo G, è un insieme di multigrafi etichettati dove i nodi possono essere variabili o costanti, e dove gli archi etichettati sono espressioni regolari. Il risultato è il grafo che soddisfa tutti i grafi della query.

Anche nell'algebra qui proposta vengono utilizzate delle query basate su espressioni regolari, ma a differenza del linguaggio G, non abbiamo query grafiche, ma utilizziamo degli operatori più simili allo stile adottato nell'algebra relazionale. Inoltre, anche nel nostro caso comparare la nostra algebra con l'algebra relazionale sembra essere altrettanto semplice, poiché utilizzando gli insiemi di path ciò che facciamo è semplicemente dare una visione diversa del grafo, che più rassomiglia ad una relazione.

A partire da tale linguaggio sono anche state proposte diverse estensioni. Tra quelle più rappresentative troviamo G+ [38] e Graphlog [39].

G+ Questo lavoro [38] ha meglio chiarito la forma del risultato finale di una query. In particolare viene introdotto un grafo riassuntivo (summary graph), che viene utilizzato come un nuovo elemento da specificare insieme alla query; esso rappresenta come deve essere ristrutturato un risultato ottenuto da una query.

GRAPHLOG In Graphlog [39] differisce da G+ per un modello dei dati più generale, per l'uso della negazione e per la trattabilità computazionale. In G+ è stato infatti tralasciato uno studio della complessità, evidenziando il fatto che avrebbero dovuto studiare meglio come potere implementare algoritmi più efficienti. Inoltre, il potere espressivo di Graphlog viene comparato con Datalog, fornendo quindi dei risultati più soddisfacenti rispetto ai precedenti lavori.

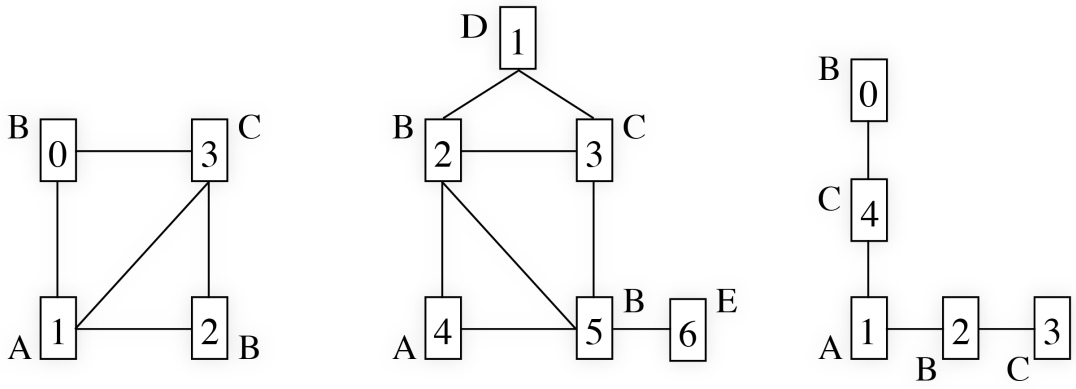
Anche in Graphlog vengono introdotti degli operatori di aggregazione, chiamati: *aggregazione* e *riepilogo di path*. Il primo può aggregare degli insiemi di valori all'interno degli archi. Il secondo invece, chiamato operatore di riepilogo, può riassumere informazioni lungo i path, rispondendo a domande come: "cerca la lunghezza del path più piccolo tra due nodi". Tali operatori sembrano pertanto permettere di rispondere alle stesse domande di quelle dell'operatore di aggregazione del presente

elaborato. Tuttavia dobbiamo comunque pensare che quest'ultimo lavora con insiemi di dati differenti, e che potrebbe pertanto avere delle differenze in termini di complessità e potere espressivo.

### 3.9.2 GLIDE

Glide [40] discende da due query language: Xpath [41] e Smart [42]. In Xpath, progettato per lavorare con XML, le query sono espresse utilizzando complesse espressioni per path, dove i filtri e le condizioni di matching sono espresse all'interno della notazione stessa dei nodi. Allo stesso modo in Glide vengono espresse delle espressioni per grafi anziché per path. Smart, al contrario è stato progettato per lavorare con database di molecole. Glide adotta la stessa notazione per esprimere i cicli, ma la generalizza per poterla utilizzare in un contesto applicativo generale, non più legato alle molecole.

Una query del linguaggio Glide può essere vista come una rappresentazione lineare di un albero, generato da una Deep First Search (DFS) eseguita sul grafo di una query. Glide viene utilizzato in un modello dei dati che assume il fatto che ogni nodo del grafo ha associato un identificativo numerico ed anche un'etichetta, queste query possono essere descritte sotto forma di una stringa. I nodi vengono rappresentati utilizzando le loro label, separati utilizzando il simbolo '/'. I rami vengono racchiusi dalle parentesi, mentre i cicli vengono rappresentati spezzando un arco ed etichettandolo utilizzando un intero. I vertici dell'arco spezzato sono poi rappresentati con la loro etichetta, seguita dal simbolo '%', dall'intero e dal simbolo '/'. Se poi lo stesso nodo è un vertice di più archi spezzati allora l'etichetta del nodo viene seguita da più simboli '%' con corrispettivo intero.




(a) Graph g1 (b) Graph g2 (c) Graph g3

**Figura 3.9.1:** L'esempio riportato nell'articolo che parla di Glide [40]. Il grafo $g1$ è rappresentato dall'espressione 'A%1%2/B/C%2/B%1/', il grafo $g2$ è rappresentato dall'espressione 'A%3/B%1%2%3/(E/)C%2/D%1/', mentre il grafo $g3$ è rappresentato dall'espressione 'B/C/A/B/C/'.

Le componenti non specificate del grafo sono poi descritte per mezzo di wildcard: '*', '.', '+' e '?'. La prima rappresenta nessuno o più nodi, la seconda rappresenta un singolo nodo, la terza rappresenta uno o più nodi, mentre la quarta rappresenta nessuno o un nodo.

Questo lavoro può essere considerato simile all'algebra che abbiamo proposto, poiché anche nel nostro caso, utilizzando la notazione dei path pattern, riusciamo a rappresentare un path generico. Nel nostro caso, l'identificativo del nodo è sottinteso: quando scriviamo il path pattern con i nodi $a \rightarrow b$ in realtà ci stiamo riferendo ai nodi $a$ e $b$ che sono unici nel grafo (non stiamo parlando delle loro etichette), in una implementazione reale questi nodi devono essere identificati in qualche modo, siano essi degli identificativi numerici o dei puntatori in locazioni di memoria.

Dal nostro punto di vista Glide semplifica il fatto che i dati potrebbero non essere delle semplici etichette, ma possono essere arbitrariamente complessi e il matching

può essere altrettanto complesso, necessitando di operatori come il '≤', che di fatto rappresenta un insieme di valori ammissibili. Dal nostro punto di vista, l'operatore di selezione, utilizzando anche i path pattern, sopperisce a queste mancanze, assomigliando molto più all'operatore di selezione dell'algebra relazionale dove vengono specificati gli elementi che si vogliono, ed a parte le condizioni.

Da notare invece che la notazione riportata in Glide permette di esprimere cicli, cosa che non è possibile nella nostra algebra, utilizzando le attuali definizioni di path pattern. Nonostante ciò, anche l'insieme di ritorno è differente, e fare un confronto di questo tipo potrebbe non essere appropriato.

### 3.9.3 Partially Ordered Regular Languages for Graph Queries

Flesca e Greco, in due articoli [31, 32], parlano di un'algebra per fare query su grafi, sfruttando anche loro le espressioni regolari.

Il lavoro di questi due autori è incentrato sui dati che troviamo tutti i giorni sul Web, ovvero gli ipertesti. Questi formano un grafo che è considerevolmente grande se si pensa all'intero World Wide Web. Gli stessi autori osservano come una tipica query che ha la forma "trova tutti gli oggetti raggiungibili da un certo nodo in modo che il path che li colleghi formi una certa parola", può essere onerosa in un'applicazione pratica, per via della grandezza spropositata dei dati che si stanno trattando.

Per questo motivo gli autori dei due articoli hanno fatto un'osservazione molto naturale, ovvero quella di non ritornare tutto l'insieme di dati che costituisce il risultato, ma solamente una sua determinata porzione, rispettando un ordinamento definito dall'utente. Questo è ciò che viene ad esempio fatto attualmente nei motori di ricerca.

Per fare ciò hanno definito un linguaggio regolare parzialmente ordinato, come un'estensione dei classici linguaggi regolari. Due stringhe $s_1$ ed $s_2$, che denotano due path

all'interno del grafo, si dice che $s_1 > s_2$ nel senso che $s_1$ viene preferito al path $s_2$. La cosa interessante è quella che non è necessario esplorare tutto il grafo per poter dare questa risposta limitata, ed anche che lo stesso utente può definire le preferenze che desidera. In seguito sono stati definiti anche le corrispettive estensioni in termini di automi che riconoscono tali linguaggi regolari, e sono state studiate le complessità per cercare in un grafo per mezzo di queste query con path parzialmente ordinate.

Questo lavoro è quindi molto interessante, soprattutto perché nell'algebra che è stata definita in realtà non facciamo nessun cenno su questo problema. Anche nel nostro caso ritornare un insieme di path di una query può essere oneroso, e per query, non limitate da condizioni molto restringenti, si rischia di veder ritornato un risultato piuttosto consistente. Sarebbe quindi una buona strada considerare il lavoro di questi autori, in modo da estendere la nostra algebra, e ritornare un risultato limitato.

### 3.9.4 GRAM

Gram [21], come il lavoro precedente, definisce un linguaggio focalizzato sugli ipertesti, assumendo quindi un modello costituito da un grafo diretto etichettato. Utilizza anch'esso le espressioni regolari sui tipi di dato. Un esempio potrebbe essere un espressione che specifica di avere come primo nodo uno di tipo PAESE, poi una successione arbitraria di nodi di tipo INCROCIO, per poi finire con un nodo di tipo PAESE.

Similmente a come è stata definita l'algebra qui descritta, anche Gram utilizza un concetto di tuple, che non vengono chiamati path, ma hyperwalk. Per spiegare tale concetto utilizzano una classica espressione SQL.

**Esempio 3.9.1.** Supponiamo di avere una relazione HOTEL(NAME, ADDRESS, CITY) ed un'altra RESTAURANT(NAME, ADDRESS, CITY). Vogliamo scrivere una query SQL che estrae tutte le città con un albergo chiamato "Royal" ed un ristorante "McDonalds".

```
select HOTEL.CITY
from HOTEL, RESTAURANT
where HOTEL.NAME='Royal'
and RESTAURANT.NAME='MCDonalds'
and HOTEL.CITY=RESTAURANT.CITY
```

I valori ritornati sono quindi della città, dove ognuna di queste la possiamo immaginare come un nodo connesso ai valori di tipo HOTEL e RESTAURANT, con un arco etichettato con un indirizzo stradale. Questi valori, delle tuple quindi, sono ciò che gli autori chiamano hyperwalk.

In maniera più formale, per prima cosa dobbiamo definire il concetto di *walk*: una sequenza $n_0 e_0 n_1 e_1 ... n_{i-1} e_{i-1} n_i$ di nodi ed archi. Gli hyperwalk sono invece degli insiemi di walk. Gli autori di Gram, hanno definito quindi delle espressioni regolari per questi hyperwalk. Nell'esempio precedente l'espressione regolare che definisce tutte le tuple che hanno quella forma è "CITY addr RESTAURANT + CITY addr HOTEL". Tali espressioni possono essere utilizzate come pattern per operatori di selezione.

Per quanto riguarda gli operatori, Gram definisce un operatore di renaming, che distingue nodi di un'espressione che hanno lo stesso tipo. Nella nostra algebra ciò è superfluo, poiché possiamo riferirci a qualunque elemento utilizzando l'operatore di proiezione (in 3.4), che ritorna un elemento ben definito di un path. Inoltre, viene definito anche un operatore di selezione e concatenazione, molto simili a quelli definiti in 3.5 e 3.3. L'operatore di proiezione poi riesce a selezionare dei sottoinsiemi di walk, e delle sotto-walk. Quest'ultimo è quindi simile al nostro operatore di proiezione, dove però i sottoinsiemi di walk possono essere trovati andando ad utilizzare l'operatore di selezione, combinato con quello di sintesi. L'operatore di join invece è totalmente differente da quello proposto nella nostra algebra, poiché unisce insiemi di hyperwalk secondo opportune condizioni, in maniera quasi equivalente di ciò che viene fatto nell'algebra relazionale.

### 3.9.5 TABELLA RIASSUNTIVA

Adesso riassumeremo in un'unica tabella (i.e. Tabella 3.9.5) le differenze tra l'algebra che abbiamo definito in questo trattato e quella dei principali lavori inerenti. Per far ciò abbiamo definito poche caratteristiche principali:

- Modello utilizzato. In questo caso abbiamo notato che, nonostante il campo di applicazione, si preferisca sempre utilizzare dei grafi diretti etichettati. Da notare come solamente il nostro linguaggio abbia degli operatori studiati proprio per un modello con grafi multi-livello (nel nostro caso Multi Level Network).

- Contesto: il campo di applicazione, dove tale linguaggio dovrebbe essere utilizzato. Abbiamo notato che andando avanti con gli anni sempre più si sia delineato un contesto, ed in particolare il contesto Web.

- Operatori. Molto spesso di selezione che estrapolano sotto-grafi. Lavori meno recenti tendono ad utilizzare soprattutto operatori per la manipolazione di grafi.

- Espressioni Regolari: l'uso delle espressioni regolari per il linguaggio. Questo aspetto è stato molto importante poiché ci ha confermato il fatto che utilizzarle è stata una buona scelta. Come la maggio parte di questi lavori, anche altri lavori [43, 44, 45, 46], che non descrivevano propriamente un linguaggio, parlano dell'uso di espressioni regolari per utilizzarle nei grafi, descrivendo anche molti altri aspetti di complessità ed espressività.

- Potere espressivo: come il linguaggio venga considerato espressivo. Con il passare degli anni abbiamo notato che sempre di più si tende a costruire linguaggi che risolvano task ben precisi ed utili per il particolare campo d'applicazione, mentre in passato si tendeva a comparare il potere espressivo con linguaggi noti conosciuti essere molto espressivi.

- Indici: la presenza o meno nei vari lavori di considerazioni sugli indici, che aiutino ad ottimizzare le prestazioni del linguaggio. Tale aspetto è per noi importante poiché vorremmo in futuro approfondirlo.
- Ricorsione: il fatto di avere nel linguaggio un concetto di ricorsione. Non è stato inserito nella tabella poiché tutti hanno un concetto di ricorsione, indispensabile quando si parla di grafi.

| Nome | Modello | Contesto | Operatori | Espressioni Regolari | Potere Espressivo | Indici |
|---|---|---|---|---|---|---|
| Logical DB Model [47] (1984) | digraph | generico | due operatori che creano una coppia di nodi connessi tra loro | no | no | no |
| G [37] (1987) | digraph | generico | query grafiche (selezione) | sì | mapping con algebra relazionale | no |
| Graphlog [39] (1990) | digraph | generico | come G, con aggregazione | sì | comparato a datalog | no |
| G-Log [48] (1992) | digraph | generico | query grafiche | no | comparato a datalog | no |
| GOOD [4] (1990) | digraph | generico | add nodo, add arco, rem nodo, rem arco, astrazione (crea nuovi nodi a partire da proprietà del grafo) | no | comprato all'algebra relazionale | no |
| Glide [40] (2002) | grafi (non diretti) (non etichettati) | generico | selezione | sì | no | sì |
| Flesca-Greco [32, 31] (1999) | digraph | Web | selezione (in stile SQL) | sì | no | no |
| Gram [21] (1992) | digraph | ipertesti | selezione, renaming, proiezione, join, concatenazione | sì | no | no |
| Multi Level Network (2013) | digraph (multi-livello) | social network | selezione, proiezione, concatenazione, sintesi, raggruppamento, join | sì | no | no |

**Tabella 3.9.5:** Tabella riassuntiva

# CAPITOLO 4

# ALGORITMI E PROPRIETÀ

In questo capitolo discuteremo in maniera più pratica gli operatori della nostra algebra che abbiamo definito precedentemente. Ovvero, dopo aver definito come questi si comportano (dato un input, quale sia il risultato che ci attendiamo), discuteremo come ognuno di questi possa essere implementato, considerando anche tutti i dettagli necessari per far in modo che la loro esecuzione sia efficiente.

Tuttavia, data una determinata query dove al suo interno vengono utilizzati diversi operatori, il DBMS può pianificare in diversi modi come processarla e come produrre il risultato. Ma quanti modi diversi abbiamo per pianificare? Qual'è quello più efficiente? Cercheremo pertanto anche di discutere come i singoli operatori si rapportano agli altri (proprietà), in modo da fare ulteriori ottimizzazioni.

In accordo con l'architettura descritta da Ioannidis [49], la query può essere ottimizzata in due fasi differenti. La prima viene chiamata *Rewriter Stage*, dove, viene generato un insieme di query equivalenti: la query iniziale viene riscritta in altre equivalenti, che *potrebbero* essere più efficienti. Tali riscritture vengono fatte solamente guardando le caratteristiche statiche della query, senza tener conto delle caratteristiche del DBMS o dei dati che vi sono contenuti in un preciso istante. Per quest'ultimo

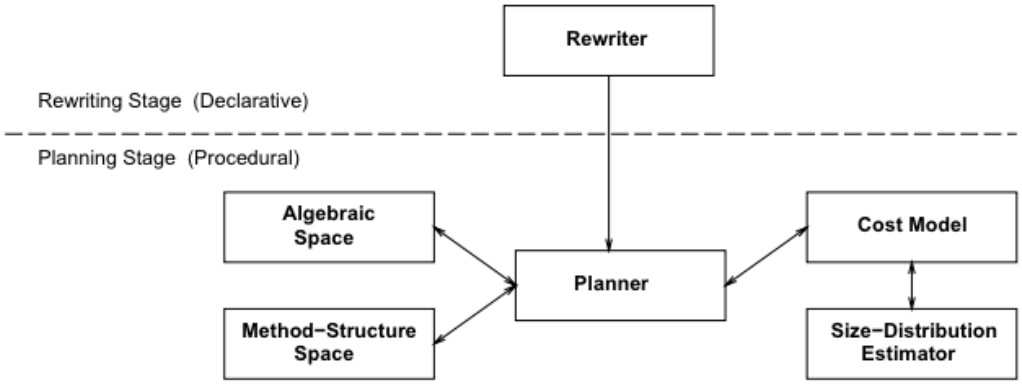



**Figura 4.0.1:** Architettura descritta da Ioannidis [49].

motivo non possiamo sapere se una determinata riscrittura è più efficiente di un'altra, necessitando di avere un insieme di query equivalenti (solamente nel caso in cui una riscrittura è conosciuta essere sempre più efficiente di un'altra la query originaria può essere eliminata). Tale insieme viene esaminato in una seconda fase chiamata *Planning Stage*, dove invece le varie query vengono esaminate, cercando di capire quale potrebbe essere la meno costosa. Tali considerazioni vengono fatte tenendo conto di stime sui dati presenti (i.e. Size-Distribution Estimator) come ad esempio la frequenza di alcuni valori chiave della query, oppure tenendo conto di una stima del costo in tempo di esecuzione (i.e Cost Model) come ad esempio il tempo impiegato dalle operazioni di I/O, oppure tenendo conto dei dettagli implementativi di ogni operatore coinvolto (i.e. Method-Structure Space), ed infine tenendo conto dell'ordine delle varie azioni che devono essere fatte (i.e. Algebraic Space).

## 4.1 Operatore di Proiezione

Questo sarà il primo operatore di cui parleremo. Vedremo come tale operatore abbia caratteristiche simili all'omonimo operatore dell'algebra relazionale, e scopriremo che alcune loro proprietà siano simili.

### 4.1.1 CONSIDERAZIONI OPERATORE

Per prima cosa l'algoritmo deve gestire due tipi differenti di dati: un insieme di path ed un path singolo. Ciò può essere irrilevante da un punto di vista concettuale, poiché molte volte l'insieme di un solo elemento viene inteso come l'elemento stesso, ma da un punto di vista algoritmico ciò ha molta importanza. Considerare singoli path come insiemi di un solo path pregiudicherebbe infatti la terminazione stessa dell'algoritmo (si osservi la prima condizione della definizione dell'operatore). Nonostante ciò, in seguito ci limiteremo a discutere del caso più generale dove l'operatore ha come input un insieme di path.

L'algoritmo ha bisogno di analizzare tutti i path presenti nel dato di input per poter generare l'output e pertanto l'algoritmo deve essere un $\Omega(n)$. Una possibile implementazione semplice è quella dove viene preso un path per volta dall'insieme di input e dove, per ognuno, si calcolano dapprima gli indici (i.e. $s$ ed $e$) relativi al path in esame, e successivamente si ricopiano nell'output solamente i nodi opportuni.

Da un punto di vista della memoria non abbiamo esigenze particolari: ogni elemento di input viene analizzato, trasformato e aggiunto all'insieme di output. Non c'è quindi una necessità di tenere in memoria informazioni particolari che si accumulano durante l'esecuzione dell'operatore: se l'insieme di input e quello di output vengono mantenuti in memoria secondaria, la memoria primaria necessaria è pertanto quella che basta per contenere il più grande path dell'insieme di input.

Nell'algoritmo 4.1.1 possiamo osservare un'implementazione dell'operatore. Possiamo notare come questo segua in maniera identica le definizioni dell'operatore che sono state date, e che specificavano solamente quale dovesse essere l'output da generare.

**Algorithm 1** Operatore di proiezione $\pi_{s,e}(P)$

```
R ← ∅
if P è un insieme then
    for all p ∈ P do
        R ← R ∪ π_{s,e}(p)
    end for
else
    k ← len(P)
    if s(P) > k then
        R ← { ( ) }
    else
        R ← sottopath of P from s(P) to min(k, e(P))
    end if
end if
return R
```

### 4.1.2 OTTIMIZZAZIONI

BUFFER Utilizzando più memoria primaria è invece possibile fare delle ottimizzazioni: se l'input è contenuto in una memoria secondaria può essere utile mantenere un buffer, leggendo più path per volta, in modo da ridurre i tempi necessari alla testina per identificare una posizione all'interno del disco.

Data la struttura non omogenea dei path di input (ogni path può avere lunghezze diverse dagli altri), tale ottimizzazione non è banale. Infatti, un buffer con una dimensione fissa non contiene sempre lo stesso numero di path (potrebbe addirittura essere troppo piccolo per contenerne uno): è necessario pertanto avere un buffer che sia almeno più grande del più grande path di input. Una tale informazione apparentemente non può essere ottenuta senza prima leggere i dati di input, cadendo quindi in un circolo vizioso.

Tale problema può essere risolto memorizzando gli insiemi di path in modo che contengano delle informazioni sul loro contenuto, come il numero di elementi e la dimensione massima dei path.

### 4.1.3 PROPRIETÀ

#### 4.1.3.1 DISTRIBUTIVITÀ

L'operazione di proiezione è distributiva rispetto all'operazione di unione di insiemi di path. Ovvero fare una proiezione dell'unione tra due insiemi è equivalente a fare l'unione delle due proiezioni dei singoli insiemi.

**Lemma 4.1.1.** Siano $P$ e $Q$ due insiemi di path e siano $s$ ed $e$ due indici per l'operazione di proiezione. Vale che:

$$\pi_{s,e}(P \cup Q) = \pi_{s,e}(P) \cup \pi_{s,e}(Q)$$

*Dimostrazione.* La dimostrazione segue la definizione dell'operatore di proiezione, nel suo unico caso che considera il dato di input come un insieme di path $P$:

$$\pi_{s,e}(P) = \bigcup_{p \in P} \pi_{s,e}(p)$$

Con questa osservazione possiamo subito dimostrare il lemma:

$$\begin{aligned}
\pi_{s,e}(P \cup Q) &= \bigcup_{p \in (P \cup Q)} \pi_{s,e}(p) \\
&= \{\pi_{s,e}(p) \ : \ p \in P \cup Q\} \\
&= \{\pi_{s,e}(p) \ : \ p \in P\} \cup \{\pi_{s,e}(p) \ : \ p \in Q\} \\
&= \bigcup_{p \in P} \pi_{s,e}(p) \ \cup \ \bigcup_{p \in Q} \pi_{s,e}(p) \\
&= \pi_{s,e}(P) \cup \pi_{s,e}(Q)
\end{aligned}$$

□

Mentre invece l'operatore non è distributivo rispetto all'operatore di intersezione.

**Lemma 4.1.2.** Siano $P$ e $Q$ due insiemi di path e siano $s$ ed $e$ due indici per l'operazione di proiezione. *Non* vale sempre che:

$$\pi_{s,e}(P \cap Q) = \pi_{s,e}(P) \cap \pi_{s,e}(Q)$$

*Dimostrazione.* La dimostrazione può essere fatta con un controesempio. Istanziando $P$ come l'insieme $\{(n_1, n_2, n_3)\}$ e $Q$ come l'insieme $\{(n_1, n_2, n_4)\}$, e dove $s(p) = 1$ ed $e(p) = 2$. Abbiamo che:

$$\pi_{s,e}(P \cap Q) = \pi_{1,2}(\{(n_1, n_2, n_3)\} \cap \{(n_1, n_2, n_4)\}) = \pi_{s,e}(\emptyset) = \emptyset$$

mentre invece

$$\begin{aligned}\pi_{s,e}(P) \cap \pi_{s,e}(Q) &= \pi_{1,2}(\{(n_1, n_2, n_3)\}) \ \cap \ \pi_{1,2}(\{(n_1, n_2, n_4)\}) \\ &= \{(n_1, n_2)\} \cap \{(n_1, n_2)\} \\ &= \{(n_1, n_2)\}\end{aligned}$$

□

Allo stesso modo l'operatore non è distributivo rispetto alla differenza.

**Lemma 4.1.3.** Siano $P$ e $Q$ due insiemi di path e siano $s$ ed $e$ due indici per l'operazione di proiezione. *Non* vale sempre che:

$$\pi_{s,e}(P \setminus Q) = \pi_{s,e}(P) \setminus \pi_{s,e}(Q)$$

*Dimostrazione.* La dimostrazione può essere fatta con un controesempio. Istanziando $P$ come l'insieme $\{(n_1, n_2, n_3)\}$ e $Q$ come l'insieme $\{(n_1, n_2, n_4)\}$, e dove $s(p) = 1$ ed $e(p) = 2$. Abbiamo che:

$$\pi_{s,e}(P \setminus Q) = \pi_{1,2}(\{(n_1, n_2, n_3)\} \setminus \{(n_1, n_2, n_4)\}) = \pi_{1,2}(\{(n_1, n_2, n_3)\}) = \{(n_1, n_2)\}$$

mentre invece

$$\begin{aligned}\pi_{s,e}(P) \backslash \pi_{s,e}(Q) &= \pi_{1,2}(\{(n_1, n_2, n_3)\}) \backslash \pi_{1,2}(\{(n_1, n_2, n_4)\}) \\ &= \{(n_1, n_2)\} \backslash \{(n_1, n_2)\} \\ &= \emptyset\end{aligned}$$

□

#### 4.1.3.2 IDEMPOTENZA

Nell'algebra relazionale l'operatore di proiezione si dice idempotente, poiché una serie di proiezioni sono equivalenti a quella più esterna. Ovvero:

$$\pi_{a_1,...,a_n}(\pi_{b_1,...,b_m}(R)) = \pi_{a_1,...,a_n}(R) \text{ dove } \{a_1, ..., a_n\} \subseteq \{b_1, ..., b_m\}$$

Ciò è utile per ottimizzare una query che presenta diversi operatori di proiezione annidati, sostituendoli con uno solo, lasciando inalterato il risultato. Nel caso dell'algebra relazionale, sotto certe condizioni, questo può essere fatto lasciando solamente l'operatore più esterno, ed eliminando quelli più interni.

Nella nostra algebra la presenza di indici, che sono in realtà funzioni che dipendono dall'input, ciò è più complicato, ma nonostante ciò questo tipo di ottimizzazioni possono essere fatte. Per cominciare facciamo un esempio:

**Esempio 4.1.1.** Supponiamo di avere un path $P = (n_1, n_2, ..., n_{12})$. Supponiamo poi di voler fare la proiezione $\pi_{4,5}(\pi_{4,9}(P))$, l'operatore più interno ritorna un path $(n_4, ..., n_9)$ di lunghezza 5. Ciò significa che l'operatore più esterno non ritornerà $(n_4, n_5)$, ma $(n_7, n_8)$, poiché i suoi indici si riferiscono al path prodotto dall'operatore più interno. Ciò significa che, diversamente di quanto accade nell'algebra relazionale, non possiamo sostituire direttamente gli operatori con quello più esterno, è necessario cambiare i suoi indici.

Generalizziamo allora il concetto dato nell'esempio:

**Proposizione 4.1.1.** Sia la query $\pi_{s_a,e_a}(\pi_{s_b,e_b}(P))$ una generica query annidata su un input generico $P$, dove gli indici sono descritti dalle funzioni $s_a$, $s_b$, $e_a$, $e_b$. Allora esistono sempre degli indici $s$ ed $e$ tale che:

$$\pi_{s_a,e_a}(\pi_{s_b,e_b}(P)) = \pi_{s,e}(P)$$

Dimostriamo questa proposizione in modo costruttivo, facendo vedere come possono essere create le funzioni $s$ ed $e$. Per fare ciò dobbiamo prima analizzare qual'è la lunghezza del risultato di un'operazione di proiezione.

**Proposizione 4.1.2.** Sia $P = (n_1, n_2, ..., n_k)$ un path generico, e sia $\pi_{s,e}(P)$ una generica operazione di proiezioni con gli indici $s$ ed $e$. Definiamo con la funzione $len_{\pi_{s,e}}(P)$ la lunghezza del risultato dell'operazione:

$$len_{\pi_{s,e}}(P) = \begin{cases} min(e(P), k) - s(P) & \text{se } s(P) \leq k \\ -1 & \text{altrimenti} \end{cases}$$

Tale definizione può essere provata semplicemente ispezionando i casi della definizione di operatore di proiezione. Per capire con semplicità tale definizione può essere utile avere un esempio:

**Esempio 4.1.2.** Sia $P = (n_1, ..., n_6)$ un path e sia $\pi_{s,e}(P)$ un'operazione di proiezione fatta sul path $P$ con gli indici $s(p) = 3$ ed $e(p) = 5$. Il risultato di tale operazione è il path $(n_3, n_4, n_5)$ di lunghezza 2. Con la definizione precedente possiamo anche sapere prima di compiere l'operazione quale sarà la lunghezza di tale risultato: in questo caso $len_{\pi_{s,e}}(P) = min(6,5) - 3 = 2$.

Una volta che possiamo prevedere quale sarà la lunghezza del risultato di un'operazione di proiezione, possiamo unire gli indici in modo opportuno. Tuttavia, visto che

gli indici prendono in input un path $p$, utilizzandolo per recuperarne la lunghezza, dobbiamo poter generare dei path fittizi. Questo modo di procedere è ovviamente costoso in termini di tempo e memoria, che ci serve a noi per semplificare la dimostrazione; in una reale implementazione la stessa cosa può essere fatta sostituendo il corpo delle funzioni in maniera opportuna (e.g. $len(p)$ sostituito con un numero) con un costo praticamente nullo.

**Definizione 4.1.1.** Sia $n \in \mathbb{N} \cup \{-1\}$ un numero. Si definisce $CP(n)$ una funzione che costruisce un path $P$ di lunghezza $n$. I nodi contenuti in tale path non sono importanti ai fini dei nostri scopi. Con $CP(-1)$ viene costruito un path di lunghezza $-1$, ovvero il path nullo `()`.

Dopo avere descritto tutti gli elementi di cui avevamo bisogno per dimostrare in maniera costruttiva la proposizione 4.1.1, possiamo procedere a descrivere il modo con la quale possono essere creati gli indici.

**Lemma 4.1.4.** Sia la query $\pi_{s_a,e_a}(\pi_{s_b,e_b}(P))$ una generica query annidata su un input generico $P$, dove gli indici sono descritti dalle funzioni $s_a$, $s_b$, $e_a$, $e_b$. Gli indici $s$ ed $e$ tale che $\pi_{s_a,e_a}(\pi_{s_b,e_b}(P)) = \pi_{s,e}(P)$ sono definiti nel seguente modo:

$$s(P) = s_b(P) - 1 + s_a(CP(len_{\pi_{s_b,e_b}}(P)))$$

$$e(P) = s_b(P) - 1 + e_a(CP(len_{\pi_{s_b,e_b}}(P)))$$

Negli indici $s_b(P) - 1$ funge da offset, in modo da traslare in avanti gli indici $s_a$ ed $e_a$ in modo opportuno. In seguito, i valori di quest'ultimi indici vengono calcolati su dei path fittizi, come se fossero il risultato dell'operatore di proiezione più interno.

Per capire meglio come questi indici vengono creati, riprenderemo l'esempio 4.1.1, facendo vedere questa volta come siamo in grado di trasformare i due operatori annidati in uno solo, a beneficio del costo di esecuzione di tale query.

**Esempio 4.1.3.** Supponiamo di avere un path $P = (n_1, n_2, ..., n_{12})$. Supponiamo poi di voler fare la proiezione $\pi_{4,5}(\pi_{4,9}(P))$ dove, l'operatore più interno ritorna un path $(n_4, ..., n_9)$ di lunghezza 5, mentre l'operatore più esterno prende il quarto e quinto nodo di quest'ultimo, con il risultato finale di $(n_7, n_8)$. In accordo con quanto detto nel lemma 4.1.4, possiamo eseguire le due proiezioni con una sola: $\pi_{4,5}(\pi_{4,9}(P)) = \pi_{s,e}(P)$. In questo esempio possiamo calcolare gli indici come $s(p) = 4-1+4$ e $e(p) = 4-1+5$. Riassumendo abbiamo $\pi_{4,5}(\pi_{4,9}(P)) = \pi_{7,8}(P)$

Proviamo anche ad osservare un caso limite, in modo da capire meglio il concetto dell'offset:

**Esempio 4.1.4.** Supponiamo di avere un path $P = (n_1, n_2, ..., n_{12})$. Supponiamo poi di voler fare la proiezione $\pi_{4,5}(\pi_{1,9}(P))$ dove, l'operatore più interno ritorna un path $(n_1, ..., n_9)$ di lunghezza 8, mentre l'operatore più esterno prende il quarto e quinto nodo di quest'ultima, con il risultato finale di $(n_4, n_5)$. In questo caso possiamo vedere come l'operatore interno non toglie nodi prima del risultato, ed infatti l'offset $s_b(P) - 1 = 0$. In accordo con quanto detto nel lemma 4.1.4, anche in questo caso possiamo eseguire le due proiezioni con una sola: $\pi_{4,5}(\pi_{1,9}(P)) = \pi_{4,5}(P)$.

Per capire invece il perché utilizziamo i path fittizi allo scopo di calcolare gli indici possiamo fare un ulteriore esempio.

**Esempio 4.1.5.** Supponiamo di avere un path $P = (n_1, n_2, ..., n_{12})$. Supponiamo poi di voler fare la proiezione $\pi_{1,len(p)}(\pi_{4,9}(P))$ dove, l'operatore più interno ritorna un path $(n_4, ..., n_9)$ di lunghezza 5, mentre l'operatore più esterno toglie l'ultimo nodo da quest'ultimo, con il risultato finale di $(n_4, ..., n_8)$. In accordo con quanto detto nel lemma 4.1.4, possiamo eseguire le due proiezioni con una sola: $\pi_{4,5}(\pi_{4,9}(P)) = \pi_{s,e}(P)$, dove gli indici sono definiti come $s(p) = 4-1+1$ e $e(p) = 4-1+len(CP(len_{\pi_{4,9}}(p)) = 3+len(CP(5)) = 8$. Riassumendo abbiamo $\pi_{1,len(p)}(\pi_{4,9}(P)) = \pi_{4,8}(P)$.

Questa proprietà, come abbiamo visto, è molto importante poiché permette di compattare sequenze di proiezioni in una sola, risparmiando tempo di calcolo ed utilizzo

di memoria. Tale proprietà può essere sempre sfruttata come una riscrittura della query, poiché indipendentemente dal DBMS dai dati che si stanno elaborando.

#### 4.1.3.3 INDICI COSTANTI

Se l'operazione di proiezione viene fatta con indici costanti si possono fare alcune ottimizzazioni. Ricordiamo che gli indici dell'operatore sono le funzioni $s$ ed $e$, che diciamo essere costanti se $\nexists p, p' \,.\, s(p) \neq s(p')$.

Con tale ipotesi non importa prendere tutti i path e calcolare gli indici su ognuna di esse: questi possono essere calcolati preventivamente. In alcuni casi ciò è semplice, poiché le funzioni $s$ ed $e$ sono a loro volta costanti, in altri casi invece si deve tener conto della natura dei dati di input. Un esempio è quello dove $s(p) = 1$ ed $e(p) = len(p) - 2$: se i path di input hanno tutti la stessa lunghezza gli indici sono costanti.

Per capire se gli indici sono costanti per un determinato input non è furbo analizzare tutti i path di input, ciò introdurrebbe un peggioramento delle prestazioni. Piuttosto tale analisi deve essere fatta tenendo conto delle informazioni riassuntive dei dati (e.g. come detto precedentemente la dimensione massima dei path, e una dimensione minima per avere un informazione riassuntiva che dica che tutti i path hanno ugual lunghezza).

#### 4.1.3.4 LETTURA PARZIALE CON INDICI COSTANTI

In presenza di indici costanti si possono fare ulteriori ottimizzazioni. Infatti sappiamo già le porzioni di path che ci interessano, potremmo non avere la necessità di leggere un path per intero per costruire il risultato finale. Per capire meglio lasciamoci guidare da un esempio.

**Esempio 4.1.6.** Supponiamo che una query $\sigma_{a\to*\to e}(G)$, ritorni tutti i path che dal nodo $a$ giungono al nodo $e$. Tralasciando i dettagli del grafo, supponiamo che tale operatore ritorni l'insieme di path $I = \{\{a,b,c,d,e\},\{a,f,g,e\},\{a,i,l,m,e\}\}$. Vorremmo prendere il secondo nodo di ogni path (i.e. $\{b,f,i\}$), poiché ad esempio ci interessa conoscere chi, partendo dal nodo $a$, è il prossimo nodo che ci permette di poter comunicare con $e$. Ciò può essere fatto con la semplice operazione di proiezione $\pi_2(I) = \{b,f,i\}$.

Possiamo notare dall'esempio come in realtà non abbiamo bisogno di prendere i path per intero, possiamo semplicemente prendere il secondo elemento di ognuno. In questo particolare esempio ciò potrebbe apparire irrilevante, ma con path molto grandi invece la differenza può essere enorme.

Se i path sono ad esempio salvati in una memoria secondaria ciò influisce positivamente sulle prestazioni, visto che alcuni blocchi, che sappiamo già essere inutili, non vengono letti.

**Esempio 4.1.7.** Supponiamo di avere un insieme $I$ di 1000 path di lunghezza 200 ognuno, e supponiamo di voler fare una proiezione $\pi_2(I)$ che come nell'esempio precedente recupera il secondo nodo di ogni path. Supponiamo che nel disco, dove sono state salvati i path, ogni blocco può contenere al più 10 identificativi di un nodo del grafo, ovvero può contenere al più dieci nodi di un path. La dimensione necessaria a contenere ogni path è quindi di 20 blocchi. Ciò significa che per ottenere il risultato finale è necessario soltanto posizionarsi nel primo blocco che contiene il primo path dell'insieme $I$ e poi leggere un blocco ogni venti. Infine di ogni blocco letto prendere soltanto il secondo nodo. In questo modo anziché leggere 20000 blocchi basta leggerne solamente 1000.

Se i path invece non sono stati salvati, ma devono essere calcolati, fungendo solamente come risultato intermedio tra il creatore e l'operazione di proiezione, ciò può avere ulteriori benefici. Si evita addirittura di esplorare il grafo: ciò è quello che vedremo quando parleremo delle proprietà dell'operatore di selezione.

## 4.2 Operatore di Selezione

L'operatore di selezione è indubbiamente l'operatore più importante della nostra algebra, scelto con molta cura in modo che, messo in relazione agli altri operatori, creasse un'algebra consistente. Tale operatore è indubbiamente differente dalla maggior parte di quelli che possono essere trovati in letteratura, poiché si basa sul fatto di ritornare insiemi di path piuttosto che sotto-grafi. Ciononostante l'accoppiata dell'operatore di selezione con quello di sintesi fornisce l'usuale risultato. Ricordiamo che ciò è stato fatto proprio per riuscire a dare un risultato che contenesse più informazioni rispetto al un singolo grafo, ed è stato fondamentale per riuscire a realizzare l'operatore di aggregazione. D'altra parte è comunque necessario considerare l'aspetto della dimensione del risultato, che può essere anche consistente.

In questa sezione forniremo comunque gli aspetti implementativi più basilari, fornendo comunque degli spunti per ottimizzazioni, impiegando delle proprietà utili in tal senso. Vedremo come abbiamo utilizzato in parte alcuni concetti già noti delle espressioni regolari, in modo da avere un ulteriore base che può aiutare a far progredire più celermente il lavoro.

### 4.2.1 Considerazioni operatore

L'idea di base è quella di utilizzare gli automi a stati finiti per riuscire a riconoscere i path all'interno del grafo, proprio come viene fatto in [50, 51] nelle espressioni regolari. Nonostante ciò abbiamo comunque notevoli differenze tra espressioni regolari e path pattern. Per prima cosa non abbiamo da analizzare delle stringhe ma dei grafi. Inoltre il nostro scopo non è quello di dire se una stringa (nel nostro caso un path) è riconosciuta o meno, ma invece quello di trovare tutti i possibili path (del grafo) che soddisfano tale path pattern. Infine ricordiamo che non abbiamo un vero e proprio alfabeto da poter utilizzare nelle espressioni (e.g. l'espressione `'a*b'` che riconosce

tutte le stringhe formate da un numero arbitrario di `'a'` che terminano con `'b'`), bensì gli elementi dei path pattern sono identificatori di nodi.

L'automa generato a partire da un path pattern servirà pertanto come uno strumento utile per la ricerca dell'insieme di path che costituisce il risultato dell'operatore di selezione. Vediamo adesso come può essere costruito.

#### 4.2.1.1 COSTRUZIONE AUTOMA NON DETERMINISTICO A STATI FINITI

Adesso illustreremo come possiamo trasformare un path pattern in un automa non deterministico a stati finiti. Per riferirci alle varie componenti che andremo ad illustrare, indicando con $P$ un generico path pattern, $A(P)$ sarà l'automa corrispondente.

Ogni automa ha delle caratteristiche comuni ben definite: ha uno stato iniziale (indicato da una freccia, come in Figura 4.2.1) ed uno stato finale (indicato con due cerchi concentrici, come in Figura 4.2.2). In generale ogni automa $A(P)$ di un path pattern $P$ può essere rappresentato come un generico stato composto da uno stato iniziale ed uno finale, indicato come in Figura 4.2.3.

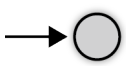


**Figura 4.2.1:** Nodo iniziale

**Figura 4.2.2:** Nodo finale

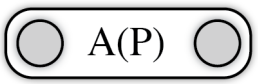



**Figura 4.2.3:** Automa astratto

Ricordiamo che con gli automi delle espressioni regolari [51] inizialmente si pone come carattere corrente quello iniziale della stringa da analizzare, e ci si muove dallo stato iniziale con una transizione etichettata $a$ soltanto se tale carattere è proprio $a$. In tal caso il carattere corrente diventa il successivo. Si continua in questo modo fino a raggiungere uno stato finale, ce decreta l'accettazione della stringa.

**Esempio 4.2.1.** Facendo un esempio, se avessimo l'espressione regolare `'a*b'`, l'automa risultante sarebbe il seguente:

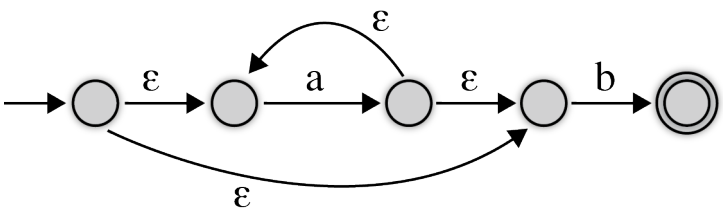



**Figura 4.2.4:** Esempio di un automa generato a partire dall'espressione regolare `'a*b'`

Supponiamo adesso di voler analizzare la stringa `'ab'`, questa viene analizzata nel seguente modo: si pone come carattere corrente la prima lettera `'a'`, si esegue una $\epsilon$-transizione per giungere al secondo stato e quindi si esegue la transizione etichettata `a` per giungere al terzo stato (a questo punto il carattere corrente diventa la seconda lettera `'b'`), compiendo un'altra $\epsilon$-transizione si giunge al quarto stato, ed infine con la transizione etichettata `b` si arriva allo stato di accettazione e la stringa viene riconosciuta.

Nel nostro caso le cose sono diverse. Non abbiamo nessuna stringa, il primo "carattere" diventa pertanto uno qualsiasi dei nodi presenti nel grafo. La transizione etichettata `a` significa che abbiamo visitato il nodo $a$, in altre parole eseguiamo la transizione etichettata `a` solamente se dal nodo corrente esiste un arco che ci porta sul nodo $a$.

Per definire come viene creato l'automa $A(P)$, di un path pattern $P$, iniziamo dagli elementi base dei path pattern:

- Se $P = \emptyset$ l'automa corrispondente $A(P)$ è il seguente:

**Figura 4.2.5:** Automa corrispondente al pattern $\emptyset$

Ovvero non c'è un modo per arrivare dallo stato iniziale a quello finale e pertanto nessun path sarà mai valido. Ricordiamo a tal proposito che con $P = \emptyset$ abbiamo $M(P) = \emptyset$.

- Se $P = \epsilon$ l'automa corrispondente $A(P)$ è il seguente:

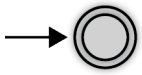


**Figura 4.2.6:** Automa corrispondente al pattern $\epsilon$

Ovvero lo stato finale è anche quello iniziale e la ricerca pertanto termina subito. Ricordiamo a tal proposito che con $P = \epsilon$ abbiamo $M(P) = \{ ( ) \}$.

- Se $P = n$ l'automa corrispondente $A(P)$ è il seguente:

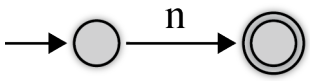



**Figura 4.2.7:** Automa corrispondente al pattern $n$

Ovvero lo stato finale è raggiungibile soltanto se dal nodo corrente esiste un arco tale da raggiungere il nodo $n$. All'inizio dell'esplorazione non abbiamo un nodo corrente e pertanto tale transizione viene fatta se il nodo $n$ esiste all'interno del grafo (successivamente $n$ diventa il nodo corrente). Ricordiamo che con $P = n$ abbiamo $M(P) = \{ (n) \}$, ovvero il solo path accettato da questo path pattern è quello che contiene il nodo $n$ singolarmente.

- Se $P = \%$ l'automa corrispondente $A(P)$ è il seguente:

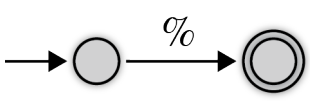



**Figura 4.2.8:** Automa corrispondente al pattern %

Ovvero lo stato finale è raggiungibile soltanto se dal nodo corrente riusciamo a raggiungere un qualsiasi nodo (che nell'automa abbiamo indicato con l'etichetta %). Anche in questo caso inizialmente non abbiamo un nodo corrente e pertanto tale transizione viene fatta per un qualsiasi nodo del grafo. Ricordiamo infatti che con $P = \%$ abbiamo $M(P) = \{ (n) : n \in N \}$ dove $N$ è l'insieme dei nodi contenuti nel grafo.

- Se $P = ?$ l'automa corrispondente $A(P)$ è il seguente:

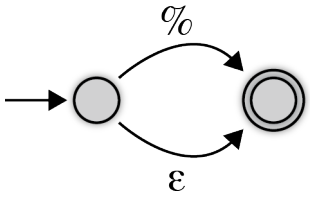



**Figura 4.2.9:** Automa corrispondente al pattern ?

Ovvero lo stato finale viene raggiunto sia che dal nodo corrente riusciamo a raggiungere un qualsiasi nodo, sia che rimaniamo fermi (i.e. non cambiando il nodo corrente).

- Se $P = *$ l'automa corrispondente $A(P)$ è il seguente:

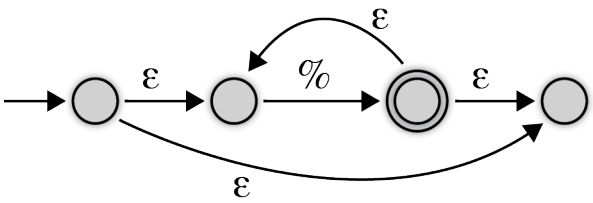



**Figura 4.2.10:** Automa corrispondente al pattern $*$

Ovvero lo stato finale viene raggiunto visitando un numero arbitrario (anche nullo) di nodi. Ricordiamo infatti che con $P = *$ abbiamo $M(P) = \{ p : p$ è path di G$\}$ dove $G$ indica il grafo analizzato.

Una volta che abbiamo la possibilità di costruire gli elementi di base dell'automa, possiamo adesso comporre dei path pattern più complessi, dove abbiamo delle sotto-formule. Ciò è possibile grazie al fatto che ogni automa rispecchia la forma descritta dalla Figura 4.2.3.

- Se $P = P_1 \rightarrow P_2$ l'automa corrispondente $A(P)$ è il seguente:

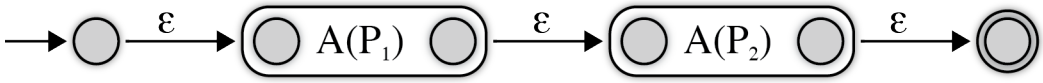



**Figura 4.2.11:** Automa corrispondente al pattern $P_1 \rightarrow P_2$

Ovvero lo stato finale viene raggiunto solamente dopo aver terminato entrambe le sotto-formule del path pattern, rispettando l'ordine.

- Se $P = P_1|P_2$ l'automa corrispondente $A(P)$ è il seguente:

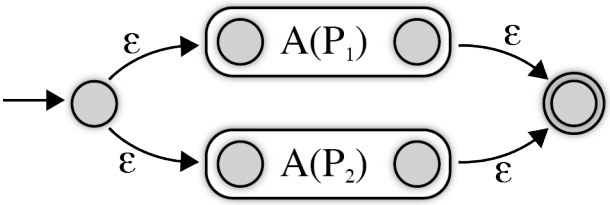



**Figura 4.2.12:** Automa corrispondente al pattern $P_1|P_2$

Ovvero il lo stato finale viene raggiunto solamente dopo aver terminato una delle due sotto-formule.

Per essere più esplicativi faremo adesso un esempio.

**Esempio 4.2.2.** Sia $P = a \rightarrow *$ un path pattern. L'automa risultante $A(P)$ può essere costruito facendo dapprima l'automa $A(a)$, poi l'automa $A(*)$, e successivamente

unendoli come abbiamo mostrato precedentemente nella definizione (i.e. $A(P_1 \rightarrow P_2)$). Il risultato finale è il seguente

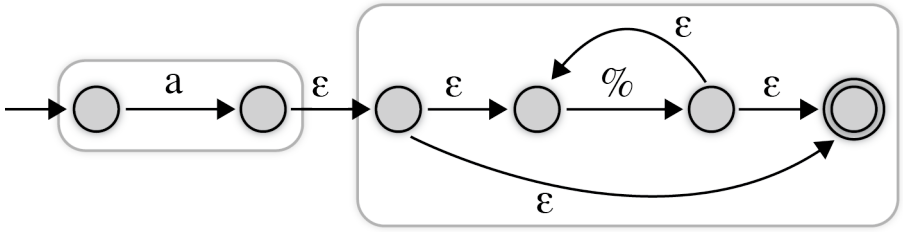



**Figura 4.2.13:** Automa dell'esempio

#### 4.2.1.2 UTILIZZO DELL'AUTOMA

Riprendiamo adesso l'esempio precedente, cercando di capire come può essere utilizzato in un Network Level.

**Esempio 4.2.3.** Sia $G = (N, A, \nu, d)$ un Network Level, rappresentato graficamente dalla seguente figura:

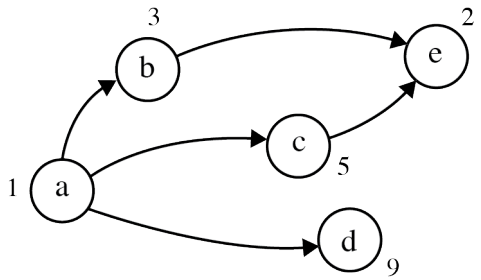



**Figura 4.2.14:** Network Level dell'esempio 4.2.3

Supponiamo adesso di voler selezionare tutti i path che partono dal nodo $a$, utilizzando l'operatore di selezione ed il path pattern $P$ che abbiamo definito nell'esempio 4.2.2

$$\sigma_{a \rightarrow *}(G)$$

Tale operatore, una volta costruito l'automa del path pattern, come descritto nell'esempio 4.2.2, lo sfrutterà nel modo seguente.

Si posiziona lo stato dell'automa al primo stato (quello più a sinistra nella figura dell'esempio 4.2.2), in questo momento non c'è un nodo corrente e per effettuare il primo cambio di stato dell'automa tale nodo diventa proprio il nodo $a$. A questo punto viene svolta l'unica transizione, etichettata $\epsilon$, che porta lo stato dell'automa al terzo (partendo da sinistra). A questo punto notiamo che possono essere fatte due possibili transizioni. La prima porta ad uno stato di accettazione, e di conseguenza il path composto dal solo nodo $a$ è già un path corretto, che può essere restituito insieme al risultato finale della query di selezione. La seconda invece porta al terzo stato (partendo sempre da sinistra), e quindi obbligatoriamente al quarto, dove è necessario aggiungere un altro nodo al path. Supponiamo adesso che tale transizione avvenga aggiungendo il nodo $d$: in questo caso non abbiamo altra scelta che dover terminare ed andare nello stato di accettazione ($d$ non ha archi uscenti), il path $(a,d)$ può essere quindi restituito anch'esso come risultato. Il passaggio dal quarto al quinto stato può però poter avvenire aggiungendo nodi come $b$ o $c$, e di conseguenza generando altri path. Riassumendo, il risultato è formato dall'insieme di path $\{(a),(a,b),(a,c),(a,d),(a,b,e),(a,c,e)\}$

Dall'ultimo esempio possiamo notare che, per poter restituire tutti i path che soddisfano un determinato path pattern, è necessario comportarsi in modo completamente differente da come avviene nelle espressioni regolari, poiché il nostro scopo non è quello di arrivare ad uno stato finale, bensì quello di esplorare tutti i modi possibili per raggiungerlo; inoltre non abbiamo delle stringhe, che possono essere interpretate come grafi che hanno un solo arco uscente che porta dal carattere corrente a quello successivo, ma abbiamo a che fare con dei grafi che possono avere più di un arco uscente, ed anche con la possibilità che generino cicli.

In generale la ricerca dei path deve soddisfare le seguenti:

- Nel caso uno stato presenti transizioni differenti queste vengano tutte fatte (come abbiamo visto nell'esempio precedente quando avevamo più di un $\epsilon$-transizione).
- Nel caso un nodo abbia più di un arco uscente che soddisfi un cambio di stato dell'automa, questi vengano tutti considerati (come abbiamo visto nell'esempio precedente quando il cambio di stato poteva essere soddisfatto sia dall'arco che portava al nodo $b$, $c$ o $d$).
- Non è possibile effettuare una transizione dello stato dell'automa utilizzando un nodo già visitato.

#### 4.2.1.3 VINCOLI

Nel caso in cui l'operatore di selezione dovesse avere dei vincoli che limitano lo spazio delle soluzioni è necessario ovviamente considerarli.

Ricordiamo che l'operatore di selezione è definito nel seguente modo:

$$\sigma_{P_G,F(p)}(G) = \{p \,:\, p \vDash P_G \wedge F(p)\}$$

dove $P_G$ è il path pattern che identifica il path con una struttura valida, mentre $F(p)$ è un predicato che, preso un path valido, determina se questo ha dei contenuti idonei per poter essere restituito come risultato.

La soluzione più naturale è la seguente: quando ci troviamo in uno stato finale (di accettazione), significa che il path $p$ che abbiamo costruito soddisfa il path pattern, e tale path può essere aggiunto al risultato solamente se vale il predicato $F(p)$.

### 4.2.2 OTTIMIZZAZIONI

#### 4.2.2.1 DEPTH-FIRST E BREADTH-FIRST SEARCH

La costruzione del risultato della selezione può essere implementato con un algoritmo di ricerca che esplora il Network Level, passato come input, ed anche l'automa costruito a partire dal path pattern passato anch'esso come input. D'altra parte sappiamo che l'esplorazione di queste due tipologie di grafi può essere fatta sostanzialmente in due modi differenti: Depth-first search e Breadth-first search.

Nel primo caso è evidente come la costruzione dei path, che costituiscono il risultato finale, sia fatta per passi successivi, ovvero viene costruito un path per volta. Nel secondo caso invece il risultato finale viene costruito tutto insieme, aggiungendo ad ogni path del risultato ogni volta un nodo in più.

La prima tipologia di ricerca è sicuramente più efficiente in termini di memoria, poiché non vengono aggiunti tutti i nodi del livello n-esimo dell'albero di ricerca, ma viene esplorato solamente un ramo per volta. Ciò è altrettanto vero quando ci si trova a gestire dei grafi simili a quelli dei social network, dove notiamo fenomeni come quello che viene chiamato in letteratura "small world" [52]. In quest'ultimo caso infatti ci si ritrova ad avere, circa al quarto livello dell'albero di ricerca, la stragrande maggioranza dei nodi presenti nell'intero grafo.

L'algoritmo di ricerca dell'operatore di selezione deve quindi analizzare un ramo per volta, possibilmente escludendo rami dell'albero ritenuti già essere inutili, proprio come vedremo in seguito.

#### 4.2.2.2 BRANCH AND BOUND UTILIZZANDO I VINCOLI

La soluzione proposta in 4.2.1.3 è sicuramente naturale e non crea problemi. Quest'ultima assicura infatti che il predicato $F(p)$ venga valutato su un path strutturalmente coerente a ciò che tale predicato si aspetta, si pensi al predicato che indaga sul

dato contenuto nel decimo nodo, ma che il path ha ancora due nodi. Ciò comunque non ci esula dal pensare ad un'altra soluzione, dove il predicato si valuta prima ancora di sapere che un path soddisfi il pattern (ovvero prima ancora di essere in uno stato di finale), in modo da evitare di ricercare path più lunghi.

**Esempio 4.2.4.** Riprendiamo il Network Level $G$ che abbiamo riportato nell'esempio 4.2.3. Supponiamo adesso di voler utilizzare l'operatore di selezione nel seguente modo:

$$\sigma_{a \rightarrow *, \pi_2(p).d=3}(G)$$

ovvero vogliamo tutti i path che partono dal nodo $a$ e che hanno il secondo nodo con un valore 3. Durante la costruzione delle soluzioni il path $(a, c)$ verrà sicuramente considerato, ma possiamo già scartarlo e scartare tutti i path costruiti a partire da esso, poiché il secondo nodo rende già falso il predicato.

Il predicato potrebbe quindi essere valutato ad ogni cambio di stato dell'automa generato a partire dal path pattern, in modo da evitare la espansione di path che sappiamo essere già non valide. Ciò limita la ricerca delle soluzioni, specialmente quando ci sono molti modi per poter formare dei path (ovvero abbiamo un elevato branching factor).

Se pensiamo tale predicato come una congiunzione di predicati che devono quindi tutti essere veri:

$$F(p) = A_1(p) \wedge A_2(p) ... \wedge A_n(p)$$

Possiamo affermare che il path $p$ *non* può essere restituito come risultato se $\exists i \, . \, \neg A_i(p)$, ma sotto alcune condizioni:

- Un predicato $A_i(p)$ potrebbe non essere valutabile, poiché il path ancora non ha elementi a sufficienza. Si pensi ad $A_i$ che analizza il decimo nodo, ma che il path ha ancora soltanto due nodi. Un tale predicato non deve quindi essere preso in considerazione.

- Un predicato $A_i(p)$ potrebbe essere valutabile, ma restituisce un valore negativo soltanto perché il path ancora non è completa. Si pensi ad esempio al predicato $A_i(p) = len(p) > 10$: il path che si sta costruendo, e che ad esempio è ancora lungo due nodi, non può essere preventivamente scartato. Tale predicato deve quindi essere utilizzato solamene in corrispondenza di uno stato di accettazione.

In generale possiamo quindi scartare un path $p$ e tutti i path $v$ (con $p \subset v$) se:

- $A_i(p)$, che restituisce un valore negativo, valuta il dato presente in un nodo.
- $A_i(p)$, che restituisce un valore negativo, valuta il dato presente di un arco.
- $A_i(p)$, che restituisce un valore negativo, valuta la lunghezza massima del path.

Durante la fase di implementazione vedremo che tale ottimizzazione è stata implementata. In particolare per realizzare qualcosa di generale è stato utilizzato il seguente approccio. Il predicato $F(p)$ può ritornare, oltre ai classici valori booleani $True$ e $False$, anche un altro valore $DefinitelyFalse$. Quest'ultimo significa semplicemente che il path $p$ non è valido e non lo sarà neanche un qualsiasi altro path ottenuto espandendo $p$. Un esempio è quello del predicato $F(p)$ che ritiene valido il path $p$ solamente se ha lunghezza $n$. Questo predicato ritornerà $False$ per tutti i path con lunghezza minore di $n$, ritornerà $True$ per tutti quelli con lunghezza uguale ad $n$, ed infine ritornerà $DefinitelyFalse$ per tutti quelli con lunghezza maggiore di $n$.

### 4.2.3 PROPRIETÀ

#### 4.2.3.1 OPERAZIONI ANNIDATE ED OPERATORE DI SINTESI

Come abbiamo già visto, l'operatore di selezione ritorna un insieme di path, mentre invece accetta come input soltanto un Network Level. Ciò potrebbe a prima vista

far notare un difetto nella definizione che limita l'espressività dell'algebra. Tuttavia l'operatore di sintesi è stato definito appositamente per risolvere questo problema.

A volte è conveniente comunque utilizzare l'operatore di sintesi. Ciò è vero per un semplice motivo: l'operatore di selezione ritorna un risultato che può essere molto più grande, rispetto al risultato ritornato dall'operatore di sintesi. Per capire tale concetto facciamo un esempio.

**Esempio 4.2.5.** Supponiamo di avere un Network Level $G$ strutturato come un albero ad $n$ livelli, dove un padre ha un arco su tutti i suoi figli. La radice di tale albero è costituita dal nodo $a$, mentre il branching factor è di 10. Supponiamo di voler fare l'operazione di selezione $\sigma_{a\rightarrow *}(G)$, che restituisce tutti i percorsi possibili dalla radice ad uno dei suoi discendenti. Abbiamo che tale operatore restituisce per prima cosa la radice ($a$), in seguito 10 path di due elementi, dove il primo nodo è sempre il nodo $a$ ed il secondo nodo è invece uno dei suoi figli. Ciò continua fino al livello n-esimo. Riassumendo, il risultato dell'operatore di selezione è costituito da un path di lunghezza 1, 10 path di lunghezza 2, $10^2$ path di lunghezza 3, $10^{n-1}$ path di lunghezza $n$. Il risultato finale è pertanto composto da un numero di nodi pari a: $\sum_{i=1}^{n} i * 10^{i-1}$. Al contrario $G$ è composto di un numero di nodi di molto inferiore: $\sum_{i=1}^{n} 10^{i-1}$.

In generale quindi abbiamo queste due problematiche. Da una parte vorremmo, come abbiamo visto nell'esempio precedente, avere tutti i possibili percorsi dalla radice ad uno dei nodi sottostanti, dall'altra però abbiamo bisogno di un notevole spazio di memoria. *Nel caso in cui non ci interessi il singolo path*, ma soltanto un sotto-grafo, l'operatore di selezione può invece essere utilizzato insieme all'operatore di sintesi.

Un'implementazione dell'operatore di selezione può sfruttare il fatto che tale operatore venga utilizzato insieme all'operatore di sintesi, in modo da non generare più inutilmente un grande input e poi in seguito il sotto-grafo, ma generare direttamente il sotto-grafo (i.e. sotto-Network Level) del Network Level di partenza che costituisce il risultato della query.

#### 4.2.3.2 LIMITARE L'OUTPUT

Si immagini uno scenario dove vogliamo capire se due nodi $a$ e $b$ sono connessi. In tal caso abbiamo bisogno di una query che, utilizzi l'operatore di selezione, e che utilizzi un path pattern che potrebbe essere il seguente: $a \rightarrow * \rightarrow b$. In seguito basterebbe controllare che il risultato di questa query non sia nullo.

Per un simile scenario non abbiamo quindi bisogno di avere tutti i path dal nodo $a$ al nodo $b$, ma abbiamo bisogno di un numero limitato di path (in questo caso ne basta uno).

Altri lavori, che hanno un operatore di selezione simile, hanno già affrontato il problema [32, 31], definendo un ordine tra i risultati che vengono ritornati, e dando quindi la possibilità di poter ritornare solamente quelli più rilevanti.

Allo stesso modo, in questo caso, un'operazione del tipo:

$$(\sigma_{a \rightarrow * \rightarrow b}(G)) \neq \emptyset$$

può essere sempre ottimizzata ricorrendo ad algoritmi appositi, che non esplorano tutto l'automa e tutto il grafo, ma che cercano direttamente il path migliore.

Un esempio in questo senso potrebbe essere l'utilizzo di A* [53], in nodi che ad esempio sono distribuiti geograficamente, ma in generale in tutte quelle situazioni in cui si riesce a definire una stima euristica.

#### 4.2.3.3 PROPRIETÀ DEI PATH PATTERN

Come abbiamo potuto notare, nella definizione di path pattern manca il simbolo "+" comunemente noto quando si parla di espressioni regolari, e che indica una sequenza che deve avere almeno un elemento.

**Proposizione 4.2.1.** Sia $P_1 = \%$ un path pattern che indica un path di un solo elemento, e sia $P_2 = *$ un path pattern che indica un generico path che può essere nullo. Il path pattern $P_3 = +$, che indica un generico path non nullo, può essere scritto mediante il path pattern $P_4 = P_1 \rightarrow P_2 = P_3$.

Oltre a questa particolarità possono essere osservate anche altre proprietà che riguardano i path pattern.

**Proposizione 4.2.2** (Commutatività dell'alternazione)**.** Siano $P_1$ e $P_2$ due path pattern. Vale che il path pattern $P_1|P_2$ è semanticamente equivalente al path pattern $P_2|P_1$.

*Dimostrazione.* Osservando le definizioni di path pattern sappiamo che $M(P_1|P_2) = \{p : p \in M(P_1) \vee p \in M(P_2)\}$, e che $M(P_2|P_1) = \{p : p \in M(P_2) \vee p \in M(P_1)\}$. Per commutatività dell'operazione logico $\vee$, i due insiemi sono equivalenti, e pertanto anche la semantica dei due path pattern. □

**Proposizione 4.2.3** (Associatività dell'alternazione)**.** Siano $P_1$, $P_2$ e $P_3$ tre path pattern. Vale che il path pattern $(P_1|P_2)|P_3$ è semanticamente equivalente al path pattern $P_1|(P_2|P_3)$.

*Dimostrazione.* Osservando le definizioni di path pattern sappiamo che $M((P_1|P_2)|P_3) = \{p : p \in M(P_1|P_2) \vee p \in M(P_3)\}$. Osservando che $M(P_1|P_2) = \{p : p \in M(P_1) \vee p \in M(P_2)\}$, sappiamo che $p \in M(P_1|P_2) \iff (p \in M(P_1) \vee p \in M(P_2))$. Detto ciò possiamo affermare che $M((P_1|P_2)|P_3) = \{p : p \in M(P_1) \vee p \in M(P_2) \vee p \in M(P_3)\}$, e, per via della commutatività dell'operatore logico $\vee$, possiamo concludere che $M((P_1|P_2)|P_3) = M(P_1(P_2|P_3))$ e di conseguenza che la semantica dei due path pattern sia equivalente. □

**Proposizione 4.2.4** (Associatività della concatenazione)**.** Siano $P_1$, $P_2$ e $P_3$ tre path pattern. Vale che il path pattern $(P_1 \rightarrow P_2) \rightarrow P_3$ è semanticamente equivalente al path pattern $P1 \rightarrow (P2 \rightarrow P3)$.

*Dimostrazione.* Osservando le definizioni di path pattern sappiamo che $M((P_1 \rightarrow P_2) \rightarrow P_3) = \{\, p.p_3 \,:\, p \in M(P_1 \rightarrow P_2) \,\wedge\, p_3 \in M(P_3) \,\wedge\, p.p_3$ è un path di $G\,\}$. Sappiamo inoltre che

$$p \in M(P_1 \rightarrow P_2) \iff p = p_1.p_2 \wedge p_1 \in M(P_1) \wedge p_2 \in M(P_2) \wedge p_1.p_2 \text{ è un path di } G$$

Di conseguenza possiamo riscrivere $M((P_1 \rightarrow P_2) \rightarrow P_3)$ come l'insieme $\{p_1.p_2.p_3 \,:\, p_1 \in M(P_1) \wedge p_2 \in M(P_2) \,\wedge\, p_3 \in M(P_3) \wedge p_1.p_2$ è un path di $G \wedge p_2.p_3$ è un path di $G\}$. Per via della commutatività dell'operatore logico $\wedge$, possiamo ottenere lo stesso insieme per $M(P_1 \rightarrow (P_2 \rightarrow P_3))$ e pertanto possiamo concludere che $M((P_1 \rightarrow P_2) \rightarrow P_3) = M(P_1 \rightarrow (P_2 \rightarrow P_3))$ e che la semantica dei due path pattern sia equivalente. □

**Proposizione 4.2.5** (Identità dell'alternazione)**.** Sia $P_1$ un path pattern. Vale che il path pattern $P_1$ è semanticamente equivalente al path pattern $\emptyset|P_1$ ed anche al path pattern $P_1|\emptyset$.

*Dimostrazione.* Osservando le definizioni di path pattern sappiamo che $M(P_1|\emptyset) = \{p : p \in M(P_1) \vee p \in M(\emptyset)\}$. Visto che sappiamo dalla semantica dei path pattern che $M(\emptyset) = \emptyset$, possiamo riscrivere l'insieme precedente come $M(P_1|\emptyset) = \{p : p \in M(P_1)\}$ che è a sua volta lo stesso di $M(\emptyset|P_1)$ e di $M(P_1)$. E ciò conclude che i tre path pattern siano semanticamente equivalenti. □

## 4.3 Operatore di Aggregazione

L'operatore di aggregazione è molto simile a quello dell'algebra relazionale. Quest'ultimo esegue l'operazione generando un insieme di tuple, dove gli attributi raggruppati non vengono ripetuti. Volendo fare un esempio, il raggruppamento di una relazione *R(Identificativo, Importo)*, specificando come attributo da raggruppare solamente il primo, genera un insieme di tuple dove non è possibile avere due tuple

con stessi identificativi. Inoltre, nell'algebra relazionale, gli attributi da raggruppare, insieme a nuovi attributi che contengono informazioni di sintesi (e.g. massimo, media), sono i soli che possono comparire nel risultato finale.

Analogamente, nel nostro operatore di aggregazione, abbiamo voluto seguire una filosofia simile. Ciò risulta semplice se si pensa che il risultato dell'operatore di selezione è definito come un insieme di path, simili alle tuple dell'algebra relazionale. Per determinare quali sono gli attributi che vengono raggruppati si guarda solamente il parametro *c* dell'operatore, ovvero la funzione che crea i path che costituiscono il risultato finale: sono i nodi di tali path ad essere raggruppati, non è possibile avere nel risultato due path con gli stessi nodi. In altre parole, per chi è abituato a quello dell'algebra relazionale, non vengono specificati gli attributi sul quale fare il raggruppamento, e da un'altra parte gli attributi che si vogliono come risultato (con il vincolo che i soli attributi che possono comparire nel risultato sono un sottoinsieme di quelli specificati per essere raggruppati).

### 4.3.1 CONSIDERAZIONI OPERATORE

L'algoritmo di aggregazione più semplice a cui possiamo pensare è il seguente:

**Algorithm 2** Operatore di aggregazione $\alpha_{G;c;a}(P)$

$H \leftarrow \emptyset$
$N_H \leftarrow \emptyset$
$A_H \leftarrow \emptyset$
$v_H \leftarrow \emptyset$
$d_H \leftarrow \emptyset$
**for all** $p \in P$ **do**
  $c_p \leftarrow c(p)$ ▷ calcola il path aggregato
  **if** $\exists\, h_p \in H$ t.c. $h_p = c_p$ **then**
    aggiorna i valori aggregati di $c_p$ memorizzati in $v_h$ e $d_h$
  **else**
    $H \leftarrow H \cup \{c_p\}$
    **for all** $n \in c_p$ **do**
      $d_H \leftarrow d_H \cup \{(n, G.d(n))\}$
      $N_H \leftarrow N_H \cup n$
    **end for**
    **for all** $n, n' \in c_p$ dove $n$ e $n'$ sono nodi adiacenti **do**
      $v_H \leftarrow v_H \cup \{((n,n'), G.v((n,n')))\}$
      $A_H \leftarrow A_H \cup (n,n')$
    **end for**
    inizializza i valori aggregati di $c_p$ memorizzandoli in $v_h$ e $d_h$
  **end if**
**end for**
**return** $(N_H, A_H, v_H, d_H)$

Come possiamo osservare nell'algoritmo ciò che viene fatto è molto semplice. Per prima cosa vengono inizializzati gli insiemi $H$, $N_H$, $A_H$ come insiemi vuoti, lo stesso viene fatto anche con le funzioni $v_H$ e $d_H$ che abbiamo descritto dal punto di vista insiemistico. Per ogni path dell'input $P$ si indaga se $c(p)$ produce un path precedentemente esaminato. Nel caso in cui $c(p)$ è un path nuovo, i nodi e gli archi di tale path vengono aggiunti rispettivamente negli insiemi $N_H$ e $A_H$, mentre il path stesso viene aggiunto nell'insieme $H$. Allo stesso modo anche i valori di tali nodi ed archi

vengono ricopiati, popolando le funzioni $v_H$ e $d_H$. Gli assegnamenti contenuti in $a$ vengono invece analizzati in modo da popolare ulteriormente queste due ultime funzioni, inizializzando i valori aggregati (e.g. nel caso di un operazione aggregata di massimo viene aggiunto il valore contenuto nel path in esame, nel caso di conteggio invece viene inizializzato il valore con il numero 1). Nel caso invece il path è già stato precedentemente esaminato allora vengono solamente modificati i valori contenuti nelle funzioni $v_H$ e $d_H$ in accordo con gli assegnamenti in $a$ dei valori aggregati (e.g. viene incrementato il conteggio di uno, oppure si determina un nuovo massimo).

### 4.3.2 OTTIMIZZAZIONI

Come possiamo notare nell'algoritmo che abbiamo mostrato, l'insieme $H$ viene utilizzato soltanto per determinare se un path sia stato già precedentemente analizzato. Tale insieme $H$ corrisponde, per come viene illustrato sia nelle definizioni di questo operatore, ad un risultato intermedio, mentre il risultato vero e proprio corrisponde ad una sintesi di questo risultato.

Per tal motivo l'operatore potrebbe essere visto come composto di due parti distinte: una prima che costruisce l'insieme di path, una seconda che invece altro non è che un operazione di sintesi su tale insieme. Nonostante ciò potrebbe invece essere utile non avere tale suddivisione, ottimizzando il modo in cui l'insieme $H$ viene memorizzato ed ottimizzando anche il tempo necessario a determinare se un path è contenuto in $H$ o meno.

Per fare ciò è necessaria una struttura dati che permetta di memorizzare un insieme di path, dove l'obbiettivo potrebbe non essere più quello di sapere quali siano i path in esso contenuti, ma piuttosto sapere se un path è contenuto o meno. Questo concetto è diffuso nel mondo dell'informatica, e si basa sulla perdita di informazioni che non sono utili, ad esempio utilizzando delle funzioni one-way, solitamente chiamate funzioni di hashing.

#### 4.3.2.1 BLOOM FILTER

Una struttura dati nota, che può venire in aiuto in questo tipo di situazione, è conosciuta in letteratura come Bloom filter [54]. Questa è una struttura dati probabilistica, efficiente in termini di memoria, che viene utilizzata per determinare se un elemento è (o non è) un membro di un insieme $S$. Tale struttura viene detta probabilistica poiché è possibile avere dei falsi positivi, ovvero è possibile che una query su tale struttura dati riconosca erroneamente un elemento appartenente all'insieme. Al contrario, non sono possibili dei falsi negativi: se si determina che un elemento non appartiene all'insieme, ciò è sicuramente vero.

Un'altra particolarità del Bloom filter è il fatto che gli elementi possono essere soltanto aggiunti. Ma ciò comunque non è limitante per i nostri scopi. Si noti invece che più è alto il numero di elementi presenti nell'insieme e più è alta la probabilità che si verifichino falsi positivi.

DESCRIZIONE Per prima cosa si definisce un valore numerico $m$, che può cambiare in base al numero di elementi attesi nell'insieme. Il Bloom Filter si basa su un array di $m$ bit $B = \{b_0, b_1, ..., b_{m-1}\}$, inizialmente inizializzato con tutti i bit a 0. Inoltre, devono essere definite $k$ funzioni hash differenti (definiamo $H = \{h_1, h_2, ..., h_k\}$ come l'insieme che le contiene), che mappano ogni elemento dell'insieme $S$ in una delle $m$ posizioni dell'array di bit in maniera uniforme, ovvero $\forall i \in [1,k] \,.\, h_i : S \to [0, m-1]$.

Vale la seguente relazione:

$$\forall i \,.\, b_i = 1 \iff \exists\, s \in S, h \in H \,.\, h(s) = i \tag{4.1}$$

Supponendo di avere un insieme $S$ vuoto, all'inserimento di un elemento in questo insieme bisogna anche aggiornare in maniera analoga l'array $B$ utilizzato dal Bloom Filter, in modo che venga mantenuta la relazione (4.1). Da notare che, nel caso in

cui $S$ non sia vuoto, il Bloom Filter può essere comunque costruito iterando ogni elemento di $S$, ripetendo per ognuno l'aggiornamento.

L'aggiornamento dell'array $B$, in seguito all'inserimento di un elemento nell'insieme $S$, può essere descritto mediante questa semplice funzione:

**Algorithm 3** Inserimento elemento nel Bloom Filter

```
function INSERT(s, H, B)
    for all h ∈ H do
        B[h(s)] = 1
    end for
end function
```

In seguito, per testare che un elemento non appartiene all'insieme, basta semplicemente calcolare le posizioni di ogni funzione hash, e controllare il valore corrispondente sull'array $B$. Se anche solo una funzione hash restituisce una posizione $i$, e l'array $B$ ha un valore nullo nella sua i-esima posizione, allora l'elemento è sicuramente non presente nell'insieme.

La verifica che un elemento sia presente o meno nell'insieme $S$, controllando l'array $B$, può essere descritta mediante questa semplice funzione:

**Algorithm 4** Verifica elemento appartenente all'insieme

```
function ISINSET(s, S, H, B)
    for all h ∈ H do
        if B[h(s)] = 0 then
            return False
        end if
    end for
    return True
end function
```

Da notare che la funzione impiega tempo $O(1)$ e spazio $O(1)$.

PROBABILITÀ DEL FALSO POSITIVO Come già detto, possono essere presenti dei falsi positivi, per via della composizione della struttura dati. Infatti, se $\forall h \in H$ abbiamo $B[h(s)] = 1$, non è detto che l'elemento sia presente nell'insieme $S$. Possiamo però calcolare la probabilità che si verifichi un falso positivo.

Per prima cosa dobbiamo ribadire ancora una volta che le funzioni hash selezionino le varie posizioni in modo uniforme. In seguito, dopo aver inserito $n$ elementi su un array di lunghezza $m$, la probabilità che un particolare bit sia ancora a 0 è la seguente:

$$P(b_i = 0) = \left(1 - \frac{1}{m}\right)^{k*n}$$

Facendo un esempio semplice per capire, con $k = 1$, $m = 3$, e $n = 1$, abbiamo che solo un bit dei tre diventa 1. La probabilità che un particolare bit $b_i$ (con $i \in [1,3]$) sia 0 è di $(1 - 1/3)^1$, ovvero 2/3.

In seguito possiamo dire [55] che la probabilità di un falso positivo è la seguente:

$$\left(1 - \left(1 - \frac{1}{m}\right)^{k*n}\right)^k \approx \left(1 - e^{\frac{k*n}{m}}\right)^k$$

Si può notare come la probabilità di avere falsi positivi diminuisca all'aumentare di $m$, come anche al decrescere di $n$. L'approssimazione può essere minimizzata con $k = ln(2) * \frac{m}{n}$, che diventa:

$$\left(\frac{1}{2}\right)^k$$

Ad esempio, con $n/m = 10$ e $k = 7$, la probabilità di avere un falso positivo è di circa 0.008.

SCOPI Il Bloom Filter può essere utilizzato nell'algoritmo di aggregazione, che abbiamo mostrato precedentemente. Infatti, tale algoritmo controlla se $\exists\ h_p \in H$ t.c. $h_p = c_p$, ovvero se il path correntemente analizzato fa già parte o meno dell'insieme $H$. Nel caso in cui faccia parte dell'insieme è necessario comunque ricercare la sua posizione, ed in seguito aggiornandone i suoi valori; nel caso invece non faccia parte dell'insieme è necessario soltanto aggiungere l'elemento nell'insieme $H$, senza neanche analizzare l'insieme $H$.

In tal senso vediamo come il Bloom Filter si adatta perfettamente ai nostri scopi, ottimizzando la parte dell'algoritmo interessata a ricercare se un elemento sia già stato analizzato o meno.

## 4.4 OPERATORE DI JOIN

L'operatore di Join sicuramente è il più complesso tra quelli che abbiamo definito, principalmente per il semplice fatto di coinvolgere più livelli, e per la sua semantica che, come vedremo, crea e distrugge archi in base al coupling tra i livelli coinvolti.

Cominciamo per prima cosa con il descrivere in maniera astratta l'algoritmo di join, in seguito poi scriveremo uno pseudo-codice che meglio sintetizza tale algoritmo. Infine, come abbiamo fatto per gli altri operatori, faremo delle considerazioni per ottimizzare al meglio l'operatore, anche mettendolo in relazione con gli altri, se possibile.

### 4.4.1 CONSIDERAZIONI OPERATORE

Dati due Network Level $G_1 = (N_1, A_1, v_1, d_1)$ e $G_2 = (N_2, A_2, v_2, d_2)$ ed un accoppiamento $(G_1, G_2, M, w)$ tra i due livelli, si definisce l'operazione di join $G_1 \bowtie_{M, f_d \cdot f_v} G_2$ come quella che genera un nuovo Network Level $G = (N, A, v, d)$, rispettando la definizione in 3.8.2.

#### 4.4.1.1 DESCRIZIONE ALGORITMO

Pertanto, l'algoritmo che implementa questo operatore deve, come prima cosa, creare un Network Level vuoto, che una volta popolato costituirà il risultato finale $G$.

Successivamente si crea un hashmap $M_h$ che avrà la funzione di tenere traccia dove ogni nodo dei due livelli coinvolti verrà utilizzato. Per esempio se $G_1$ ha un nodo $n_1$ associato con un nodo $l_1$ di $G_2$, il join tra i due livelli genererà un nodo che, convenzionalmente, possiamo identificare con $[n_1, l_1]$. L'hashmap $M_h$ allora memorizzerà il fatto che $n_1$ è stato utilizzato per $[n_1, l_1]$. Questa hashmap vedremo che sarà utile per la generazione e la fusione degli archi, poiché ogni nodo del risultato viene attraverso la fusione di più livelli degli operandi, ma anche ogni nodo dei due operandi può essere utilizzato per creare più nodi del risultato.

Dopo questa fase di preparazione, l'algoritmo comincia iterando ogni nodo $n \in N_1$, costruendo per ognuno un nodo $x_{C_M(n)}$, che formerà il risultato. Infatti, ogni nodo di questo viene aggiunto all'insieme $N$. Inoltre il dato di tali nodi vengono ricopiati: $d(x_{C_M(n)}) = d_1(n)$. Viene inoltre presa nota di come ogni nodo $n$ viene utilizzato per formare il risultato finale, ovvero $M_h(n) = \{x_{C_M(n)}\}$.

Come per la definizione dell'operatore di join, che abbiamo dato precedentemente, per ogni nodo $n$ analizzato viene creato un insieme $C_M(n)$, che non è altro che l'insieme dei nodi (nel livello $G_2$) associati ad $n$ (che invece si trova su $G_1$).

Durante l'analisi di un nodo $n$ viene analizzato ogni suo nodo associato, ovvero ogni nodo $l \in C_M(n)$. Ogni nodo $l$ viene difatti utilizzato per costituire il nodo $x_{C_M(n)}$, e pertanto si assegna ad $M_h(l)$ l'insieme $\{x_{C_M(n)}\}$. Visto che il nodo $l$ può essere già stato utilizzato per costruire altri nodi, nel caso in cui $M_h(l)$ non sia vuoto non deve essere fatto l'assegnamento, bensì un'aggiunta: $M_h(l) = M_h(l) \cup \{x_{C_M(n)}\}$.

Non occorre però tenere nota di come i vari nodi del livello $G_2$ contribuiscono alla creazione dei nodi del risultato, è necessario anche integrare il dato del nodo $x_{C_M(n)}$.

Per ogni $l$ che si sta analizzando è necessario pertanto fare la seguente operazione: $d(x_{C_M(n)}) = f_d(d(x_{C_M(n)}), d_2(l))$.

Una volta iterati tutti i nodi di $N_1$, si passa alla costruzione degli archi. Per ogni arco $(n, n') \in A_1$, si crea un arco $(x_{C_M(n)}, x_{C_M(n')})$ e si inserisce in $A$. Ovviamente, visto che ogni nodo $n$ (e $n'$) può essere utilizzato per diversi $x_{C_M(n)}$ (e $x_{C_M(n')}$), allora ciò va fatto per ogni possibile $x_{C_M(n)}$ (e $x_{C_M(n')}$). Per fare ciò è necessario quindi iterare, come appena detto, ogni arco $(n, n') \in A_1$ e per ognuno iterare ogni nodo $x \in M_h(n)$. Per ognuno di questi iterare ogni nodo $x' \in M_h(n')$ e creare l'arco $(x, x')$, aggiungendolo poi ad $A$. Tale operazione deve poi essere ripetuta anche per ogni arco presente in $A_2$.

Per popolare anche i valori associati a tali archi è necessario, durane l'analisi di un generico nodo $(n, n') \in A_1$, per ogni arco $(x, x')$ che si crea, modificare la funzione $v$ nel seguente modo: se $v((x, x'))$ non è ancora definito si inizializza il suo valore facendo $v((x, x')) = v_1((n, n'))$, mentre se invece è definito si integra il valore facendo $v((x, x')) = f_v(v_1((n, n')), v((x, x')))$. Da notare invece come, durante l'analisi di un generico nodo $(n, n') \in A_2$, è necessario utilizzare $v_2$ anziché $v_1$.

#### 4.4.1.2 ALGORITMO

Adesso descriveremo l'algoritmo in maniera più dettagliata, fornendo un suo pseudo-codice. Il funzionamento di tale algoritmo, nel caso non si comprendano alcuni punti, segue esattamente la descrizione precedente.

**Algorithm 5** Operatore di Join $G_1 \bowtie_{M, f_d . f_v} G_2$

```
G ← new network level
M_h ← new map
for all n ∈ G_1.N do
    x ← new node
    add x to G.N
    G.d(x) ← G_1.d(n)
    M_h(n) = {x}
    C_M = C(n, M)
    for all l ∈ C_M do
        if M_h(l) is void then
            M_h(l) ← {x}
        else
            M_h(l) ← M_h(l) ∪ {x}
        end if
        G_d(x) ← f_d(G.d(x), G_2.d(l))
    end for
end for
ArcExp(G_1.A, G_1.v, M_h, G, f_v)
ArcExp(G_2.A, G_2.v, M_h, G, f_v)
return G
```

Come è possibile notare dall'algoritmo appena illustrato, questo necessita di due ulteriori funzioni:

- $C(n, M)$ che crea l'insieme $C_M(n)$ (come descritto nella definizione del join, ovvero tutti i nodi del secondo livello che sono associati ad $n$) considerando il mapping $M$.
- ArcExp($A$, $v$, $M_h$, $G$, $f_v$) che in base ai valori di $M_h$ espande gli archi, ovvero crea tutti i nuovi archi che potrebbero nascere in seguito alla fusione dei due livelli, e fonde quelli che invece hanno vertici che sono stati fusi.

**Algorithm 6** $C_M(n)$

```
function C(n, M)
    R ← {}
    for all (n1, n2) ∈ M do
        if n1 = n then
            R ← R ∪ {n2}
        end if
    end for
    return R
end function
```

**Algorithm 7** $ArcExp$

```
function ArcExp(A, v, Mh, G, fv)
    for all (n1, n2) ∈ A do
        for all x1 ∈ Mh(n1) do
            for all x2 ∈ Mh(n2) do
                G.A ← G.A ∪ {(x1, x2)}
                if G.v((x1, x2)) is null then
                    G.v((x1, x2)) ← v((n1, n2))
                else
                    G.v((x1, x2)) ← fv(G.v((x1, x2)), v((n1, n2)))
                end if
            end for
        end for
    end for
end function
```

#### 4.4.1.3 Considerazioni e ottimizzazioni

Il funzionamento di $M_h$ La prima nota importante è considerare l'importanza di $M_h$. Questo memorizza come un determinato nodo (appartenente ai grafi passati per argomento) viene utilizzato per formare i nuovi nodi. Per riuscire a capire il suo funzionamento ed anche la sua importanza faremo adesso un esempio grafico.

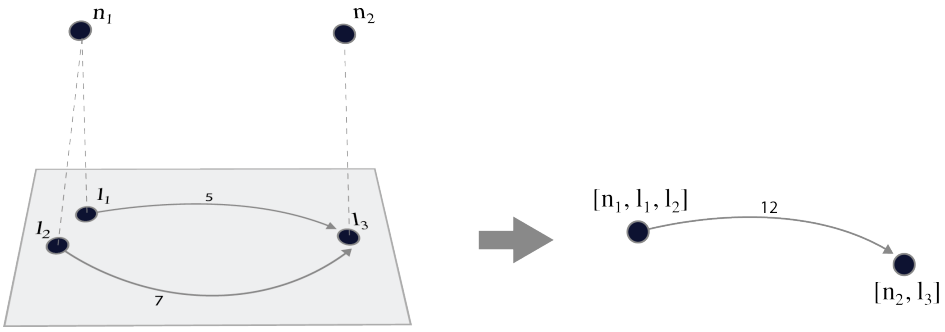



**Figura 4.4.1:** Esempio di un join, dove un nodo del livello passato come secondo argomento viene utilizzato da più nodi del livello passato come primo argomento.

**Esempio 4.4.1.** Come possiamo notare dalla parte sinistra della Figura 4.4.1, abbiamo due Network Level, dove il primo è composto dai nodi $n_1$ e $n_2$, mentre il secondo è composto dai nodi $l_1$, $l_2$ e $l_2$. L'accoppiamento tra i due livelli viene descritto invece graficamente attraverso le linee tratteggiate (i.e. il nodo $n_1$ è associato ai nodi $l_1$ e $l_2$, mentre $n_2$ solamente con $l_3$). La funzione $f_d$ è definita come quella che esegue la somma dei valori degli archi.

L'operatore di join, chiamato per unire questi due network level, rispettando l'accoppiamento descritto, fornirà come risultato il Network Level illustrato nella parte destra della Figura 4.4.1. Analizziamo però adesso $M_h$, che l'algoritmo utilizza per creare il risultato.

L'algoritmo di join, analizzando il nodo $n_1$, creerà un nuovo nodo $x$, che nella Figura 4.4.1 abbiamo identificato con $[n_1, l_1, l_2]$ ed assegna come $M_h(n_1)$ proprio questo nodo. In seguito l'algoritmo crea l'insieme $C_M(n_1)$ costituito dai nodi $l_1$ e $l_2$; quindi passa all'analisi di ognuno di questi. Sia per $l_1$ che per $l_2$ viene assegnato lo stesso valore, ovvero $M_h(l_1) = \{[n_1, l_1, l_2]\}$ e $M_h(l_2) = \{[n_1, l_1, l_2]\}$. In altre parole l'informazione che ci stiamo memorizzando con $M_h$ è che $n_1$ (come anche $l_1$ ed $l_2$) è stato utilizzato per creare il nuovo nodo, e che quindi i nuovi archi devono tenere conto di questo fatto. Lo stesso viene fatto per il nodo $n_2$, dove avremo $M_h(n_2) = M_h(l_3) = \{[n_2, l_3]\}$.

Infatti, durante l'ArcExp, come viene utilizzato $M_h$? In questa procedura vengono prima di tutto analizzati tutti gli archi dei Network Level di partenza. Quello che ci

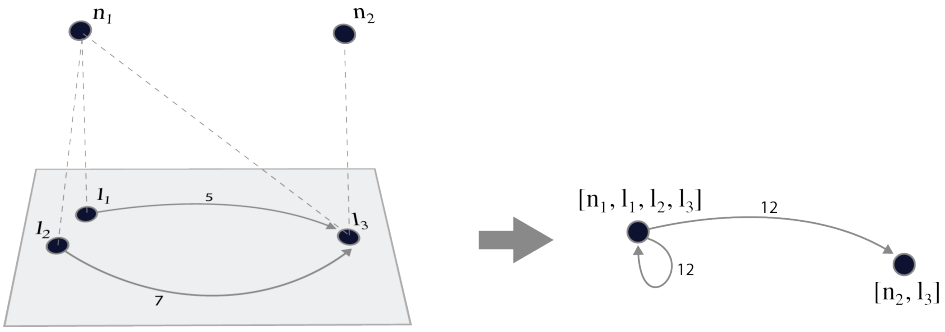



**Figura 4.4.2:** Esempio di un join, dove un nodo del livello passato come secondo argomento viene utilizzato da più nodi del livello passato come primo argomento. I livelli di partenza sono identici a quelli in Figura 4.4.1, ma ciò che cambiano sono gli accoppiamenti.

interessa è il secondo livello, dove abbiamo gli archi $(l_1, l3)$ e $(l_2, l_3)$, corrispettivamente con i valori 5 e 7. Per ogni vertice di questi archi la procedura analizza ogni nodo di $M_h$. In questo caso ogni vertice ha solamente un nodo in $M_h$, poiché ognuno ha contribuito per la formazione di un solo nodo del risultato. Durante l'analisi dell'arco $(l_1, l_3)$ viene pertanto aggiunto all'insieme degli archi del risultato l'arco $([n_1, l_1, l_2], [n_2, l_3])$ e gli viene dato il valore 5. Durante l'analisi dell'arco $(l_2, l_3)$ invece viene aggiunto nuovamente lo stesso arco (poiché $M_h(l_2) = M_h(l_1)$), ma ricordiamo che tale aggiunta non ha in realtà effetto perché l'arco era appena stato aggiunto nell'insieme. Ciò che cambia è invece il valore del nodo, che da 5 diventa 12.

Quest'ultimo esempio mostra come viene utilizzato $M_h$ per determinare i nuovi archi, ma possiamo fare un ulteriore esempio per capire meglio, ovvero quando in una particolare entry di $M_h$ abbiamo più elementi (i.e. quando un nodo contribuisce per la creazione di più nodi).

**Esempio 4.4.2.** Come possiamo osservare dalla Figura 4.4.2, abbiamo un input che è del tutto identico a quello della Figura 4.4.1, con l'unica differenza che l'accoppiamento cambia. Il nodo $n_1$ è accoppiato anche con il nodo $l_3$ (che a sua volta è accoppiato anche con $n_2$).

Ciò cambia ovviamente anche il risultato, ma in particolar modo cambia $M_h$. Infatti, il nodo $l_3$ non solo contribuisce alla formazione del nodo $[n_1, l_1, l_2, l_3]$ ma anche di $[n_2, l_3]$. Ciò significa che $M_h(l_3)$ questa volta sarà $\{[n_1, l_1, l_2, l_3], [n_2, l_3]\}$.

La conseguenza di ciò si nota nell'ArcExp, dove nell'analisi dell'arco $(l_1, l_3)$, si creano questa volta due archi: $([n_1, l_1, l_2, l_3], [n_2, l_3])$ e $([n_1, l_1, l_2, l_3], [n_1, l_1, l_2, l_3])$. Lo stesso accade nell'analisi dell'arco $(l_2, l_3)$.

In generale possiamo quindi notare come gli accoppiamenti possono cambiare il risultato (cosa che avevamo già notato parlando della definizione del join), ma anche come l'algoritmo gestisce tali situazioni, utilizzando $M_h$, ed infine generando il risultato atteso.

OTTIMIZZAZIONI PER $M_h$ Come abbiamo detto $M_h$ può essere una hashmap, dove per un nodo $n$ viene associato un insieme di nodi. Preme dire comunque che, in maniera dipendente dall'implementazione effettiva del sistema che memorizza i dati, potrebbe essere conveniente anche non utilizzare un'hashmap per questo scopo, in modo da ottimizzare questa parte dell'algoritmo.

Ad esempio, potrebbe essere plausibile che la reale implementazione del sistema, dato $n$ che identifica univocamente il nodo $n$ presente in un certo Network Level, possa trovare il corrispettivo nodo (con i suoi relativi dati) in maniera istantanea (si pensi ad esempio agli usuali puntatori in memoria centrale).

Perciò potrebbe essere utile non creare una vera hashmap $M_h$, ma memorizzare tali informazioni direttamente nei nodi dei Network Level coinvolti. Ad esempio, considerando strutturato il dato di un nodo, possiamo recuperare l'insieme che nell'hashmap indichiamo con $M_h(n)$, prendendo il dato $d(n).mh$, dove $d$ è la classica funzione associata al Network Level che ci permette di recuperare il dato di un certo nodo.

Ciò potrebbe essere vantaggioso in termini di memoria e di velocità, ma ci sono ulteriori dettagli, che dipendono dall'effettiva implementazione, che devono essere controllati. Ad esempio la cancellazione di questi dati dai nodi, una volta che l'operazione di join viene conclusa, o l'impossibilità in questo modo di poter realizzare un sistema concorrente che possa fare più join contemporaneamente.

### 4.4.2 PROPRIETÀ

#### 4.4.2.1 ASIMMETRIA DELL'OPERATORE

Come abbiamo già notato, questo operatore di join non è simmetrico. Abbiamo visto come in alcuni esempi l'inversione degli operandi (i.e. $B \bowtie A$ anziché $A \bowtie B$) avrebbe generato risultati differenti. Adesso cercheremo di capire un po' meglio questa asimmetria, con un semplice esempio.

**Esempio 4.4.3.** Supponiamo di avere due Network Level $A = (N_A, A_A, v_A, d_A)$ e $B = (N_B, A_B, v_B, d_B)$. Il livello $A$ ha due nodi (i.e. $\{n_1, n_2\}$), mentre Il livello $B$ ne ha quattro (i.e. $\{v_1, v_2, v_3, v_4\}$). Questi sono associati come in figura, ovvero, se $C = (A, B, M, w)$ è il Couple che li associa, $M = \{(n_1, v_2), (v_2, n_1), (n_1, v_1), (v_1, n_1), (n_1, v_3), (v_3, n_1), (n_2, v_1), (v_1, n_2), (n_2, v3), (v_3, n_2), (n_2, v_4), (v_4, n_2)\}$.

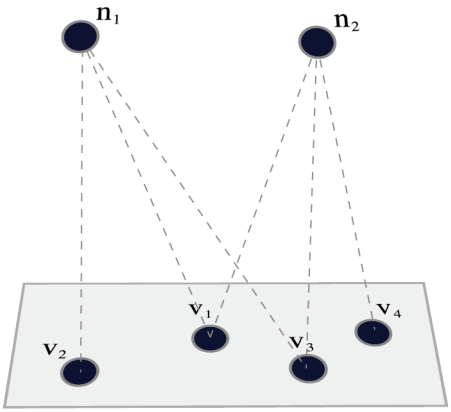



**Figura 4.4.3:** Network Level $A$ e $B$ dell'esempio 4.4.3.

Provando a fare un'operazione di join di $A$ con $B$ (i.e. $A \bowtie B$), notiamo che il nodo $n_1$ viene fuso con $v_1$, $v_2$ e $v_3$; mentre $n_2$ viene fuso con $v_1$, $v_3$ e $v_4$. Con questa osservazione possiamo subito parlare di una prima osservazione, ovvero i nodi del secondo operando possono essere coinvolti più volte, come nel caso di $v_2$ e $v_3$.

Provando invece a fare l'operazione inversa, cioè $B \bowtie A$, notiamo che sono i nodi del livello sottostante che vengono fusi con quelli del livello soprastante. Ovvero, $v_2$ viene fuso con $n_1$, $v_1$ con $n_1$ e $n_2$, $v_3$ con $n_2$ e $n_1$, ed infine $v_4$ con $n_2$. Il Network Level risultante è quindi anche abbastanza differente, nonostante i dati dei nodi e degli archi potrebbero essere identici. Nel primo caso infatti abbiamo un Network Level di due nodi, che possiamo identificare con la notazione utilizzata in Figura 3.8.1: $[n_1, v_1, v_2, v_3]$ e $[n_2, v_1, v_3, v_4]$. Mentre nel secondo caso, abbiamo quattro nodi, che con la medesima notazione sono: $[v_2, n_1]$, $[v_1, n_1, n_2]$, $[v_3, n_1, n_2]$ e $[v_4, n_2]$.

Dopo aver analizzato l'esempio precedente, possiamo formalizzare l'osservazione fatta, poiché è importante tenerne conto, per trarne vantaggio, ma anche per capire meglio il funzionamento di questo operatore.

**Proposizione 4.4.1.** Come in definizione 3.8.1, $\forall n \in N_1$, indichiamo l'insieme $C_M(n) = (n_1, n_2, ..., n_k)$ come l'insieme dei nodi di $G_2$ che sono associati ad $n$, secondo il mapping $M$ tra i nodi dei due livelli.

Vale che $\exists G1, G2, C$ dove: $G_1 = (N_1, A_1, v_1, d_1)$ e $G_2 = (N_2, A_2, v_2, d_2)$ sono due Network Level, e dove $C = (G_1, G_2, M, w)$ è un accoppiamento (i.e. Couple) tra $G_1$ e $G_2$. In modo che valga la seguente proposizione:

$$\exists n, n' \text{ tale che } C_M(n) \cap C_M(n') \neq \emptyset$$

# CAPITOLO 5

# IMPLEMENTAZIONE

Nei capitoli precedenti abbiamo definito il modello dei dati che permette di rappresentare i grafi multi-livello, cercando di essere sufficientemente espressivi da poter rappresentare tutti i modelli presi in esame che riguardassero questo argomento. In seguito abbiamo definito un'algebra, definendo i vari operatori per l'estrazione e la manipolazione di dati all'interno di questi grafi multi-livello. Infine abbiamo anche potuto approfondire meglio i dettagli implementativi di ciascun operatore, fornendo anche degli algoritmi ad alto livello, e cercando di trovare delle proprietà che potessero essere sfruttate per fare delle ottimizzazioni.

In questo capitolo parleremo della fase di implementazione, costruita solo grazie ai passi fatti in precedenza, di cui abbiamo appena fatto un breve riassunto. Tale implementazione è comunque da considerarsi un prototipo, certamente non un prodotto finito che possa essere utilizzato per scopi commerciali. Nasce soprattutto con l'idea di poter verificare le idee proposte precedentemente:

- Verificando la correttezza degli algoritmi che sono stati illustrati.
- Verificando il modo con cui possono essere utilizzati gli operatori per produrre determinati risultati.

- Verificando la correttezza delle definizioni del modello e degli operatori, nello specifico l'interazione tra il modello e gli operatori e l'interazione tra gli operatori.
- Verificando alcune delle ottimizzazioni che sono state proposte per gli algoritmi.

Nasce anche con l'intento di poter fornire una visione ancora più dettagliata del modello e degli operatori, con delle definizioni precise del modello, utilizzando i costrutti di un reale linguaggio di programmazione, e con gli algoritmi che implementano gli operatori, eliminando tutte le possibili ambiguità.

Il prototipo, di circa 3500 righe di codice, è stato scritto utilizzando il linguaggio Java. Quest'ultimo è stato scelto per via dei numerosi dettagliati costrutti che aiutano a definire strutture dati complesse senza lasciare spazio ad ambiguità, per via del complesso sistema di auto-generazione della documentazione, ed infine per via dell'ormai nota popolarità del linguaggio che potrà permettere in futuro una più semplice integrazione con altri sistemi.

Il progetto viene mantenuto su un repository *git*, ed è reperibile su *Github* al seguente indirizzo:

`https://github.com/miromannino/master-thesis-prototype`

## 5.1 STRUTTURA DEL PROTOTIPO

Per avere una visione schematizzata del prototipo, descriveremo sinteticamente la struttura del prototipo. Ciò può essere molto utile per chi volesse cominciare a collaborare al progetto, e necessita delle informazioni generali per poterlo comprendere.

La struttura del progetto è composta da tre parti distinte, che vengono memorizzate in tre directory separate:

**src** codice sorgente del prototipo.

**test** codice sorgente dei test che vengono effettuati sul prototipo.

**out** output dei test. In particolare file *dot* (*Graphviz dot* [56]) che descrivono i grafi generati dai test. Contiene anche un particolare script chiamato `dotToPng.sh` che crea automaticamente le immagini *png* a partire dai file *dot*.

### 5.1.1 SRC

La parte relativa al codice sorgente del prototipo, viene memorizzata all'interno del package `it.miromannino.multilevelnetwork`, e si compone a sua volta di quattro parti distinte, memorizzate in package separati.

**generic** contiene tutte le classi generiche che non vengono fornite dalla libreria standard di Java, ma che sono utili per il progetto. Un esempio è il `MutableInt`.

**inout** contiene tutte le classi relative all'importazione e l'esportazione di dati, come ad esempio NetworkLevelDotExport che esporta un Network Level nel formato Graphviz dot [56], oppure PathPatternAutomatonDotExport che esporta un automa relativo ad un path pattern nel formato Graphviz dot.

**model** contiene tutte le classi necessarie a definire il modello.

**operator** contiene tutte le classi che definiscono gli operatori, tutte le definizioni delle callback necessarie per costruire predicati e gli indici.

#### 5.1.1.1 PACKAGE: MODEL

La parte relativa al modello, che definisce le strutture fondamentali per poter rappresentare i Network Level, contiene le seguenti classi:

`Arc` classe che definisce un arco, con il nodo di partenza `fromNode`, il nodo di arrivo `toNode` ed il dato associato `data`. Detiene tutti i metodi per poter recuperare a manipolare tali informazioni, per comparare gli archi, e per generare un hashcode (utile per la memorizzazione degli archi in un `HashMap`).

`ArcIterator` il complesso iteratore che enumera tutti gli archi di un Network Level (con la possibilità di eliminazione dell'elemento corrente), partendo da una rappresentazione completamente differente fatta di nodi e link.

`Couple` classe che definisce un Couple, che mantiene una lista di associazioni (definiti come `CoupleLink`) che partono da un livello e finiscono in un altro.

`CoupleLink` classe che definisce un link di un Couple, con un nodo di partenza e uno di arrivo, ed il dato associato.

`LevelsCoupling` classe che definisce un Levels Coupling, che memorizza una lista di Couple.

`Link` classe che definisce un arco in uscita, memorizzato da un nodo. Mantiene l'informazione sul nodo di arrivo e il dato associato all'arco, e detiene tutti i metodi per poter recuperare a manipolare tali informazioni, per comparare i link, e per generare un hashcode (utile per la memorizzazione dei link in un `HashMap`).

`NetworkLevel` classe che definisce un Network Level, mantenendo un identificativo e un insieme di nodi memorizzati su una mappa. Tale classe fornisce i metodi per aggiungere, rimuovere, ed iterare nodi ed archi. Si ricorda che l'aggiunta o la rimozione degli archi, come anche la loro iterazione, è virtuale e mappata sulla reale memorizzazione, fatta di `Node` e `Link`.

`Node` classe che definisce un nodo. Memorizza un identificativo, un dato associato, ed una mappa di link (i.e. archi in uscita). Contiene anche tutti i metodi per manipolare e recuperare tali informazioni, per iterare gli archi in uscita, e per generare un hashcode (utile per la memorizzazione dei nodi in un `HashMap`).

`Path` classe che definisce un path. Memorizza al suo interno una lista di nodi e detiene i metodi per recuperare e manipolare tali nodi, per conoscere la lunghezza della path, per comparare due path e per generare un hashcode (utile per la memorizzazione delle path in un `HashMap`).

`PathSet` classe che definisce un insieme di path, con i relativi metodi per la manipolazione ed il recupero di tali path, per la loro iterazione e per determinare velocemente l'appartenenza di una path a tale insieme.

#### 5.1.1.2 PACKAGE: OPERATOR

`pathpattern` package che definisce tutte le classi utili per la costruzione di un path pattern e la costruzione del relativo automa.

`Automaton` classe che definisce un automa relativo ad un path pattern. Il suo costruttore prende come parametro un path pattern ed ha l'incarico di generare i nodi e i link dell'automa. Tiene traccia solamente del nodo iniziale e finale, visto che gli altri nodi possono essere recuperati navigando l'automa.

`AutomatonLink` classe che definisce un link tra un nodo dell'automa ed un altro. Memorizza al suo interno anche l'azione necessaria per eseguire la transizione.

`AutomatonLinkActionType` enum che definisce il tipo dell'azione: transizione epsilon, nodo generico o nodo specifico.

`AutomatonNode` classe che definisce un nodo dell'automa. Memorizza una lista di AutomatonLink verso i vicini.

`PathPattern` classe astratta che contiene tutte le classi specifiche dei path pattern: `EmptyPath`, `GraphNode`, `GenericNode`, `OptionalGenericNode`, `GenericPath`, `Concatenation`, `Alternation`.

`Aggregation` definisce il metodo che implementa l'operatore di aggregazione. Al suo interno è anche definita la classe astratta `CFunction` e la classe astratta `Assignements`, utili per definire le funzioni di callback necessarie per richiamare tale operatore.

`Concatenation` definisce il metodo che implementa l'operatore di concatenazione.

`Join` definisce i metodi che implementano l'operatore di join. Al suo interno è anche definita la classe astratta `DataJoin`, utile per definire la funzione di callback necessaria per richiamare tale operatore.

`Pathcondition` definisce la classe astratta per costruire un predicato su un path, utile in maniera totalmente esclusiva per l'operatore di selezione. Al suo interno sono state definite delle path condition concrete come: `TruePredicate` che ritorna sempre `True`, oppure `SpecificLengthPredicate` che definisce in maniera efficiente il predicato che controlla la lunghezza di una path.

`Projection` definisce i metodi che implementano l'operatore di proiezione. Al suo interno è anche definita la classe astratta `Index`, utile per definire gli indici necessari per richiamare tale operatore. Viene anche definita la classe concreta `ConstantIndex` che realizza un indice costante.

`Selection` definisce i metodi che implementano l'operatore di selezione.

`Synthetis` definisce i metodi che implementano l'operatore di sintesi.

### 5.1.2 TEST

I test, realizzati mediante l'uso di JUnit, mirano a verificare la correttezza dell'implementazione del modello e degli operatori. Questi test sono organizzati per categorie, ognuna racchiusa in una delle seguenti classi:

`AggregationTest` definisce i Network Level opportuni ed effettua dei test mirati a verificare la correttezza dell'operatore di aggregazione.

`ConcatenationTest` definisce delle path e degli insiemi di path opportuni ed effettua dei test mirati a verificare la correttezza dell'operatore di concatenazione.

`DotNetworkLevelExportTest` definisce dei Network Level e ne effettua la loro esportazione.

`JoinTest` definisce i Network Level opportuni ed effettua dei test mirati a verificare la correttezza dell'operatore di join.

`ModelTest` definisce dei Network Level, nodi ed archi, popolandoli anche con determinati dati, e ne verifica il corretto comportamento.

`PathPatternTest` definisce dei path pattern opportuni e verifica la corretta creazione dei relativi automi.

`ProjectionTest` definisce delle path e degli insiemi di path opportuni ed effettua dei test mirati a verificare la correttezza dell'operatore di proiezione.

`SelectionTest` definisce i Network Level opportuni ed effettua dei test mirati a verificare la correttezza dell'operatore di selezione.

`SynthetisTest` definisce i Network Level opportuni, effettua delle selezioni su di essi ed effettua dei test mirati a verificare la correttezza dell'operatore di sintesi.

## 5.2 MODELLO

Il modello del Network Level è stato rappresentato utilizzando le classi di Java. Per intendere meglio questo concetto un Network Level, un nodo, e un arco, vengono rappresentati rispettivamente con le classi `NetworkLevel`, `Node` e `Arc`. Per gli scopi per il quale è stato pensato il prototipo, tali classi vengono istanziate in memoria centrale, lo stesso luogo dove gli operatori manipolano tali oggetti. Ciò non significa che il passo da fare per avere un'implementazione che manipoli dati in memoria secondaria (come succede nei DBMS commerciali) sia molto lontana, poiché sappiamo che Java stesso fornisce dei meccanismi per la persistenza di oggetti in memoria secondaria. L'esempio più nodo in questo senso sono le Java Persistence API [57].

Adesso cominceremo ad elencare tutte le varie classi definite per rappresentare il modello dei dati.

### 5.2.1 NODE

Un oggetto di tipo `Node` rappresenta un generico nodo di un Network Level. Questo viene identificato con una stringa alfanumerica (e.g. `'a'`, oppure `'node1'`), in modo che possa essere rappresentato graficamente sia all'interno del grafo stesso che all'interno di un path. Questo può anche contenere un dato qualsiasi, e non viene imposta nessuna limitazione sul suo tipo, neanche in relazione agli altri nodi (e.g. un nodo può avere come dato un numero, mentre un altro può avere come dato un documento XML).

Notiamo già una prima piccola differenza rispetto alle definizioni che abbiamo dato precedentemente. Il dato del nodo viene memorizzato all'interno del nodo, e viene acceduto utilizzando lo stesso nodo (e.g. per avere il dato del nodo `n` faremo `n.getData()`), al contrario di come veniva fatto nelle definizioni del Network Level, dove per accedere al dato del nodo $n$, contenuto nel Network Level $G = (N, A, \nu, d)$,

veniva fatto $d(n)$. Ciò è stato fatto per semplificare, poiché in questo modo l'informazione, che appartiene al nodo, viene memorizzata e gestita dal nodo stesso, piuttosto che dal Network Level al quale appartiene. Conseguenza di ciò è anche un guadagno in termini di tempo e spazio.

Un'altra differenza è la memorizzazione degli archi. Piuttosto che mantenere l'insieme degli archi presenti nel Network Level in un insieme, è il nodo stesso a mantenerli. In particolare ogni nodo mantiene un insieme di archi uscenti, chiamati `Link`. Quest'ultimo mantiene due informazioni: il nodo che viene puntato (chiamato anche nodo di destinazione), e il dato stesso dell'arco. Anche in quest'ultimo caso possiamo notare come le informazioni che venivano memorizzate all'interno del Network Level vengono memorizzate all'interno dei nodi. In questo caso il motivo, ancora più evidentemente, deriva dal voler ottimizzare gli algoritmi degli operatori. Infatti ad esempio, per trovare tutti i vicini di un nodo $n$, si sarebbe dovuto cercare nell'insieme degli archi del Network Level tutti quelli con $n$ come nodo di partenza.

Riassumendo, la classe `Node` può essere descritta in maniera sintetica nel seguente modo:

```
class Node {
  String id;
  Object data;
  Map<Node, Link> links;

  Node(String id, Object data, Link[] links) { /* ... */ }
  Node(String id, Object data) { /* ... */ }
  Node(String id) { /* ... */ }

  Object getData() { /* ... */ }
  void setData(Object data) { /* ... */ }
  boolean addNeighbor(Node n) { /* ... */ }
  boolean addNeighbor(Node n, Object data) { /* ... */ }
```

```
  Link removeNeighbor(Node n) { /* ... */ }
  boolean hasNeighbor(Node n) { /* ... */ }
  Object getNeighborLinkData(Node n) { /* ... */ }
  boolean setNeighborLinkData(Node n, Object data) { /* ... */ }
  Iterator<Link> getLinksIterator() { /* ... */ }
  Iterator<Node> getNeighborIterator() { /* ... */ }
  /* altri metodi meno importanti ... */
}
```

Possiamo osservare come i vari link vengano memorizzati in una mappa, dove il link memorizzato in corrispondenza della chiave $k$ ha come nodo puntato proprio $k$: in questo modo è possibile recuperare velocemente il link specifico che punta ad un particolare nodo $k$ in tempo $O(1)$ in media.

Il link viene definito nella classe `Link`, che può essere descritta in maniera sintetica nel seguente modo:

```
class Link {
  Node destinationNode;
  Object data;

  Link(Node destinationNode, Object data) { /* ... */ }
  Link(Node destinationNode) { /* ... */ }

  public Object getData() { /* ... */ }
  public void setData(Object data) { /* ... */ }
  public Node getDestinationNode() { /* ... */ }
  /* altri metodi meno importanti ... */
}
```

### 5.2.2 NetworkLevel

Un generico Network Level viene identificato con una stringa alfanumerica (e.g. `'nl'`, oppure `'networkLevel1'`), in modo che possa essere rappresentato graficamente. Questo non memorizza nient'altro che un insieme di nodi, ma fornisce all'esterno una visione simile a quella della definizione di Network Level che è stata data precedentemente. In particolare può fornire l'insieme dei nodi presenti, l'insieme degli archi (si utilizza un'apposita classe `Arc`), e quindi anche i relativi dati dei nodi e degli archi.

```
class NetworkLevel {
  String id;
  Map<String, Node> N;

  NetworkLevel(String id) { /* ... */ }

  String getId() { /* ... */ }
  Node addNewNode(String id, Object data) { /* ... */ }
  Node addNewNode(String id) { /* ... */ }
  boolean addNewArc(Node fromNode, Node toNode, Object data) { /* ... */ }
  boolean addNewArc(Node fromNode, Node toNode) { /* ... */ }
  boolean containsNode(Node n) { /* ... */ }
  boolean containsArc(Node fromNode, Node toNode) { /* ... */ }
  Iterator<Node> getNodeIterator() { /* ... */ }
  ArcIterator getArcIterator() { /* ... */ }
  /* altri metodi meno importanti ... */
}
```

È possibile notare dai metodi come possono essere inseriti dei nuovi nodi (tale metodo restituisce un riferimento al nodo che è stato creato ed inserito), e nuovi archi. In quest'ultimo caso viene creato l'arco prendendo il primo nodo ed aggiungendo un

`Link` verso il secondo. In seguito, oggetti complessi come l'`ArcIterator` aiutano poi ad avere una visione simile a quella delle definizioni del Network Level che sono state date, poiché estrapola i link dei vari nodi fornendo la visione di un insieme di archi. L'uso degli iteratori aiuta infatti ad avere una visione completa delle componenti del Network Level, senza dover fare copie o trasformazioni, dando anche la possibilità, mentre si itera, di poter fare delle manipolazioni.

### 5.2.3 PATH

Il path viene rappresentato come una lista di nodi, dove i vari metodi per la lunghezza e la posizione dei nodi sono state rivisitate per soddisfare le definizioni che sono state date. Per prima cosa un path può essere creato vuoto, oppure inizializzato con una serie di nodi. In seguito sappiamo che il path $(a \to b \to c)$ ha lunghezza 2, e per questo motivo il metodo `getLength()` ha una semantica un po' più inusuale rispetto a quello che potremmo aspettarci. Abbiamo poi i metodi per ottenere o modificare un nodo che si trova in una determinata posizione, e quelli per aggiungere o rimuovere nodi in fondo al path (utilizzati pesantemente dall'operatore di selezione). Anche il path fornisce degli iteratori, utili per poter scorrere i nodi presenti senza dover fare delle copie, con la possibilità di poter modificare anche il path stesso.

```
class Path implements Iterable<Node> {
  List<Node> nodes;

  Path(Node[] nodes) { /* ... */ }
  public Path(int initialCapacity) { /* ... */ }
  public Path() { /* ... */ }
  public Path(Path path) { /* ... */ }

  Node getNodeAtPosition(int pos) { /* ... */ }
  void setNodeAtPosition(int pos, Node n) { /* ... */ }
```

```
  void appendNode(Node n) { /* ... */ }
  Node removeLastNode() { /* ... */ }
  boolean contains(Node n) { /* ... */ }
  int getLength() { /* ... */ }
  @Override Iterator<Node> iterator() { /* ... */ }
  /* altri metodi meno importanti ... */
}
```

### 5.2.4 PathSet

Allo stesso modo, è stato necessario rappresentare insiemi di path, creando un'apposita classe `PathSet`. Tale classe è molto semplice, ed espone i classici metodi che possono essere trovati nelle implementazioni dei `Set` di Java, come `containsPath()`, `addPath()`, `removePath()`, e `getSize()`.

### 5.2.5 LevelsCoupling e Couple

Per rappresentare i Levels Coupling è stata creata un apposita classe che, allo stesso modo della definizione, mantiene una lista di Couple. Anche per quest'ultimo è stata creata una classe, che ha il compito di mantenere una quadrupla, come specificato nella sua definizione: il Network Level di partenza, quello di arrivo, un insieme di coppie di nodi (dove il primo nodo della coppia appartiene al Network Level di partenza, ed il secondo a quello di arrivo), e infine una serie di valori associati ad ogni coppia. Ogni coppia, con il corrispettivo valore associato, vengono rappresentati con la classe `CoupleLink`, di cui però non descriviamo i dettagli.

## 5.3 OPERATORI

### 5.3.1 PROIEZIONE

L'operatore più semplice è quello della proiezione. Per via della natura polimorfa dell'operatore, vengono definiti i seguenti metodi:

```
PathSet project(Index s, Index e, PathSet p);
PathSet project(Index pos, PathSet p);
Path project(Index s, Index e, Path p);
Path project(Index pos, Path p);
```

Il primo metodo esegue l'operatore di proiezione di un insieme di path, mentre il terzo esegue l'operatore di proiezione di un singolo path (da notare che il singolo path non equivale ad un insieme con un solo path), entrambi con un indice di inizio `s` e uno di fine `e`. Il secondo ed il quarto metodo invece effettuano rispettivamente le stesse operazioni del primo e del terzo, ma con un indice di inizio e di fine identico, chiamato `pos`. Questi ultimi sono metodi di comodo, esattamente come viene fatto nelle definizioni dell'operatore di proiezione che abbiamo dato precedentemente.

Per fornire un indice sappiamo che è in realtà necessario fornire una funzione $f(p)$ tale che, preso un path $p$, venga restituito un numero, ovvero l'indice può cambiare in base al path che viene analizzato. Per questo motivo un `Index` è in realtà una classe astratta definita nel modo seguente:

```
abstract class Index {
  abstract int get(Path p);
}
```

Definendo una classe derivata è possibile definire l'indice che si vuole. Esistono comunque già classi derivate predefinite, come `ConstantIndex`, che fornisce sempre lo stesso indice per un qualunque path.

### 5.3.2 SELEZIONE

L'operatore di selezione è senza dubbio quello più complesso che è stato definito. La sua implementazione è pertanto altrettanto complessa, ma cercheremo comunque di fornirne una descrizione generale. Per prima cosa l'operatore è definito nel seguente modo:

```
PathSet select(NetworkLevel nl, PathPattern pattern, PathCondition pc);
```

Viene preso come input il Network Level sul quale viene fatta l'operazione, un Path Pattern ed un predicato che stabilisce se un path, conforme ad un certo pattern, è da ritenere valido o meno. L'operatore restituisce poi un `PathSet`, che costituiscono il risultato, esattamente come è stato definito dalle definizioni date in precedenza.

#### 5.3.2.1 PATHCONDITION

La path condition è il predicato che stabilisce se un path conforme al pattern è da ritenere valido o meno. Tale predicato riceve quindi in input un path, e può analizzarlo maggiormente, ispezionando cose come la lunghezza, i dati dei nodi o i dati degli archi. Il predicato è stato implementato senza l'utilizzo di linguaggi particolari che permettessero la scrittura di formule logiche, ma è stato implementato come una funzione di callback, come è stato fatto per gli indici dell'operatore di proiezione. La definizione di PathCondition è infatti la seguente:

```
abstract class PathCondition {
  enum ConditionResult {
    True,
    False,
    DefinitelyFalse
  }
```

```
  abstract ConditionResult predicate(Path p);
}
```

Il risultato del predicato, come possiamo osservare, non restituisce solamente 'vero' o 'falso', ma anche un particolare tipo di valore negativo. Quest'ultimo viene chiamato `DefinitelyFalse` e significa che non solo il predicato è falso per il path $p$, ma che lo sarà per un qualunque path $p'$, ottenuto espandendo il path $p$. Un esempio è il predicato che è vero soltanto quando il path è lunga $n$. In questo caso il predicato dovrebbe restituire `False` con path di lunghezza minore di $n$, `True` con path di lunghezza uguale ad $n$, e `DefinitelyFalse` con path di lunghezza maggiore a $n$. L'uso di quest'ultimo valore negativo è comunque da considerarsi un'ottimizzazione e potrebbe anche non essere utilizzato: fa parte delle ottimizzazioni di Branch and Bound sull'operatore di selezione di cui abbiamo parlato nel precedente capitolo.

Come per gli indici dell'operatore di proiezione, il `PathCondition` ha delle implementazioni di semplici predicati, come quello di cui abbiamo appena accennato nell'esempio, che controlla che il path sia lungo esattamente $n$.

#### 5.3.2.2 PathPattern

Un path pattern viene costruito attraverso le sottoclassi di `PathPattern`. Abbiamo infatti:

- `EmptyPath` che identifica il pattern di un path vuoto, che abbiamo indicato precedentemente con il simbolo $\emptyset$.
- `GraphNode` che identifica il pattern di un nodo specifico.
- `GenericNode` che identifica il pattern di un generico nodo, che abbiamo indicato precedentemente con il simbolo %.

- `OptionalGenericNode` che identifica il pattern di un generico nodo opzionale, che abbiamo indicato precedentemente con il simbolo ?.
- `GenericPath` che identifica il pattern di una generico path, ricordiamo può anche essere vuoto. Indicata precedentemente con il simbolo $*$.
- `Concatenation` che identifica il pattern di una concatenazione di un path pattern $p_1$ con un altro $p_2$. Indicata precedentemente con $p_1 \rightarrow p_2$.
- `Alternation` che identifica il pattern di una alternazione di un path pattern $p_1$ con un altro $p_2$. Indicata precedentemente con $p_1|p_2$.

Per specificare un path pattern non è stato implementato nessun linguaggio, con il relativo parser. La costruzione avviene invece utilizzando i costruttori di tali classi, lo stesso lavoro che farebbe un parser. Se ad esempio vogliamo costruire il path pattern $(a|b) \rightarrow \%$, utilizzeremo i costruttori nel seguente modo:

```
PathPattern p = new PathPattern.Concatenation(
          new PathPattern.Alternation(
            new PathPattern.GraphNode(a),
            new PathPattern.GraphNode(b)
          ),
          new PathPattern.GenericNode()
        );
```

#### 5.3.2.3 Automaton

La classe `Automaton` rappresenta l'automa di un path pattern, ma ha il compito della costruzione dell'automa. Infatti il costruttore di tale classe prende in input un path pattern.

`Automaton`, una volta costruito l'automa, memorizza solamente il riferimento al nodo iniziale. Ogni nodo dell'automa viene definito con la classe `AutomatonNode`, e

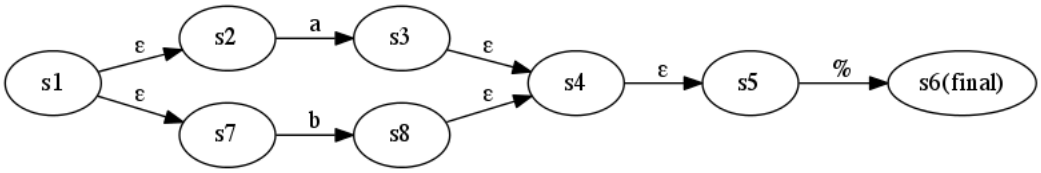



**Figura 5.3.1:** Rappresentazione grafica dell'automa del path pattern $(a|b) \rightarrow \%$. Il grafico è stato creato utilizzando *Graphviz dot* [56], mentre il file *dot* è stato creato utilizzando la classe `PathPatternAutomatonDotExport`, appositamente implementato per questo scopo.

memorizza al suo interno una lista di `AutomatonLink`, che definiscono i vicini che è possibile raggiungere, oltre che l'azione da compiere per poterli raggiungere. Un `AutomatonLink` memorizza infatti tre informazioni: il nodo di destinazione, il tipo di azione da compiere (i.e. `AutomatonLinkActionType`, che può essere un'azione $\epsilon$, un'azione di un nodo generico, o un'azione di un nodo specifico), e un nodo (solamente nel caso in cui il tipo dell'azione sia un nodo specifico).

Facciamo adesso un esempio per capire meglio. Supponiamo di voler creare un automa a partire dal path pattern `p` dell'esempio precedente, facendo `new Automaton(p)`. L'automa risultante è quello mostrato in Figura 5.3.1. Facendo un esempio, il nodo $s2$ ha un `AutomatonLink`, di tipo `SpecificNode`, specificando come nodo $a$, e come nodo di destinazione $s3$.

#### 5.3.2.4 Esempio

Per rendere le cose più concrete, vista la complessità dell'operatore, facciamo un esempio. Supponiamo di avere un Network Level come in Figura 5.3.2.

Vorremmo adesso estrarre dal grafo tutti i modi possibili per raggiungere il nodo $b$, partendo dal nodo $a$. Per fare ciò specifichiamo come path pattern $a \rightarrow * \rightarrow b$, e come path condition il predicato sempre vero.

```
Selection.select(nl1,
  new PathPattern.Concatenation(
```

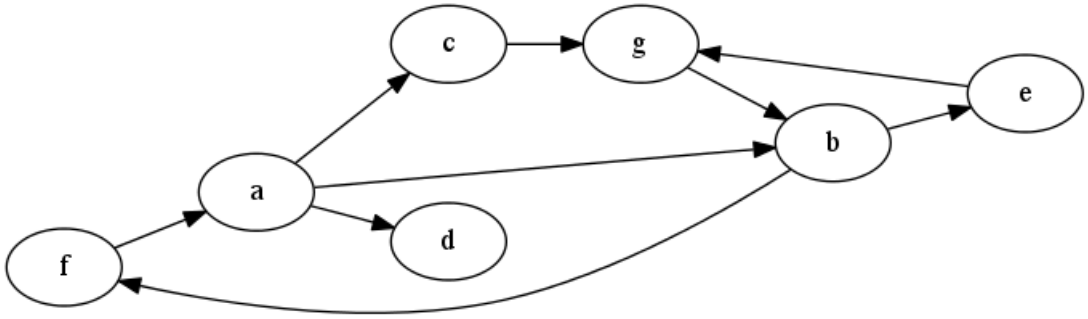



**Figura 5.3.2:** Rappresentazione grafica del Network Level `nl1`. Il grafico è stato creato utilizzando *Graphviz dot* [56], mentre il file *dot* è stato creato utilizzando la classe `NetworkLevelDotExport`, appositamente implementata per questo scopo.

```
    new PathPattern.GraphNode(a),
    new PathPattern.Concatenation(
      new PathPattern.GenericPath(),
      new PathPattern.GraphNode(b)
    )
  ),
  new PathCondition.TruePredicate());
```

Il risultato che otteniamo è l'insieme formato dal path $(a \rightarrow c \rightarrow g \rightarrow b)$ e $(a \rightarrow b)$.

### 5.3.3 AGGREGAZIONE

L'operatore di aggregazione che, dato un insieme di path, restituisce un Network Level, nei modi che sono stati definiti precedentemente, è definito nel modo seguente:

```
NetworkLevel aggregate(PathSet ps,
                       CFunction cFunc,
                       Assignments a,
                       String resultLevelID,
                       String newNodeID);
```

Il primo argomento è l'insieme di path di input che devono essere analizzate. Il secondo argomento è invece una funzione, quella che è stata chiamata $c$, che prende in input un path e restituisce un path (il path aggregato). Il terzo argomento descrive invece la serie di assegnamenti, sotto forma di una funzione di callback. Come quarto argomento viene preso ciò che verrà utilizzato come identificativo del Network Level che costituirà il risultato. Tale argomento non viene preso in considerazione nella definizione degli operatori dati in precedenza, ma è necessario invece nell'implementazione. Analogamente, anche il quinto argomento è necessario solamente nell'implementazione. Questo è un identificativo che costituisce il prefisso di tutti i nuovi nodi che verranno creati (e.g. se il prefisso è 'n' i tre nuovi nodi che verranno creati saranno 'n1', 'n2', 'n3').

La funzione $c$ viene definita con la classe `CFunction`.

```
abstract class CFunction {
  abstract Path cFunction(Path p);
}
```

Come sappiamo, tale funzione può restituire un path, riusando i nodi presenti nel path di input (e.g. data $(a \to b \to c)$ restituisce $(a \to b)$), ma può anche creare nuovi nodi. Per far ciò la funzione può utilizzare un particolare nodo, definito staticamente: `Aggregation.NewNode`. Tale nodo verrà analizzato dall'operatore di aggregazione per creare nuovi nodi, se necessario, rispettando il comportamento dell'operatore (i dettagli di come l'operatore crei tali nodi è comunque omessa, poiché abbastanza complessa).

La funzione incaricata di fare gli assegnamenti viene definita con la classe `Assignments`:

```
abstract class Assignments {
  abstract void changeData(Path aggregatePath, Path currentPath);
}
```

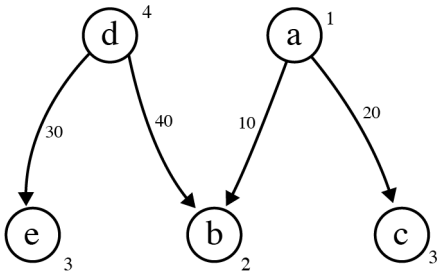



**Figura 5.3.3:** Network Level di input.

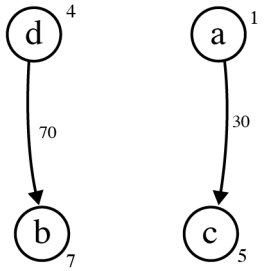



**Figura 5.3.4:** Network Level risultante.

Questa prende un path aggregato (i.e. `aggregatePath`), creato a partire dal path corrente (i.e. `currentPath`), e cambia i dati dei nodi e degli archi del path aggregato in maniera opportuna. Tale comportamento è differente da quello dato nelle definizioni, dove gli assegnamenti erano soltanto una serie di coppie $LV = RV$, e costituisce un metodo alternativo altrettanto potente.

ESEMPIO Proviamo adesso a fare un piccolo esempio per capire meglio l'utilizzo di questo operatore. Supponiamo di avere un Network Level `nl1`, rappresentato in Figura 5.3.3. Possiamo notare come ogni nodo ha un determinato dato (e.g. il nodo $a$ ha il valore 1, mentre $e$ ha il valore 5) ed ogni arco ha un determinato dato (e.g. l'arco $a \rightarrow b$ ha il valore 10, mentre l'arco $d \rightarrow e$ ha il valore 30). Supponiamo poi di avere costruito il seguente insieme di path (che chiamiamo `pathSet`), utilizzando l'operatore di selezione:

$$\{(a \rightarrow b), (a \rightarrow c), (d \rightarrow e), (d \rightarrow b)\}$$

Supponiamo adesso di voler aggregare i vicini dei nodi $a$ e $d$ (quelli in alto nella figura), in modo che un nodo soltanto rappresenti il vicinato. Inoltre, quest'ultimo nodo dovrà avere come dato la somma dei dati dei nodi che rappresenta (e.g. il nodo che rappresenta il vicinato di $a$ dovrà avere il valore 5, poiché $b$ ha valore 2 e $c$ ha valore 3). Infine, anche gli archi devono avere la somma dei valori degli archi che vengono

aggregati (e.g. la somma degli archi uscenti di $a$ è 30, e l'arco tra $a$ e il nodo che rappresenta i vicini di $a$ dovrà avere come valore proprio 30). Riassumendo, vogliamo ottenere il Network Level, come quello rappresentato in Figura 5.3.4.

Per ottenere questo risultato dobbiamo definire la funzione $c(p)$ in modo che restituisca sempre il primo nodo (che può essere soltanto $a$ o $d$) e che restituisca un secondo nuovo nodo. In particolare:

```
Aggregation.CFunction cf = new Aggregation.CFunction() {
  @Override
  public Path cFunction(Path p) {
    return new Path(new Node[] {p.getNodeAtPosition(1), Aggregation.NewNode});
  }
};
```

Gli assegnamenti invece dovranno effettuare la somma degli archi e dei nodi. In particolare:

```
Aggregation.Assignments a = new Aggregation.Assignments() {
  @Override
  public void changeData(Path aggregatePath, Path currentPath) {
    //aggregate function SUM of all the values in the destination node
    if (aggregatePath.getNodeAtPosition(2).getData() == null) {
      aggregatePath.getNodeAtPosition(2).setData(
        currentPath.getNodeAtPosition(2).getData()
      );
    } else {
      aggregatePath.getNodeAtPosition(2).setData(
        (Integer)currentPath.getNodeAtPosition(2).getData() +
        (Integer)aggregatePath.getNodeAtPosition(2).getData()
      );
    }
```

```
    //aggregate function SUM of all the values in the arcs
    if (aggregatePath.getNodeAtPosition(1).getNeighborLinkData(
      aggregatePath.getNodeAtPosition(2)) == null) {
      aggregatePath.getNodeAtPosition(1).setNeighborLinkData(
        aggregatePath.getNodeAtPosition(2),
        currentPath.getNodeAtPosition(1).getNeighborLinkData(
          currentPath.getNodeAtPosition(2)
        )
      );
    } else {
      aggregatePath.getNodeAtPosition(1).setNeighborLinkData(
        aggregatePath.getNodeAtPosition(2),
        (Integer)currentPath.getNodeAtPosition(1).getNeighborLinkData(
          currentPath.getNodeAtPosition(2)
        ) + (Integer)aggregatePath.getNodeAtPosition(1).getNeighborLinkData(
          aggregatePath.getNodeAtPosition(2)
        )
      );
    }

  }
};
```

Infine per aggregare, possiamo richiamare l'operatore di aggregazione nel seguente modo:

```
Aggregation.aggregate("risNl1", pathSet, cf, a, "nn");
```

Questo creerà un nuovo Network Level `risNl1`, dove i nuovi nodi saranno nominati con il prefisso 'nn' (nella Figura 5.3.4 infatti possiamo notare i nodi $nn1$ e $nn2$).

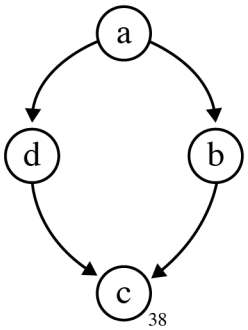



**Figura 5.3.5:** Rappresentazione grafica del Network Level generato a partire dalla sintesi dell'insieme di path `ins1`.

### 5.3.4 SINTESI

L'operatore di sintesi che, dato un insieme di path restituisce un Network Level unendo tutti i vari path, è definito nel modo seguente:

```
NetworkLevel synthesize(String resultLevelID, PathSet ps);
```

Il primo argomento, analogamente a come viene fatto per l'operatore di aggregazione, è l'identificativo che verrà utilizzato per creare il nuovo Network Level che costituirà il risultato. Il secondo argomento è l'insieme di path che dovranno essere elaborati. Da notare che non è necessario avere un riferimento del Network Level dove sono salvati i nodi dei path di input, poiché i dati si possono recuperare direttamente esaminando i nodi stessi.

Per quanto riguarda gli identificativi dei nodi che verranno creati e inseriti nel Network Level che costituisce il risultato, non è necessario specificare un prefisso (come succedeva nell'operatore di selezione). Infatti, vengono utilizzati gli identificativi dei nodi che sono contenuti nei path di input.

Facciamo un esempio per capire meglio come si può utilizzare questo operatore. Supponiamo di avere un insieme di path descritto nel seguente modo:

$$\{(a \rightarrow b), (a \rightarrow b \rightarrow c), (a \rightarrow d), (d \rightarrow c)\}$$

Diciamo anche che il nodo *c* ha un dato numerico del valore 38. Supponendo che il `PathSet` che contiene tale insieme si chiami `ins1`, possiamo sintetizzare tale insieme, creando un nuovo Network Level, identificato con 'nl1S', nel seguente modo:

```
Synthesis.synthesize('nl1S', ins1);
```

Il risultato grafico di tale insieme può essere osservato in Figura 5.3.5.

### 5.3.5 JOIN

L'operatore di Join che, dati due Network Level, restituisce un nuovo Network Level fondendoli in maniera opportuna, è stato definito nel modo seguente:

```
NetworkLevel join(String resultLevelID,
         NetworkLevel a,
         NetworkLevel b,
         Couple c,
         DataJoin fd,
         DataJoin fv);
```

Il primo argomento, analogamente a come viene fatto per l'operatore di aggregazione, è l'identificativo che verrà utilizzato per creare il nuovo Network Level che costituirà il risultato. Gli altri argomenti invece seguono la definizione che è stata data del Join, rispettivamente: i due Network Level, un accoppiamento tra i due che descrive come sono messi in relazione, e due funzioni che determinano sia come devono essere fusi i dati dei nodi e degli archi. Queste ultime funzioni di callback possono essere definite a sua volta utilizzando la classe `DataJoin`, definita nel modo seguente:

```
abstract class DataJoin {
  public abstract Object join(Object a, Object b);
}
```

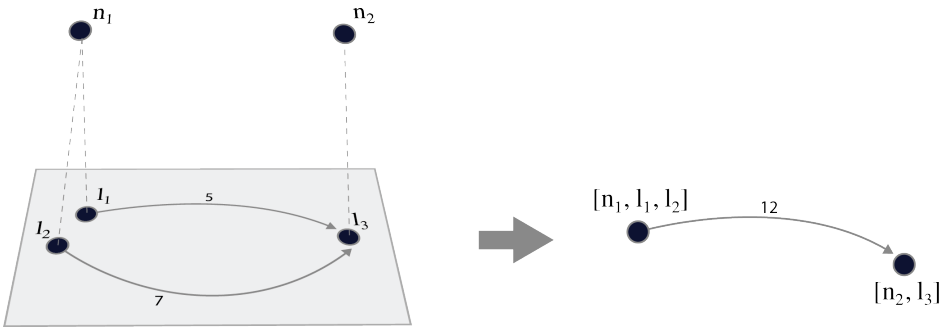



**Figura 5.3.6:** Esempio di un join, dove si vuole fondere il livello sovrastante con quello sottostante, e dove gli archi comuni vengono fusi facendo la somma dei loro valori.

Per capire come può essere utilizzato questo operatore, riprendiamo l'Esempio 4.4.1, riportato sinteticamente in Figura 5.3.6.

In particolare sappiamo in questo caso che la funzione $f_v$ è definita come quella che restituisce la somma dei valori degli archi che è necessario fondere, mentre non importa come venga definita la funzione $f_d$. Per questo esempio, osservando la Figura 5.3.6, assumiamo che il livello soprastante sia stato opportunamente creato e memorizzato in `nl1`, mentre quello sottostante in `nl2`. Inoltre assumiamo che l'accoppiamento tra questi due livelli sia stato definito come in figura, e memorizzato in `couple`. Il join può quindi essere effettuato nel seguente modo:

```
Join.DataJoin fv = new Join.DataJoin() {
  @Override
  public Object join(Object a, Object b) {
    return ((Integer)a + (Integer)b);
  }
};

NetworkLevel nlris = Join.join("nlris", nl1, nl2, couple, null, fv);
```

Il risultato di tale operazione, memorizzato in `nlris`, è lo stesso che può essere osservato nel Network Level di destra, in Figura 5.3.6.

# CAPITOLO 6

# CONCLUSIONI

Citando diversi lavori, ci siamo convinti del fatto che ormai non solo è importante analizzare tipi di dati strutturati come un grafo, ma che è altrettanto importante analizzare più grafi, correlandoli tra di loro. Queste relazioni possono essere gerarchiche, per semplificare reti troppo complesse; oppure temporali, permettendo l'analisi dell'evoluzione di una rete; oppure possono riguardare la stessa rete, con gli stessi nodi, ma curando connessioni di tipo differente; ed infine possono essere tra reti totalmente differenti, che riguardano dati della stessa natura (e.g. Facebook e Twitter) o di natura differente (e.g. rete urbana e rete di comunicazione telefonica).

Abbiamo pertanto creato un modello dei dati, chiamato *Multi Level Network*, definito come una struttura dati formata da una serie di reti differenti (i.e. Network Level) relazionate tra di loro (grazie ai Levels Coupling). In seguito abbiamo dimostrato, con delle semplici dimostrazioni di riduzione, come, data una determinata struttura dati tra quelle citate, fosse possibile rappresentarla utilizzando i Multi Level Network, confermando pertanto l'espressività del modello.

In seguito abbiamo definito un linguaggio che permettesse l'analisi di questi dati, e la manipolazione del modello definito. Anche in questo campo abbiamo ricercato e rassegnato i diversi lavori che affrontano questo tema, cercando di capire i pregi e i

difetti di ogni approccio. L'algebra, basata su path, si oppone alla vecchia concezione dell'estrazione di sotto-grafi, presente in vecchi lavori che affrontano questi temi. Allo stesso modo, la definizione di pattern su questi path è computazionalmente più semplice rispetto alla ricerca di sotto-grafi isomorfi.

È stato pertanto definito l'*operatore di selezione* che, dato un determinato Network Level, restituisce un'insieme di path conformi ad un path pattern e ad un predicato. Tale operatore può a sua volta essere utilizzato per estrarre un sotto-grafo, utilizzando un operatore che abbiamo chiamato *operatore di sintesi*, che fonde tutti i path di un insieme. In seguito è stato definito il complesso *operatore di aggregazione*, che permette di aggregare nodi ed archi, ed i relativi dati, utilizzando funzioni note come sommatoria, media, massimo o minimo. Gli *operatori di concatenazione e di proiezione* permettono invece di poter manipolare i path e gli insiemi di path. Con l'intento di analizzare meglio diversi livelli, è stato anche definito l'*operatore di join* che, dati due Network Level, li fonde osservando le relazioni che incorrono tra di essi. La definizione di questi operatori offrono risultati interessanti. Molti lavori infatti si limitano alla definizione di un operatore di selezione, e raramente invece vengono definiti operatori di aggregazione. Lo stesso operatore di selezione effettua delle estrazioni ad un livello di dettaglio maggiore rispetto all'estrazione di un sotto-grafo. Infine l'operatore di join è uno dei primi a considerare la manipolazione di queste reti multi-livello.

La terza fase del lavoro è stata quella di studiare le proprietà, le considerazioni implementative e le ottimizzazioni degli operatori. Con le proprietà si sono potuti conoscere meglio gli operatori in sé e come questi si integrano tra di loro. Con le considerazioni implementative si è potuto capire come poter realizzare i vari operatori. Infine, le considerazioni sulle ottimizzazioni hanno fornito spunti per migliorare l'efficienza in termini di tempo e spazio.

L'ultima fase del lavoro è stata quella di rendere tutto più concreto, creando un prototipo in Java di circa 3500 righe di codice, in modo da fornire una visione ancora più

dettagliata ed esente da ambiguità, del modello e degli operatori. Si è potuto quindi verificare la correttezza degli algoritmi che sono stati illustrati, verificare il modo con cui possono essere utilizzati gli operatori per produrre determinati risultati, verificare la correttezza delle definizioni del modello e degli operatori, verificare l'interazione tra il modello e gli operatori e l'interazione tra gli operatori, ed infine verificare alcune delle ottimizzazioni che sono state proposte.

## 6.1 SVILUPPI FUTURI

Sicuramente il lavoro svolto necessita di ulteriori sviluppi futuri, per cercare di capire meglio come il modello e il linguaggio possano essere efficientemente utilizzati nei sistemi commerciali esistenti.

Per proseguire in questo senso bisognerebbe eseguire dei test di larga scala, utilizzando dei dati reali messi a disposizione da servizi come Twitter o Facebook, cercando anche di capire il tipo di analisi che interessa. Vista la generalità del modello che è stato definito, e visto anche che il linguaggio proposto è stato pensato per poter esprimere interrogazioni su sistemi reali (in particolar modo per i social network), ciò non dovrebbe essere complicato, ma richiede comunque un'attenta riflessione.

Il prototipo è un altro aspetto da migliorare, poiché è solamente un punto di partenza, nonostante sia già abbastanza complesso. Bisognerebbe raffinare gli operatori cercando di definire linguaggi specifici che aiutino l'utente a richiamarli, come succede ad esempio con SQL. Bisognerebbe anche cercare di focalizzarsi maggiormente sulla realizzazione di un sistema che implementi algoritmi specifici per manipolare i dati, anche quando questi si trovano in memoria secondaria. Infine bisognerebbe cercare di comparare il sistema con altri sistemi esistenti, valutandone sia l'efficacia che l'efficienza.

Infine sarebbe utile capire meglio se l'algebra necessita di ulteriori operatori, che possano essere utili per formulare determinate interrogazioni. La formulazione di

questi nuovi operatori dovrà essere comunque conforme alla filosofia generale del linguaggio, ed è necessario quindi confermare che quest'ultima non sia limitante.

## 6.2 CONCLUSIONI FINALI

Il lavoro svolto è stato certamente lungo e complesso. Quando si definiscono modelli e linguaggi bisogna sempre pensare ad un qualcosa di semplice, espressivo (in modo da risolvere i problemi oggetto di studio), e coerente (gli operatori tra loro devono potersi integrare perfettamente, come anche il modello con gli operatori). Per questo motivo il modello e il linguaggio proposto non sono altro che il frutto di continui cambiamenti e raffinamenti, ciò che viene descritto è soltanto il risultato di tale processo.

La definizione del modello è nata con l'intento di creare qualcosa di abbastanza generale per modellare tutti quegli scenari che sono stati descritti. Allo stesso tempo la semplicità e la coerenza di questo modello è stato di vitale importanza per le fasi successive del lavoro, dove è stato necessario sfruttare il modello, dandolo per assodato. La definizione degli operatori è stata altrettanto complessa, soprattutto perché si è cercato di definire un'algebra semplice, con operatori che risolvessero problemi reali, nonostante dovessero non limitarsi ad analizzare un problema in particolare. Difficile è anche stato trovare un accordo tra i vari operatori, facendo integrare l'input e l'output di ognuno, garantendo la possibilità di effettuare operazioni annidate, utilizzando anche operatori differenti. Il prototipo invece è stato utile per poter verificare il lavoro svolto, rendendo tutte le definizioni e idee un qualcosa di concreto.

La necessità di integrare diversi sistemi tra loro, ognuno con dati organizzati a grafi, necessita di un approccio simile a quello che qui è stato adottato. Sempre di più i dati di molti sistemi si allontanano dalle rigide strutture del modello relazionale. Pertanto, sosteniamo che il lavoro presenta un'ottima base per poter proseguire la ricerca in questo campo, e potrebbe dare ottimi spunti per i nuovi sistemi futuri per la gestione dei dati.

# BIBLIOGRAFIA


[1] Frank Wm Tompa. A data model for flexible hypertext database systems. *ACM Transactions on Information Systems (TOIS)*, 7(1):85–100, 1989.

[2] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L Wiener. The lorel query language for semistructured data. *International journal on digital libraries*, 1(1):68–88, 1997.

[3] Peter Buneman. Semistructured data. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 117–121. ACM, 1997.

[4] Marc Gyssens, Jan Paredaens, and Dirk Van Gucht. A graph-oriented object database model. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 417–424. ACM, 1990.

[5] Sihem Amer-Yahia, Laks Lakshmanan, and Cong Yu. Socialscope: Enabling information discovery on social content sites. *arXiv preprint arXiv:0909.2058*, 2009.

[6] Anton Dries, Siegfried Nijssen, and Luc De Raedt. A query language for analyzing networks. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 485–494. ACM, 2009.

[7] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. nsparql: A navigational language for rdf. *Web Semantics: Science, Services and Agents on the World Wide Web*, 8(4):255–270, 2010.

[8] Marcelo Arenas, Claudio Gutierrez, and Jorge Pérez. An extension of sparql for rdfs. In *Semantic Web, Ontologies and Databases*, pages 1–20. Springer, 2008.

[9] Royi Ronen and Oded Shmueli. Soql: A language for querying and creating data in social networks. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 1595–1602. IEEE, 2009.

[10] Mauro San Martın, Claudio Gutierrez, and Peter T Wood. Snql: A social networks query and transformation language. *cities*, 5:r5, 2011.

[11] Peter J Mucha, Thomas Richardson, Kevin Macon, Mason A Porter, and Jukka-Pekka Onnela. Community structure in time-dependent, multiscale, and multiplex networks. *Science*, 328(5980):876–878, 2010.

[12] Michael Balzer and Oliver Deussen. Level-of-detail visualization of clustered graph layouts. In *Visualization, 2007. APVIS'07. 2007 6th International Asia-Pacific Symposium on*, pages 133–140. IEEE, 2007.

[13] Reinhard Diestel. Graph theory. 2005, 2005.

[14] Michele Berlingerio, Michele Coscia, Fosca Giannotti, Anna Monreale, and Dino Pedreschi. Foundations of multidimensional network analysis. In *Advances in Social Networks Analysis and Mining (ASONAM), 2011 International Conference on*, pages 485–489. IEEE, 2011.

[15] Luca Rossi Matteo Magnani. Pareto distance for multi-layer network analysis.

[16] Matteo Magnani and Luca Rossi. The ml-model for multi-layer social networks. In *Advances in Social Networks Analysis and Mining (ASONAM), 2011 International Conference on*, pages 5–12. IEEE, 2011.

[17] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Englewood Cliffs, 2001.

[18] Marc Gyssens, Jan Paredaens, and Dirk Van Gucht. A graph-oriented object model for database end-user interfaces. In *ACM SIGMOD Record*, volume 19, pages 24–33. ACM, 1990.

[19] Jan Hidders. *A graph-based update language for object-oriented data models*. PhD thesis, Technische Universiteit Eindhoven, 2001.

[20] Jan Hidders and Jan Paredaens. Goal, a graph-based object and association language. 1993.

[21] Bernd Amann and Michel Scholl. Gram: a graph data model and query languages. In *Proceedings of the ACM conference on Hypertext*, pages 201–211. ACM, 1992.

[22] Ralf Hartmut Güting. Graphdb: Modeling and querying graphs in databases. In *VLDB*, volume 94, pages 12–15. Citeseer, 1994.

[23] Marc Levene and Alexandra Poulovassilis. The hypernode model and its associated query language. In *Information Technology, 1990.'Next Decade in Information Technology', Proceedings of the 5th Jerusalem Conference on (Cat. No. 90TH0326-9)*, pages 520–530. IEEE, 1990.

[24] Ravi Kothuri, Albert Godfrind, and Euro Beinat. *Pro oracle spatial for oracle database 11g*. Dreamtech Press, 2008.

[25] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1, 2008.

[26] Mark Graves, Ellen R Bergeman, and Charles B Lawrence. A graph-theoretic data model for genome mapping databases. In *System Sciences, 1995. Vol. V.*

*Proceedings of the Twenty-Eighth Hawaii International Conference on*, volume 5, pages 32–41. IEEE, 1995.

[27] Lisbeth Bergholt, Jacob S Due, Thomas Hohn, Jørgen L Knudsen, Kirsten H Nielsen, Thomas S Olesen, and Emil Hahn Pedersen. Database management systems: Relational, object-relational, and object-oriented data models. *Centre For Objekt Teknology*, 1998.

[28] Alfred V Aho and Jeffrey D Ullman. *Foundations of computer science*, volume 2. Computer Science Press New York, 1992.

[29] John Adrian Bondy and Uppaluri Siva Ramachandra Murty. *Graph theory with applications*, volume 290. Macmillan London, 1976.

[30] Edgar F Codd. A relational model of data for large shared data banks. In *Pioneers and Their Contributions to Software Engineering*, pages 61–98. Springer, 2001.

[31] Sergio Flesca and Sergio Greco. Partially ordered regular languages for graph queries. In *Automata, Languages and Programming*, pages 321–330. Springer, 1999.

[32] Sergio Flesca and Sergio Greco. Querying graph databases. In *Advances in Database Technology—EDBT 2000*, pages 510–524. Springer, 2000.

[33] Charu C Aggarwal and Haixun Wang. *Managing and mining graph data*, volume 40. Springer, 2010.

[34] H. He. Querying and mining graph databases. *Ph.D. Thesis, UCSB*, 2007.

[35] Anna Lubiw. Some np-complete problems similar to graph isomorphism. *SIAM Journal on Computing*, 10(1):11–21, 1981.

[36] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[37] Isabel F Cruz, Alberto O Mendelzon, and Peter T Wood. A graphical query language supporting recursion. *ACM SIGMOD Record*, 16(3):323–330, 1987.

[38] Isabel F Cruz, Alberto O Mendelzon, and Peter T Wood. G+: Recursive queries without recursion. In *Proceedings of the Second International Conference on Expert Database Systems*, pages 355–368, 1988.

[39] Mariano P Consens and Alberto O Mendelzon. Graphlog: a visual formalism for real life recursion. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 404–416. ACM, 1990.

[40] Rosalba Giugno and Dennis Shasha. Graphgrep: A fast and universal method for querying graphs. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 2, pages 112–115. IEEE, 2002.

[41] James Clark, Steve DeRose, et al. Xml path language (xpath) version 1.0, 1999.

[42] CA James, D Weininger, and J Delany. Daylight theory manual-daylight 4.71. *Daylight Chemical Information Systems*, 2000.

[43] Dan Stefanescu and Alex Thomo. Enhanced regular path queries on semistructured databases. In *Current Trends in Database Technology–EDBT 2006*, pages 700–711. Springer, 2006.

[44] Alberto O Mendelzon and Peter T Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258, 1995.

[45] André Koschmieder and Ulf Leser. Regular path queries on large graphs. In *Scientific and Statistical Database Management*, pages 177–194. Springer, 2012.

[46] Serge Abiteboul and Victor Vianu. Regular path queries with constraints. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 122–133. ACM, 1997.

[47] Gabriel M Kuper and Moshe Y Vardi. A new approach to database logic. In *Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 86–96. ACM, 1984.

[48] Jan Paredaens, Peter Peelman, and Letizia Tanca. G-log: A graph-based query language. *Knowledge and Data Engineering, IEEE Transactions on*, 7(3):436–453, 1995.

[49] Yannis E Ioannidis. Query optimization. *ACM Computing Surveys (CSUR)*, 28(1):121–123, 1996.

[50] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.

[51] Chia-Hsiang Chang and Robert Paige. From regular expressions to dfa's using compressed nfa's. In *Combinatorial Pattern Matching*, pages 90–110. Springer, 1992.

[52] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.

[53] Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. *Artificial intelligence: a modern approach*, volume 74. Prentice hall Englewood Cliffs, 1995.

[54] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[55] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In *Algorithms–ESA 2006*, pages 684–695. Springer, 2006.

[56] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz—open source graph drawing tools. In *Graph Drawing*, pages 483–484. Springer, 2002.

[57] Oracle. Java Persistence API. http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html.

[58] Peter T Wood. Query languages for graph databases. *ACM SIGMOD Record*, 41(1):50–60, 2012.

[59] Lei Sheng, Z Meral Ozsoyoglu, and Gultekin Ozsoyoglu. A graph query language and its query processing. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 572–581. IEEE, 1999.

[60] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. Rewriting of regular expressions and regular path queries. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 194–204. ACM, 1999.