Università degli Studi di Udine
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Corso di Laurea Magistrale in Informatica

Tesi di Laurea Magistrale

Le Simulazioni Lasche

Definizione, Applicazioni e Computazione Distribuita

Candidato
Alessio Mansutti

Relatore
Prof. Marino Miculan

Co-relatore
Dott. Marco Peressotti

Anno accademico 2015-2016

Contatti dell'Istituto
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine — Italia
+39 0432 558400
http://www.dimi.uniud.it/

Ringraziamenti

Desidero innanzitutto ringraziare il Prof. Marino Miculan ed il Dott. Marco Peressotti, rispettivamente relatore e co-relatore di questa tesi, non solo per le molteplici osservazioni fattemi riguardanti questo lavoro ma per avermi permesso negli ultimi anni di prendere parte alla loro attività di ricerca, dalla quale ho tratto numerosi insegnamenti e spunti di riflessione.

Ringrazio il Prof. Andrea Corradini, dell'Università di Pisa, per vari suggerimenti riguardanti le *simulazioni lasche*, argomento principale di questa tesi. Eventuali complimenti riguardanti il nome di queste vanno a lui.

Infine ringrazio le persone a me più care: la mia famiglia ed i miei amici, ai quali dedico questo lavoro. Tra loro, un grazie particolare va a mia zia Silvia e ad Andrea per aver rivolto una parte del loro tempo alla lettura di questo scritto e avermi suggerito numerosi cambiamenti per aumentarne la chiarezza espositiva.

Indice

1

Introduzione

I grafi sono ormai da molti anni uno strumento ubiquo e fondamentale per la rappresentazione ed il recupero di informazioni. Essi abbinano intuitività ed eleganza alla possibilità di esprimere strutture topologiche complesse, come per esempio la struttura di un sistema distribuito. Il loro utilizzo è talmente comune e rilevante da essere ormai un argomento centrale nei curriculum di laurea in scienze matematiche ed informatiche, trattato in molteplici corsi e sotto diversi punti di vista. All'interno della loro teoria, ricca di risultati profondi dalle forti componenti applicative, uno degli ambiti più approfonditi negli ultimi anni è quello del pattern matching.

Il *pattern matching* su grafi studia le metodologie per l'individuazione automatica di particolari strutture all'interno di un grafo. Più precisamente viene richiesto di trovare porzioni di un grafo che soddisfino una query definita formalmente — solitamente attraverso un grafo, un linguaggio o un predicato logico. A queste porzioni viene assegnata una semantica ben precisa, specifica ad un determinato problema, che può essere utilizzata per studiare e trarre conclusioni sulle proprietà dell'intero grafo. Le prime implementazioni di algoritmi per il pattern matching — non limitato ai grafi — risalgono alle fine degli anni settanta, grazie all'algoritmo di J.R. Ullmann per la computazione dell'isomorfismo di sottografo [42] e l'aggiunta al text editor QED delle funzioni di ricerca tramite espressioni regolari ad opera di K. Thompson [41]. Per capire l'importanza del pattern matching su grafi basta osservare quanto venga utilizzato nei più diversi settori di ricerca. In biologia computazionale questo viene utilizzato nel sequenziamento di proteine e lo studio dell'albero filogenetico [34]. In chimica e chemioinformatica viene applicato allo studio di sistemi molecolari, al fine di predirne l'evoluzione [1, 7]. Nelle scienze forensi ed in sociologia viene utilizzato per la profilazione di persone, molto utilizzata negli ultimi anni grazie alla repentina crescita dei social network [15]. Infine — anche se questo elenco è ben lontano dal essere completo — il pattern matching su grafi è utilizzato in informatica per la verifica automatica di proprietà di sistema, per la risoluzione di problemi legati alla progettazione di reti e, più in generale, nell'ambito del data mining [27, 35, 43].

Tuttavia, anche proprio a causa dell'ubiquità del pattern matching su grafi nei diversi settori di ricerca, in letteratura si trovano numerose nozioni in grado di identificare strutture molto specifiche all'interno di un grafo le quali sono poi utilizzate per risolvere problemi altrettanto specifici. In questo contesto risulta quindi problematico riuscire ad individuare facilmente strutture nel grafo corrispondenti simultaneamente a più di queste nozioni. Molto spesso infatti, i pattern cercati corrispondono ad una composizione di più nozioni di similarità tra grafi che, per mancanza di alternative, vanno controllate

una ad una utilizzando talvolta anche diversi strumenti e portando quindi un aggravio in termini di prestazioni. A complicare ancor più questo scenario è il tasso di crescita delle dimensioni dei grafi sui quali viene compiuto il pattern matching: scenari come l'*internet of things* e i *social network* richiedono l'analisi di grafi con centinaia di milioni di nodi. È sempre più importante dunque la necessità fornire soluzioni algoritmiche che permettano la computazione delle varie nozioni di pattern matching rispetto a grafi di notevoli dimensioni. Essendo molte di queste nozioni associate a problemi NP-difficili, una delle soluzioni spesso proposte a questo problema va nella direzione dei sistemi ed algoritmi distribuiti [4, 17, 18, 28].

A fronte di questi due problemi, in questo lavoro viene proposta una nuova nozione di relazione tra grafi in grado di essere utilizzata per risolvere problemi di pattern matching. In questa nozione, che prende il nome di *simulazione lasca*, la semantica delle queries — rappresentate tramite dei grafi opportunamente decorati — è tale da permettere l'individuazione di molteplici pattern interessanti e piuttosto diversi tra loro. Una buona parte della tesi sarà dunque incentrata sullo studio delle simulazioni lasche e all'analisi delle relazioni che intercorrono tra questa nuova nozione e le sue alternative presenti in letteratura.

Accanto alla definizione delle simulazioni lasche e allo studio delle loro applicazioni verrà introdotto un algoritmo in grado di computarle. L'algoritmo proposto sarà parallelo e distribuito, cercando quindi di rispondere alla problematica legata alle dimensioni dei grafi sui quali viene eseguito il pattern matching. Questo è reso necessario anche in virtù del fatto che, come sarà formalmente dimostrato, il calcolo delle simulazioni lasche è un problema NP-completo.

1.1 Lavori Correlati

Come già accennato, le simulazioni lasche sono collocate in uno scenario molto vasto e comprendente problemi studiati approfonditamente per decenni; come per esempio:

- il problema dell'*isomorfismo di sottografo* [42], nel quale dati due grafi G e Q, viene richiesto di trovare un sottografo di G isomorfo a Q. È forse il problema di pattern matching su grafi più conosciuto e studiato, con algoritmi risalenti agli anni settanta e numerose applicazioni in informatica e chemioinformatica [1, 7, 27];

- il *pattern matching di linguaggi regolari* su grafi [22], secondo il quale va ricercato un cammino in un grafo con archi etichettati in modo tale da formare una parola appartenente al linguaggio. Questo problema ha importanti applicazioni nei database grafici ed è spesso studiato in relazione a logiche modali come LTL [13, 31];

- le *simulazioni di grafo* [36], nel quale dati due grafi G e Q si richiede esista una porzione di G che *simuli* Q rispetto le transizioni uscenti dai nodi di quest'ultimo. Le simulazioni di grafo sono utilizzate per la profilazione utenti, in bioinformatica e urbanistica [15, 17].

Rispetto a questi problemi, le simulazioni lasche godono di alcune caratteristiche piuttosto interessanti, le quali motivano la scelta compiuta in questo lavoro di introdurre nell'ambito del pattern matching un'ulteriore nozione di similarità tra grafi.

Per prima cosa, il calcolo delle simulazioni lasche sussume molti problemi presenti in letteratura — come per esempio i tre problemi appena citati. Le riduzioni di questi problemi alle simulazioni lasche risultano piuttosto intuitive, anche grazie ad una semplice notazione grafica definita per le queries di queste ultime pensata per renderne immediata la comprensione. La riduzione opposta, qualora possibile, non appare invece altrettanto intuitiva — come per esempio nel caso dell'isomorfismo di sottografo, essendo anch'esso un problema NP-completo.

In secondo luogo, le simulazioni lasche sono state pensate principalmente per permettere l'identificazione in un grafo di strutture che possano essere viste come il risultato dell'applicazione di molteplici problemi diversi di pattern matching, andando così a proporre una soluzione al problema brevemente esposto nella sezione precedente. Per esempio, poniamo di voler verificare l'esistenza all'interno di un grafo di un cammino di lunghezza arbitraria terminante in un nodo facente parte di un sottografo dalla struttura ben precisa. Una query di questo tipo può essere vista come una composizione del problema di isomorfismo di sottografo con un problema di raggiungibilità tra nodi. Le simulazioni lasche permettono di risolvere richieste di questo tipo passando per una traduzione di queries relative a diversi problemi ed andando ad applicare su queste un'operazione di composizione. La query risultante da questa operazione di composizione sarà tale da identificare correttamente i sottografi richiesti, rispetto al problema del calcolo delle simulazioni lasche. Ancora una volta, come per le riduzioni, queste traduzioni e composizioni sono piuttosto semplici e mantengono l'immediatezza espressiva delle queries di partenza.

Per quanto riguarda invece l'algoritmo distribuito per il calcolo delle simulazioni lasche, si è scelto di utilizzare un algoritmo generico già istanziato con successo per altri problemi, come ad esempio per la computazione di *embedding* di *bigrafi* ad opera di M. Miculan, M. Peressotti ed il sottoscritto [29]. La generalità e modularità di questo algoritmo permette di ottenere facilmente un buon *proof of concept* riguardante la fattibilità del calcolo distribuito delle simulazioni lasche, ancorché meno performante di alcuni algoritmi ad-hoc per problemi di pattern matching [4, 28].

1.2 Organizzazione della tesi

La tesi è suddivisa in due parti. Nella prima parte, composta da tre capitoli, verranno introdotte e studiate le simulazioni lasche. Più nel dettaglio, Capitolo 2 introdurrà le definizioni di guest, ovvero le query per simulazioni lasche, e host, *i.e.* il grafo sul quale si intende compiere la verifica di proprietà. Il capitolo si concluderà con la definizione formale delle simulazioni lasche. In seguito, in Capitolo 3, vedremo quali relazioni esistono tra le simulazioni lasche ed alcune nozioni comunemente utilizzate nel pattern matching su grafi. In particolare, ci focalizzeremo sui problemi di *isomorfismo di sottografo*, *simulazioni di grafo* e sul pattern matching di *linguaggi regolari*, mostrando come questi possano essere ridotti alle simulazioni lasche. Il capitolo mostrerà inoltre un esempio piuttosto esauriente di come queste possano essere utilizzate per comporre query provenienti dai problemi appena elencati. Utilizzando anche alcuni risultati attinenti a queste riduzioni proseguiremo andando a dimostrare formalmente l'NP-completezza del problema del vuoto per simulazioni lasche. Questo verrà fatto in Capitolo 4, dove verrà effettuata una riduzione di tale problema ad un modello di programmazione lineare mista. Con questo risultato terminerà lo studio delle simulazioni lasche.

La seconda parte sarà incentrata sullo studio di un algoritmo parallelo e distribuito per il problema del vuoto delle simulazioni lasche. Questo algoritmo verrà definito a partire da un algoritmo generico per il calcolo distribuito di predicati, detto di Amalgamazione Distribuita e introdotto in Capitolo 5. Una volta studiati i requisiti dell'algoritmo, questo verrà istanziato al caso delle simulazioni lasche in Capitolo 6. Nel fornire l'istanziazione, studieremo anche un metodo che permetta la riduzione del numero di messaggi scambiati tra i nodi della rete coinvolti nella computazione. Il capitolo terminerà quindi con l'analisi di complessità dell'algoritmo, data per l'appunto in termini di messaggi scambiati complessivamente.

Accanto a queste due parti prettamente teoriche viene discussa in appendice una progettazione ed implementazione dell'algoritmo distribuito per il problema del vuoto delle simulazioni lasche. In particolare Appendice A studierà l'implementazione dell'algoritmo di Amalgamazione Distribuita introdotto in Capitolo 5, mentre Appendice B ne mostrerà l'istanziazione nel caso delle simulazioni lasche, secondo quanto detto in Capitolo 6.

Il grafo in Figura 1.1 riassume la struttura della tesi, mostrando la suddivisione delle sue parti e sottolineando le dipendenze che intercorrono tra i vari capitoli.

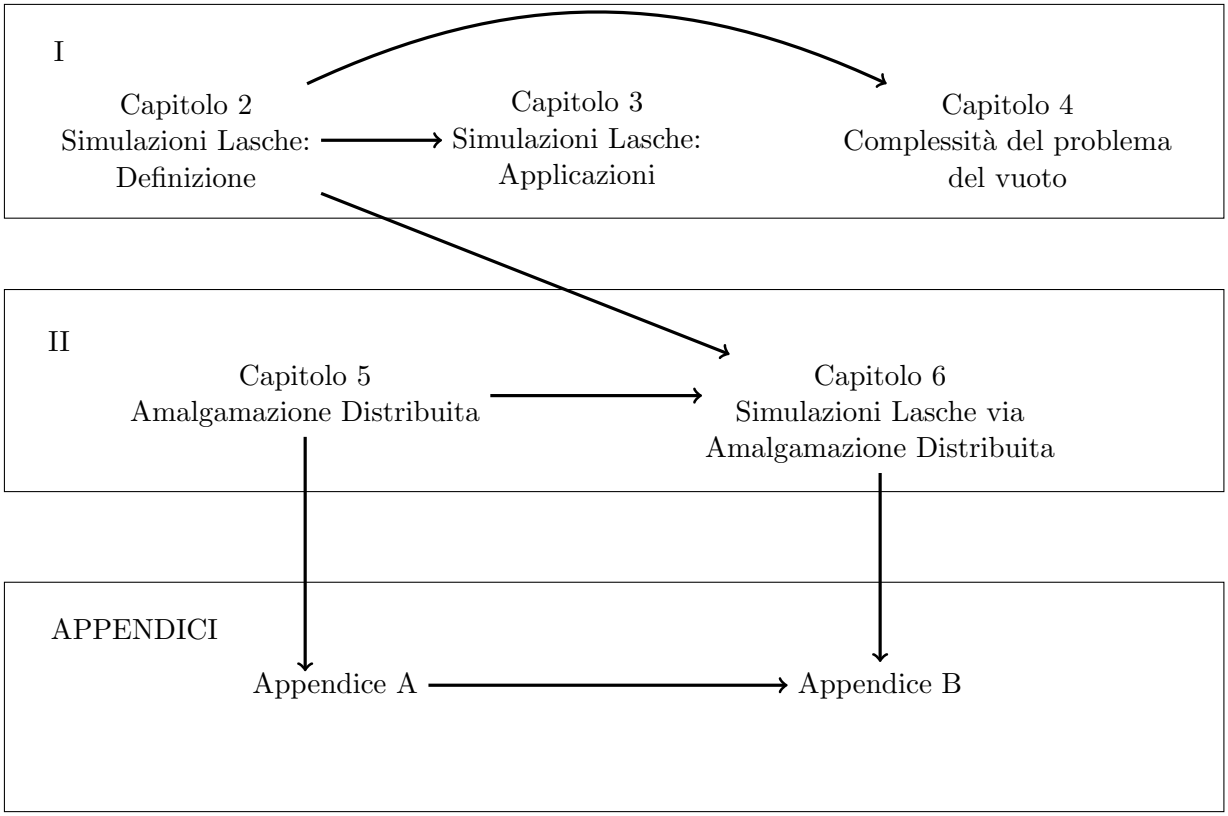


Figura 1.1: Struttura della tesi

I

Le Simulazioni Lasche

2

Simulazioni Lasche: Definizione

Le *simulazioni lasche* sono delle relazioni di similarità tra grafi pensate per essere utilizzate come strumento di verifica di proprietà di sistema. Esse ricadono nel problema generale di pattern matching su grafi e generalizzano importanti nozioni di similarità come *isomorfismo di sottografo* e *simulazione di grafo* — come vedremo in Capitolo 3. Inoltre, le query utilizzate per la verifica di queste nozioni vengono tradotte nelle corrispondenti query — d'ora in poi chiamate *guest* — per simulazioni lasche in modo intuitivo. Altra caratteristica importante è data dalla possibilità di unire query corrispondenti a nozioni diverse in un unico guest. Infine, i guests per simulazioni lasche possono essere descritti attraverso un formalismo grafico che li rende di immediata comprensione.

In questo capitolo daremo la definizione formale di *host* — il grafo rappresentante il sistema sul quale eseguire verifiche — e di guest, per poi definire le simulazioni lasche.

2.1 Definizioni preliminari

Prima di poter definire host e guest per simulazioni lasche, è necessario richiamare alcune nozioni fondamentali e spesso utilizzate nel campo del pattern matching su grafi. La prima di queste nozioni è quella di *multigrafo diretto etichettato.*

Definizione 2.1 (Multigrafo diretto etichettato)**.** *Un* multigrafo diretto etichettato (Σ, V, E) *è una tripla composta da un insieme di simboli* Σ *(chiamato alfabeto), un insieme finito* V *di nodi (o vertici) ed un insieme* $E \subseteq V \times \Sigma \times V$ *di archi.*

Denoteremo con $s(e)$*,* $\sigma(e)$*, e* $t(e)$ *rispettivamente la prima, seconda e terza proiezione di un arco* $e \in E$*,* i.e.

- $s \triangleq \pi_1$*, dove* $s(e) \in V$ *è chiamato* nodo sorgente *di* e*;*
- $\sigma \triangleq \pi_2$*, dove* $\sigma(e) \in \Sigma$ *è l'*etichetta *di* e*;*
- $t \triangleq \pi_3$*, dove* $t(e) \in V$ *è chiamato* nodo terminazione *di* e*.*

Un arco $e \in E$ si dirà entrante *in $v \in V$ qualora $t(e) = v$, mentre si dirà* uscente *se $s(e) = v$. Infine, definiamo le seguenti funzioni* $\text{in}, \text{out} : V \to \mathcal{P}(E)$*:*

$$\text{in}(v) \triangleq \{e \mid t(e) = v\}$$
$$\text{out}(v) \triangleq \{e \mid s(e) = v\}$$

ovvero, dato un nodo $v \in V$, in *e* out *ritornano rispettivamente l'insieme degli archi entranti ed uscenti da v.*

Si noti come Definizione 2.1 non permetta l'esistenza, tra due nodi, di più archi con la stessa etichetta. Questa caratteristica, che normalmente non è presente nella definizione di multigrafo, è invece spesso utilizzata in ambito di verifica di proprietà di sistema basata su grafi. In questo, i nodi del grafo rappresentano configurazioni del sistema[1] mentre gli archi rappresentano transizioni — o più in generale relazioni — tra configurazioni. Due transizioni identiche tra due configurazioni risultano quindi indistinguibili dal punto di vista osservazionale e non appariranno nel multigrafo corrispondente. Per un più preciso studio sulle metodologie utilizzate per la traduzione di un sistema nella sua astrazione grafica, problema non affrontato in questo lavoro, richiamiamo alla lettura di [8, 20, 26, 35, 39].

Al fine di semplificare l'esposizione, è utile introdurre il dominio dei *cammini non vuoti* di un multigrafo — ovvero l'insieme di tutti i cammini composti da almeno un arco.

Definizione 2.2 (Dominio dei cammini non vuoti)**.** *Sia $G = (\Sigma, V, E)$ un multigrafo diretto etichettato. Denotiamo con $\mathbb{P}_G$ l'*insieme di tutti i cammini non vuoti *in G. Formalmente*

$$\mathbb{P}_G \triangleq \bigcup_{n \in \mathbb{N}^+} \{(e_0, \ldots, e_n) \in E^n \mid \forall i \in \{1, \ldots, n\}\ s(e_i) = t(e_{i-1})\}$$

Le funzioni sorgente, destinazione *ed* etichetta *di un arco, definite in Definizione 2.1, sono estese per ogni cammino $(e_0, \ldots, e_n) \in \mathbb{P}_G$ come segue:*

$$s((e_0, \ldots, e_n)) \triangleq s(e_0) \qquad s : \mathbb{P}_G \to V$$
$$t((e_0, \ldots, e_n)) \triangleq t(e_n) \qquad t : \mathbb{P}_G \to V$$
$$\sigma((e_0, \ldots, e_n)) \triangleq \sigma(e_0) \cdot \ldots \cdot \sigma(e_n) \qquad \sigma : \mathbb{P}_G \to \Sigma^+$$

Siano $v, v' \in V$ due nodi. Denotiamo con $\mathbb{P}_G(v, v')$ l'insieme di tutti i cammini con sorgente v e terminazione v', i.e.

$$\mathbb{P}_G(v, v') \triangleq \{\rho \in \mathbb{P}_G \mid s(\rho) = v \wedge t(\rho) = v'\}$$

Questa definizione tornerà utile in Sezione 2.3 per definire le simulazioni lasche. Si noti come i cammini vengano rappresentati con sequenze di archi e non di nodi, come viene comunemente fatto in teoria dei grafi. La motivazione dietro a questa scelta è ovvia: lavorando con multigrafi, una sequenza

[1] Le configurazioni di un sistema possono per esempio rappresentare stati di un programma, entries di un database oppure, più semplicemente, nodi di una rete.

di nodi non corrisponde ad un singolo cammino ma può, in generale, denotare un numero esponenziale di questi. Possiamo ora definire le nozioni di host e guest.

2.2 Hosts e Guests

Nel ambito del pattern matching, una relazione di similarità tra grafi può essere vista come una qualsiasi relazione tra un grafo, chiamato *host*, rappresentante il sistema da analizzare ed un secondo grafo, chiamato *guest*, rappresentante una specifica che deve essere soddisfatta dall'host. Informalmente, diremo che esiste un *match* tra un guest G ed un host H, rispetto ad una data definizione di relazione di similarità, se esiste una relazione di quel tipo tra G e H. A seconda delle condizioni richieste nella definizione di relazione utilizzata si ottengono nozioni diverse di similarità: per esempio, richiedendo come relazione una funzione iniettiva che, per quanto riguarda gli archi, sia commutativa rispetto alle funzioni s, t di Definizione 2.1, si ottiene la relazione cercata nel problema d'isomorfismo di sottografo — la cui formalizzazione verrà vista in Sezione 3.1. Aggiungendo a questa definizione la suriettività si ottiene invece l'isomorfismo di grafi. Spesso quindi, gli hosts e i guests non sono altro che due grafi (o multigrafi) e tutte le informazioni sulla semantica del tipo di pattern matching sono date dalla nozione di similarità utilizzata. Questo tuttavia non è vero per le simulazioni lasche, dove parte della semantica viene data da parametri ausiliari decoranti il guest.

Definizione 2.3 (Host)**.** *Un* host *è un multigrafo diretto etichettato.*

Definizione 2.4 (Guest)**.** *Un grafo* guest $G = (\Sigma, V, E, \mathcal{M}, \mathcal{U}, \mathcal{E}, \mathcal{C})$ *è una settupla tale che*

- (Σ, V, E) *è un multigrafo diretto etichettato*
- $\mathcal{M}, \mathcal{U}, \mathcal{E} \subseteq V$ *sono tre insiemi di vertici, chiamati rispettivamente* must*,* unique *e* exclusive*.*
- $\mathcal{C} : V \to \coprod_{v \in V} \mathcal{P}^2(\text{out}(v))$*, è una* funzione di scelta *tale che* $\mathcal{C}(v) \subseteq \mathcal{P}(\text{out}(v))$*.*

La semantica dei tre insiemi ausiliari e della funzione di scelta sarà data formalmente quando definiremo le simulazioni lasche — Sezione 2.3. Intuitivamente, in una simulazione lasca

- devono apparire tutti i nodi nell'insieme *must*;
- i nodi nell'insieme *unique* possono essere associati solo ad un elemento dell'host;
- i nodi nell'insieme *exclusive* sono associati a nodi dell'host che non sono in relazione con altri elementi del guest;
- la funzione di scelta ha lo scopo di modellare la simulazione sugli archi. Essa attribuisce ad ogni nodo del guest degli insiemi di archi, tutti a lui uscenti. Questi insiemi possono essere visti come dei *pattern di transizione*, ovvero indicano come stati (e quindi configurazioni) dell'host possono transire verso altri stati. Un nodo nell'host, per poter essere in relazione con uno nodo del guest, deve rispettare almeno uno di questi pattern.

Si noti come la funzione di scelta possa tornare un numero di insiemi esponenziale rispetto al numero di archi del guest. Tuttavia, come vedremo in Capitolo 3, non solo i guests normalmente sono di piccole

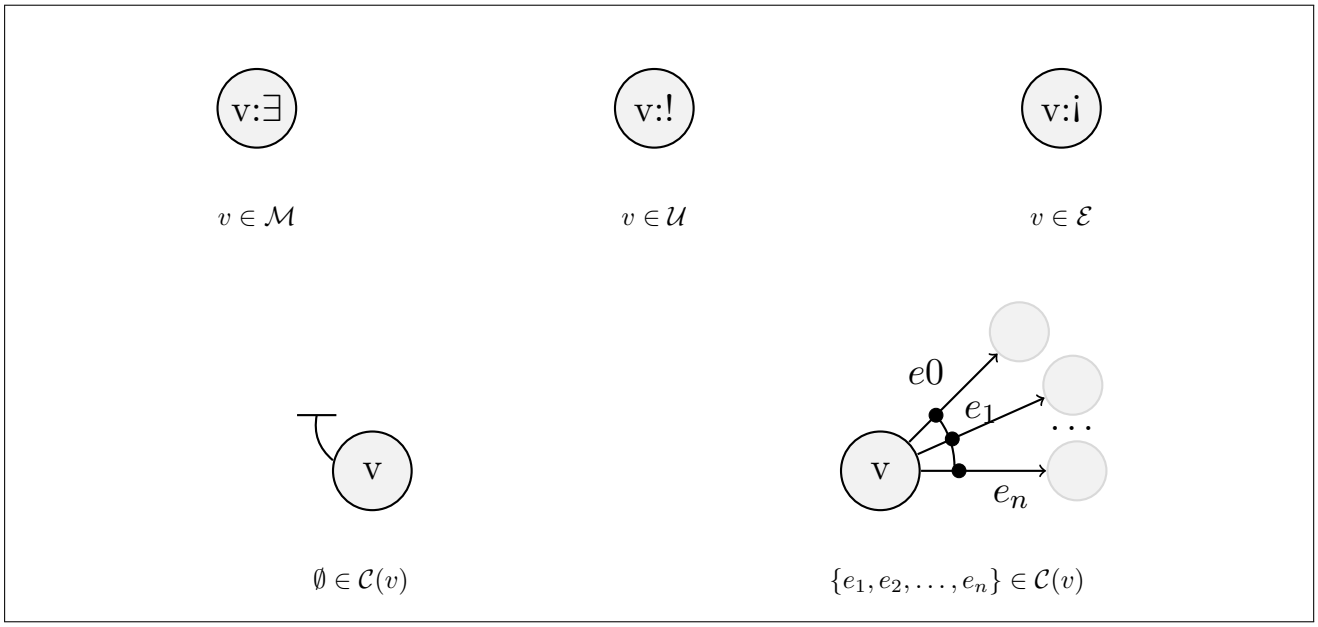


Figura 2.5: Notazione grafica per descrivere grafi guest.

dimensioni, ma nella maggior parte dei casi — ad esempio nella traduzione di altre nozioni di pattern matching in simulazioni lasche — questi insiemi sono al più lineari rispetto al numero di archi.

La definizione di guest è accompagnata con una semplice notazione grafica, rappresentata in Figura 2.5. Come si può notare, l'appartenenza di un nodo ai tre insiemi *must*, *unique* e *exclusive* viene rappresentata attraverso un'opportuna decorazione del nodo stesso. Per quanto riguarda la funzione di scelta $\mathcal{C}$ invece, dato un nodo v, un insieme in $\mathcal{C}(v)$ viene rappresentato con una corda che unisce gli archi di tale insieme[2]. Infine, un caso particolare è riservato alla presenza dell'insieme vuoto in $\mathcal{C}(v)$: in tale eventualità, il nodo v sarà decorato con un *arco tappato* (⌒).

Esempio 1. Figura 2.8 mostra, a sinistra, la rappresentazione di un guest attraverso la notazione appena introdotta. In tale guest, $\mathcal{M} = \{m\}$ mentre $\mathcal{U} = \mathcal{E} = \emptyset$. La funzione di scelta è invece definita come

$$\begin{aligned}\mathcal{C} &= \lambda x.\{\{e\} \mid e \in \mathit{out}(x)\} \cup \{\emptyset \mid \mathit{out}(x) = \emptyset\} \\ &= \lambda x. \begin{cases} \{\{(u,a,u)\},\{(u,a,m)\}\} & \text{se } x = u \\ \{\{(m,b,v)\}\} & \text{se } x = m \\ \{\emptyset\} & \text{se } x = v \end{cases}\end{aligned}$$

2.3 Simulazioni Lasche

In seguito, siano $H = (\Sigma, V_H, E_H)$ e $G = (\Sigma, V_G, E_G, \mathcal{M}, \mathcal{U}, \mathcal{E}, \mathcal{C})$ rispettivamente un host ed un guest. Per semplicità supponiamo sia definita una relazione di uguaglianza sugli elementi dell'alfabeto Σ; le seguenti definizioni sono tuttavia generalizzabili ad un qualsiasi preordine.

[2] L'intersezione tra archi di un insieme di $\mathcal{C}(v)$ e la corda a lui corrispondente sono evidenziati per evitare incoerenze nel caso di grafi non planari, dove un arco non appartenente a tale insieme potrebbe intersecare, nella rappresentazione grafica, la sua corda.

Al fine di semplificare l'esposizione definiamo la seguente *funzione di raggiungibilità duale*, che verrà poi utilizzata nella definizione di simulazione lasca.

Definizione 2.6 (Funzione di raggiungibilità duale)**.** *Sia $\eta \subseteq E_G \times E_H$ una relazione tra archi di G ed archi di H. Su η introduciamo una funzione di* raggiungibilità duale *$\Delta_\eta : V_G \times V_H \to \mathcal{P}(V_G \times V_H)$, definita come segue:*

$$\Delta_\eta(u,v) \triangleq \left\{ (u',v') \in V_G \times V_H \;\middle|\; \begin{array}{l} \exists (e_0,\ldots,e_n) \in \mathbb{P}_G(u,u') \\ \exists (e'_0,\ldots,e'_n) \in \mathbb{P}_H(v,v') \\ \forall i \in \{0,\ldots,n\}\ (e_i,e'_i) \in \eta \end{array} \right\}$$

Intuitivamente, data una coppia di nodi $(u,v) \in V_G \times V_H$, la funzione di raggiungibilità duale torna le coppie di nodi $(u',v') \in V_G \times V_H$ terminazione di due cammini, uno nel guest e uno nell'host, aventi sorgente u e v rispettivamente e i cui archi sono a due a due nella relazione η.

Possiamo ora definire le simulazioni lasche, contributo principale di questo capitolo.

Definizione 2.7 (Simulazione lasca)**.** *Una* simulazione lasca *di G in H è una coppia di relazioni (ϕ,η) con $\phi \subseteq V_G \times V_H$ e $\eta \subseteq E_G \times E_H$ tali che:*

(SL 1) *tutti i vertici di G nell'insieme must $\mathcal{M}$ occorrono in ϕ,* i.e.

$$\forall u \in \mathcal{M}\ u \in \pi_1(\phi)$$

(SL 2) *se un vertice di G appartiene all'insieme unique $\mathcal{U}$, allora può essere in relazione al massimo con un vertice di H:*

$$\forall u \in \mathcal{U}\ |\phi(u)| \leq 1$$

dove $\phi(u) \triangleq \{v \mid (u,v) \in \phi\}$;

(SL 3) *se un vertice di G appartiene all'insieme exclusive $\mathcal{E}$, allora i nodi di H in relazione con esso non possono essere in relazione con altri nodi di G. Formalmente,*

$$\forall u \in \mathcal{E}\ \forall v \in V_G \backslash \{u\}\ \phi(u) \cap \phi(v) = \emptyset$$

(SL 4) *ϕ e η preservano sorgenti e destinazioni, inoltre η mette in relazione unicamente coppie di archi con la stessa etichetta,* i.e. *per ogni $(e,e') \in \eta$*

$$(s(e),s(e')) \in \phi$$
$$(t(e),t(e')) \in \phi$$
$$\sigma(e) = \sigma(e')$$

(SL 5) *H* simula *G rispetto alla funzione di scelta $\mathcal{C}$: per ogni $(u,v) \in \phi$ esiste $\gamma \in \mathcal{C}(u)$ tale che γ è* selezionato *in η rispetto a v. Inoltre, per ogni $(e,e') \in \eta$ esiste $\gamma \in \mathcal{C}(s(e))$ tale che $e \in \gamma$ e γ è* selezionato *in η rispetto a $s(e')$.*

In questa condizione,

$$\gamma \text{ è selezionato in } \eta \text{ rispetto a } v \iff \forall e \in \gamma \; \exists e' \in out(v) \; (e, e') \in \eta$$

(SL 6) *La simulazione rispetta la raggiungibilità per quanto riguarda i nodi nell'insieme* must, i.e. *per ogni* $(u, v) \in \phi$ *e* $w \in \mathcal{M}$ *se* $\mathbb{P}_G(u, w) \neq \emptyset$ *allora esiste* $v' \in V_H$ *tale che* $(w, v') \in \Delta_\eta(u, v)$.

Con $\mathbb{S}^{G \to H}$ *denoteremo il dominio di tutte le simulazioni lasche del guest* G *nell'host* H. *Diremo inoltre che* $v \in V_H$ simula $u \in V_G$ *in una simulazione lasca* $(\phi, \eta) \in \mathbb{S}^{G \to H}$ *qualora* $(u, v) \in \phi$.

Come già anticipato nella sezione precedente, le condizioni (SL 1), (SL 2), (SL 3) definiscono la semantica degli insiemi *must*, *unique*, *exclusive*. A seconda delle proprietà di questi parametri del guest, la simulazione cercata cambia notevolmente. Per esempio, data una simulazione lasca $(\phi, \eta) \in \mathbb{S}^{G \to H}$, la condizione (SL 2) impone a tutti i nodi dell'insieme *unique* di essere simulati al massimo da un solo nodo in H: nel caso in cui $\mathcal{U} = V_G$, la relazione ϕ sarà dunque una funzione parziale. In particolare:

- (SL 1) impone che ogni elemento di $\mathcal{M}$ compaia nella simulazione lasca;
- (SL 2) impone che la simulazione lasca, ridotta all'insieme unique, sia una funzione parziale;
- (SL 3) impone che la simulazione lasca, sia iniettiva rispetto ai nodi nell'insieme exclusive $\mathcal{E}$, ovvero che vertici in V_H in relazione con un elemento di $\mathcal{E}$ non possano essere in relazione con nessun altro elemento di V_G.

Condizione (SL 4) indica come un arco $e \in E_G$ del guest può essere in relazione con un arco $e' \in E_H$ solamente se sorgente e terminazione di e sono in relazione rispettivamente con sorgente e terminazione di e', ed i due archi hanno la stessa etichetta.

Condizione (SL 5) introduce il concetto di *selezione* di insiemi di archi: ad ogni coppia $(u, v) \in \phi$ deve corrispondere la selezione di almeno un insieme in $\mathcal{C}(u)$, dove un insieme è detto selezionato se tutti i suoi archi sono mappati in η con archi la cui sorgente è il nodo v. La seconda parte di (SL 5) indica invece che ogni coppia di archi in η è frutto di una selezione. Per capire meglio questo vincolo, prendiamo in considerazione l'esempio di Figura 2.8, prestando attenzione al nodo u ed alla porzione di simulazione lasca a lui associata: dato che $(u, x) \in \phi$, deve esistere un insieme di $\mathcal{C}(u) = \{\{(u, a, u)\}, \{(u, a, m)\}\}$ selezionato rispetto ad x. Tale insieme è $\{(u, a, u)\}$ e la selezione porta l'arco $(u, a, u) \in E_G$ ad essere associato, in η, a $(x, a, y) \in E_H$. Condizione (SL 4) impone quindi che, oltre a x, anche y simuli u. Condizione (SL 5) richiede quindi la selezione di un insieme in $\mathcal{C}(u)$ rispetto ad y: questa volta viene selezionato l'insieme contenente l'arco (u, a, m), che in η risulta dunque in relazione con $(y, a, w) \in E_H$. In Capitolo 3 approfondiremo l'importanza della condizione (SL 5) mostrando come questa sia una generalizzazione del concetto di simulazione definito nel problema di *simulazione di grafo*.

Infine, Condizione (SL 6) può essere vista come una generalizzazione della condizione d'accettazione per automi a stati finiti — rispetto al problema di pattern matching di espressioni regolari —, dove il ruolo degli stati finali è compiuto dai nodi dell'insieme *must*[3]. Sia $(\phi, \eta) \in \mathbb{S}^{G \to H}$ una simulazione lasca; dato un nodo $u \in E_G$ per il quale esiste un cammino terminante con un nodo $w \in \mathcal{M}$, Condizione (SL 6)

[3]Questa analogia, così come il problema di pattern matching per espressioni regolari, verrà meglio formalizzata in Sezione 3.3

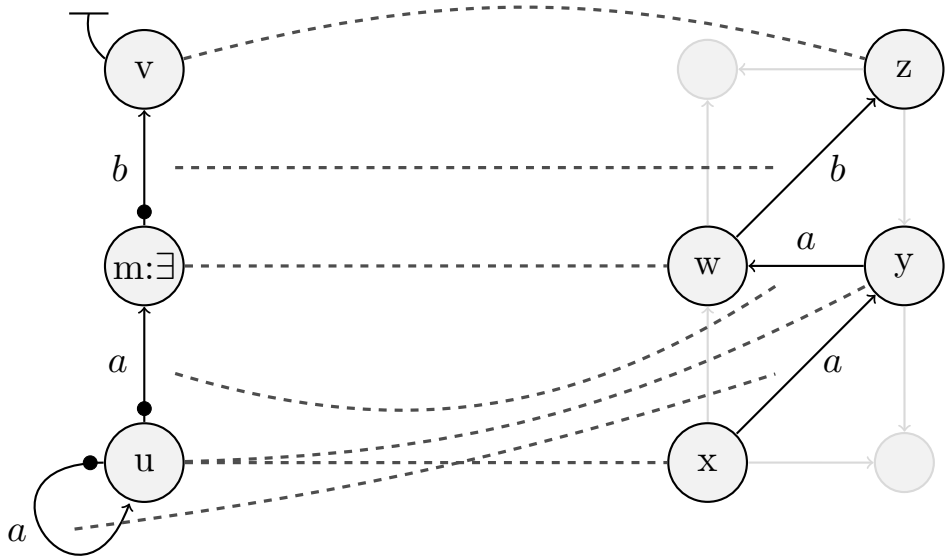


Figura 2.8: Un grafo guest (a sinistra) e un grafo host. Una simulazione lasca tra guest e host è evidenziata.

richiede che ad ogni occorrenza $(u, v) \in \phi$ corrisponda, tramite la simulazione lasca, un cammino nell'host con sorgente v e terminazione un nodo che simula w. Si noti l'utilizzo della funzione di raggiungibilità duale Δ_η, la quale garantisce l'uguaglianza tra il cammino cercato nell'host ed uno dei cammini di $\mathbb{P}_G(u, w)$.

In ambito di simulazioni su grafi, importanti risultati sono dati dalla definizione di un ordine sul dominio delle simulazioni e dall'individuazione della più grande simulazione, ottenuta per unione di tutte le simulazioni, chiamata *similarità* [17, 21, 36]. In generale, questi risultati non sono possibili nel caso delle simulazioni lasche in quanto la loro unione — definita come unione point-wise di relazioni $(\phi, \eta) \cup (\phi', \eta') \triangleq (\phi \cup \phi', \eta \cup \eta')$ — può violare i vincoli delle condizioni (SL 2) e (SL 3). Vale tuttavia il seguente risultato:

Proposizione 2.9 (Massima simulazione lasca)**.** *Sia $G = (\Sigma, V_G, E_G, \mathcal{M}, \mathcal{U}, \mathcal{E}, \mathcal{C})$ un guest e sia H un host. Se $\mathcal{U} = \mathcal{E} = \emptyset$, allora $\bigcup \mathbb{S}^{G \to H}$ è una simulazione lasca.*

Esempio 2. Riprendiamo nuovamente l'host ed il guest di Figura 2.8 per mostrare ora un esempio di simulazione lasca. Come già descritto in Esempio 1, il guest è tale che $\mathcal{M} = \{m\}$, $\mathcal{U} = \mathcal{E} = \emptyset$ e la funzione di scelta è definita come

$$\mathcal{C} = \lambda x.\{\{e\} \mid e \in out(x)\} \cup \{\emptyset \mid out(x) = \emptyset\}$$

Sia $(\phi, \eta) \in \mathbb{S}^{G \to H}$ una simulazione lasca. Dalla definizione di $\mathcal{C}$, Condizione (SL 5) fa sì che, per $n \in \{u, m\}$ e per ogni nodo $w \in \phi(n) \subseteq V_H$, deve esistere almeno una coppia di archi $(e, e') \in \eta$ tale che $s(e) = n$ e $s(e') = w$. Inoltre, essendo $m \in \mathcal{M}$, si ha che m deve apparire in ϕ e, per quanto appena detto riguardo (SL 5), per ogni nodo in $\phi(m) \subseteq V_H$ deve esistere un arco uscente etichettato con b la

cui terminazione è appartenente a $\phi(v)$. Infine, nel caso in cui u sia simulato da un nodo di H, la condizione (SL 6) assicura che per ogni sua occorrenza esisterà una coppia di cammini (ρ, ρ'), uno nel guest e uno nell'host, di soli edge accoppiati in η e tali che $t(\rho) = m$.

Da questo deduciamo quindi che ogni simulazione lasca di questo guest rappresenta, nell'host, cammini non vuoti $(e_0, e_1, \ldots, e_n)$ arbitrariamente lunghi tali che $\forall i < n\ \sigma(e_i) = a$ e $\sigma(e_n) = b$. Il guest in figura è quindi paragonabile al matching su grafo dell'espressione regolare $a^\star b$, *i.e.* da uno stato $w \in V_H$ inizierà un cammino ρ tale che $\sigma(\rho)$ appartiene al linguaggio generato da $a^\star b$ se e solo se esiste una simulazione lasca (ϕ, η) tale che $w \in \phi(u) \cup \phi(m)$.

L'esempio appena illustrato mostra come esista una relazione tra simulazioni lasche e altri strumenti per la verifica di proprietà di sistema — come, per l'appunto, i linguaggi regolari. Queste relazioni sono collegate alla computazione di $\mathbb{S}^{G \to H}$, che sarà dunque l'argomento centrale di questo lavoro. Al fine di studiare tale problema, il prossimo capitolo sarà dedicato al confronto e alla risoluzione di problemi noti in letteratura in termini di simulazioni lasche, al fine anche di inquadrare meglio il potere espressivo di queste ultime.

3

Simulazioni Lasche: Applicazioni

Come descritto in Capitolo 2, le simulazioni lasche ricadono nell'ambito del pattern matching su grafi. La letteratura riguardante questo campo dell'informatica è molto vasta, così come vasto è il panorama di nozioni definite per distinguere particolari famiglie di pattern e gli algoritmi utilizzati per computarle [18, 21, 35, 41, 42]. Risulta quindi necessario motivare la decisione di definire una nuova nozione di similarità tra grafi. Alla difesa di questa scelta è dedicato questo capitolo, articolato principalmente in due parti.

Nella prima parte verranno ridotti tre problemi noti — isomorfismo di sottografo, simulazione di grafo e pattern matching di linguaggi regolari — alle simulazioni lasche. Una importante caratteristica di queste riduzioni è che consistono unicamente in traduzioni da query di uno di questi problemi in guests per simulazioni lasche. Più precisamente, le query vengono decorate aggiungendo un'opportuna funzione di scelta e specifici insiemi must, unique ed exclusive. Successivamente si mostra come queste decorazioni rendano la definizione delle simulazione lasche equivalente a quella della nozione considerata. Queste traduzioni inoltre, grazie alla notazione grafica dei guest, mantengono inalterata l'espressività ed intuitività delle query.

La seconda parte è invece dedicata all'analisi di alcune proprietà delle simulazioni lasche. In particolare verranno definite due operazioni sui loro guests e mostrato come queste possano essere utilizzate per comporre facilmente query provenienti da diverse nozioni — *e.g.* una query per linguaggi regolari con una per isomorfismo di sottografo — in un unico guest. Infine verrà mostrato un esempio completo dove queste operazioni vengono utilizzate per generare guests in grado di verificare l'isomorfismo di sottografo per grafi decorati con linguaggi regolari sugli archi.

3.1 Isomorfismo di Sottografo

Iniziamo col confrontare le simulazioni lasche con il problema di trovare un isomorfismo di sottografo; ovvero determinare se un grafo (host) contenga un sottografo isomorfo ad un secondo grafo (query). Questo problema ha importanti applicazioni nella progettazione di reti [27], nel settore della bioinformatica ed in quello della chemioinformatica — dove viene usato per l'analisi delle interazioni tra molecole [1,7]. Nella seguente definizione, il problema è esteso per i multigrafi diretti etichettati introdotti in Sezione 2.1.

Definizione 3.1 (Isomorfismo di sottografo)**.** *Siano* $H = (\Sigma, V_H, E_H)$ *e* $Q = (\Sigma, V_Q, E_Q)$ *due multigrafi diretti etichettati, chiamati rispettivamente* host *e* query*. Esiste un sottografo di* H *isomorfo a* Q *qualora*

Figura 3.2: Una query per isomorfismo di sottografo (a sinistra) e la sua traduzione in un guest per simulazioni lasche (a destra).

esista una coppia di iniezioni $\phi_{SG} : V_Q \hookrightarrow V_H$ *e* $\eta_{SG} : E_Q \hookrightarrow E_H$ *tali che, per ogni* $e \in E_Q$ *valgono le tre seguenti condizioni:*

$$\begin{aligned} \phi_{SG} \circ s(e) &= s \circ \eta_{SG}(e) \\ \phi_{SG} \circ t(e) &= t \circ \eta_{SG}(e) \\ \sigma(e) &= \sigma \circ \eta_{SG}(e) \end{aligned}$$

L'isomorfismo di sottografo è mostrato essere NP-completo da S. A. Cook, tramite riduzione a 3-SAT [11]. La sua complessità e la sua importanza a livello applicativo l'hanno reso uno dei problemi più studiati dal punto di vista algoritmico [7, 12, 42]. Vogliamo ora mostrare come, tramite guests opportunamente definiti, sia possibile capire se esiste un isomorfismo di sottografo — ovvero, risolvere un problema decisionale booleano — utilizzando le simulazioni lasche.

Proposizione 3.3 (L'isomorfismo di sottografo è verificabile tramite simulazioni lasche)**.** *Siano* $H = (\Sigma, V_H, E_H)$ *e* $Q = (\Sigma, V_Q, E_Q)$ *rispettivamente un host ed una query. Sia inoltre* G *un guest tale che*

$$\begin{aligned} G &= (\Sigma, V_Q, E_Q, \mathcal{M}, \mathcal{U}, \mathcal{E}, \mathcal{C}) \\ \mathcal{M} = \mathcal{U} = \mathcal{E} &\triangleq V_Q \\ \mathcal{C} &\triangleq \lambda v.\{out(v)\} \end{aligned}$$

Esiste un sottografo di H *isomorfo a* Q *se e solo se esiste una simulazione lasca di* G *in* H*.*

Dimostrazione. La dimostrazione viene fatta mostrando come l'aggiunta delle definizioni di $\mathcal{M}$, $\mathcal{U}$, $\mathcal{E}$ e $\mathcal{C}$ alla definizione di simulazione lasca porti quest'ultima ad essere equivalente a Definizione 3.1. Per semplificare l'esposizione, sia $(\phi, \eta) \in \mathbb{S}^{G \to H}$ una simulazione lasca. Se $\mathcal{M} = \mathcal{U} = \mathcal{E} = V_Q$ allora le condizioni (SL 1), (SL 2) e (SL 3) impongono che ϕ sia una funzione totale iniettiva. Inoltre, la condizione (SL 5), insieme alla definizione della funzione di scelta $\mathcal{C}$, impone che ogni arco debba essere mappato: questa condizione, insieme all'iniettività di ϕ e la condizione (SL 4), rende anche ψ una funzione totale iniettiva. Infine, Condizione (SL 4) è equivalente alle condizioni di buona formazione delle due iniezioni di Definizione 3.1. Possiamo concludere quindi che, attraverso le condizioni aggiuntive

dovute alla definizione di $\mathcal{M}$, $\mathcal{U}$, $\mathcal{E}$ e $\mathcal{C}$, Definizione 2.7 risulta equivalente a Definizione 3.1. Inoltre, se $(\phi, \eta) \in \mathbb{S}^{G \to H}$ allora ϕ e η sono esattamente le due iniezioni che descrivono l'isomorfismo. □

Nota. Nella dimostrazione precedente non è stato fatto alcun accenno alla condizione (SL 6). Questo perché nel caso in cui $\mathcal{C} = \lambda v.\{\text{out}(v)\}$, Condizione (SL 5) impone che, dato un elemento $(u, v) \in \phi$, tutti gli archi in in out(u) appaiano in η e siano associati con archi di out(v). Questa proprietà, insieme alla condizione (SL 4), garantisce che (SL 6) sia sempre rispettata.

Esempio 3. Figura 3.2 mostra una query per isomorfismo di sottografo e la sua corrispondente traduzione in un guest per simulazioni lasche. Si noti come la traduzione non modifichi la struttura del grafo ma compia unicamente una decorazione dei vari nodi del grafo, mantenendo così pressoché inalterata l'espressività della query.

Abbiamo così dimostrato che le simulazioni lasche sono espressive almeno quanto il problema di isomorfismo di sottografo. Questo è un primo importante bound alla complessità del calcolo di simulazioni lasche: è un problema NP-difficile.

3.2 Simulazione di Grafo

Passiamo ora a considerare, utilizzando le simulazioni lasche, il problema di trovare una *simulazione di grafo*. Questo tipo di relazione tra grafi viene utilizzato nell'analisi di database sociali — come ad esempio analisi delle relazioni tra utenti di un social network [15] —, in bioinformatica e urbanistica [17].

Definizione 3.4 (Simulazione di grafo)**.** *Siano $H = (\Sigma, V_H, E_H)$ e $Q = (\Sigma, V_Q, E_Q)$ due multigrafi diretti etichettati, chiamati rispettivamente* host *e* query*. Esiste una* simulazione di grafo *di Q in H se e solo se esiste una relazione $\mathcal{R} \subseteq V_Q \times V_H$ tale che*

- *per ogni nodo $u \in V_Q$ esiste un nodo $v \in V_H$ tale che $(u, v) \in \mathcal{R}$;*
- *per ogni coppia $(u, v) \in \mathcal{R}$ ed ogni arco $e \in out(u)$ esiste un arco $e' \in out(v)$ tale che $\sigma(e) = \sigma(e')$ e $(t(e), t(e')) \in \mathcal{R}$.*

Il problema di computare una simulazione di grafo è mostrato essere polinomiale grazie all'algoritmo definito da B. Bloom e R. Paige in [6] e successivamente ripreso da M. R. Henzinger, T. A. Henzinger e P. W. Kopke in [21] — dove viene mostrato essere computabile in $O((|V_H| + |V_Q|)(|E_H| + |E_Q|))$ tempo. Oltre alla loro facilità, le simulazioni di grafo permettono di verificare proprietà interessanti. Si prenda per esempio la query di Figura 3.5: una simulazione di questo grafo corrisponde, nel host, a cammini di sole a dove ogni nodo del cammino è anche connesso ad un altro nodo, tramite un arco etichettato con b. A differenza dell'isomorfismo di sottografo le simulazioni di grafo permettono quindi di cercare facilmente pattern con un numero arbitrario di nodi.

Queste due caratteristiche — facilità del problema ed il tipo di proprietà verificabili — hanno portato le simulazioni su grafo ad essere utilizzate in bioinformatica, nell'analisi di rete e, sopratutto, nel controllo di pattern in grafi sociali — e.g. i database utenti dei maggiori social network — anche grazie ad una riformulazione dell'algoritmo di B. Bloom e R. Paige in ambito distribuito [17].

Figura 3.5: Una query per simulazione di grafo (a sinistra) e la sua traduzione in un guest per simulazioni lasche (a destra).

Come fatto per l'isomorfismo di sottografo, vogliamo ora mostrare come guests opportunamente definiti permettano di risolvere il problema decisionale di simulazioni di grafo utilizzando le simulazioni lasche.

Proposizione 3.6 (Le simulazioni di grafo sono verificabili tramite simulazioni lasche)**.** *Siano* $H = (\Sigma, V_H, E_H)$ *e* $Q = (\Sigma, V_Q, E_Q)$ *rispettivamente un host e una query. Sia inoltre* $G = (\Sigma, V_Q, E_Q, \mathcal{M}, \mathcal{U}, \mathcal{E}, \mathcal{C})$ *un guest tale che*

$$\begin{aligned} \mathcal{M} &\triangleq V_Q \\ \mathcal{U} = \mathcal{E} &\triangleq \emptyset \\ \mathcal{C} &\triangleq \lambda v.\{out(v)\} \end{aligned}$$

Esiste una simulazione di grafo di Q *in* H *se e solo se esiste una simulazione lasca di* G *in* H*.*

Dimostrazione. Similmente a quanto fatto per Proposizione 3.3, è facile vedere come, aggiungendo a Definizione 2.7 le condizioni dovute alle definizioni di $\mathcal{M}$, $\mathcal{U}$, $\mathcal{E}$ e $\mathcal{C}$, otteniamo Definizione 3.4. Sia $(\phi, \eta) \in \mathbb{S}^{G \to H}$. Se $\mathcal{M} = V_Q$ allora (SL 1) impone l'appartenenza di tutti i nodi del guest alla prima componente di ϕ. Condizione (SL 1) risulta quindi equivalente alla prima condizione di Definizione 3.4. Inoltre, $\mathcal{C} = \lambda v.\{\text{out}(v)\}$ porta le due condizioni (SL 4) e (SL 5) ad essere equivalenti alla seconda condizione di Definizione 3.4. Infine, $\mathcal{U} = \mathcal{E} = \emptyset$ rende sempre verificate le condizioni (SL 2) e (SL 3). Definizione 2.7 risulta quindi equivalente a Definizione 3.4. Inoltre, se $(\phi, \eta) \in \mathbb{S}^{G \to H}$ allora ϕ è una simulazione di grafo di Q in H. □

Come accennato in Sezione 2.3, dalla dimostrazione appena vista si può notare come la seconda condizione di simulazione di grafo sia un caso particolare di Condizione (SL 5) — insieme alla condizione (SL 4) — delle simulazioni lasche. Inoltre, come spiegato nella nota al termine della dimostrazione di Proposizione 3.3, queste due condizioni garantiscono la validità di Condizione (SL 6). Figura 3.5 mostra il guest risultante dalla traduzione di una query per simulazione di grafo. Come per l'isomorfismo di sottografo, si noti come la traduzione non modifichi la struttura del grafo e mantenga inalterata l'espressività della query.

3.3 Linguaggi Regolari

I linguaggi regolari definiscono sequenze finite di lettere (chiamate *stringhe* o *parole*) prese da un alfabeto finito. Possono essere descritte tramite *espressioni regolari* o attraverso *automi a stati finiti* [22]. Sono largamente utilizzati in ambito del pattern matching, prevalentemente nell'elaborazione di stringhe e testi, anche se non mancano applicazioni anche nel pattern matching su grafi [3, 31].

In questa sezione ci restringiamo ai linguaggi regolari ϵ-free, ovvero privi della parola vuota ϵ. Questa restrizione, che semplifica notevolmente i risultati tra poco esposti, è piuttosto comune in ambito di pattern matching: nel caso si voglia cercare infatti, per esempio all'interno di un testo, una stringa appartenente ad un linguaggio regolare, l'appartenenza della parola vuota all'interno di questo fa sì che ogni posizione del testo sia una soluzione al problema di matching. Questa riduzione non rimuove dunque alcun caso significativo.

Definizione 3.7 (Linguaggi regolari ϵ-free)**.** *Sia Σ un alfabeto. L'insieme vuoto $\emptyset$ è un* linguaggio regolare ϵ-free*. Per ogni $a \in \Sigma$, l'insieme singoletto $\{a\}$ è un* linguaggio regolare ϵ-free*. Se A e B sono due* linguaggi regolari ϵ-free*, allora lo sono anche le seguenti:*

$$A \cdot B \triangleq \{vw \mid v \in P \wedge w \in Q\}$$
$$A \mid B \triangleq A \cup B$$
$$A^+ \triangleq \bigcup_{n \in \mathbb{N}^+} A^n \qquad \text{dove } A^1 = A \text{ e } A^{n+1} = A \cdot A^n$$

In [44] è mostrato come ogni *linguaggio regolare* senza parola vuota ϵ può essere espresso con un linguaggio regolare ϵ-free.

Definiremo ora il problema del pattern matching per linguaggi regolari ϵ-free, per poi far vedere come può essere risolto utilizzando le simulazioni lasche. Per semplicità, ci restringiamo ai linguaggi non vuoti. Esattamente come per la prima restrizione, il linguaggio vuoto $\emptyset$ non rappresenta un caso significativo.

Definizione 3.8 (Pattern matching di linguaggi regolari ϵ-free non vuote (RE-PM))**.** *Siano $H = (\Sigma, V_H, E_H)$ e $\mathcal{L}$ rispettivamente un host ed un linguaggio regolare ϵ-free tale che $\mathcal{L} \neq \emptyset$. Un* match *di $\mathcal{L}$ in H è un cammino $\rho \in \mathbb{P}_H$ tale che $\sigma(\rho) \in \mathcal{L}$.*

Per ridurre il problema RE-PM alla verifica di simulazioni lasche introduciamo gli automi a stati finiti non deterministici (*NFA*). Ricordiamo inoltre che gli NFA hanno la stessa espressività dei linguaggi regolari [41].

Definizione 3.9 (Automa a stati finiti non deterministico (NFA))**.** *Un NFA è una quintupla $N = (\Sigma, Q, \Delta, q_0, F)$ composta da*

- *un alfabeto Σ*
- *un insieme finito di stati Q*
- *una* funzione di transizione $\Delta : Q \times \Sigma \to \mathcal{P}(Q)$
- *uno* stato iniziale $q_0 \in Q$

- *un insieme di stati* accettanti *(o* finali*)* $F \subseteq Q$.

Diremo che un NFA $N = (\Sigma, Q, \Delta, q_0, F)$ accetta *una parola* $w = a_0, a_1, \ldots, a_n \in \Sigma^\star$ *se esiste una sequenza di stati* $r_0, r_1, \ldots, r_{n+1}$ *di* Q *tale che:*

- $r_0 = q_0$
- *per ogni* $i = 0, \ldots, n$, $r_{i+1} \in \Delta(r_i, a_i)$
- $r_{n+1} \in F$

Ogni linguaggio regolare non vuoto e senza ϵ può essere tradotto in un NFA dove lo stato iniziale non ha archi entranti, esiste un unico stato finale e quest'ultimo è privo di archi uscenti. Questo può essere mostrato partendo dalla costruzione di K. Thompson [41] che partendo da espressioni regolari darà luogo, in tempo lineare, ad un NFA $N = (\Sigma, Q, \Delta, q_0, \{f\})$ con un solo stato finale differente da quello iniziale (dato che ϵ non appartiene al linguaggio). A partire da questa costruzione, l'automa da noi cercato può essere definito come $N' = (\Sigma, Q \cup \{q_0', f'\}, \Delta', q_0', \{f'\})$ dove

- per ogni $a \in \Sigma$, $\Delta'(q_0', a) \triangleq \Delta'(q_0, a)$ e $\Delta'(f', a) \triangleq \emptyset$
- per ogni $q \in Q$ e per ogni $a \in \Sigma$, $\Delta'(q, a) \triangleq \Delta(q, a) \cup \{f' \mid f \in \Delta(q, a)\}$

Grazie all'introduzione degli NFA ricaviamo facilmente il seguente risultato.

Teorema 3.10 (Decidere RE-PM è polinomiale)**.** *Siano* $H = (\Sigma, V_H, E_H)$ *e* $\mathcal{L}$ *rispettivamente un host ed un linguaggio regolare* ϵ*-free tale che* $\mathcal{L} \neq \emptyset$*. Il problema decisionale legato a RE-PM, ovvero decidere se esiste un cammino* $\rho \in \mathbb{P}_H$ *tale che* $\sigma(\rho) \in \mathcal{L}$*, è polinomiale.*

Dimostrazione. Sia $N = (\Sigma, Q, \Delta, q_0, \{f\})$ un NFA il cui stato iniziale non ha archi entranti, l'unico stato finale è privo di archi uscenti e tale da accettare esattamente il linguaggio $\mathcal{L}$, secondo la costruzione appena vista. Per ogni $v \in V_H$, si prenda l'NFA

$$\begin{aligned} \mathrm{NH}_v &= (\Sigma,\ Q \times V_H \cup \{\star\},\ \Delta',\ (q_0, v),\ \{f\} \times V_H) \\ \Delta'((q,u),a) &\triangleq \{(q', u') \mid q' \in \Delta(q,a) \wedge (u, a, u') \in E_H\} \cup \{\star\} \\ \Delta'(\star, a) &\triangleq \{\star\} \end{aligned}$$

ottenuto per prodotto cartesiano degli stati di N con quelli di H, dove la funzione di transizione Δ' segue sia la funzione Δ che gli archi E_H, ovvero

$$(q', u') \in \Delta'((q,u),a) \iff q' \in \Delta(q,a) \wedge (u, a, u') \in E_H$$

e dove $\star$ è un nuovo stato pozzo. Sia $w = a_0, a_1, \ldots, a_n \in \Sigma^\star$ una parola accettata da NH_v. Per la definizione di accettazione di un automa nondeterministico, deve esistere una sequenza di stati $r_0, r_1, \ldots, r_{n+1}$ tali che

- $r_0 = (q_0, v)$
- per ogni $i = 0, \ldots, n$, $r_{i+1} \in \Delta'(r_i, a_i)$

- $r_{n+1} \in \{f\} \times V_H$

Per quanto detto su Δ' e dalla definizione dell'insieme di stati accettanti di NH_v, seguono i due seguenti risultati: w è accettata da N, rispetto alla sequenza di stati $\pi_1(r_0), \pi_1(r_1), \ldots, \pi_1(r_{n+1})$. Analogamente, il cammino

$$\rho = (\pi_2(r_0), a_0, \pi_2(r_1)), (\pi_2(r_1), a_1, \pi_2(r_2)), \ldots, (\pi_2(r_n), a_n, \pi_2(r_{n+1})) \in \mathbb{P}_H$$

è tale che $\sigma(\rho) = w$. Si noti infine che la dimensione dei $|V_H|$ automi generati è $\mathcal{O}(|Q||V_H|+|Q|^2|\Sigma||E_H|)$. Il problema di decidere se esista un cammino $\rho \in \mathbb{P}_H$ tale che $\sigma(\rho) \in \mathcal{L}$ può quindi essere ricondotto al problema del vuoto di $|V_H|$ automi dalle dimensioni polinomiali rispetto a N e H, dove tale cammino esiste se e solo se uno di questi automi accetta un linguaggio diverso da $\emptyset$. Il problema decisionale di RE-PM risulta quindi essere polinomiale, essendo polinomiale la complessità del problema del vuoto per NFA [24] — il cui algoritmo verifica la raggiungibilità di uno degli stati finali dallo stato iniziale. □

Passiamo ora a mostrare come guests opportunamente definiti permettano di risolvere il problema decisionale di *RE-PM* utilizzando le simulazioni lasche.

Proposizione 3.11 (RE-PM è verificabile tramite simulazioni lasche)**.** *Sia $H = (\Sigma, V_H, E_H)$ un host. Sia $N = (\Sigma, Q, \Delta, q_0, \{f\})$ un NFA dove lo stato iniziale q_0 non ha archi entranti e l'unico stato finale f è privo di archi uscenti. Sia inoltre $G = (\Sigma, Q, E, \mathcal{M}, \mathcal{U}, \mathcal{E}, \mathcal{C})$ un guest tale che*

$$\begin{aligned} E &\triangleq \{(p, a, q) \in Q \times \Sigma \times Q \mid q \in \Delta(p, a)\} \\ \mathcal{M} &\triangleq \{q_0, f\} \\ \mathcal{U} = \mathcal{E} &\triangleq \emptyset \\ \mathcal{C} &\triangleq \lambda v.\{\{e\} | e \in \mathrm{out}(v)\} \cup \{\emptyset \mid out(v) = \emptyset\} \end{aligned}$$

Esiste un cammino $\rho \in \mathbb{P}_H$ tale che $\sigma(\rho)$ è accettato da N, se e solo se esiste una simulazione lasca di G in H.

Dimostrazione. ($\Rightarrow$): se $\sigma(\rho) = a_0, a_1, \ldots, a_n$ è accettato da N allora, per la definizione data di G e dato che $q_0, f \in \mathcal{M}$, devono esistere due nodi $v, v' \in V_H$ tali che $(f, v') \in \Delta_\eta((q_0, v))$. Inoltre,

- Dalla definizione di accettazione per NFA, esiste un cammino $r_0, r_1, \ldots, r_{n+1}$ tale che $r_0 = q_0$, $r_n = f$ e $\forall i \in \{0, \ldots, n\}$ $r_{i+1} \in \Delta(r_i, a_i)$.
- Sia $(\phi, \eta) \in \mathbb{S}^{G \to H}$ una simulazione lasca. La funzione di scelta $\mathcal{C}$ è definita in modo tale che, per ogni coppia di nodi $(u, v) \in \phi$ con $u \neq f$, dobbiamo almeno una coppia $(e, e') \in \eta$ tale che $s(e) = u$ e $s(e') = v$. Questo è equivalente a compiere un passo di transizione nel NFA.

Possiamo quindi concludere che (SL 6) è verificata. Anche le altre condizioni di simulazione lasca sono banalmente verificate guardando alle definizioni di $\mathcal{M}$, $\mathcal{U}$, $\mathcal{E}$ e $\mathcal{C}$ — come fatto per Proposizioni 3.3 3.6. Esiste quindi una simulazione lasca da G in H.

($\Leftarrow$): se esiste una simulazione lasca (ϕ, η) di G in H, allora — per la condizione (SL 6) — in (ϕ, η) deve esistere $v, v' \in V_H$ s.t. $(f, v') \in \Delta_\eta((q_0, v))$, dato che $q_0, f \in \mathcal{M}$. Inoltre, dalla definizione di E, il

Figura 3.12: L'NFA rappresentante l'espressione regolare $(ab)^+$ (a sinistra) e la sua traduzione in un guest per simulazioni lasche (a destra).

guest G rispetta la funzione di transizione Δ del NFA. Dunque, il cammino duale iniziante in (q_0, v) e terminante in (f, v') deve produrre una parola accettata da N. □

Figura 3.12 mostra il guest risultato dalla traduzione di un NFA accettante il linguaggio regolare generato dall'espressione $(ab)^+$. Ancora una volta, si noti la semplicità della traduzione, la quale non modifica la struttura del grafo dell'automa mantenendone inalterata l'espressività.

3.4 Composizione e Quozientazione di Guests

Dai risultati delle sezioni precedenti si può notare come le simulazioni lasche siano computazionalmente complesse almeno quanto il problema di *isomorfismo di sottografo* e più complesse, per esempio, delle *simulazione di grafo*. Non c'è dunque alcun motivo per utilizzarle nel caso in cui si vogliano computare soluzioni a questi problemi. Esse trovano tuttavia applicazione nel caso in cui si vogliano comporre query provenienti da diverse nozioni. Per esempio, potremmo voler cercare un cammino in un grafo, rappresentabile tramite una query di *RE-PM* — Definizione 3.8 —, terminante in un nodo dal quale parte uno specifico grafo rappresentato tramite query di isomorfismo di sottografo — Definizione 3.1. Le simulazioni lasche permettono di generare query di questo tipo attraverso due operazioni, composizione e quozientazione, presentate in questa sezione.

Intuitivamente, dati due guests ed una relazione $\mathcal{R}$ sui vertici di questi, l'operazione di composizione connette i due guests mettendo in comune la parte in $\mathcal{R}$ e lasciando inalterata la restante parte dei due guest.

Definizione 3.13 (Composizione di guest)**.** *Siano G ed F due guest.*

$$G = (\Sigma, V_G, E_G, \mathcal{M}_G, \mathcal{U}_G, \mathcal{E}_G, \mathcal{C}_G) \qquad F = (\Sigma, V_F, E_F, \mathcal{M}_F, \mathcal{U}_F, \mathcal{E}_F, \mathcal{C}_F)$$

Sia inoltre $\mathcal{R} \subseteq V_G \times V_F$ una relazione tra vertici di G e F. La composizione di G e F rispetto a $\mathcal{R}$ è

definita come

$$
\begin{aligned}
G \bigsqcup_{\mathcal{R}} F &\triangleq (\Sigma, V_\sqcup, E_\sqcup, \mathcal{M}_\sqcup, \mathcal{U}_\sqcup, \mathcal{E}_\sqcup, \mathcal{C}_\sqcup) \\
V_\sqcup &\triangleq \mathcal{R} \cup \{(v, \star) \mid v \in V_G \setminus \pi_1(\mathcal{R})\} \cup \{(\star, v) \mid v \in V_F \setminus \pi_2(\mathcal{R})\} \\
E_\sqcup &\triangleq \{((u,v), a, (u',v')) \in V_\sqcup \times \Sigma \times V_\sqcup \mid (u,a,u') \in E_G \ \vee \ (v,a,v') \in E_F\} \\
\mathcal{M}_\sqcup &\triangleq \{(u,v) \in V_\sqcup \mid u \in \mathcal{M}_G \ \vee \ v \in \mathcal{M}_F\} \\
\mathcal{U}_\sqcup &\triangleq \{(u,v) \in V_\sqcup \mid u \in \mathcal{U}_G \ \vee \ v \in \mathcal{U}_F\} \\
\mathcal{E}_\sqcup &\triangleq \{(u,v) \in V_\sqcup \mid u \in \mathcal{E}_G \ \vee \ v \in \mathcal{E}_F\}
\end{aligned}
$$

$$
\mathcal{C}_\sqcup(u,v) \triangleq \left\{ \left\{ \begin{array}{l} ((u,v), a_1, (u_1, v_1')) \\ \dots \\ ((u,v), a_n, (u_n, v_n')) \end{array} \right\} \cup \left\{ \begin{array}{l} ((u,v), b_1, (u_1', v_1)) \\ \dots \\ ((u,v), b_m, (u_m', v_m)) \end{array} \right\} \;\middle|\; \begin{array}{l} \{(u, a_1, u_1), \dots, (u, a_n, u_n)\} \in \mathcal{C}_G(u) \\ \{(v, b_1, v_1), \dots, (v, b_m, v_m)\} \in \mathcal{C}_F(v) \end{array} \right\}
$$

dove $V_\sqcup \subseteq (V_G \cup \{\star\}) \times (V_F \cup \{\star\})$ *per un nuovo elemento* $\star \notin V_G \cup V_F$ *ed estendiamo le funzioni di scelta* $\mathcal{C}_G$ *e* $\mathcal{C}_H$ *ponendo* $\mathcal{C}_G(\star) = \mathcal{C}_F(\star) = \{\emptyset\}$*, in modo da definire correttamente* $\mathcal{C}_\sqcup$*.*

Esempio 4. Vediamo l'effetto della composizione di due guests tramite l'esempio riportato in Figura 3.14. In alto sono rappresentati due guest: il primo (a sinistra), già visto in Figura 2.8, è equivalente all'espressione regolare $a^\star b$; il secondo è invece la rappresentazione di una query per isomorfismo su grafi. La figura evidenzia inoltre la relazione $\mathcal{R}$ tra i vertici dei due grafi: più precisamente, nel primo guest, il vertice denominato v è messo in relazione con tutti i vertici del secondo guest. Il risultato della composizione è riportato nell'ultimo grafo della figura: seguendo la definizione di $V_\sqcup$, le coppie di vertici in $\mathcal{R}$ diventano nodi del guest risultante, mentre i singoli vertici in $\pi_1(\mathcal{R})$ o $\pi_2(\mathcal{R})$ vengono rimossi. Gli archi uscenti ed entranti da nodi rimossi sono reintrodotti sui nodi di $\mathcal{R}$. I tre insiemi must, unique e exclusive vengono uniti opportunamente: dato un nodo $(u,v) \in V_\sqcup$, esso sarà nel insieme $\mathcal{M}_\sqcup$, $\mathcal{U}_\sqcup$ o $\mathcal{E}_\sqcup$ se almeno uno tra v e u è rispettivamente in un insieme must, unique o exclusive. La funzione di scelta di un vertice (u,v) sarà tale che ogni suo insieme rappresenti una scelta per u in $\mathcal{C}_G(u)$ ed una scelta per v in $\mathcal{C}_F(v)$. Il risultato è un guest le cui simulazioni lasche rappresentano cammini di sole a terminanti in un nodo dal quale esce almeno una transizione etichettata con b che termina in un nodo appartenente all'isomorfismo di sottografo rappresentato dal guest in alto a destra.

Si noti come, qualora la relazione utilizzata per la composizione sia $\mathcal{R} = \emptyset$, si ottenga l'unione disgiunta dei due guest. D'altro canto, ponendo $\mathcal{R} = V_G \times V_F$ si ottiene una nozione di prodotto cartesiano per i guest. Passiamo ora a definire l'operatore di quozientazione che, data una relazione di equivalenza sui nodi di un guest, unisce tutti i nodi nella stessa classe d'equivalenza.

Definizione 3.15 (Quozientazione di guest)**.** *Sia* $G = (\Sigma, V, E, \mathcal{M}, \mathcal{U}, \mathcal{E}, \mathcal{C})$ *un guest e sia* $\sim \subseteq V \times V$ *una relazione di equivalenza sui nodi di* G*. Il quoziente di* G *rispetto a* R *è definito come il guest*

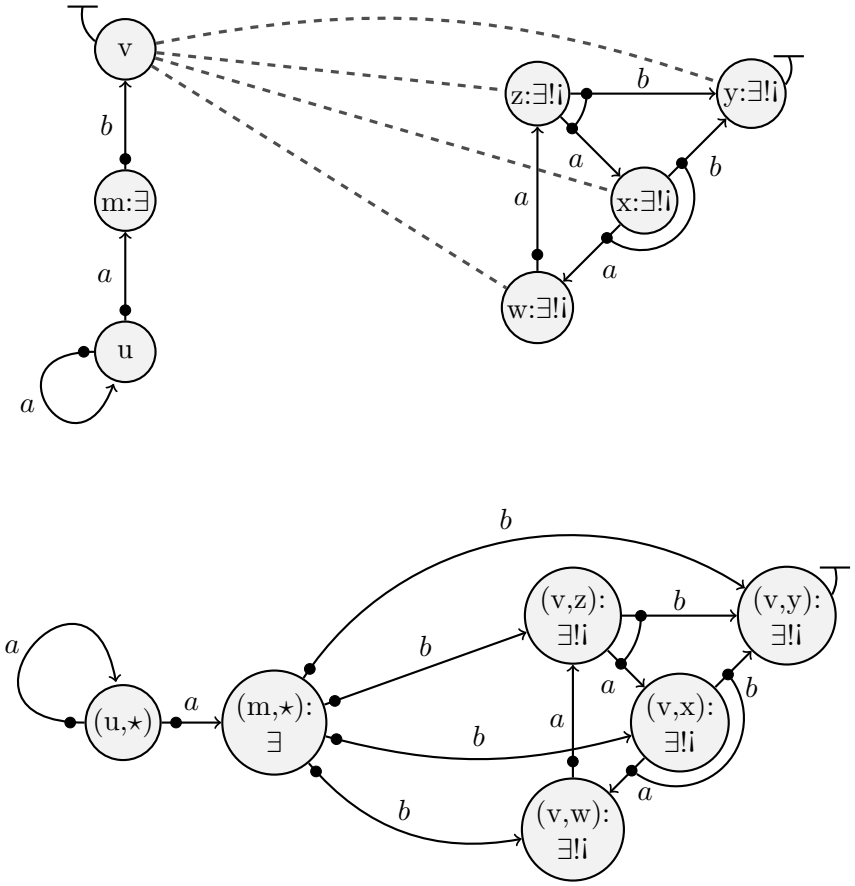


Figura 3.14: In alto, due guests ed una relazione $\mathcal{R}$ tra essi. In basso, il risultato della composizione dei due guests utilizzando $\mathcal{R}$.

$G/_\sim \triangleq (\Sigma, V/_\sim, E', \mathcal{M}/_\sim, \mathcal{U}/_\sim, \mathcal{E}/_\sim, \mathcal{C}')$, *dove*

$$E' \triangleq \{([v]_\sim, a, [w]_\sim) \mid (v, a, w) \in E\}$$

$$\mathcal{C}' \triangleq \lambda r. \left\{ \bigcup_{\gamma \in \{\gamma_1, \ldots, \gamma_n\}} \{([v]_\sim, a, [w]_\sim) \mid (v, a, w) \in \gamma\} \;\middle|\; \begin{array}{l} \exists v_1, \ldots, v_n \in V \\ \quad r = \{v_1, \ldots, v_n\} \;\wedge \bigwedge_{i \in \{1,\ldots,n\}} \gamma_i \in \mathcal{C}(v_i) \end{array} \right\}$$

dove, dato un insieme $A \subseteq B$ ed una relazione d'equivalenza $\sim$ sugli elementi di B, $A/_\sim$ è l'insieme quoziente di A e dato $a \in A$, $[a]_\sim \in A/_\sim$ è l'elemento rappresentante un insieme della partizione indotta da $\sim$, ovvero $[a]_\sim = \{b \in A \mid b \sim a\}$.

Infine, considerato che i guests risultanti da operazioni di composizione e quozientazione hanno un insieme di nodi diversi dai guests di partenza, può essere utile introdurre una nozione di sostituzione dei nodi del guest. Intuitivamente, la sostituzione — che può essere vista come una rinomina — andrà a rimpiazzare uno o più nodi con nuovi vertici analoghi a quelli rimossi.

Definizione 3.16 (Sostituzione di nodi)**.** *Sia $G = (\Sigma, V, E, \mathcal{M}, \mathcal{U}, \mathcal{E}, \mathcal{C})$ un guest, A un insieme (di nuovi vertici), e sia $\theta : V \hookrightarrow A \cup V$ una funzione iniettiva. Definiamo il guest risultato della* sostituzione dei

nodi *di G secondo la funzione θ come* $G\theta = \{\Sigma, \theta(V), E\theta, \theta(\mathcal{M}), \theta(\mathcal{U}), \theta(\mathcal{E}), \mathcal{C}\theta\}$*, tale che*

$$E\theta \triangleq \{(\theta(v), a, \theta(w)) \mid (v, a, w) \in E\}$$

$$\mathcal{C}\theta \triangleq \lambda r. \left\{ \left\{ \begin{matrix} (r, a_1, \theta(w_1)) \\ (r, a_2, \theta(w_2)) \\ \dots \\ (r, a_n, \theta(w_n)) \end{matrix} \right\} \middle| \exists v \in V : \ \theta(v) = r \wedge \left\{ \begin{matrix} (v, a_1, w_1) \\ (v, a_2, w_2) \\ \dots \\ (v, a_n, w_n) \end{matrix} \right\} \in \mathcal{C}(v) \right\}$$

dove, data una funzione $f : X \to Y$ ed un insieme $X' \subseteq X$, indichiamo con $f(X')$ l'immagine di X' tramite f.

Grazie alle operazioni di composizione e quozientazione, oltre che ai risultati delle precedenti sezioni, possiamo utilizzare le simulazioni lasche per comporre query da diverse nozioni di similarità tra grafi. Nella prossima sezione vedremo un esempio completo dove questi concetti vengono applicati.

3.5 Grafi con decorazioni regolari

In letteratura si trovano numerosi studi sull'unione e la modifica di nozioni di similarità tra grafi, con lo scopo di comprendere casi più generali o di interesse per un certo ambito di ricerca. Tra questi studi compaiono diversi tentativi di mettere insieme morfismi di grafi con linguaggi regolari [3, 16]. In questi approcci, gli archi delle query sono etichettati con linguaggi regolari e devono essere mappati in cammini dell'host, in modo simile a quanto visto per i linguaggi regolari in Definizione 3.8. Utilizzeremo ora i risultati delle sezioni 3.3 e 3.4 per mostrare come definire un problema di questo tipo in termini di simulazioni lasche.

Definizione 3.17 (Grafo con decorazioni regolari)**.** *Un* grafo con decorazioni regolari $G = (\Sigma, V, E, \mathcal{L})$ *è una quadrupla composta da un alfabeto Σ, un insieme finito V di nodi, un insieme $E \subseteq V \times V$ di archi ed una funzione di etichettatura $\mathcal{L} : E \to RE_\Sigma$ che associa un linguaggio regolare ad ogni arco — dove $RE_\Sigma \subseteq \mathcal{P}(\Sigma^\star)$ denota l'insieme di tutti i linguaggi regolari su alfabeto Σ.*

Similmente a quanto fatto per Definizione 2.1, indichiamo con $s(e)$ e $t(e)$ rispettivamente la prima e la seconda proiezione di un arco $e \in E$.

Si noti come, in questo caso, lavoreremo con grafi e non multigrafi. Benché estendibili anche ai multigrafi, infatti, i risultati in questa sezione risulterebbero meno intuitivi e renderebbero difficile focalizzarci sugli aspetti chiave presentati — ovvero l'utilizzo delle operazioni definite in Sezione 3.4.

Definizione 3.18 (Isomorfismo di sottografo con decorazioni regolari (RE-SGISO))**.** *Sia $H = (\Sigma, V_H, E_H)$ un host e sia $Q = (\Sigma, V_Q, E_Q, \mathcal{L})$ un grafo con decorazioni regolari, denominato* query*. Esiste un* isomorfismo di sottografo con decorazioni regolari *di H rispetto a Q qualora esista una coppia di iniezioni $\phi : V_Q \hookrightarrow V_H$ e $\eta : E_Q \hookrightarrow \mathbb{P}_H$ tale che, per ogni $e \in E_Q$ valgono le seguenti condizioni:*

$$\phi \circ s(e) = s \circ \eta(e)$$
$$\phi \circ t(e) = t \circ \eta(e)$$
$$\sigma \circ \eta(e) \in \mathcal{L}(e)$$

Inoltre, imponiamo che gli archi appartenenti a cammini nell'immagine di η non possano avere come sorgente o terminazione elementi dell'immagine di ϕ, eccezione fatta per sorgente e terminazione dell'intero cammino; formalmente:

$$\forall (e_0, \ldots, e_n) \in \eta(E_Q)\ \forall i \in \{1, \ldots, n\}\ s(e_i) \notin \phi(V_Q)$$

dove ricordiamo $\eta(E_Q)$ e $\phi(V_Q)$ essere rispettivamente le immagini di E_Q tramite η e di V_Q tramite ϕ.

Si noti come η associ ogni arco $e \in V_Q$ con un cammino non vuoto dell'host: insieme alla condizione che $\sigma \circ \eta(e) \in \mathcal{L}(e)$, questa definizione ci permette di restringerci ai linguaggi regolari ϵ-free non vuoti. Passiamo ora a mostrare come definire un guest per simulazioni lasche che permetta di decidere il problema RE-SGISO, utilizzando le operazioni di composizione e quozientazione di Sezione 3.4.

Proposizione 3.19 (RE-SGISO è verificabile tramite simulazioni lasche)**.** *Siano $H = (\Sigma, V_H, E_H)$ e $Q = (\Sigma, V_Q, E_Q, \mathcal{L})$ rispettivamente un host ed una query per RE-SGISO. Esiste un guest G per simulazioni lasche tale che*

$$\textit{esiste un isomorfismo di sottografo con decorazioni regolari di } H \textit{ rispetto a } Q \iff \mathbb{S}^{G \to H} \neq \emptyset$$

Dimostrazione. Al fine di fornire un supporto grafico alle operazioni fatte durante questa dimostrazione, verrà utilizzata Figura 3.20 come esempio di riduzione da query di RE-SGISO a guest per simulazioni lasche. Il primo grafo di questa figura rappresenta una possibile query. In Sezione 3.3 è strato dimostrato come ogni linguaggio regolare ϵ-free diverso da vuoto $\mathcal{L}$ sia caratterizzabile con un guest per simulazioni lasche. Dato un arco $e \in E_Q$, denotiamo con $N_e = (\Sigma, V_e, q_e, \{f_e\})$ un *NFA* per $\mathcal{L}(e)$ dove lo stato iniziale q_e non ha archi entranti e l'unico stato finale f_e è privo di archi uscenti. Seguendo i risultati descritti in Sezione 3.3, è noto che un *NFA* con queste proprietà deve esistere. Sia G_e il guest risultato della traduzione di N_e, secondo quanto detto in Proposizione 3.11. Si noti come l'insieme dei nodi di G_e sia l'insieme V_e dei vertici del *NFA*. Consideriamo ora il seguente insieme di guest:

$$\mathcal{G} \triangleq \{G_e \mid e \in E_Q\}$$

Ogni guest $G_e \in \mathcal{G}$ rappresenta, oltre all'arco $e \in E_Q$, il linguaggio a lui associato. In G_e, così come per l'NFA, q_e non ha archi entranti e f_e è privo di archi uscenti. Inoltre, questi due nodi sono gli unici nell'insieme must di G_e. Si noti inoltre che, per ipotesi di linguaggio ϵ-free, $q_e \neq f_e$. Nel grafo Q, tuttavia, possono essere presenti dei self-loop — come per esempio l'arco (u, u) nel primo grafo di Figura 3.20. Per gestire questa eventualità possiamo utilizzare l'operatore di quozientazione, in modo tale da unire q_e a f_e. Per ogni guest $G_e \in \mathcal{G}$ introduciamo dunque una relazione di equivalenza $\sim_e \subseteq V_e \times V_e$ definita come chiusura riflessiva, simmetrica e transitiva di:

$$\begin{cases} \{(q_e, f_e)\} & \text{se } s(e) = t(e) \\ \emptyset & \text{altrimenti} \end{cases}$$

In altri termini, $G_e/_{\sim_e}$ è equivalente a G_e se $s(e) \neq t(e)$. In caso contrario, q_e e f_e vengono uniti in $G_e/_{\sim_e}$. Questo ci permette di gestire i casi di self-loop. La seconda illustrazione di Figura 3.20 mostra

i quattro guests risultanti dalla traduzione degli archi della query, dopo la quozientazione. Si noti come nel guest relativo al self-loop (u, u) vi sia solo un nodo nel insieme must: su questo guest, infatti, la quozientazione ha unito $q_{(u,u)}$ e $f_{(u,u)}$.

Una seconda proprietà da considerare è l'iniettività dei vertici di Q. Per introdurla, consideriamo il guest $F = (\Sigma, V_Q, \emptyset, V_Q, V_Q, V_Q, \lambda v.\{\emptyset\})$, composto unicamente dai vertici di Q, non connessi tra loro ed appartenenti ai tre insiemi must, unique e exclusive. Applichiamo ora l'operatore di composizione per comporre F ai vari guests di $\mathcal{G}$. Le composizioni verranno fatte sequenzialmente, sempre sul risultato dell'operazione precedente: supponiamo per questo motivo un ordinamento totale (qualsiasi) $\leq$ sugli elementi di $\mathcal{G}$ — o equivalentemente sugli archi di Q —, in modo da identificare univocamente la sequenza di operazioni da eseguire. Dato che il risultato della composizione è un guest con un nuovo insieme di vertici, per semplificare i passaggi, dopo ogni composizione verrà fatta una sostituzione che riporti i vertici di Q a far parte del guest. Dato un guest $G_e \in \mathcal{G}$, definiamo la relazione $R_e \triangleq \{(s(e), [q_e]_{\sim_e}), (t(e), [f_e]_{\sim_e})\} \subseteq V_Q \times V_e/_{\sim_e}$. La relazione R_e ha lo scopo di collegare in F i vertici $s(e)$ e $t(e)$ con il guest G_e che rappresenta l'arco e di Q. Definiamo il guest $G = (\Sigma, V, E, \mathcal{M}, \mathcal{U}, \mathcal{E}, \mathcal{C})$ come risultato della seguente computazione

$$G = \text{fold}_{\leq}(\lambda A.\lambda(B, e).(A \sqcup_{R_e} B)\theta,\ F,\ \{(G_e/_{\sim_e}, e) \mid G_e \in \mathcal{G}\})$$

dove la sostituzione θ e la funzione *fold* sono definite rispettivamente come

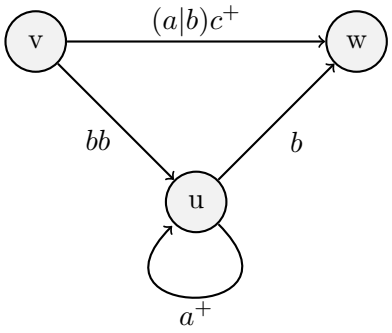
$$\theta = \lambda w. \begin{cases} v & \text{se } \exists u \in \bigcup_{e \in E_Q}(V_e) \cup \{\star\}\ w = (v, u)\ \wedge v \in V_Q \\ w & \text{altrimenti} \end{cases}$$

$$\text{fold}_{\leq}(f,\ A,\ S) = \begin{cases} A & \text{se } S = \emptyset \\ \text{fold}_{\leq}(f,\ f(A, B),\ S \setminus \{B\}) & \text{dove } B \text{ elemento minimo in } S \text{ rispetto a } \leq \end{cases}$$

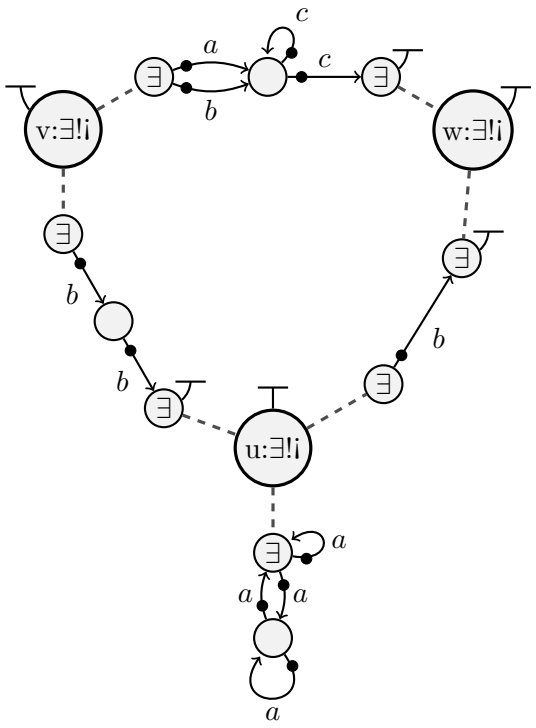
In particolare, la sostituzione θ ha il compito di rinominare i vertici che prima della composizione corrispondevano a nodi di V_Q, in modo che questi ricompaiano nel guest risultante. In Figura 3.20, la seconda illustrazione mostra il guest F e la relazione usata per effettuare la composizione tra F ed i vari guests di $\mathcal{G}$. Il risultato di tali operazioni è mostrato nella terza illustrazione.

Il guest G caratterizza Q. I vertici di V_Q sono gli unici nodi in G che appartengono agli insiemi must, unique ed exclusive. La definizione di $\mathcal{C}$ garantisce che per ogni $e \in E_Q$ esista almeno un arco in G, con sorgente un nodo di V_Q e derivante dal guest G_e, mappato nella simulazione lasca. Questo arco corrisponde al primo passo di transizione dell'automa nondeterministico N_e, e Condizione (SL 6) garantisce esso faccia parte di un cammino terminante in uno stato di V_Q. Tale cammino inoltre rappresenta una parola accettata da N_e. Questo garantisce che le simulazioni lasche di G in H non violino le prime tre proprietà di Definizione 3.18. L'ultima condizione sulle path è inoltre verificata dal fatto che i vertici in V_Q siano exclusive. Ne consegue che se esiste una simulazione lasca di G in H, allora esiste un isomorfismo di sottografo con decorazioni regolari di H rispetto a Q, e viceversa. □

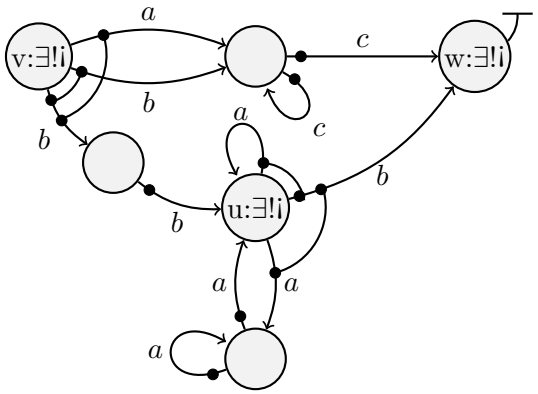
Abbiamo così mostrato come poter utilizzare guests e simulazioni lasche per risolvere il problema RE-SGISO. Questa nozione, che generalizza l'isomorfismo di sottografo, permette di verificare facilmente alcuni pattern molto interessanti, come per esempio la presenza di cicli con un numero pari (o dispari)



1. Una query per isomorfismo di sottografo con decorazioni regolari.



2. Guests di simulazioni lasche utilizzati per generare il guest corrispondente alla query di cui sopra. I nodi v, w e u appartengono al guest chiamato F nella dimostrazione di Proposizione 3.19, mentre gli altri guests appartengono all'insieme $\mathcal{G}$. Gli archi tratteggiati corrispondono alle relazioni utilizzate per la composizione dei guest.



3. Guest risultante dalle composizioni rappresentate nella figura precedente.

Figura 3.20: Passi di traduzione di query per RE-SGISO in guests per simulazioni lasche. In alto è rappresentata la query di partenza, mentre l'ultima illustrazione mostra il risultato della sua traduzione.

Figura 3.21: A sinistra, la query di RE-SGISO per determinare la presenza di cicli (di soli caratteri a o b) con un numero pari di archi. A destra la sua traduzione in guest per simulazioni lasche.

di archi, come rappresentato in Figura 3.21, oppure la presenza di cammini diversi che si ricongiungono in un determinato nodo.

Con questo risultato si conclude l'analisi di alcune applicazioni delle simulazioni lasche, durante il quale abbiamo potuto dare alcuni accenni sulla loro complessità, grazie al confronto fatto con altre nozioni di similarità tra grafi presenti in letteratura. Focalizzeremo ora la nostra attenzione sullo studio di complessità delle simulazioni lasche, in modo da completare formalmente quanto iniziato in questo capitolo.

4

Complessità del problema del vuoto

I risultati di Capitolo 3, oltre a mostrare la semplicità delle simulazioni lasche nel rappresentare altre nozioni usate per il pattern matching su grafi, danno già alcuni accenni sulla loro complessità. In particolare, da Sezione 3.1 si evince come il problema decisionale ad esse associato sia NP-difficile. In questo capitolo siamo dunque interessati a studiare formalmente la complessità del seguente problema

Definizione 4.1 (Problema del vuoto per simulazioni lasche)**.** *Siano H e G rispettivamente un host ed un guest. Il problema del vuoto per simulazioni lasche consiste nel verificare se $\mathbb{S}^{G \to H} = \emptyset$.*

Lo studio di complessità qui presentato è diviso in due parti: inizieremo con tradurre le simulazioni lasche in un modello di soddisfacibilità per la programmazione lineare mista e, grazie a questo, mostreremo come il problema appena definito sia NP-completo. In seguito andremo a fare alcune considerazioni su questo risultato, in relazione ai vincoli richiesti dalle simulazioni lasche.

4.1 Risoluzione mediante Programmazione Lineare Intera

Un modello per la programmazione lineare è un programma di ottimizzazione o soddisfacibilità di un problema, rappresentato da un insieme di vincoli e provvisto, nel caso si intenda eseguire un'ottimizzazione, di una funzione da massimizzare (o minimizzare) denominata *funzione obiettivo*. Un programma per la programmazione lineare, nella sua forma canonica, può essere espresso con il seguente sistema di disequazioni[1] (riga 2) e funzione obiettivo (riga 1):

$$\begin{aligned} &\text{maximize} && c^T x \\ &\text{subject to} && Ax \leq b \end{aligned}$$

dove x è un vettore di n variabili a valori reali mentre c b e A sono rispettivamente un vettore di lunghezza n, un vettore di lunghezza m ed una matrice $m \times n$ di coefficienti reali. Nel caso di problema di soddisfacibilità — da noi utilizzato per la modellazione delle simulazioni lasche —, la funzione obiettivo non è presente e viene richiesto unicamente di assegnare ad ogni variabile un valore in modo tale da non violare alcuna disequazione.

[1] Si noti come le equazioni possano essere espresse in un programma per la programmazione lineare come una coppia di disequazioni.

Rispetto alla programmazione lineare, la *programmazione lineare intera* richiede l'interezza ai valori assegnati ad certo sottoinsieme di variabili. Un problema dove non viene richiesta l'interezza di tutte le variabili viene anche chiamato di *programmazione lineare mista*. Per uno studio completo delle tecniche di risoluzione per modelli di programmazione lineare (intera) rimandiamo il lettore a [5, 37].

Introdurremo ora un modello di programmazione lineare mista per il calcolo di simulazioni lasche. È bene notare come questo modello, seppur risolva il problema formulato in Definizione 4.1, non sia pensato per avere buone prestazioni[2] ma solo per fornire una soluzione di partenza al problema e permettere uno studio della sua complessità.

In seguito, siano $H = (\Sigma, V_H, E_H)$ e $G = (\Sigma, V_G, E_G, \mathcal{M}, \mathcal{U}, \mathcal{E}, \mathcal{C})$ rispettivamente un host ed un guest. Iniziamo col descrivere le variabili del modello. Esse possono essere suddivise nei seguenti cinque insiemi

$$
\begin{aligned}
X &= \{x_{u,v} \mid u \in V_G \ \wedge \ v \in V_H\} \\
Y &= \{y_{u,v,\alpha,u',v'} \mid \alpha \in \Sigma \ \wedge \ (u,\alpha,u') \in E_G \ \wedge \ (v,\alpha,v') \in E_H\} \\
Z &= \{z_{u,v,\gamma} \mid u \in V_G \ \wedge \ \gamma \in \mathcal{C}(u) \ \wedge \ v \in V_H\} \\
C &= \{c^e_{u,v,\gamma} \mid u \in V_G \ \wedge \ \gamma \in \mathcal{C}(u) \ \wedge \ e \in \gamma \ \wedge \ v \in V_H\} \\
F &= \left\{ f^{g,h,m}_{u,v,\alpha,u',v'} \;\middle|\; \begin{array}{l} \alpha \in \Sigma \ \wedge \ (u,\alpha,u') \in E_G \ \wedge \ (v,\alpha,v') \in E_H, \\ g \in V_G \ \wedge \ h \in V_H \ \wedge \ m \in \{m \in \mathcal{M} \mid \mathbb{P}_G(g,m) \neq \emptyset\} \end{array} \right\}
\end{aligned}
$$

dove richiediamo che le variabili in X, Y, Z e C abbiano valori interi in $\{0, 1\}$ mentre le variabili in F siano a valori reali in $[0, 1]$. Si noti come, per semplicità di esposizione, non saranno introdotti i vettori di variabili (o coefficienti) ma andremo ad esplicitare le singole variabili (o coefficienti). La struttura del modello descritta poc'anzi può infatti essere riscritta come

$$
\begin{aligned}
\text{maximize} \quad & \sum_{j \in \{1,\dots,n\}} c_j x_j \\
\text{subject to} \quad & \sum_{j \in \{1,\dots,n\}} A_{i,j} x_j \leq b_i \quad i \in \{1, \dots, m\}
\end{aligned}
$$

Descriviamo ora la semantica degli insiemi di variabili appena introdotti. Per farlo supponiamo esista una simulazione lasca $(\phi, \eta) \in \mathbb{S}^{G \to H}$ e che le variabili del modello di programmazione lineare siano istanziate in modo da riflettere tale soluzione:

- per ogni $u \in V_G$ e $v \in V_H$, $x_{u,v} \in X$ ha valore pari a 1 se e solo se $(u, v) \in \phi$. L'insieme X contiene $|V_G||V_H|$ variabili;
- per ogni $e = (u, \alpha, u') \in E_G$ e $e' = (v, \alpha, v') \in E_H$, aventi la stessa etichetta α, $y_{u,v,\alpha,u',v'} \in Y$ è impostata a 1 se e solo se $(e, e') \in \eta$. Si noti dunque come le variabili in Y corrispondano a coppie di archi in $E_G \times E_H$ con la stessa etichetta. La cardinalità di Y è in $\mathcal{O}(|E_G||E_H|)$.
- per ogni $u \in V_G$, $v \in V_H$ ed ogni insieme $\gamma \in \mathcal{C}(u)$ ritornato dalla funzione di scelta applicata a u, $z_{u,v,\gamma}$ ha valore pari a 1 se e solo se γ è *selezionato* in η rispetto a v — secondo la definizione di selezione data per Condizione (SL 5). La cardinalità di Z è in $\mathcal{O}(|V_G||V_H| \max_{u \in V_G}(|\mathcal{C}(u)|))$

[2] Più nello specifico, il numero di variabili è molto elevato, non sono aggiunti vincoli per gestire le simmetrie del problema e viene risolto un problema multiflusso in modo naive, al posto di utilizzare un modello a generazione di colonne.

- l'insieme C è un insieme di variabili ausiliarie per Z. Per ogni $u \in V_G$, $v \in V_H$, $\gamma \in \mathcal{C}(u)$ e arco $e \in \gamma$, la variabile $c^e_{u,v,\gamma}$ ha valore pari a 1 se e solo se esiste un arco $e' \in \text{out}(v)$ tale che $(e, e') \in \eta$. La cardinalità di C è in $\mathcal{O}(|V_G||E_G||V_H| \max_{u \in V_G}(|\mathcal{C}(u)|))$

- per ogni $e = (u, \alpha, u') \in E_G$ e $e' = (v, \alpha, v') \in E_H$, aventi la stessa etichetta α, per ogni $u \in V_G$, $v \in V_H$ e per ogni nodo $m \in \mathcal{M}$ appartenente all'insieme must e tale per cui esiste in G una path da u in m, $f^{g,h,m}_{u,v,\alpha,u',v'}$ esprime una quantità di flusso passante per la coppia di archi (e, e'). Queste variabili verranno utilizzate per risolvere un problema multiflusso — introdotto in seguito — necessario per verificare venga rispettata Condizione (SL 6). La cardinalità di F è in $\mathcal{O}(|V_G|^2|E_G||V_H||E_H|)$.

Il numero (elevato) di variabili è dunque in $\mathcal{O}(|V_G||E_G||V_H|(|V_G||E_H| + \max_{u \in V_G}(|\mathcal{C}(u)|)))$.

Passiamo ora a descrivere i vincoli del modello, andando a tradurre ogni condizione di Definizione 2.7 nel corrispondente insieme di disequazioni lineari. Per quanto riguarda Condizione (SL 1), ricordiamo come questa imponga ad ogni vertice dell'insieme must di apparire nella simulazione lasca. Possiamo esprimere questa condizione con le seguenti $|\mathcal{M}|$ disequazioni

$$\sum_{h \in V_H} x_{u,h} \geq 1 \qquad \forall u \in \mathcal{M}$$

In una soluzione al modello di programmazione lineare mista, ognuna di queste disequazioni, ciascuna con $|V_H|$ variabili, deve essere soddisfatta. Dato $u \in \mathcal{M}$, essendo l'insieme X un insieme di variabili intere $\{0, 1\}$, perché la disequazione ad esso associata venga rispettata è necessario che almeno una delle variabili in X associate ad u abbia valore 1. Per la semantica data alle variabili in X, questo corrisponde a richiedere che u compaia nella simulazione lasca, come richiesto da Condizione (SL 1).

Condizione (SL 2) è analoga a quanto visto per la prima condizione. In essa viene richiesto che ogni vertice dell'insieme unique appaia al massimo associato ad un vertice dell'host. Possiamo esprimere questa condizione attraverso le seguenti $|\mathcal{U}|$ disequazioni a $|V_H|$ variabili.

$$\sum_{h \in V_H} x_{u,h} \leq 1 \qquad \forall u \in \mathcal{U}$$

Dato $u \in \mathcal{U}$, grazie all'interezza delle variabili in X, perché la disequazione ad esso associata venga rispettata, è necessario che al massimo una delle variabili associate a u abbia valore 1. Questo corrisponde a richiedere sia verificata Condizione (SL 2).

Passiamo ora a codificare Condizione (SL 3). Essa richiede che, per ogni vertice $u \in \mathcal{E}$ dell'insieme exclusive, qualora u venga associato nella simulazione lasca ad un vertice h dell'host, allora h non possa essere associato a nessun altro nodo v del guest. In termini di programmazione lineare, questo corrisponde a richiedere che le due variabili intere $x_{u,h}$ e $x_{v,h}$ non possano assumere contemporaneamente il valore 1. Condizione (SL 3) può quindi facilmente essere espressa con il seguente sistema di $\mathcal{O}(|\mathcal{E}||V_G||V_H|)$ disequazioni, ciascuna con due variabili.

$$x_{u,h} + x_{v,h} \leq 1 \qquad \forall u \in \mathcal{E}\ \forall v \in V_G \setminus \{u\}\ \forall h \in V_H$$

Si noti infatti come, assegnando il valore 1 ad entrambe le variabili $x_{u,h}$ e $x_{v,h}$, la disequazione venga violata.

Condizione (SL 4) richiede che, qualora un arco $e \in E_G$ del guest ed un arco $e' \in E_H$ vengano associati in una simulazione lasca, allora anche $s(e)$ e $s(e')$ siano in relazione tra loro. Lo stesso deve inoltre valere per $t(e)$ e $t(e')$ ed infine i due archi devono necessariamente avere la stessa etichetta. Per quanto riguarda quest'ultimo vincolo — sulle etichette — si noti come esso venga rispettato dalla definizione dell'insieme Y, le cui variabili corrispondono a coppie di archi con la stessa etichetta. Gli altri due vincoli su sorgenti e terminazioni possono invece essere espressi con le seguenti $\mathcal{O}(|E_G||E_H|)$ disequazioni, ciascuna con due variabili.

$$\begin{aligned}
&\forall \alpha \in \Sigma\ \forall (u, \alpha, u') \in E_G\ \forall (v, \alpha, v') \in E_H \\
&\quad y_{u,v,\alpha,u',v'} \leq x_{u,v} \\
&\quad y_{u,v,\alpha,u',v'} \leq x_{u',v'}
\end{aligned}$$

Qualora la variabile $y_{u,v,\alpha,u',v'}$ associata a due archi $(u, \alpha, u') \in E_G$ e $(v, \alpha, v') \in E_H$ assuma valore 1, perché le due disequazioni sopra definite vengano rispettate entrambe le variabili $x_{u,v}$ e $x_{u',v'}$ devono assumere valore 1. Dalla semantica data per le variabili in X, $x_{u,v} = 1$ corrisponde a richiedere che le sorgenti u e v dei due archi siano in relazione nella simulazione lasca, mentre $x_{u',v'}$ è pari a 1 se e solo se le due terminazioni sono in relazione. Si noti infine come, correttamente, il vincolo non imponga che qualora una delle variabili di X assuma valore 1, allora anche le variabili in Y ad essa collegate assumano valore 1. Questo sistema di disequazioni, quindi, corrisponde alla condizione (SL 4).

Passiamo ora a considerare la condizione (SL 5). In essa si richiede che per ogni coppia di vertici $(u, v) \in V_G \times V_H$ appartenente ad una simulazione lasca esista un insieme di $\mathcal{C}(u)$ *selezionato* rispetto a v. Inoltre, per ogni coppia di archi $(e, e') \in E_G \times E_H$ appartenente alla simulazione deve esistere un insieme $\gamma \in \mathcal{C}(s(e))$ selezionato rispetto a $s(e')$ tale che $e \in \gamma$. Per esprimere questo vincolo è necessario utilizzare l'insieme Z: una variabile $z_{u,v,\gamma} \in Z$ deve assumere valore 1 solo qualora $\gamma \in \mathcal{C}(u)$ venga selezionato rispetto a v. Nel caso in cui $z_{u,v,\gamma} \in Z$ sia pari a 1, deve inoltre valere $x_{u,v} = 1$. Iniziamo dunque col codificare questi vincoli, i quali mettono in relazione l'insieme X con l'insieme Z, attraverso le seguenti disequazioni.

$$\begin{aligned}
z_{u,v,\gamma} &\leq x_{u,v} && \forall u \in V_G\ \forall \gamma \in \mathcal{C}(u)\ \forall v \in V_H \\
x_{u,v} &\leq \sum_{\gamma \in C(u)} z_{u,v,\gamma} && \forall u \in V_G\ \forall v \in V_H
\end{aligned}$$

Il primo insieme di $\mathcal{O}(|V_G||V_H| \max_{u \in V_G}(|\mathcal{C}(u)|))$ disequazioni, ciascuna di due variabili, impone che $z_{u,v,\gamma}$ possa essere pari a 1 solo qualora $x_{u,v}$ assuma valore 1. Questo corrisponde a dire che la selezione di un elemento $\gamma \in \mathcal{C}(u)$ rispetto a v può essere fatta solo se u è in relazione con v nella simulazione lasca. Il secondo insieme di $|V_G||V_H|$ disequazioni in $\mathcal{O}(\max_{u \in V_G}(|\mathcal{C}(u)|))$ variabili invece impone che, qualora $x_{u,v}$ sia pari a 1 — ovvero (u, v) appartiene alla simulazione lasca — allora almeno una delle variabili $z_{u,v,\gamma}$, al variare di $\gamma \in \mathcal{C}(u)$, deve assumere valore 1, ovvero almeno un insieme in $\mathcal{C}(u)$ venga selezionato rispetto a v. Prendiamo ora, per una simulazione lasca $(\phi, \eta) \in \mathbb{S}^{G \to H}$, la definizione di

selezione:

$$\gamma \in \mathcal{C}(u) \text{ è selezionato in } \eta \text{ rispetto a } v \iff \forall e \in \gamma\ \exists e' \in \text{out}(v)\ (e, e') \in \eta$$

Ricordando la semantica data alle variabili negli insiemi Y, Z e C, questa definizione può essere riformulata con la seguente coppia di doppie implicazioni

$$\begin{aligned} z_{u,v,\gamma} = 1 &\iff \forall e \in \gamma\ c^e_{u,v,\gamma} = 1 \\ c^{(u,\alpha,u')}_{u,v,\gamma} = 1 &\iff \exists (v,\alpha,v') \in \text{out}(v)\ y_{u,v,\alpha,u',v'} = 1 \end{aligned}$$

Vediamo ora come aggiungere queste due doppie implicazioni al modello di programmazione lineare mista. Per quanto riguarda la prima, interpretando il valore 0 come *false* ed il valore 1 come *true*, otteniamo la formula equivalente $z_{u,v,\gamma} = \bigwedge_{e\in\gamma} c^e_{u,v,\gamma}$. Un'equivalenza logica di questo tipo può essere tradotta in programmazione lineare intera come segue:

$$z = c_1 \wedge c_2 \wedge \cdots \wedge c_n \iff 0 \leq \sum_{i\in\{1,\ldots,n\}} c_i - nz \leq n-1$$

Infatti, nel caso in cui z sia *true* (ovvero assuma valore 1), è necessario che ogni variabile c_i, $i \in \{1,\ldots,n\}$, sia pari a 1, in caso contrario si avrà $\sum_{i\in\{1,\ldots,n\}} c_i - nz < 0$ ed il vincolo sarà violato. Se invece z viene posto a *false*, allora non tutte le variabili c_i possono essere pari a 1, in quanto $\sum_{i\in\{1,\ldots,n\}} 1 > n-1$. Questo vale tuttavia solo per $n > 0$. Se n è pari a 0, caso corrispondente all'appartenenza dell'insieme vuoto all'insieme ritornato dalla funzione di scelta, non va aggiunto alcun vincolo, dato che la prima doppia implicazione è banalmente verificata. Grazie alle considerazioni appena fatte, possiamo legare le variabili di Z con quelle di C attraverso il seguente sistema di disequazioni in $\mathcal{O}(|V_G||V_H| \max_{u\in V_G}(|\mathcal{C}(u)|))$, ciascuna con un numero di variabili in $\mathcal{O}(|E_G|)$.

$$\begin{aligned} &\forall u \in V_G\ \forall \gamma \in \mathcal{C}(u)\ \forall v \in V_H \text{ s.t. } \gamma \neq \emptyset \\ &\quad |\gamma| z_{u,v,\gamma} \leq \sum_{e\in\gamma} c^e_{u,v,\gamma} \\ &\quad \sum_{e\in\gamma} c^e_{u,v,\gamma} \leq |\gamma|(z_{u,v,\gamma} - 1) \end{aligned}$$

Passiamo a considerare la seconda doppia implicazione. Interpretando nuovamente i valori 0 e 1 rispettivamente come *false* e *true*, questa può essere riscritta come $c^{(u,\alpha,u')}_{u,v,\gamma} = \bigvee_{(v,\alpha,v')\ \in\ \text{out}_\alpha(v)} y_{u,v,\alpha,u',v'}$, dove $\text{out}_\alpha(v) \triangleq \{e \in \text{out}(v) \mid \sigma(e) = \alpha\}$. Un'equivalenza di questo tipo può essere tradotta in programmazione lineare come segue

$$c = y_1 \vee y_2 \vee \cdots \vee y_n \iff 0 \leq nc - \sum_{i\in\{1,\ldots,n\}} y_i \leq n-1$$

Si noti infatti come, nel caso in cui c venga posto a *true*, è necessario che almeno una variabile y_i,$i \in \{1,\ldots,n\}$, assuma il valore 1, altrimenti sarà violata la limitazione superiore di $n-1$. Se invece c viene posto a *false*, nessuna variabile y_i può essere pari a 1, altrimenti $nc - \sum_{i\in\{1,\ldots,n\}} y_i < 0$.

In questo caso, diversamente dalla prima doppia implicazione, se $\text{out}_\alpha(v)$ è vuoto $c_{u,v,\gamma}^{(u,\alpha,u')}$ deve essere necessariamente posto a 0. Possiamo quindi esprimere la seconda doppia implicazione attraverso il seguente sistema di disequazioni in $\mathcal{O}(|V_G||E_G||V_H| \max_{u \in V_G}(|\mathcal{C}(u)|))$, ciascuna con un numero di variabili in $\mathcal{O}(|E_H|)$.

$$\begin{aligned}
&\forall u \in V_G \ \forall \gamma \in \mathcal{C}(u) \ \forall (u,\alpha,u') \in \gamma \ \forall v \in V_H \\
&\qquad \sum_{(v,\alpha,v') \ \in \ \text{out}_\alpha(v)} y_{u,v,\alpha,u',v'} \leq |\text{out}_\alpha(v)| c_{u,v,\gamma}^{(u,\alpha,u')} && \text{se } \text{out}_\alpha(v) \neq \emptyset \\
&\qquad |\text{out}_\alpha(v)| c_{u,v,\gamma}^{(u,\alpha,u')} \leq |\text{out}_\alpha(v)| - 1 + \sum_{(v,\alpha,v') \ \in \ \text{out}_\alpha(v)} y_{u,v,\alpha,u',v'} && \text{se } \text{out}_\alpha(v) \neq \emptyset \\
&\qquad c_{u,v,\gamma}^{(u,\alpha,u')} = 0 && \text{se } \text{out}_\alpha(v) = \emptyset
\end{aligned}$$

Abbiamo così terminato la modellazione di *selezione* in programmazione lineare intera. Questo conclude la prima parte della condizione (SL 5): nel caso in cui una variabile $x_{u,v}$ assuma valore 1, allora deve esistere un $\gamma \in \mathcal{C}(u)$ tale che $z_{u,v,\gamma}$ sia pari a 1. Rispetto a tale variabile, per ogni $(u,\alpha,u') \in \gamma$ deve valere $c_{u,v,\gamma}^{(u,\alpha,u')} = 1$ e questo comporta debba esistere un arco $(v,\alpha,v') \in \text{out}(v)$ tale che $y_{u,v,\alpha,u',v'}$ assuma valore 1. Per terminare la codifica di Condizione (SL 5) non resta imporre che, qualora una coppia di archi $(e,e') \in E_G \times E_H$ appartenga alla simulazione lasca, allora deve esistere $\gamma \in \mathcal{C}(s(e))$ selezionato rispetto a $s(e')$ tale che $e \in \gamma$. Ricordando la semantica data per l'insieme di variabili C, possiamo rappresentare questo vincolo facilmente attraverso le seguenti disequazioni.

$$y_{u,v,\alpha,u',v'} \leq \sum_{\substack{\gamma \in C(u) \\ (u,\alpha,u') \in \gamma}} c_{u,v,\gamma}^{(u,\alpha,u')} \qquad \forall \alpha \in \Sigma \ \forall (u,\alpha,u') \in E_G \ \forall (v,\alpha,v') \in E_H$$

Il numero di disequazioni in questo ultimo sistema appartiene a $\mathcal{O}(|E_G||E_H|)$ e vi sono $\mathcal{O}(\max_{u \in V_G}(|\mathcal{C}(u)|))$ variabili per disequazione.

Infine, consideriamo la condizione (SL 6). Essa richiede che, rispetto ad una coppia $(u,v) \in V_G \times V_H$ appartenente alla simulazione lasca e dato un vertice $w \in \mathcal{M}$ dell'insieme must, se w è raggiungibile da u nel guest, allora deve esistere un nodo $v' \in V_H$ tale che (w,v') è raggiungibile da (u,v) — tramite la funzione di raggiungibilità duale introdotta in Definizione 2.6. Per capire meglio questa condizione, introduciamo il grafo di una simulazione lasca.

Definizione 4.2 (Grafo di una simulazione lasca)**.** *Sia $(\phi,\eta) \in \mathbb{S}^{G \to H}$ una simulazione lasca. Essa sottintende un multigrafo diretto etichettato (Σ,ϕ,E_η) avente come alfabeto Σ l'alfabeto del guest G (o dell'host H), le coppie in ϕ costituiscono i vertici del grafo e η definisce l'insieme di archi come segue*

$$E_\eta \triangleq \{((u,v),\alpha,(u',v')) \mid ((u,\alpha,u'),(v,\alpha,v')) \in \eta\}$$

Si noti come, rispetto alla definizione appena data, per ogni arco $((u,v),\alpha,(u',v')) \in E_\eta$ esiste una variabile $y_{u,v,\alpha,u',v'} \in Y$. Rispetto ad una soluzione del modello di programmazione lineare mista che stiamo definendo, infatti, ad una simulazione lasca $(\phi,\eta) \in \mathbb{S}^{G \to H}$ corrisponderà un istanziazione delle

variabili tale che

$$y_{u,v,\alpha,u',v'} = 1 \iff ((u,\alpha,u'),(v,\alpha,v')) \in \eta \iff ((u,v),\alpha,(u',v')) \in E_\eta$$

Condizione (SL 6) richiede che per ogni vertice $(u,v) \in \phi$ e per ogni vertice $w \in \mathcal{M}$, se $\mathbb{P}_G(u,w) \neq \emptyset$ allora deve esistere nel grafo un cammino da (u,v) ad un vertice $t \in \phi$ tale che $\pi_1(t) = w$. Il metodo più utilizzato per codificare vincoli di questo tipo in programmazione lineare è attraverso l'impostazione di un problema (multi)flusso. Un problema di flusso consiste nel far viaggiare lungo le connessioni di una rete (ovvero gli archi di un grafo) una *quantità di flusso* prodotto da un nodo sorgente e che deve giungere in uno o più nodi destinazione. Ovviamente perché questo problema ammetta una soluzione è necessaria l'esistenza di un cammino dalla sorgente alla destinazione: per questo motivo, i problemi di flusso possono essere utilizzati per risolvere problemi di raggiungibilità tra nodi.

In seguito sia (Σ, V, E) un multigrafo diretto etichettato, $s \in V$ un nodo sorgente e $T \subseteq V$ un insieme di nodi terminazione. Introduciamo ora un modello di programmazione lineare usato per la risoluzione di problemi di raggiungibilità, studiati tuttavia come problemi di flusso:

$$\sum_{e \in \text{out}(u)} f_e = \sum_{e \in \text{in}(u)} f_e \qquad \forall u \in V \setminus (\{s\} \cup T)$$

$$\sum_{e \in \text{out}(s)} f_e = 1$$

$$\sum_{e \in \text{in}(s)} f_e = 0 \qquad \text{se } s \notin T$$

dove le variabili f_e, $e \in E$, sono a valori reali nell'intervallo $[0,1]$ e rappresentano la quantità di flusso passante per l'arco e, da $s(e)$ verso $t(e)$. Si noti come l'ultima equazione venga aggiunta al modello unicamente se $s \notin T$. Prendiamo in esame il modello nel caso in cui $s \in T$. La sorgente s produrrà una quantità di flusso pari a 1 inviata arbitrariamente attraverso gli archi da essa uscenti — comportamento descritto, nel modello, dalla penultima equazione. Questa quantità di flusso viaggerà nella rete, tuttavia qualora giunga ad un vertice non appartenente a T essa dovrà anche uscire da tale vertice. Per questo motivo, ad ognuno di questi nodi, viene aggiunto un vincolo — rappresentato dal primo insieme di equazioni — che impone l'equivalenza tra flusso entrante da un nodo e quello uscente da questo. Ne consegue che, essendo la quantità di flusso prodotta dalla sorgente maggiore di 0, il problema avrà soluzione unicamente se il flusso può raggiungere almeno una delle terminazioni in T. Lo stesso vale nel caso in cui $s \notin T$, dove però l'aggiunta dell'ultima equazione va ad imporre che il flusso prodotto dalla sorgente non possa tornare in questa. Infatti, non essendo la sorgente in questo caso anche un terminazione, il flusso deve raggiungere necessariamente un nodo diverso da questa[3]. Si noti una particolarità del modello appena proposto: nel caso in cui $T = \{s\}$ viene comunque richiesto un cammino da s in se stesso che coinvolga almeno un arco. In particolare, quindi, il modello può anche essere utilizzato per controllare la presenza di cicli che coinvolgano un determinato nodo, ponendo per l'appunto questo come sorgente ed unica terminazione.

[3]Si noti come, se l'istanza del problema ammette soluzione, allora esiste sicuramente una soluzione dove il cammino collegante la sorgente s ad un nodo di T non contiene cicli. Per questo motivo è possibile porre a 0 il flusso entrante nella sorgente.

Il modello appena introdotto può essere utilizzato, con poche modifiche, per verificare Condizione (SL 6). Abbiamo tuttavia bisogno di un modello per problemi di flusso per ogni coppia $(u, v) \in V_G \times V_H$ e per ogni $w \in \mathcal{M}$ tale che $\mathbb{P}_G(u, w) \neq \emptyset$.[4] In questo caso, dove bisogna gestire diversi tipi di flusso separatamente, si parla spesso di *problemi multiflusso*. Per rappresentare Condizione (SL 6) in programmazione lineare utilizzeremo l'insieme di variabili F. In particolare, con la variabile $f^{g,h,m}_{u,v,\alpha,u',v'} \in F$ rappresenteremo il flusso passante lungo l'arco corrispondente alla variabile $y_{u,v,\alpha,u',v'}$, rispetto al problema di flusso avente come sorgente $(g, h) \in V_G \times V_H$ e come terminazioni l'insieme $\{(m, t) \mid t \in V_H\}$. Perché il flusso passante per $f^{g,h,m}_{u,v,\alpha,u',v'}$ possa essere diverso da 0, devono necessariamente essere pari a 1 le variabili $y_{u,v,\alpha,u',v'}$ e $x_{g,h}$ — ovvero, rispetto ad una simulazione lasca $(\phi, \eta) \in \mathbb{S}^{G \to H}$ deve valere $((u, v), \alpha, (u', v')) \in E_\eta$ e $(g, h) \in \phi$. Questo vincolo può essere espresso con le seguenti $\mathcal{O}(|V_G|^2|E_G||V_H||E_H|)$ disequazioni in due variabili

$$\forall g \in V_G\ \forall m \in \{m \in \mathcal{M} \mid \mathbb{P}(g, m) \neq \emptyset\}\ \forall h \in V_H\ \forall \alpha \in \Sigma\ \forall (u, \alpha, u') \in E_G\ \forall (v, \alpha, v') \in E_H$$
$$f^{g,h,m}_{u,v,\alpha,u',v'} \leq y_{u,v,\alpha,u',v'}$$
$$f^{g,h,m}_{u,v,\alpha,u',v'} \leq x_{g,h}$$

Si noti quindi come, qualora $x_{g,h}$ non assuma valore 1, le disequazioni dei problemi di flusso aventi $(g, h) \in V_G \times V_H$ come sorgente debbano comunque essere verificate, ma senza che nessuna quantità di flusso passi per le variabili in esse presenti. Questa particolarità può essere aggiunta al modello per problemi di flusso descritto poc'anzi andando ad indicare che la sorgente (g, h) produca un'unità di flusso unicamente se $x_{g,h} = 1$. Possiamo dunque rappresentare Condizione (SL 6) con il seguente sistema di equazioni:

$$\forall g \in V_G\ \forall m \in \{m \in \mathcal{M} \mid \mathbb{P}(g, m) \neq \emptyset\}\ \forall h \in V_H$$
$$\sum_{\substack{\alpha \in \Sigma \\ (u,\alpha,u') \in E_G \\ (v,\alpha,v') \in E_H}} f^{g,h,m}_{u,v,\alpha,u',v'} = \sum_{\substack{\alpha \in \Sigma \\ (u',\alpha,u) \in E_G \\ (v',\alpha,v) \in E_H}} f^{g,h,m}_{u',v',\alpha,u,v} \qquad \forall (u, v) \in V_G \times V_H \setminus (\{(g, h)\} \cup \{(m, x) \mid x \in V_H\})$$
$$\sum_{\substack{\alpha \in \Sigma \\ (g,\alpha,u) \in E_G \\ (h,\alpha,v) \in E_H}} f^{g,h,m}_{g,h,\alpha,u,v} = x_{g,h}$$
$$\sum_{\substack{\alpha \in \Sigma \\ (u,\alpha,g) \in E_G \\ (v,\alpha,h) \in E_H}} f^{g,h,g}_{u,v,\alpha,g,h} = 0 \qquad \text{se } g \neq m$$

Si noti come questo sistema istanzi fedelmente il modello per problemi di flusso prima introdotto, con la sola differenza nella quantità di flusso generata dalla sorgente, equivalente al valore della variabile ad essa associata. Come limitazione superiore, questo sistema appartiene a $\mathcal{O}(|V_G|^3|V_H|^2)$, ed ogni singola equazione ha un numero di variabili in $\mathcal{O}(|E_G||E_H|)$.

Per quanto detto durante questa sezione, tutti i sistemi di disequazioni introdotti formano un modello di programmazione lineare mista in grado di risolvere esattamente il problema del vuoto per simulazioni lasche di Definizione 4.1. In particolare, sia $M^{G \to H}$ il modello generato per un guest G ed un host H.

[4] L'insieme di nodi dell'insieme must raggiungibili da un determinato nodo del guest può essere computato in tempo polinomiale tramite un algoritmo di ricerca su grafo.

Vale la seguente doppia implicazione

$$\mathbb{S}^{G\to H} \neq \emptyset \iff M^{G\to H} \text{ ammette almeno una soluzione}$$

Inoltre, sia $S = (X_S, Y_S, Z_S, C_S, F_S)$ una soluzione computata a partire da $M^{G\to H}$, con X_S, Y_S, Z_S, C_S e F_S assegnamenti per le variabili degli insiemi X, Y, Z, C e F rispettivamente. Per la semantica data agli insiemi X e Y, la coppia di relazioni (ϕ, η) definita come

$$\phi = \{(u, v) \mid X_S \ni x_{u,v} = 1\}$$
$$\eta = \{((u, \alpha, u'), (v, \alpha, v')) \mid Y_S \ni y_{u,v,a,u',v'} = 1\}$$

è una simulazione lasca per G in H.

4.2 Complessità

Il modello proposto nella sezione precedente ha un numero di disequazioni avente una limitazione superiore in $\mathcal{O}(|V_G|^3|V_H|(|E_H| + \max_{u\in V_G}(|\mathcal{C}(u)|)))$, mentre il numero di variabili per disequazioni appartiene a $\mathcal{O}(|E_G||E_H| + \max_{u\in V_G}(|\mathcal{C}(u)|))$. Queste due limitazioni valgono anche rispettivamente per le righe e le colonne della matrice dei coefficienti del modello. Seppur polinomiale, il numero di variabili in questo caso è piuttosto elevato. Per questo motivo, come già detto all'inizio della sezione precedente, il modello proposto non rappresenta un buon programma per risolvere la computazione di simulazioni lasche. Il problema principale risiede nel numero di variabili necessario per risolvere il problema multiflusso collegato a Condizione (SL 6). Come proposto in [37], una migliore soluzione per problemi di questo tipo richiede un modello a generazione di colonne. Tuttavia, il bound polinomiale trovato ci permette facilmente di individuare la classe di complessità alla quale appartiene il problema del vuoto per le simulazioni lasche. Questo perché la risoluzione di modelli di soddisfacibilità per la programmazione lineare intera (e mista), dove tutte le variabili alla quale viene richiesta l'interezza hanno valori in $\{0, 1\}$ ed il numero di disequazioni, così come il numero di variabili per disequazione, ha un bound polinomiale, è un problema in NP; come mostrato da R. Karp in [25]. Sapendo inoltre, da Sezione 3.1, come le simulazioni lasche possano essere utilizzate per risolvere il problema decisionale per isomorfismo di sottografo, deduciamo che il problema del vuoto per simulazioni lasche è NP-completo.

Questo risultato porta ad un'importante osservazione: come mostrato in Sezione 3.1, nel caso in cui la funzione di scelta di un guest sia definita come $\lambda v.\{\text{out}(v)\}$, Condizione (SL 6) è sempre rispettata. Nondimeno, il problema del vuoto per simulazioni lasche risulta NP-completo anche in questo caso. La condizione (SL 6) risulta quindi ininfluente in termini di espressività e possiamo affermare che, data una coppia di guest e host, esista una riduzione di questa in una coppia di guest e host per simulazioni lasche private di Condizione (SL 6); e viceversa. D'altro canto, questa condizione permette di esprimere molte nozioni, come per esempio i linguaggi regolari, in modo facile ed intuitivo. Per questo motivo — ed anche proprio in virtù del fatto che essa non porta alcun aggravio in termini di complessità — continueremo a richiedere che le simulazioni lasche rispettino Condizione (SL 6).

II

Algoritmo distribuito per Simulazioni Lasche

5

Amalgamazione Distribuita

La complessità delle simulazioni lasche, studiata nel capitolo precedente, è tale da rendere il calcolo locale di queste poco praticabile nel caso in cui l'host sia di notevoli dimensioni. Intendiamo dunque approcciare il problema dal punto di vista del calcolo distribuito e parallelo, come fatto in letteratura per numerose nozioni di pattern matching e model checking su grafi [4, 17, 18, 28].

In questo capitolo introdurremo un algoritmo generale e decentralizzato per il calcolo distribuito di predicati, denominato *Amalgamazione Distribuita*. Questo algoritmo è stato istanziato da M. Miculan, M. Peressotti e dal sottoscritto in [29] per la computazione di *embedding* di *bigrafi*, un metalinguaggio introdotto da R. Milner per la progettazione e l'analisi di sistemi ad agenti [33]. Intuitivamente, ogni processo coinvolto nell'algoritmo mantiene localmente una porzione della rappresentazione del sistema da verificare ed una collezione di risultati parziali, computata a partire da una famiglia di queries. Le soluzioni (totali) vengono poi cercate a partire da quelle parziali tramite cooperazione tra processi.

Dettagli sulla progettazione e implementazione dell'algoritmo di Amalgamazione Distribuita sono presentati in Appendice A, mentre in Capitolo 6 mostreremo come questo possa essere istanziato per risolvere il problema del vuoto delle simulazioni lasche.

5.1 Calcolo distribuito di predicati

L'algoritmo di *Amalgamazione Distribuita* può essere visto come un'applicazione in ambito di algoritmi per sistemi distribuiti del modello di esecuzione di un sistema ad agenti — utilizzato in ambito di intelligenza artificiale e progettazione di sistemi intelligenti [19]. In questo, gli agenti sono entità autonome in grado di compiere azioni dipendenti dalle informazioni ricevute dall'ambiente nel quale sono collocati, come per esempio i messaggi ricevuti da altri agenti. Queste entità utilizzano un modello di esecuzione piuttosto generale, rappresentato in Figura 5.1: un agente, ricevuto un evento dall'ambiente esterno, lo analizza, pianifica l'operazione da compiere e la esegue, tutto a seconda delle informazioni in suo possesso, dettate dalla sua conoscenza. Si noti come il passo di esecuzione possa andare a modificare la conoscenza di un agente, aggiornandola.

Nel caso dell'algoritmo di Amalgamazione Distribuita, gli agenti sono processi che cercano di coordinarsi per trovare la soluzione ad una query (o un predicato). La loro conoscenza locale corrisponde ad un frammento del sistema — per esempio, una porzione di host — sul quale computare la soluzione. Rispetto ad una query, gli agenti utilizzeranno la loro conoscenza per ricavare delle soluzioni parziali ed

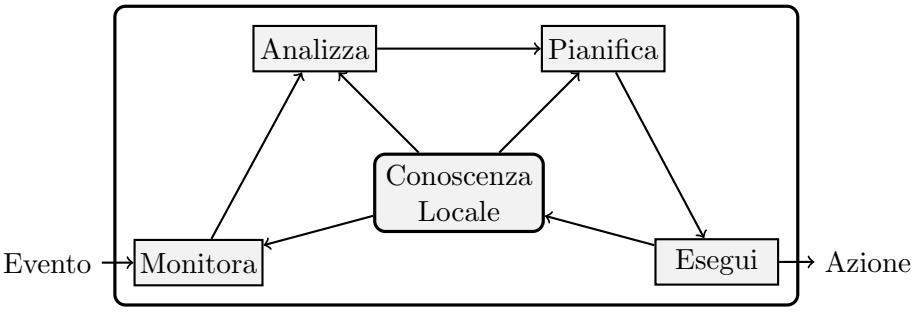


Figura 5.1: Modello di esecuzione di un agente.

inviare queste ultime ad altri processi del sistema, cercando di ottenere così una soluzione completa per unione di soluzioni parziali. Per garantire la correttezza e completezza delle soluzioni computate dai processi, viene considerato il seguente problema astratto di calcolo distribuito di predicati.

Sia $(\mathcal{K}, \sqcup)$ un semireticolo finito e sia $\mathcal{G} \subseteq \mathcal{K}$ un insieme — solitamente il più piccolo — di generatori, ovvero di elementi tali da generare tramite operazione di join $\sqcup$ tutti gli elementi di $\mathcal{K}$, *i.e.* in termini di minimo punto fisso deve valere $\mathcal{K} = \mathrm{lfp}(\lambda X.\{e \sqcup e' \mid e \in X \ \wedge \ e' \in X\} \cup \mathcal{G})$. Dato un predicato p su $\mathcal{K}$, siamo interessati a computare i valori $s \in \mathcal{K}$ tali da rendere vero p, trovando s a partire dai generatori in $\mathcal{G}$ tramite applicazioni di $\sqcup$. Indicheremo con $p(s)$ la verità di p rispetto ad s. Rispetto al problema appena definito, gli elementi di $\mathcal{K}$ possono essere interpretati come computazioni di p effettuate dai vari processi. Queste possono essere *complete*, ed in tal caso è possibile verificare se soddisfino o meno il predicato p, oppure *parziali*, ovvero possono essere composte (o continuate) con altre computazioni effettuate da altri processi nel tentativo di completarle. Più precisamente, se ogni processo, tramite la sua conoscenza locale, è in grado di computare un sottoinsieme di $\mathcal{G}$ e l'unione di questi sottoinsiemi è pari a $\mathcal{G}$, allora tramite scambio e join di computazioni da parte dei processi è possibile ricostruire tutte le soluzioni in $\mathcal{K}$. Qualora un elemento $s \in \mathcal{K}$ tale che $p(s)$ è computato, esso è consegnato come output dell'algoritmo. In caso contrario, qualora nessun elemento soddisfi il predicato, l'algoritmo terminerà indicando l'assenza di soluzioni. È quindi necessario definire:

(CDP 1) un modo di partizionare la rappresentazione di un sistema — come per esempio un host — ed di assegnare ogni parte ad un processo;

(CDP 2) $\mathcal{K}$, ovvero l'insieme (finito) delle computazioni (o soluzioni), parziali e complete;

(CDP 3) Un semireticolo su $\mathcal{K}$, ovvero sia definita su tale insieme un'operazione di join (o unione) di computazioni parziali $\sqcup : \mathcal{K} \times \mathcal{K} \to \mathcal{K}$ associativa, commutativa e idempotente. Si noti come questa operazione induce un ordinamento parziale $\leq$ definito per ogni $x, y \in \mathcal{K}$ come $x \leq y \overset{\triangle}{\iff} x \sqcup y = y$;

(CDP 4) un insieme finito di generatori $\mathcal{G} \subseteq \mathcal{K}$. Essendo $\mathcal{K}$ finito, questo insieme esiste sempre. Richiediamo tuttavia che ogni elemento di $\mathcal{G}$ sia computabile localmente — ovvero senza scambio di messaggi — a partire dalla parte di sistema assegnata ad un processo;

(CDP 5) una nozione di instradamento che garantisca lo scambio di computazioni tra processi e tale da permettere la computazione di ogni soluzione, rispetto ad ogni query. Ovvero per ogni predicato p ed ogni soluzione $s \in \mathcal{K}$ tale che $p(s)$, l'instradamento deve garantire esista un processo che computi o riceva l'insieme di generatori $G \subseteq \mathcal{G}$ tale che $\bigsqcup G = s$;

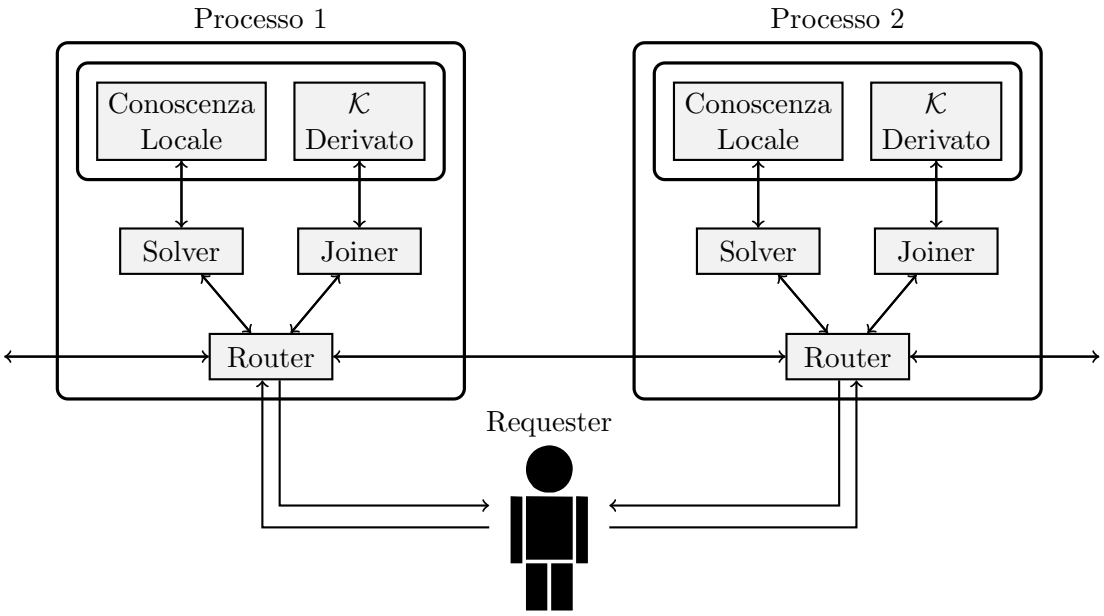


Figura 5.2: Moduli dell'algoritmo di Amalgamazione Distribuita.

(CDP 6) un modo per individuare quando una computazione sia completa e il suo risultato soddisfi un certo predicato;

(CDP 7) il criterio d'arresto dell'algoritmo, ovvero un modo per individuare quando, rispetto a $\mathcal{K}$, non esistano più nuove soluzioni per un predicato.

5.2 L'algoritmo di Amalgamazione Distribuita

Passiamo ora a definire meglio i ruoli dei vari elementi visti nella sezione precedente e ad indicare le varie parti dell'algoritmo. Ogni processo dell'algoritmo di Amalgamazione Distribuita è essenzialmente diviso in cinque moduli, di cui tre sottoprocessi e due basi di dati rappresentanti la conoscenza del processo.

In seguito, sia $(\mathcal{K}, \sqcup)$ un semireticolo e $\mathcal{G}$ un insieme di suoi generatori tale da verificare (CDP 3). Con riferimento alla Figura 5.2:

- la *conoscenza locale* mantiene il frammento del sistema assegnato ad un processo, secondo quanto definito in (CDP 1);
- la base di dati denominata $\mathcal{K}$*-derivato* mantiene, per ogni predicato, l'insieme delle computazioni parziali ricevute o generate dal sottoprocesso *joiner*;
- il sottoprocesso *solver* ha il compito di ricevere dal *router* le queries trasmesse dal *requester* — che può essere visto come un utente che richiede la soluzione di un predicato — interrogare la conoscenza locale computando in questo modo parte di $\mathcal{G}$ ed inviare queste soluzioni al router per l'instradamento;
- il sottoprocesso *joiner* riceverà computazioni parziali da parte del router ed interrogherà $\mathcal{K}$*-derivato* per ricavare nuove computazioni che potrebbero portare a soluzioni per il predicato analizzato. Tali computazioni verranno salvate in $\mathcal{K}$-derivato ed inviate nuovamente al router;

```
1 Solver: Event handler on receive {query, Q}
2     while un'altra computazione locale di Q esiste, rispetto alla conoscenza locale do
3         send {local_computation, next_computation(Q), Q} to Router
4     send {no_computations, Q} to Router
```

```
1 Joiner: Event handler on receive {partial_computation, S, Q}
2     K[Q] ← K[Q] ∪ {S}
3     while esiste G ∈ K[Q] tale che S ⊔ G ∉ K[Q] do
4         K[Q] ← K[Q] ∪ {S ⊔ G}
5         send {partial_computation, S ⊔ G, Q} to Router
6     send {no_computations, Q} to Router
```

Figura 5.3: Amalgamazione Distribuita - Handler messaggi per Solver e Joiner.

- il sottoprocesso *router* implementerà l'instradamento di (CDP 5) ed i controlli di (CDP 6) e (CDP 7). Più precisamente, questo processo si occupa di trasmettere al solver le queries del requester, trasmettere al joiner la computazioni parziali ricevute dal solver e da altri processi, inviare ad altri processi le computazioni parziali del joiner ed infine segnalare al requester le soluzioni totali o la mancanza di queste.

Figura 5.3 e Figura 5.5 mostrano lo pseudocodice di quanto appena descritto per i tre sottoprocessi e per il requester. Si noti come le porzioni di codice presentate vengano eseguite dai processi alla ricezione di un messaggio. Il codice per il solver dovrebbe essere autoesplicativo. Per quanto riguarda il joiner, $\mathcal{K}[Q]$ rappresenta la parte di $\mathcal{K}$-derivato attinente alla query Q. Si noti come ogni computazioni venga salvata in $\mathcal{K}[Q]$: questo meccanismo di *bookkeeping* permette di inviare ogni computazione un'unica volta. Per quanto riguarda Figura 5.5, i due frammenti di codice proposti verranno esaminati approfonditamente nella sezione successiva, in quanto mostrano come implementare il criterio d'arresto richiesto in (CDP 7) indipendentemente dalle definizioni di conoscenza locale e di predicato. Per il momento ci limiamo ad osservare come il router permetta la comunicazione tra solver, joiner e requester, mentre quest'ultimo si metterà in attesa delle risposte inviategli dal router rispetto ad un messaggio di query $\{\texttt{query}, Q\}$ — non mostrato nello pseudocodice del requester.

Sotto ipotesi che valgano le condizioni indicate al termine di Sezione 5.1, rispetto alla descrizione dei processi della precedente sezione, vale il seguente risultato:

Proposizione 5.4 (Computazione delle soluzioni)**.** *Sia p un predicato. Per ogni $v \in \mathcal{K}$ se vale $p(v)$ allora esiste un processo il cui router spedirà v al requester di p.*

Dimostrazione. Le condizioni sui generatori $\mathcal{G}$ di $\mathcal{K}$ dettate da (CDP 4) impone che questi siano tutti computabili da almeno un solver di un processo. Queste soluzioni verranno spedite secondo la definizione di instradamento imposta da (CDP 5) ai joiners, i quali applicheranno l'operazione di join descritta in (CDP 3). Se l'instradamento garantisce le condizioni di (CDP 5), allora tutte le computazioni $v \in \mathcal{K}$ tali che $p(v)$ — computabile grazie a (CDP 6) — verranno spedite dai moduli joiners e solvers ai routers, i quali provvederanno ad inoltrarle al requester. □

5.3 Criterio d'arresto distribuito

Grazie a quanto detto in Proposizione 5.4, sappiamo che l'algoritmo di Amalgamazione Distribuita consegnerà al requester ogni soluzione. Queste, tuttavia, proverranno da processi differenti. È quindi necessario individuare un meccanismo che permetta al requester di capire quando tutte le soluzioni sono state trovate — come richiesto in (CDP 7). Intuitivamente, l'algoritmo termina qualora tutti i processi, rispetto ad una data query, abbiano completato tutte le operazioni — *i.e.* i solvers hanno trovato tutti i generatori e i joiners abbiano unito tutte le computazioni parziali — e non vi siano messaggi in transito nella rete. In altre parole, viene richiesto che il sistema formato dai vari processi si sia stabilizzato. Per semplicità, lavoriamo sotto le seguenti assunzioni:

Assunzione 5.6. *I messaggi spediti da un processo sono sempre ricevuti dal destinatario. Inoltre, i processi non terminano in modo anomalo.*

Il problema della terminazione di un algoritmo distribuito, o della stabilizzazione di un sistema, è ampiamente affrontato in letteratura [14, 23, 30]. In particolare, sotto Assunzione 5.6, l'algoritmo di S.T. Huang, presentato in [23] risulta essere adatto a risolvere il problema nel caso di Amalgamazione Distribuita.

Definizione 5.7 (Algoritmo di Huang per la terminazione distribuita)**.** *Sia* Proc *un insieme di processi coinvolti nell'algoritmo distribuito e sia R un processo (anche esterno a* Proc*) richiedente una computazione distribuita. Diremo che un processo in* Proc *è inattivo qualora non stia eseguendo alcuna computazione locale. Rispetto ad una query Q,*

- *inizialmente tutti i processi in* Proc *sono inattivi;*
- *ogni processo $P \in$ Proc mantiene un valore $w_{P,Q}$, inizialmente pari a 1;*
- *ogniqualvolta un processo $P \in$ Proc intende inviare un messaggio, $w_{P,Q}$ viene diviso in due parti δ_1 e δ_2, entrambe diverse da 0. In seguito, il processo aggiornerà $w_{P,Q}$ impostandolo al valore δ_1 ed invierà δ_2 insieme al messaggio;*
- *ogniqualvolta un processo $P \in$ Proc riceve un messaggio, il valore δ_2 allegato con esso va ad incrementare $w_{P,Q}$;*
- *ogniqualvolta un processo $P \in$ Proc torna inattivo, esso invia il valore di $w_{P,Q}$ ad R.*

Il processo R mantiene, per ogni processo $P \in$ Proc, l'ultimo valore $w_{P,Q}$ da esso ricevuto — nel caso in cui P non abbia mai spedito il suo valore a R, poniamo $w_{P,Q} = 0$. La computazione distribuita termina qualora R riesca a verificare l'uguaglianza

$$|\text{Proc}| = \sum_{P \in \text{Proc}} w_{P,Q}$$

Per uno studio completo di questo algoritmo rimandiamo il lettore a [23]. Intuitivamente, i valori $w_{P,Q}$ permettono di capire se esistono ancora messaggi in transito nella rete: qualora un processo P_1 invii un messaggio ad un processo P_2 e poi diventi inattivo, la somma calcolata da R sarà inferiore a |Proc| almeno fino alla ricezione del messaggio e successiva inattività di P_2.

```
1  Router: Event handler on receive M
2      switch M do
3          case {query, Q} from Requester
4              R[Q] ← Requester
5              W[Q] ← 1                                   // Inizializza algoritmo di Huang per Q
6              S[Q] ← true                                // false quando solver inattivo
7              J[Q] ← 0                                   // 0 quando joiner inattivo
8              send {query, Q} to Solver
9          case {local_computation, S, Q} from Solver
10         or {partial_computation, S, Q} from Joiner
11             if S è completa e soddisfa Q then
12                 send {solution, S, Q} to R[Q]
13             else if S non è completa then
14                 receiver ← getReceiver(S,Q)
15                 if receiver = Joiner then
16                     J[Q] ← J[Q] + 1
17                     send {partial_computation, S, Q} to receiver
18                 else
19                     (δ1, δ2) ← split(W[Q])             // δ1, δ2 > 0 e δ1 + δ2 = W[Q]
20                     W[Q] ← δ1
21                     send {partial_computation, S, Q, δ2} to receiver
22         case {partial_computation, S, Q, w} from OtherProcess
23             W[Q] ← W[Q] + w
24             J[Q] ← J[Q] + 1
25             send {partial_computation, S, Q} to Joiner
26         case {no_computations, Q} from Solver
27             S[Q] ← false                               // Solver impostato ad inattivo
28             if J[Q] = 0 then
29                 send {node_off, Q, W[Q]} to Solver     // Se anche joiner inattivo, allora processo inattivo
30         case {no_computations, Q} from Joiner
31             J[Q] ← J[Q] − 1
32             if J[Q] = 0 and not S[Q] then
33                 send {node_off, Q, W[Q]} to Solver     // Processo inattivo se solver e joiner inattivi
34     endsw
```

```
1  Requester: Event handler on receive M
2      switch M do
3          case {solution, S, Q}
4              newSolution(S, Q)
5          case {node_off, Q, w} from Process
6              W[Q][ Process ] ← w
7              let Proc l'insieme di tutti i processi in
8                  if |Proc| = ∑_{P∈Proc} W[Q][P] then
9                      noOtherSolutions(Q)
10     endsw
```

Figura 5.5: Amalgamazione Distribuita - Handler messaggi per Router e Requester.

L'algoritmo appena descritto può essere utilizzato per risolvere quanto richiesto da (CDP 7), dove il requester ha il ruolo di R in Definizione 5.7 ed il sottoprocesso router si occupa di gestire i valori $w_{P,Q}$. Per applicare l'algoritmo è dunque necessario unicamente indicare quando i processi possono essere considerati inattivi. In seguito, sia Q una query e si faccia inoltre riferimento a Figura 5.3 e Figura 5.5. Per quanto specificato nella sezione precedente pare evidente che un processo sia inattivo quando, rispetto a Q, il solver ha inviato tutte le computazioni locali ed il joiner ha eseguito tutti i join dovuti ai messaggi a lui recapitati. Essendo l'algoritmo di Amalgamazione Distribuita generico sulla struttura di solver e joiner, richiediamo dunque che questi segnalino espressamente al router il termine delle loro computazioni. Più precisamente, terminato l'invio di tutte le soluzioni locali, il solver invierà al router un messaggio $\{\texttt{no\_computations}, Q\}$ — riga 4 dello pseudocodice del solver in Figura 5.3 —, e sarà così considerato inattivo da quest'ultimo. Analogamente, per ogni messaggio ricevuto, il joiner invierà $\{\texttt{no\_computations}, Q\}$ al termine della computazione dei join per tale messaggio. Il joiner verrà considerato inattivo qualora il numero di messaggi di questo tipo da esso inviati corrisponda al numero di messaggi inviati dal router al joiner stesso.

Per implementare i controlli appena descritti, il router necessita di

- un valore booleano $S[Q]$, inizializzato a *true* e portato a *false* al momento della ricezione di $\{\texttt{no\_computations}, Q\}$ da parte del solver — righe 6 e 27 dello pseudocodice relativo al router, Figura 5.5;
- un contatore $J[Q]$, inizializzato a 0, incrementato ad ogni invio di un messaggio al joiner e decrementato al momento della ricezione di $\{\texttt{no\_computations}, Q\}$ da parte di quest'ultimo — righe 7, 16, 24 e 31.

Dunque, qualora $S[Q]$ sia *false* e $J[Q]$ sia pari a 0, il router concluderà che i due sottoprocessi sono inattivi ed invierà al requester un messaggio di inattività contenente il valore $W[Q]$ — che, per un processo P, corrisponde al valore $w_{P,Q}$ di Definizione 5.7 —, come richiesto dall'algoritmo di Huang. Abbiamo così definito il criterio d'arresto dell'algoritmo di Amalgamazione Distribuita richiesto da (CDP 7).

6

Simulazioni Lasche via Amalgamazione Distribuita

Capitolo 5 introduce un algoritmo generale per la computazione di predicati in ambiente distribuito. Studieremo ora un'istanziazione di questo per risolvere il problema del vuoto delle simulazioni lasche, introdotto in Definizione 4.1. Ricordiamo come, perché l'algoritmo possa essere istanziato correttamente, risulti necessario aggiungere ad esso le definizioni richieste dalle condizioni (CDP1-6). Dobbiamo quindi indicare

- un modo per partizionare un host ed assegnare ogni sua parte ad un processo — come richiesto in (CDP 1). Questo argomento verrà trattato in Sezione 6.1;
- una nozione di *simulazione lasca parziale*, che comprenda anche le simulazioni lasche e costituisca l'insieme delle computazioni — come indicato in (CDP 2). Su questo insieme deve inoltre essere definita un'operazione di join binaria — Condizione (CDP 3) — e un insieme di generatori computabili localmente — Condizione (CDP 4). Le definizioni riguardanti questa parte saranno introdotte in Sezione 6.2;
- una nozione d'instradamento che permetta la computazione di ogni soluzione — secondo quanto detto in (CDP 5). Questa nozione verrà studiata in Sezione 6.5;
- un criterio che permetta ad un processo di individuare quando una simulazione lasca è stata computata — Condizione (CDP 6) —; definito in Sezione 6.2.

Per semplicità, in seguito considereremo unicamente guests per i quali la soluzione vuota $(\emptyset, \emptyset)$ non è una simulazione lasca. Da Definizione 2.7 e rispetto ad un guest G ed un host H, si può notare come $(\emptyset, \emptyset) \notin \mathbb{S}^{G \to H}$ se e solo se l'insieme must di G non è vuoto. Si noti come, rispetto al problema del vuoto per simulazioni lasche, restringerci ai soli guests con insieme must diverso dall'insieme vuoto non elimini alcun caso interessante.

6.1 Host distribuito

Per poter applicare correttamente l'algoritmo di Amalgamazione Distribuita è necessario indicare come distribuire un host rispetto ad una famiglia di processi, secondo quanto detto in (CDP 1). In questa

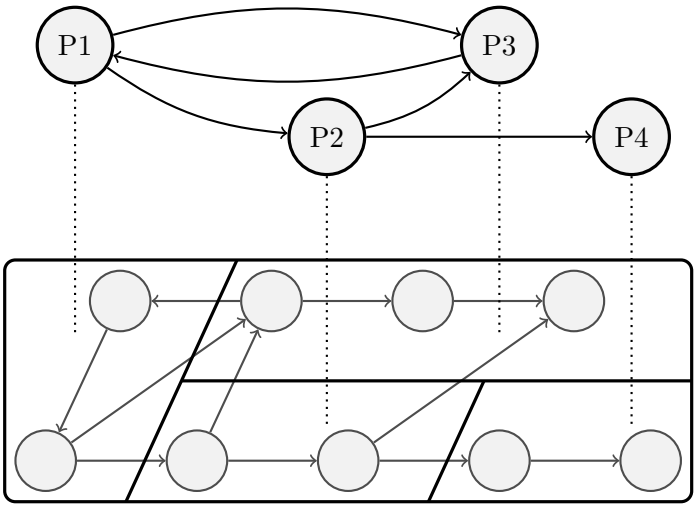


Figura 6.3: Strato di rete semantico (in alto) dovuto alla partizione dell'host (in basso) tra quattro processi.

sezione illustreremo questa idea di *host distribuito* e mostreremo che la sua partizione definisce uno *strato di rete semantico*. In seguito, sia $H = (\Sigma, V, E)$ un grafo host e denotiamo con Proc una famiglia di processi. Iniziamo con introdurre il concetto di partizione di host, nella quale, intuitivamente, ogni vertice di H viene assegnato ad un unico processo di Proc.

Definizione 6.1 (Partizione di un host)**.** *Una partizione di un host H rispetto a* Proc *è una mappa $\mathcal{P} : V \to$ Proc assegnante ogni vertice di H ad un processo. Diremo che $v \in V$ è* locale *a $P \in$ Proc, o analogamente che P è il* possessore *di v, qualora $\mathcal{P}(v) = P$. Questa nozione è estesa agli archi di H: un processo P è detto essere* possessore *di un arco $e \in E$ se $\mathcal{P} \circ s(e) = P$ oppure $\mathcal{P} \circ t(e) = P$. Inoltre, un arco è detto essere* locale *a P se $\mathcal{P} \circ s(e) = \mathcal{P} \circ t(e) = P$; in caso contrario è detto essere* condiviso *tra due processi P e Q, dove $\mathcal{P} \circ s(e) = P \neq Q = \mathcal{P} \circ t(e)$.*

La partizione induce quindi una nozione di località dell'host distribuito rispetto alla famiglia di processi. Inoltre, la nozione di archi condivisi induce una relazione di *adiacenza* tra processi. In seguito, sia $\mathcal{P}\colon V \to$ Proc una partizione sull'host H.

Definizione 6.2 (Adiacenza)**.** *Siano $P, Q \in$ Proc due processi. P è detto essere* adiacente *a Q rispetto alla partizione $\mathcal{P}$ qualora esista un arco $e \in E$ condiviso tra P e Q tale che $s(e)$ è locale a P. Denoteremo con $P \overset{\mathcal{P}}{\circ\!\!\to} Q$ l'adiacenza di P a Q rispetto a $\mathcal{P}$.*

La relazione di adiacenza definisce un grafo diretto con nodi in Proc, ovvero uno *strato di rete semantico* $\mathcal{N}_{\mathcal{P}}$. Questa rete ha un significato preciso: riflette l'adiacenza degli elementi dell'host detenuti dai singoli processi della rete. Due processi sono infatti adiacenti se e solo se possiedono componenti adiacenti in H. Questa corrispondenza tra hosts e rete semantica è rappresentata in Figura 6.3.

La rete $\mathcal{N}_{\mathcal{P}}$ è tale che il cammino minimo da un processo P ad un processo Q non può essere maggiore del cammino minimo che connette, nell'host H, nodi locali a P con nodi di Q.

Proposizione 6.4 (Cammino minimo tra due processi)**.** *Siano $v_1, v_2 \in V$ due nodi di H. Se esiste almeno un cammino da v_1 a v_2,* i.e. *$\mathbb{P}_H(v_1, v_2) \neq \emptyset$, allora il cammino minimo che connette il processo*

$\mathcal{P}(v_1)$ *al processo* $\mathcal{P}(v_2)$ *esiste e la sua lunghezza è limitata dalla lunghezza del cammino minimo in* $\mathbb{P}_H(v_1, v_2)$.

Dimostrazione. Definizioni 6.1 e 6.2 caratterizzano il quoziente indotto da $\mathcal{P}$ su H. □

Come vedremo in Sezione 6.5, il concetto di adiacenza sarà cruciale per definire l'instradamento delle soluzioni parziali lungo la rete. Si noti come il flusso delle informazioni di messaggi che passano unicamente attraverso la rete semantica $\mathcal{N}_\mathcal{P}$ possa non essere confluente. In questo caso è impossibile garantire che, utilizzando unicamente lo strato di rete semantico per l'instradamento dei messaggi, vengano computate tutte le soluzioni. Con riferimento a Figura 6.3, poniamo per esempio esista una simulazione lasca che coinvolga vertici assegnati ai processi *P2*, *P3* e *P4*: seguendo la rete semantica, i messaggi possono unicamente essere trasmessi da *P2* a *P3* e da *P2* a *P4*. In nessun modo si potranno quindi unire soluzioni parziali che coinvolgano tutti e tre i processi. Per risolvere questo problema, assumiamo dunque che tutti i processi in $\mathcal{P}(V)$ siano in grado di comunicare tra di loro, ovvero:

Assunzione 6.5. *È possibile estendere* $\mathcal{N}_\mathcal{P}$ *in modo tale che esista in essa un cammino per ogni coppia di processi.*

Con abuso di notazione, d'ora in avanti $\mathcal{N}_\mathcal{P}$ indicherà una rete semantica con questa proprietà di connessione. Questo assicura che esita sempre un punto di *rendezvous* per due messaggi e non elimina dunque la completezza dell'algoritmo distribuito per il calcolo delle simulazioni lasche. Denoteremo con $|\mathcal{N}_\mathcal{P}|$ il numero di archi nello strato di rete semantico. Si noti infine come supponiamo i processi conoscano, dato un arco in loro possesso ma non locale, l'identità del processo con il quale questo è condiviso. Anche questa informazione sarà di fondamentale importanza nella definizione di instradamento delle soluzioni parziali — Sezione 6.5.

Prima di introdurre le *simulazioni lasche parziali*, le quali corrispondono alle soluzioni parziali computate durante l'esecuzione dell'algoritmo di Amalgamazione Distribuita, terminiamo questa sezione sul partizionamento di hosts estendendo il concetto di *possesso* ad una qualsiasi relazione tra nodi di un guest e quelli di un host.

Definizione 6.6 (Possessori di una relazione)**.** *Siano* V_G *e* V_H *due insiemi (di vertici) e sia* $\mathcal{P} : V_H \to \text{Proc}$ *una mappa. L'insieme dei possessori di una relazione* $\phi \subseteq V_G \times V_H$ *rispetto a* $\mathcal{P}$*, denotato con* $\mathcal{O}_\mathcal{P}(\phi)$*, è costituito da tutti i processi che possiedono almeno un vertice in* $\pi_2(\phi)$*,* i.e. $\mathcal{O}_\mathcal{P}(\phi) \triangleq \mathcal{P}(\pi_2(\phi))$.

Questo concetto verrà a breve utilizzato per individuare i possessori di una simulazione lasca parziale.

6.2 Simulazioni Lasche Parziali

Una volta definito come distribuire la rappresentazione del sistema (l'host) tra i vari processi, l'algoritmo di Amalgamazione Distribuita richiede di fornire una definizione di soluzione parziale e un criterio per decidere quando questa sia completa, un insieme finito di generatori computabili localmente, un modo per unire soluzioni parziali ed un ordinamento su queste. Iniziamo dunque con l'affrontare il primo di questi problemi andando ad introdurre le *simulazioni lasche parziali*, relazioni rappresentanti le soluzioni parziali richieste dall'algoritmo in (CDP 2) e computate dai vari processi localmente o tramite cooperazione, *i.e.* attraverso scambio e composizione di queste.

In seguito, siano $H = (\Sigma, V_H, E_H)$ e $G = (\Sigma, V_G, E_G, \mathcal{M}, \mathcal{U}, \mathcal{E}, \mathcal{C})$ rispettivamente un host ed un guest per simulazioni lasche. Siano inoltre Proc un insieme di processi coinvolti nell'algoritmo di Amalgamazione Distribuita, $\mathcal{P} : V \to$ Proc una partizione di H rispetto a Proc ed infine $\mathbf{P} \subseteq$ Proc un sottoinsieme di processi.

Definizione 6.7 (Simulazione lasca parziale)**.** *Una* simulazione lasca parziale *di G in H, rispetto ad una partizione $\mathcal{P}$ ed un insieme di processi $\mathbf{P}$, è una coppia di relazioni (ϕ, η) con $\phi \subseteq V_G \times V_H$ e $\eta \subseteq E_G \times E_H$ tali che:*

(SLP 1) $\mathbf{P}$ *è l'insieme dei possessori di ϕ, inoltre tutti gli archi in η sono posseduti da processi in $\mathbf{P}$. Formalmente:*

$$\mathcal{O}_{\mathcal{P}}(\phi) = \mathbf{P}$$
$$\forall e \in \pi_2(\eta) : \ \mathcal{P} \circ s(e) \in \mathbf{P} \ \vee \ \mathcal{P} \circ t(e) \in \mathbf{P}$$

(SLP 2) *se un vertice di G appartiene all'insieme* unique *allora può essere in relazione al massimo con un vertice di H:*

$$\forall u \in \mathcal{U} \ |\phi(u)| \leq 1$$

dove $\phi(u) \triangleq \{v \mid (u, v) \in \phi\}$;

(SLP 3) *se un vertice di G appartiene all'insieme* exclusive*, allora i nodi di H in relazione con esso non possono essere in relazione con altri nodi di G. Formalmente,*

$$\forall u \in \mathcal{E} \ \forall v \in V_G \backslash \{u\} \ \phi(u) \cap \phi(v) = \emptyset$$

(SLP 4) *ϕ e η preservano sorgenti e destinazioni locali ai processi in $\mathbf{P}$, inoltre η mette in relazione unicamente coppie di archi con la stessa etichetta,* i.e. *per ogni $(e, e') \in \eta$*

$$\sigma(e) = \sigma(e')$$
$$\mathcal{P} \circ s(e') \in \mathbf{P} \Rightarrow (s(e), s(e')) \in \phi$$
$$\mathcal{P} \circ t(e') \in \mathbf{P} \Rightarrow (t(e), t(e')) \in \phi$$

(SLP 5) H simula localmente G *rispetto alla funzione di scelta $\mathcal{C}$ ed i processi in $\mathbf{P}$: per ogni $(u, v) \in \phi$ esiste $\gamma \in \mathcal{C}(u)$ tale che γ è* selezionato *in η rispetto a v. Inoltre, per ogni $(e, e') \in \eta$, se $\mathcal{P} \circ s(e') \in \mathbf{P}$ allora esiste $\gamma \in \mathcal{C}(s(e))$ tale che $e \in \gamma$ e γ è* selezionato *in η rispetto a $s(e')$.*

dove, come formalizzato in Definizione 2.7,

$$\gamma \text{ è selezionato in } \eta \text{ rispetto a } v \iff \forall e \in \gamma \ \exists e' \in out(v) \ (e, e') \in \eta$$

(SLP 6) *La simulazione rispetta la raggiungibilità per quanto riguarda i nodi nell'insieme* must*: per ogni $(u, v) \in \phi$ e $w \in \mathcal{M}$ se $\mathbb{P}_G(u, w) \neq \emptyset$ allora esiste $v' \in V_H$ tale che $(w, v') \in \Delta_\eta((u, v))$ oppure esiste $(e, e') \in Y_\eta$ tale che $(s(e), s(e')) \in \Delta_\eta((u, v)) \ \wedge \ \mathbb{P}_G(t(e), w) \neq \emptyset$, dove $Y_\eta \triangleq \{(e, e') \in \eta \mid$*

$\mathcal{P} \circ t(e') \notin \mathbf{P}\}$ *è l'insieme delle coppie di* η *il cui arco dell'host ha sorgente locale ad un processo di* $\mathbf{P}$ *ma terminazione locale ad un processo in* $\mathrm{Proc} \setminus \mathbf{P}$*;*

Con $p\mathbb{S}^{G\to H}_{\mathcal{P},\mathbf{P}}$ *denoteremo il dominio di tutte le simulazioni lasche parziali del guest* G *nell'host* H*, rispetto alla partizione* $\mathcal{P}$ *e l'insieme di processi* $\mathbf{P}$*. L'unione di tutti questi domini, al variare di* $\mathbf{P}$*, è l'insieme di tutte le simulazioni parziali di* G *in* H *rispetto a* $\mathcal{P}$*, ovvero*

$$p\mathbb{S}^{G\to H}_{\mathcal{P}} \triangleq \bigcup_{\mathbf{P}\subseteq \mathrm{Proc}} p\mathbb{S}^{G\to H}_{\mathcal{P},\mathbf{P}}$$

Informalmente, una *simulazione lasca parziale* può essere vista come una coppia di relazioni soggette agli stessi vincoli di una *simulazione lasca* per ogni nodo o arco di H posseduto da un processo di $\mathbf{P}$; infatti Condizione (SLP 1) impone che solo questi elementi possano apparire nella relazione. Per questo motivo, le condizioni (SLP 2) e (SLP 3) sono uguali alle rispettive condizioni di Definizione 2.7: esse infatti non vanno a porre condizioni al di fuori del codominio di ϕ. Analogamente, Condizione (SLP 4) è una naturale generalizzazione della condizione (SL 4), dove sorgenti e destinazioni di una coppia di archi $(e, e') \in \eta$ vengono messi in corrispondenza in ϕ solo qualora siano locali ad un processo di $\mathbf{P}$. Condizione (SLP 6) è invece modificata e merita un'analisi più approfondita. In essa appare fondamentale il ruolo delle coppie di archi di η con sorgente o terminazione non locali a $\mathbf{P}$. Queste coppie, come vedremo a breve, ricoprono un ruolo centrale nell'algoritmo e possono essere viste come una *frontiera* della simulazione. Attraverso questa frontiera, Condizione (SLP 6) estende Condizione (SL 6): esattamente come quest'ultima, infatti, viene richiesto che dato un elemento $(u, v) \in \phi$ ed un vertice $w \in \mathcal{M}$ dell'insieme must, da (u, v) si possa raggiungere un vertice di H che simuli w, secondo la funzione di raggiungibilità duale Δ_η introdotta in Definizione 2.6. Diversamente da Condizione (SL 6), tuttavia, qualora non sia possibile raggiungere tale vertice, viene richiesto che si possa raggiungere una coppia di nodi $(u', v') \in \phi$ per la quale esiste una coppia $(e, e') \in \eta$ di *frontiera*, con $s(e) = u'$ e $s(e') = v'$, per la quale da $t(e) \in V_G$ parta un cammino, nel guest, che termini in w. La motivazione dietro questa modifica, rispetto al vincolo per le simulazioni lasche, è semplice: qualora la raggiungibilità di un nodo must non sia verificata all'interno di una simulazione lasca parziale, essa può essere ancora recuperata tramite composizione di tale soluzione con una seconda simulazione lasca parziale. Tuttavia risulta necessario sia raggiungibile un elemento $(e, e') \in \eta$ la cui terminazione $(t(e), t(e'))$, mappata nella seconda simulazione lasca parziale, debba a sua volta raggiungere tale nodo must. Si noti infine come nella definizione di simulazioni lasche parziali non esista una condizione corrispondente a (SL 1).

Nota. Esattamente come visto per le simulazioni lasche, la computazione delle simulazioni lasche parziali è un problema NP-completo. Il modello di programmazione lineare mista visto in Capitolo 4 può, con opportune modifiche, essere infatti utilizzato per generare simulazioni lasche parziali. La rivisitazione di questo modello è spiegata all'interno di Appendice B, la quale presenta i dettagli dell'istanziazione dell'algoritmo di Amalgamazione Distribuita per il problema del vuoto delle simulazioni lasche. Il modello ivi presentato impone essenzialmente che i vincoli di simulazione lasca vengano rispettati solo dalle variabili associate a nodi e archi locali ad un determinato insieme di processi.

Prima di considerare l'operazione di unione, l'ordinamento e l'insieme di generatori richiesti dall'algoritmo distribuito, è necessario individuare sotto quali assunzioni una soluzione parziale, computata

da un certo insieme di processi, possa essere ritenuta completa, ovvero una simulazione lasca. Innanzitutto, data la sua importanza nei risultati seguenti, formalizziamo meglio il concetto di *frontiera* di una simulazione lasca parziale.

Definizione 6.8 (Frontiera di una simulazione lasca parziale)**.** *Sia* $(\phi, \eta) \in p\mathbb{S}^{G \to H}_{\mathcal{P},\mathbf{P}}$ *una simulazione lasca parziale. La sua frontiera è una coppia* (X_η, Y_η) *tale che:*

- X_η*, chiamato* parte entrante *della frontiera, è l'insieme di tutte le coppie* $(e, e') \in \eta$ *tali che la sorgente di* e' *non è locale ad alcun processo di* $\mathbf{P}$*. Da Condizione (SLP 1) ne consegue inoltre che* $\mathcal{P} \circ t(e') \in \mathbf{P}$*. Formalmente:*

$$X_\eta \triangleq \{(e, e') \in \eta \mid \mathcal{P} \circ s(e') \notin \mathbf{P} \}$$

- Y_η*, chiamato* parte uscente *della frontiera, è l'insieme di tutte le coppie* $(e, e') \in \eta$ *tali che la terminazione di* e' *non è locale ad alcun processo di* $\mathbf{P}$*. Vale inoltre che* $\mathcal{P} \circ s(e') \in \mathbf{P}$ *— Condizione (SLP 1). Formalmente:*

$$Y_\eta \triangleq \{(e, e') \in \eta \mid \mathcal{P} \circ t(e') \notin \mathbf{P} \}$$

Data la definizione di frontiera, risulta molto semplice mostrare sotto quali condizioni una simulazione lasca parziale è una soluzione completa, ovvero una simulazione lasca.

Teorema 6.9 (Individuazione di una soluzione)**.** *Sia* $(\phi, \eta) \in p\mathbb{S}^{G \to H}_{\mathcal{P}}$ *una simulazione lasca parziale e sia* (X_η, Y_η) *la sua frontiera.* (ϕ, η) *è una simulazione lasca se e solo se* $X_\eta = Y_\eta = \emptyset$ *e* $\mathcal{M} \subseteq \pi_1(\phi)$*.*

Dimostrazione. È triviale mostrare come, sotto ipotesi $X_\eta = Y_\eta = \emptyset$ e $\mathcal{M} \subseteq \pi_1(\phi)$, Definizione 6.7 coincida con Definizione 2.7. Innanzitutto, $\mathcal{M} \subseteq \pi_1(\phi)$ è equivalente alla condizione (SL 1). Le Condizioni (SLP 2) e (SLP 3) sono già equivalenti alle rispettive condizioni per simulazioni lasche, (SL 2) e (SL 3). L'ipotesi $X_\eta = Y_\eta = \emptyset$ implica che, per ogni $(e, e') \in \eta$, sia sorgente che destinazione di e' appartengano a processi di $\mathbf{P}$, ovvero

$$\forall e \in \pi_2(\eta)\ \mathcal{P} \circ s(e) \in \mathbf{P} \ \wedge\ \mathcal{P} \circ t(e) \in \mathbf{P}$$

Sotto questa ipotesi il vincolo sugli archi di Condizione (SLP 1) è verificato. Lo stesso vale per le Condizioni (SLP 4) e (SLP 5), che diventano equivalenti alle rispettive condizioni nella definizione di simulazioni lasche. Infine, anche Condizione (SLP 6) diventa equivalente alla condizione (SL 6), dato che il predicato che richiede l'esistenza di una coppia in Y_η risulta falso e quindi è necessario che, per ogni $(u, v) \in \phi$ e $w \in \mathcal{M}$ tale che $\mathbb{P}_G(u, w) \neq \emptyset$, esista $v' \in V_H$ tale che $(w, v') \in \Delta_\eta((u, v))$. □

Dato che le ipotesi $X_\eta = Y_\eta = \emptyset$ e $\mathcal{M} \subseteq \pi_1(\phi)$ implicano l'equivalenza tra definizione di simulazioni lasche e la loro versione parziale, vale inoltre il seguente corollario.

Corollario 6.10 (Indipendenza dalla partizione)**.** *Sia* $(\phi, \eta) \in \mathbb{S}^{G \to H}$ *una simulazione lasca. Per ogni insieme di processi* Proc *ed ogni partizione* $\mathcal{P} : V_H \to \text{Proc}$ *si ha* $(\phi, \eta) \in p\mathbb{S}^{G \to H}_{\mathcal{P}}$*.*

Questi risultati identificano quanto richiesto dalla condizione (CDP 6) dell'algoritmo di Amalgamazione Distribuita. A meno che non sia computabile localmente da un singolo processo, perché una soluzione venga trovata è necessario definire l'operazione di join (o unione) di simulazioni lasche parziali, secondo quanto espresso in Condizione (CDP 3).

Definizione 6.11 (Join e inclusione di simulazioni lasche parziali)**.** *Siano* $A = (\phi_A, \eta_A) \in p\mathbb{S}^{G\to H}_{\mathcal{P},\mathbf{P}}$ *e* $B = (\phi_B, \eta_B) \in p\mathbb{S}^{G\to H}_{\mathcal{P},\mathbf{Q}}$ *due simulazioni lasche parziali. Il loro* join *(o* unione*) è definito come*

$$A \sqcup B \triangleq \begin{cases} (\phi_A \cup \phi_B,\ \eta_A \cup \eta_B) & \text{if } (\phi_A \cup \phi_B,\ \eta_A \cup \eta_B) \in p\mathbb{S}^{G\to H}_{\mathcal{P},\mathbf{P}\cup\mathbf{Q}} \\ \top & \text{altrimenti} \end{cases}$$

dove $\top$ *corrisponde ad una soluzione erronea, scartata dal processo che la computa. Accanto all'unione di simulazioni lasche parziali definiamo il loro ordinamento per inclusione, ovvero*

$$A \sqsubseteq B \overset{\triangle}{\iff} \phi_A \subseteq \phi_B \ \wedge\ \eta_A \subseteq \eta_B$$

Essendo $p\mathbb{S}^{G\to H}_{\mathcal{P}}$ un insieme finito, $\sqsubseteq$ risulta essere un *ordine parziale completo*. Nella definizione appena formulata, $\top$ corrisponde ad una soluzione erronea, scartata dal processo che l'ha computata. Si noti come join e ordinamento appena definiti possano essere estesi ponendo $\top$ come estremo superiore, *i.e.* per ogni $A \in p\mathbb{S}^{G\to H}_{\mathcal{P}} \cup \{\top\}$ $A \sqsubseteq \top$ ed l'unione venga estesa con $A \sqcup \top = \top \sqcup A \triangleq \top$. Rispetto a questa estensione, considerando che l'operazione di unione appena definita è associativa, commutativa e idempotente, vale il seguente risultato:

Proposizione 6.12 (Semireticolo delle simulazioni lasche parziali)**.** $(p\mathbb{S}^{G\to H}_{\mathcal{P}} \cup \{\top\}, \sqcup)$ *è un semireticolo.*

Infine, per poter applicare l'algoritmo distribuito, è necessario definire un insieme finito di generatori e mostrare che esso permette di computare tutte (e solo) le simulazioni lasche — Condizione (CDP 4). Questo insieme di generatori è introdotto attraverso la definizione di *simulazione lasca parziale locale*.

Definizione 6.13 (Simulazione lasca parziale locale)**.** *Una* simulazione lasca parziale $(\phi, \eta) \in p\mathbb{S}^{G\to H}_{\mathcal{P},\mathbf{P}}$ *è detta* locale *rispetto ad un processo* P *se e solo se* $\mathbf{P} = \{P\}$.

Prendiamo dunque come insieme di generatori l'insieme (finito) di tutte le simulazioni lasche parziali locali, definito formalmente come

$$\mathbb{G}^{G\to H}_{\mathcal{P}} \triangleq \{(\phi, \eta) \in p\mathbb{S}^{G\to H}_{\mathcal{P},\mathbf{P}} \mid |\mathbf{P}| = 1\}$$

Legato a questo insieme di generatori, troviamo il seguente risultato di completezza.

Teorema 6.14 (Completezza)**.** *Per ogni simulazione lasca* $(\phi, \eta) \in \mathbb{S}^{G\to H}$*, ogni insieme di processi* Proc *ed ogni partizione di un host* $\mathcal{P} : V_H \to \text{Proc}$ *esiste un (unico) insieme* $\Psi \subseteq \mathbb{G}^{G\to H}_{\mathcal{P}}$ *tale che* $(\phi, \eta) = \bigsqcup \Psi$ *e ogni processo è possessore di al massimo una simulazione lasca parziale in* Ψ.

Dimostrazione. Segue direttamente da Corollario 6.10 e Definizioni 2.7, 6.7 e 6.11. Sia

$$\Psi \triangleq \left\{ (\phi_P, \eta_P) \;\middle|\; \begin{array}{l} P \in \mathcal{O}_{\mathcal{P}}(\phi) \\ \phi_P = \{(u,v) \in \phi \mid \mathcal{P}(v) = P\} \\ \eta_P = \{(e,e') \in \eta \mid \mathcal{P} \circ s(e') = P \;\vee\; \mathcal{P} \circ t(e') = P\} \end{array} \right\}$$

Vale $(\phi, \eta) = \bigsqcup \Psi$ dato che Ψ è una partizione di (ϕ, η) indotta dalla funzione $\mathcal{P}$. Inoltre, per definizione di Ψ e dato che le proprietà di Definizione 2.7 assicurano le condizioni di Definizione 6.7, tutte le $\psi \in \Psi$ sono simulazioni lasche parziali locali. Infine, sempre dalla definizione di Ψ, segue direttamente che ogni processo in Proc possiede al massimo una simulazione lasca parziale di Ψ. □

Ricordiamo inoltre come, tramite Teorema 6.9, abbiamo dimostrato sotto quali condizioni la definizione di simulazione lasca coincide con la sua variante parziale. Questo risultato, unito alla definizione di join di simulazioni lasche parziali, garantisce che l'insieme di generatori produca unicamente soluzioni corrette, secondo quanto espresso dal seguente teorema.

Teorema 6.15 (Correttezza)**.** *Sia $\Psi \subseteq \mathbb{G}_{\mathcal{P}}^{G \to H}$ un insieme di simulazioni lasche parziali locali. Se $(\phi, \eta) = \bigsqcup \Psi$ è definito, $\mathcal{M} \subseteq \pi_1(\phi)$ e la sua frontiera (X_η, Y_η) è tale che $Y_\eta = X_\eta = \emptyset$, allora (ϕ, η) è una simulazione lasca.*

Infine, si noti come Teorema 6.14 fornisca, oltre al risultato di completezza, un modo di partizionare le simulazioni lasche in modo tale che ogni processo computi al più una parte di queste. Questo permette di ridurci alle unioni di simulazioni lasche computabili da insiemi diversi di processi e le cui frontiere siano componibili.

Assunzione 6.16. *Date due simulazioni lasche parziali $(\phi, \eta) \in p\mathbb{S}_{\mathcal{P},\mathbf{P}}^{G \to H}$ e $(\phi', \eta') \in p\mathbb{S}_{\mathcal{P},\mathbf{Q}}^{G \to H}$, queste verranno composte tramite join solo qualora*

- $\mathbf{P} \cap \mathbf{Q} = \emptyset$
- *per ogni $(e, e') \in X_\eta$, se la sorgente di e' è locale ad un processo di $\mathbf{Q}$,* i.e. *$\mathcal{P} \circ s(e') \in \mathbf{Q}$, allora $(e, e') \in Y_{\eta'}$. Simmetricamente, per ogni $(e, e') \in X_{\eta'}$, se $\mathcal{P} \circ s(e') \in \mathbf{P}$ allora $(e, e') \in Y_\eta$.*
- *per ogni $(e, e') \in Y_\eta$, se la terminazione di e' è locale ad un processo di $\mathbf{Q}$,* i.e. *$\mathcal{P} \circ t(e') \in \mathbf{Q}$, allora $(e, e') \in X_{\eta'}$. Simmetricamente, per ogni $(e, e') \in Y_{\eta'}$, se $\mathcal{P} \circ t(e') \in \mathbf{P}$ allora $(e, e') \in X_\eta$.*

Si noti inoltre come la prima di queste condizioni implichi $X_\eta \cap X_{\eta'} = \emptyset$ e $Y_\eta \cap Y_{\eta'} = \emptyset$. Questa proprietà verrà utilizzata nella prossima sezione dove mostreremo come, rispetto al join di due soluzioni parziali, non sia necessario né ricontrollare completamente tutte le condizioni di simulazione lasca parziale — come invece sembrerebbe suggerire Definizione 6.11 — né sia necessario conoscere tutta la soluzione parziale. Questo ci permette di *astrarre* le simulazioni lasche parziali riducendo notevolmente il numero di soluzioni inviate lungo la rete, la loro dimensione ed il tempo necessario per calcolarne l'unione.

6.3 Interfaccia di una Simulazione Lasca Parziale

Le simulazioni lasche parziali computate da un singolo processo possono essere, nel peggiore dei casi, in numero esponenziale rispetto alla dimensione dell'host. È dunque impensabile inviare tutte queste soluzioni ai vari processi e ancor meno compierne il join. Per questo motivo risulta necessario cercare di ridurre il numero di soluzioni parziali e definire una buona strategia d'instradamento dei messaggi in modo da compiere meno unioni possibili, pur mantenendo la completezza del metodo. In questa sezione vedremo come risolvere la prima di queste due richieste, introducendo la nozione di *interfaccia* di una simulazione lasca parziale.

In seguito, siano $H = (\Sigma, V_H, E_H)$ e $G = (\Sigma, V_G, E_G, \mathcal{M}, \mathcal{U}, \mathcal{E}, \mathcal{C})$ rispettivamente un host ed un guest per simulazioni lasche. Siano inoltre Proc un insieme di processi, $\mathcal{P} : V \to \text{Proc}$ una partizione di H rispetto a Proc ed infine $\mathbf{P} \subseteq \text{Proc}$ un sottoinsieme di processi.

Informalmente, un'interfaccia può essere vista come un'astrazione di una simulazione lasca parziale, dove vengono parzialmente dimenticate le informazioni non riguardanti la sua frontiera. Questo può essere fatto in quanto, sotto Assunzione 6.16, date due simulazioni lasche parziali $(\phi, \eta) \in p\mathbb{S}^{G\to H}_{\mathcal{P},\mathbf{P}}$ e $(\phi', \eta') \in p\mathbb{S}^{G\to H}_{\mathcal{P},\mathbf{Q}}$, l'unione $(\phi \cup \phi',\ \eta \cup \eta')$ non può violare Condizioni (SLP 1), (SLP 3), (SLP 4) e (SLP 5). Infatti, $\mathbf{P} \cap \mathbf{Q} = \emptyset$ implica $\pi_2(\phi) \cap \pi_2(\phi') = \emptyset$, e dunque Condizione (SLP 3) è rispettata. Inoltre, gli altri due vincoli di Assunzione 6.16 fanno si che *H simuli localmente G* rispetto ai processi $\mathbf{P} \cup \mathbf{Q}$, rendendo quindi rispettata Condizione (SLP 5). Questi due vincoli rendono inoltre rispettata la consistenza tra $\phi \cup \phi'$ e $\eta \cup \eta'$ richiesta da Condizione (SLP 4). Infine, la condizione (SLP 1) è sempre rispettata, anche in assenza di Assunzione 6.16. Le informazioni, contenute in una simulazione lasca parziale, necessarie per verificare queste condizioni sono quindi superflue e, laddove ortogonali alle informazioni necessarie alla verifica di Condizioni (SLP 2) e (SLP 6), possono essere rimosse. Questa operazione di rimozione corrisponde all'astrazione da simulazione lasca parziale alla sua interfaccia.

Definizione 6.17 (Interfaccia di una simulazione lasca parziale)**.** *Sia $(\phi, \eta) \in p\mathbb{S}^{G\to H}_{\mathcal{P},\mathbf{P}}$ una simulazione lasca parziale, con frontiera (X_η, Y_η). La sua interfaccia è definita dalla* 8*-tupla*

$$\mathbf{P} \vdash \langle X_\eta, \mu_\mathcal{M}, \mu_Y \rangle \xrightarrow{M,\ U} \langle Y_\eta, \nu \rangle$$

nella quale

$$\begin{aligned}
M &\triangleq \mathcal{M} \cap \pi_1(\phi) \\
U &\triangleq \mathcal{U} \cap \pi_1(\phi) \\
\mu_\mathcal{M}((e,e')) &\triangleq \mathcal{M} \cap (\{t(e)\} \cup \pi_1(\Delta_\eta(t(e), t(e')))) && \mu_\mathcal{M} : X_\eta \to \mathcal{P}(\mathcal{M}) \\
\mu_Y((e,e')) &\triangleq \{(d,d') \in Y_\eta \mid (s(d), s(d')) \in \Delta_\eta(t(e), t(e')) \cup \{(t(e), t(e'))\}\} && \mu_Y : X_\eta \to \mathcal{P}(E_G \times E_H) \\
\nu((e,e')) &\triangleq \{u \in \mathcal{M} \mid t(e) = u \vee \mathbb{P}_G(t(e), u) \neq \emptyset\} && \nu : Y_\eta \to \mathcal{P}(\mathcal{M})
\end{aligned}$$

Denoteremo con $\mathcal{I}\langle p\mathbb{S}^{G\to H}_{\mathcal{P},\mathbf{P}}\rangle$ il dominio di tutte le interfacce associate alle simulazioni lasche parziali di $p\mathbb{S}^{G\to H}_{\mathcal{P},\mathbf{P}}$. Similmente, data una simulazione lasca parziale $A \in p\mathbb{S}^{G\to H}_{\mathcal{P},\mathbf{P}}$, denotiamo con $\mathcal{I}(A)$ la sua interfaccia.

Dalla definizione appena data, l'interfaccia di una simulazione lasca parziale $A = (\phi_A, \eta_A)$ è composta dall'insieme dei processi $\mathbf{P} = \mathcal{O}_{\mathcal{P}}(\phi_A)$ che possiedono A; dalla sua frontiera; da due insiemi M e U contenenti rispettivamente i vertici del guest che compaiono in ϕ_A e sono appartenenti rispettivamente agli insiemi must e unique; e da tre funzioni $\mu_{\mathcal{M}}$, μ_Y e ν.

- $\mu_{\mathcal{M}}$, dato un elemento $(e, e') \in X_\eta$, ritorna l'insieme dei nodi del guest appartenenti all'insieme must raggiungibili tramite Δ_η dalla coppia di nodi terminazione $(t(e), t(e'))$.
- μ_Y, dato un elemento $(e, e') \in X_\eta$, ritorna l'insieme degli elementi della parte uscente della frontiera Y_η che sono raggiungibili tramite Δ_η.
- ν, dato un elemento $(e, e') \in Y_\eta$, indica l'insieme dei nodi appartenenti all'insieme must per i quali esiste un cammino, nel guest, a partire da $t(e)$, più eventualmente anche $t(e)$, se questi è un nodo di $\mathcal{M}$.

Queste tre funzioni hanno un ruolo centrale nel calcolo del join di interfacce — che verrà introdotto e studiato a breve. In particolare, la funzione ν corrisponde a requisiti da soddisfare collegati con Condizione (SLP 6): si richiede infatti che a partire da un elemento $y \in Y_\eta$ appartenente alla parte uscente della frontiera di A siano raggiungibili in S tutti i nodi in $\nu(y)$, dove S è una simulazione lasca ottenuta tramite A. Durante il join di due interfacce, per soddisfare queste richieste basterà osservare le due funzioni μ, rispetto all'elemento y visto come appartenente alla parte entrante della frontiera di B, seconda simulazione lasca parziale t.c. $A \sqcup B \sqsubseteq S$. Inoltre, per quanto ν sia dipendente dalle informazioni del guest, risulta indipendente dalla struttura dell'host. I processi possono quindi precomputare la funzione

$$\widehat{\nu} \triangleq \lambda v : E_G.\{u \in \mathcal{M} \mid v = u \vee \mathbb{P}_G(v, u) \neq \emptyset\}$$

ed utilizzarla per calcolare ν come $\nu((e, e')) = \widehat{\nu}(t(e))$, senza che questa debba quindi essere inviata attraverso la rete.

Nota. Una astrazione può essere vista come una funzione $\alpha : \mathbb{C} \to \mathbb{A}$ tra un dominio concreto $\mathbb{C}$, nel nostro caso $p\mathbb{S}_{\mathcal{P}}^{G \to H}$, ed uno astratto $\mathbb{A}$, ovvero il dominio di tutte le interfacce $\mathcal{I}\langle p\mathbb{S}_{\mathcal{P}}^{G \to H}\rangle$. Solitamente α è una funzione suriettiva. Nel nostro caso inoltre, essa non è iniettiva: più simulazioni lasche parziali possono avere la stessa interfaccia. A patto di non inviare più volte un'interfaccia, stiamo effettivamente riducendo il numero di messaggi in transito tra i vari processi.

Definiamo ora il join di due interfacce per poi mostrare, in Sezione 6.4, come questo possa essere computato direttamente, senza conoscere le simulazioni lasche parziali dalle quali deriva.

Definizione 6.18 (Join di interfacce)**.** *Siano* $A = (\phi_A, \eta_A) \in p\mathbb{S}_{\mathcal{P},\mathbf{P}}^{G \to H}$ *e* $B = (\phi_B, \eta_B) \in p\mathbb{S}_{\mathcal{P},\mathbf{Q}}^{G \to H}$ *due simulazioni lasche parziali. Il* join *(o* unione*) di* $\mathcal{I}(A)$ *e* $\mathcal{I}(B)$ *è definita come*

$$\mathcal{I}(A) \sqcup \mathcal{I}(B) \triangleq \begin{cases} \mathcal{I}(A \sqcup B) & \text{se } A \sqcup B \neq \top \\ \top & \text{altrimenti} \end{cases}$$

dove, come per Definizione 6.11, $\top$ *corrisponde ad una soluzione erronea, scartata dai processi che la computano.*

Input: Due interfacce I e I' tali da rispettare Assunzione 6.16.

$$I = \mathbf{P} \vdash \langle X, \mu_{\mathcal{M}}, \mu_Y \rangle \xrightarrow{M,\ U} \langle Y, \nu \rangle$$
$$I' = \mathbf{Q} \vdash \langle X', \mu'_{\mathcal{M}}, \mu'_Y \rangle \xrightarrow{M',\ U'} \langle Y', \nu' \rangle$$

Result: Se esiste, $I \sqcup I'$. Altrimenti *Error*.

1 **if** $U \cap U' \neq \emptyset$ **then**
2 **return** *Error* // Violata Condizione (SLP 2)
3 $X^{\sqcup} \longleftarrow X \uplus X'$, $Y^{\sqcup} \longleftarrow Y \uplus Y'$,
4 $\mu^{\sqcup}_{\mathcal{M}} \longleftarrow [\mu_{\mathcal{M}}, \mu'_{\mathcal{M}}]$, $\mu^{\sqcup}_Y \longleftarrow [\mu_Y, \mu'_Y]$, $\nu^{\sqcup} \longleftarrow [\nu, \nu']$
5 **while** $X^{\sqcup} \cap Y^{\sqcup} \neq \emptyset$ // Rimuovi tutti gli elementi locali a $\mathbf{P} \cup \mathbf{Q}$
6 **do**
7 **let** *y un elemento di* $X^{\sqcup} \cap Y^{\sqcup}$ **in**
8 **if** $\nu^{\sqcup}(y) \not\subseteq \mu^{\sqcup}_{\mathcal{M}}(y) \cup \bigcup_{y' \in \mu^{\sqcup}_Y(y)} \nu^{\sqcup}(y')$ **then**
9 **return** *Error* // Violata Condizione (SLP 6)
10 $\mu^{\sqcup}_{\mathcal{M}} \leftarrow \lambda x. \begin{cases} \mu_{\mathcal{M}}(x) \cup \mu^{\sqcup}_{\mathcal{M}}(y) & \text{se } y \in \mu^{\sqcup}_Y(x) \\ \mu^{\sqcup}_{\mathcal{M}}(x) & \text{altrimenti} \end{cases}$
11 $\mu^{\sqcup}_Y \leftarrow \lambda x. \begin{cases} (\mu^{\sqcup}_Y(x) \cup \mu^{\sqcup}_Y(y)) \setminus \{x,\ y\} & \text{se } y \in \mu^{\sqcup}_Y(x) \\ \mu^{\sqcup}_Y(x) & \text{altrimenti} \end{cases}$
12 $X^{\sqcup} \longleftarrow X^{\sqcup} \setminus \{y\}$
13 $Y^{\sqcup} \longleftarrow Y^{\sqcup} \setminus \{y\}$
14 **return** $\mathbf{P} \cup \mathbf{Q} \vdash \langle X^{\sqcup},\ \mu^{\sqcup}_{\mathcal{M}},\ \mu^{\sqcup}_Y \rangle \xrightarrow{M \cup M',\ U \cup U'} \langle Y^{\sqcup},\ \nu^{\sqcup} \rangle$

Figura 6.19: Algoritmo per il join di due interfacce.

6.4 Join di Interfacce

Per poter astrarre completamente dalle simulazioni lasche parziali e considerare le loro interfacce come soluzioni parziali dell'algoritmo di Amalgamazione Distribuita è necessario poter svolgere l'operazione di join di interfacce senza conoscere le simulazioni lasche parziali corrispondenti. Solo così facendo infatti i processi potranno computare le interfacce contemporaneamente alle simulazioni lasche parziali per poi andare ad inviare e comporre solo le prime, dimenticandosi delle seconde.

Introdurremo ora un algoritmo in grado di compiere il join di interfacce direttamente, verificando inoltre che essa rispetti Condizioni (SLP 2), (SLP 6) — come spiegato all'inizio di questa sezione infatti, le altre condizioni sono garantite da Assunzione 6.16.

Nota. Nello pseudocodice di Figura 6.19, date due funzioni $f : A \to C$ e $g : B \to C$, indichiamo con $[f, g]$ la loro somma disgiunta, definita come $[f, g] : A + B \to C$ tale che

$$[f, g](x) = \begin{cases} f(x) & \text{se } x \in A \\ g(x) & \text{altrimenti}(x \in B) \end{cases}$$

Procediamo descrivendo l'algoritmo. Con riferimento allo pseudocodice in Figura 6.19, date in input due interfacce I e I', l'algoritmo ritorna l'interfaccia $I \sqcup I'$ oppure un errore nel caso in cui Condizione (SLP 2) o Condizione (SLP 6) non vengano rispettate e dunque il join delle due interfacce non sia definito. La procedura inizia con verificare (riga 1) che i due insiemi U e U' siano disgiunti:

questo corrisponde a controllare Condizione (SLP 2). L'algoritmo poi procede unendo le due interfacce, componente per componente. Da Assunzione 6.16, sappiamo che $X \cap X' = \emptyset$ e $Y \cap Y' = \emptyset$: i risultati delle loro unioni disgiunte, $X^{\sqcup}$ e $Y^{\sqcup}$, sono dunque isomorfi a $X \cup X'$ e $Y \cup Y'$ rispettivamente. Analogamente vengono computate tramite somma disgiunta le funzioni $\mu^{\sqcup}_{\mathcal{M}}$, $\mu^{\sqcup}_{Y}$ e $\nu^{\sqcup}$.

Qualora $X^{\sqcup}$ e $Y^{\sqcup}$ siano disgiunti, è facile notare come la struttura

$$\mathbf{P} \cup \mathbf{Q} \vdash \langle X^{\sqcup},\ \mu^{\sqcup}_{\mathcal{M}},\ \mu^{\sqcup}_{Y} \rangle \xrightarrow{M \cup M',\ U \cup U'} \langle Y^{\sqcup},\ \nu^{\sqcup} \rangle$$

sia un'interfaccia lasca. Sotto Assunzione 6.16, infatti, questo caso corrisponde ad unire due interfacce che non avevano alcun elemento in comune. Gli insiemi $X^{\sqcup}$ e $Y^{\sqcup}$ possono tuttavia essere tali che l'intersezione $X^{\sqcup} \cap Y^{\sqcup}$ non sia vuota. Gli archi di questa intersezione corrispondono ad elementi locali rispetto ai processi $\mathbf{P} \cup \mathbf{Q}$ e quindi non apparterrebbero alla frontiera di alcuna simulazione lasca parziale avente come interfaccia $I \sqcup I'$. La parte rimanente dell'algoritmo (righe 5-12) ha dunque il compito di rimuovere uno ad uno questi elementi, controllando di non violare Condizione (SLP 6). Rispetto ad una qualunque simulazione lasca (ϕ, η) avente interfaccia $I \sqcup I'$, per ogni elemento $(e, e') \in \eta$ da rimuovere dall'insieme $X^{\sqcup} \cap Y^{\sqcup}$, vale che $(t(e), t(e')) \in \phi$. Per questo elemento di ϕ deve valere Condizione (SLP 6): l'algoritmo si occuperà dunque di verificare questa condizione (riga 7) controllando che, rispetto a $(t(e), t(e'))$, ogni $w \in \mathcal{M}$ tale che $\mathbb{P}_G(t(e), w) \neq \emptyset$ — ovvero appartenente a $\nu((e, e'))$ — valga almeno una delle seguenti:

- w è raggiungibile tramite Δ_η, *i.e.* $w \in \mu_{\mathcal{M}}(e, e')$;
- esista $(d, d') \in Y^{\sqcup}$ tale che $\mathbb{P}_G(t(d), w) \neq \emptyset$ — ovvero per il quale vale $w \in \nu^{\sqcup}(d, d')$ — raggiungibile tramite Δ_η, *i.e.* $(d, d') \in \mu_Y((e, e'))$.

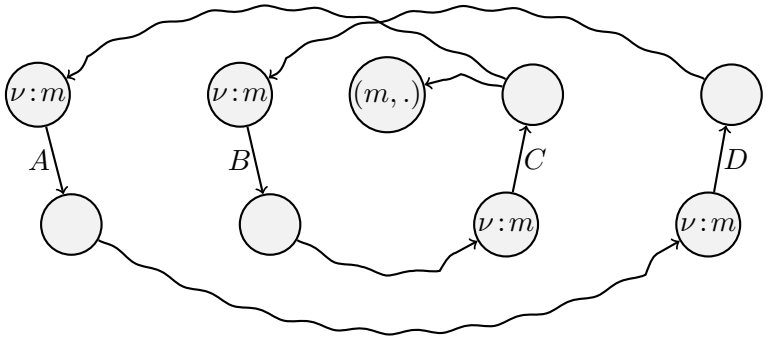
Se questo controllo ha successo, allora Condizione (SLP 6) è rispettata per $(t(e), t(e'))$ ed è quindi possibile rimuovere (e, e') da $X^{\sqcup}$ e $Y^{\sqcup}$ (righe 11 e 12), aggiornando tuttavia le funzioni $\mu^{\sqcup}_{\mathcal{M}}$ e $\mu^{\sqcup}_{Y}$ (righe 9 e 10) in modo da aggiungere agli elementi $x \in X^{\sqcup}$ tali che $(e, e') \in \nu_Y(x)$ i nodi appartenenti a $\mathcal{M}$ e gli elementi di $Y^{\sqcup}$ che erano connessi a (e, e') — si noti come nel secondo caso sia necessario rimuovere (e, e') e x da $\mu^{\sqcup}_{Y}(x)$, dato che in $I \sqcup I'$ deve valere che $X^{\sqcup} \cap Y^{\sqcup} = \emptyset$.

Esempio 5. Figura 6.20 mostra l'eliminazione di quattro elementi di $X^{\sqcup} \cap Y^{\sqcup}$ dovuta a quattro iterazioni del corpo del while dello pseudocodice di Figura 6.19.

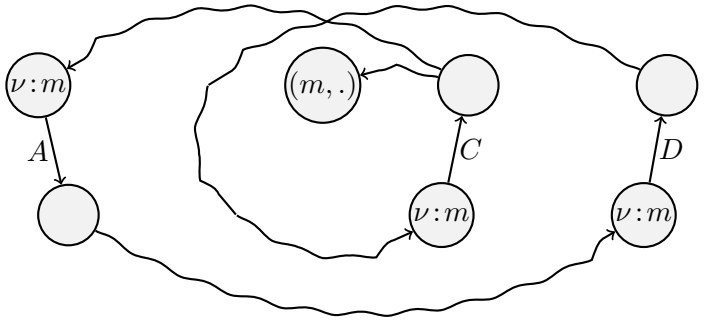
Per quanto appena descritto, l'algoritmo riesce a verificare la consistenza rispetto alle Condizioni (SLP 2) e (SLP 6) e computa esattamente il join di interfacce. Inoltre, grazie a questi risultati, da Teorema 6.9 e Definizioni 6.17, 6.18 ricaviamo direttamente la seguente proposizione.

Proposizione 6.21 (Individuazione di una soluzione: interfacce)**.** *Sia* $I = \mathbf{P} \vdash \langle X, \mu_{\mathcal{M}}, \mu_Y \rangle \xrightarrow{M,\ U} \langle Y, \nu \rangle$ *una interfaccia appartenente a* $\mathcal{I}(p\mathbb{S}^{G \to H}_{\mathcal{P}, \mathbf{P}})$*. Esiste una simulazione lasca* $A \in \mathbb{S}^{G \to H}$ *tale che* $\mathcal{I}(A) = I$ *se e solo se* $X = Y = \emptyset$ *e* $M = \mathcal{M}$*.*

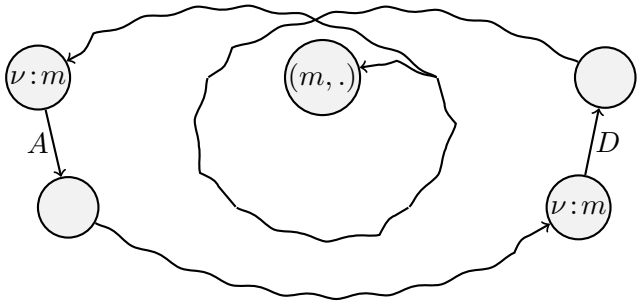
È dunque possibile utilizzare le interfacce per decidere se esiste una simulazione lasca di un guest in un host. Terminiamo l'esposizione dell'algoritmo di join fornendone uno studio di complessità.



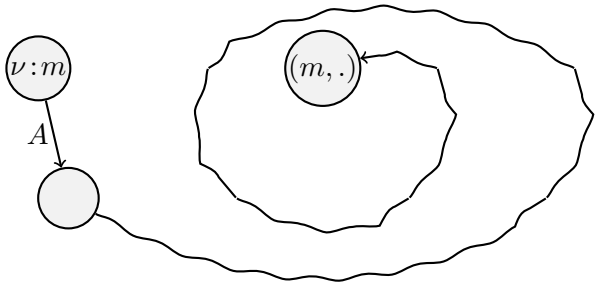
1. Gli archi A, B, C e D rappresentano elementi di $E_G \times E_H$ appartenenti a $X^{\sqcup} \cap Y^{\sqcup}$. Dalla terminazione di C è raggiungibile un elemento $(m, .) \in \phi$, con $m \in \mathcal{M}$ — ovvero $\mu^{\sqcup}_{\mathcal{M}}(C) = \{m\}$. Gli archi ondulati rappresentano le funzioni $\mu^{\sqcup}_{\mathcal{M}}$ e $\mu^{\sqcup}_{Y}$, mentre le etichette $\nu : m$ indicano che la funzione ν ritorna $\{m\}$ per gli elementi A, B, C e D — in quanto appartenenti a $Y^{\sqcup}$.



2. Alla prima iterazione del ciclo while, viene rimosso l'elemento B. Vengono quindi aggiornati $\mu^{\sqcup}_{Y}(D)$ e $\mu^{\sqcup}_{\mathcal{M}}(D)$ (invariato), dato che prima della rimozione valeva $B \in \mu^{\sqcup}_{Y}(D)$.



3. Viene rimosso l'elemento C. Ancora una volta vengono aggiornati $\mu^{\sqcup}_{Y}(D)$ e $\mu^{\sqcup}_{\mathcal{M}}(D)$, ora uguali rispettivamente a $\{A\}$ e $\{m\}$.



4. Viene rimosso l'elemento D. Vengono quindi aggiornati $\mu^{\sqcup}_{Y}(A)$ e $\mu^{\sqcup}_{\mathcal{M}}(A)$. Si noti tuttavia che $A \notin \mu^{\sqcup}_{Y}(A)$, nonostante $A \in \mu^{\sqcup}_{Y}(D)$, come da definizione di $\overline{\mu_Y}$ in Figura 6.19. In seguito l'algoritmo rimuoverà (con successo) anche A, dato che $\nu(A) = \mu^{\sqcup}_{\mathcal{M}}(A)$.

Figura 6.20: Esempio dei passi eseguiti dall'algoritmo per il join di interfacce.

Sia $I = \mathbf{P} \vdash \langle X, \mu_{\mathcal{M}}, \mu_Y \rangle \xrightarrow{M,\, U} \langle Y, \nu \rangle$ un'interfaccia. Codifichiamo questa attraverso le seguenti strutture dati:

- i processi $\mathbf{P}$ saranno codificati con un array binario di lunghezza $|\text{Proc}|$ dove ad ogni processo è stato assegnato un indice dell'array. L'unione di due array di questo tipo, vista come *or* logico elemento per elemento, ha dunque complessità $\mathcal{O}(|\text{Proc}|)$. Lo stesso vale per l'operazione di intersezione, implementata tramite l'operazione di *and* logico;
- X e Y sono due insiemi. Il costo per inserimento, ricerca e rimozione è di $\mathcal{O}(\log n)$, dove n è la cardinalità dell'insieme.
- essendo $\mathcal{M}$ e $\mathcal{U}$ fissati per un guest, M e U verranno codificati con due array binari di lunghezza rispettivamente $|\mathcal{M}|$ e $|\mathcal{U}|$, dove ad ogni vertice del guest appartenente a $\mathcal{M}$ (oppure $\mathcal{U}$) è stato assegnato un indice dell'array M (oppure U), come fatto per $\mathbf{P}$.
- le tre funzioni $\mu_{\mathcal{M}}$, μ_Y e ν verranno codificate con tre mappe. L'insieme delle chiavi di $\mu_{\mathcal{M}}$ e ν è rispettivamente X e Y, mentre i loro valori saranno array di lunghezza $|\mathcal{M}|$. μ_Y, invece, va da X in sottoinsiemi di Y. La ricerca di un elemento tramite la sua chiave è $\mathcal{O}(\log k)$, con k numero di chiavi.

Sotto le assunzioni dovute alle strutture dati appena introdotte, vale il seguente risultato di complessità.

Proposizione 6.22 (Join di interfacce: complessità)**.** *Date due interfacce*

$$\mathbf{P} \vdash \langle X, \mu_{\mathcal{M}}, \mu_Y \rangle \xrightarrow{M,\, U} \langle Y, \nu \rangle \qquad\qquad \mathbf{Q} \vdash \langle X', \mu'_{\mathcal{M}}, \mu'_Y \rangle \xrightarrow{M',\, U'} \langle Y', \nu' \rangle$$

Il loro join è computabile in $\mathcal{O}(|\text{Proc}| + Z^2(Z\log(Z) + |\mathcal{M}|))$*, dove* $Z = \max(|X| + |X'|, |Y| + |Y'|)$.

Dimostrazione. Con riferimento allo pseudocodice di Figura 6.19, il controllo di riga 1 può essere fatto in $\mathcal{O}(|\mathcal{U}|)$. Le operazioni di righe 3 e 4 consistono in unioni di insiemi e mappe, la cui complessità sarà $\mathcal{O}(Z\log(Z))$ per i due insiemi (riga 3), $\mathcal{O}(Z(\log(Z) + |\mathcal{M}|))$ per $\mu^{\sqcup}_{\mathcal{M}}$ e $\nu^{\sqcup}$ ed infine $\mathcal{O}(Z^2)$ per $\mu^{\sqcup}_Y$. Una limitazione superiore a queste operazioni risulta dunque essere $\mathcal{O}(Z(Z + |\mathcal{M}|))$. Inoltre, l'unione $\mathbf{P} \cup \mathbf{Q}$ di riga 5 ha complessità $\mathcal{O}(|\text{Proc}|)$.

Verrà in seguito effettuato il ciclo while, per un numero di iterazioni massimo pari a Z. Di questo, le due operazioni computazionalmente più complesse sono il controllo di riga 7 e l'aggiornamento delle due mappe $\mu_{\mathcal{M}}$ e μ_Y (righe 9 e 10). Per quanto riguarda il controllo

$$\nu(y) \not\subseteq \mu_{\mathcal{M}}(y) \cup \bigcup_{y' \in \mu_Y(y)} \nu(y')$$

le ricerche $\nu(y)$, $\mu_{\mathcal{M}}(y)$ e $\mu_Y(y)$ appartengono a $\mathcal{O}(\log(Z))$. Ogni unione (oltre al controllo di sottoinsieme) viene fatta rispetto all'array M e quindi ha costo $\mathcal{O}(|\mathcal{M}|)$: il costo complessivo delle unioni di $\nu(y')$, dove l'operazione $y' \in \mu_Y(y)$ e la ricerca dell'elemento vengono fatte in tempo logaritmico, è dunque $\mathcal{O}(Z(\log(Z) + |\mathcal{M}|))$, che risulta quindi essere anche una limitazione superiore alla complessità dell'intero confronto di riga 7.

Per quanto riguarda invece l'aggiornamento di $\mu_{\mathcal{M}}$ (riga 9), il controllo $y \in \mu_Y(x)$ ha costo $\mathcal{O}(\log(Z))$, mentre l'operazione $\mu_M(x) \cup \mu_M(y)$ ha complessità $\mathcal{O}(\log(Z) + |\mathcal{M}|)$. Il caso pessimo di queste operazioni si ha quando y appartiene a $\mu_Y(x)$ per tutti gli elementi $x \in X$. Essendo $|X| \leq Z$, otteniamo una limitazione superiore alle operazioni di costruzione di $\mu_{\mathcal{M}}$ pari a $\mathcal{O}(Z(\log(Z) + |\mathcal{M}|))$. Le stesse considerazioni possono essere fatte per l'aggiornamento di μ_Y, dove però, essendo questa una mappa i cui valori sono sottoinsiemi di Y, le operazioni di $\mu_Y(x) \cup \mu_Y(y) \setminus \{x, y\}$ hanno complessità $\mathcal{O}(Z \log(Z))$ e portano quindi la costruzione della mappa ad avere complessità $\mathcal{O}(Z^2 \log(Z))$.

La complessità delle operazioni nel corpo del while risulta dunque, nel caso pessimo, appartenente alla classe di complessità $\mathcal{O}(Z(Z \log(Z) + |\mathcal{M}|))$. Come già detto, questo ciclo verrà eseguito al massimo Z volte, portando così la complessità totale dell'algoritmo a $\mathcal{O}(|\text{Proc}| + Z^2(Z \log(Z) + |\mathcal{M}|))$. □

6.5 Instradamento e numero di messaggi

Per completare lo studio sulla complessità dell'algoritmo di Amalgamazione Distribuita nel caso delle simulazioni lasche è necessario trovare una limitazione superiore al numero di messaggi prodotti e scambiati tra i vari processi. Per fare questo occorre definire la nozione d'instradamento utilizzata dall'algoritmo, ovvero indicare a quali processi e secondo quale logica verrà inviata una soluzione parziale.

In seguito, siano $H = (\Sigma, V_H, E_H)$ e $G = (\Sigma, V_G, E_G, \mathcal{M}, \mathcal{U}, \mathcal{E}, \mathcal{C})$ rispettivamente un host ed un guest per simulazioni lasche. Siano inoltre Proc un insieme di processi e $\mathcal{P} : V_H \to \text{Proc}$ una partizione. L'instradamento verrà definito sotto la seguente assunzione.

Assunzione 6.23. *Sui processi in* Proc *esiste un ordinamento totale* $\leq$. *Essendo* Proc *un insieme finito, esso avrà un elemento massimo* $\max_{\leq}(\text{Proc})$. *Inoltre, ogni processo* $P \in \text{Proc}$ *può inviare messaggi ad ogni processo* $Q \in \text{Proc}$ *tale che* $P \leq Q$.

Definizione 6.24 (Instradamento di interfacce)**.** *Sia* $P \in \text{Proc}$ *un processo e sia*

$$I \;=\; \mathbf{P} \vdash \langle X, \mu_{\mathcal{M}}, \mu_Y \rangle \xrightarrow{M,\ U} \langle Y, \nu \rangle$$

un'interfaccia computata da P *— localmente o tramite join — per la quale non è verificato il criterio di arresto di Proposizione 6.21, ovvero* I *è tale che* $X \neq \emptyset$ *oppure* $Y \neq \emptyset$ *oppure* $M \neq \mathcal{M}$. *L'*instradamento *di* I *da parte di* P *è definito secondo le seguenti regole:*

- *Se* $X = Y = \emptyset$ *e* $P < \max_{\leq}(\text{Proc})$, *allora* P *invierà il messaggio al processo* $\max_{\leq}(\text{Proc})$*;*
- *Se* $X \neq \emptyset$ *oppure* $Y \neq \emptyset$, *sia* Q *il più piccolo processo non in* $\mathbf{P}$ *che possiede un arco in* $\pi_2(X \cup Y)$. *Se* $P < Q$ *allora* P *invierà il messaggio a* Q.

Nel caso in cui nessuna delle due regole sia applicabile, l'interfaccia verrà mantenuta da P. *Infine, ogni interfaccia inviata a* P *da altri processi verrà mantenuta da questo e non instradata.*

Analizziamo brevemente la definizione di instradamento appena data. La prima regola gestisce il caso in cui l'interfaccia corrisponda a parte di una simulazione lasca formata da più sottografi sconnessi dell'host. Le interfacce di questi sottografi, non avendo elementi nella frontiera che inducano l'instradamento verso un processo, verranno inviati ad un processo *default* — per semplicità l'elemento maggiore

nell'ordinamento dei processi — che avrà il compito di unirle. La seconda regola invece indica quando un processo deve inviare un'interfaccia I ad un secondo processo seguendo le informazioni contenute nella frontiera di questa: tale processo deve infatti necessariamente contribuire ad ogni soluzione che utilizzi I. Infine, nel caso in cui queste due regole non siano applicabili, le interfacce computate da un processo, così come tutte le soluzioni ricevute da questo, non saranno inviate e verranno utilizzate dal processo per eseguire il join con altre interfacce — anch'esse locali o ricevute da altri processi.

Per quanto detto, risulta chiaro come l'instradamento appena definito rispetti la condizione (CDP 5) richiesta dall'algoritmo di Amalgamazione Distribuita. La definizione di instradamento data gode inoltre di un'importante proprietà: ogni interfaccia viene inviata al massimo una volta. Grazie a questa proprietà è possibile evitare di inviare più di una volta la stessa interfaccia semplicemente implementando un meccanismo di bookkeeping interno ai vari processi. Vale inoltre il seguente bound al numero complessivo di messaggi necessario per computare una singola interfaccia.

Proposizione 6.25 (Numero messaggi per singola interfaccia)**.** *Sia* $I = \mathbf{P} \vdash \langle X, \mu_{\mathcal{M}}, \mu_Y \rangle \xrightarrow{M,\ U} \langle Y, \nu \rangle$ *un'interfaccia. Per computarla è necessario un numero di messaggi compreso tra* $|\mathbf{P}| - 1$ *e* $|\mathbf{P}|$.

Dimostrazione. Il risultato segue direttamente dalla definizione di instradamento e dal fatto che I è stata costruita a partire dalle interfacce di $|\mathbf{P}|$ simulazioni lasche parziali locali. Più precisamente, il numero di messaggi è pari a $|\mathbf{P}|$ nel caso in cui venga utilizzato il nodo *default* e questi non sia in $\mathbf{P}$. In caso contrario, sono inviati $|\mathbf{P}| - 1$ messaggi. □

Si noti come, indipendentemente dal tipo di instradamento, $|\mathbf{P}| - 1$ sia un lower bound al numero di messaggi necessario per computare una soluzione via cooperazione di $|\mathbf{P}|$ processi. Questo risultato è importante rispetto alla seguente definizione di località [17].

Definizione 6.26 (Data locality)**.** *Una classe* Q *di queries per pattern matching è detta avere* data locality *se per ogni host* $H = (\Sigma, V_H, E_H)$ *ed ogni suo nodo* $v \in V_H$*, è possibile decidere se esiste un match per una query di* Q *che utilizzi* v *ispezionando le soluzioni utilizzanti nodi fino a distanza* d *da* v*, dove* d *è determinato unicamente dalla query ed è quindi indipendente dalla dimensione di* H*.*

Per esempio, l'isomorfismo di sottografo gode di *data locality*, diversamente dalle simulazioni di grafo (e quindi, in generale, anche dalle simulazioni lasche) [17]. Questa definizione può essere estesa ai processi secondo la seguente proposizione.

Proposizione 6.27. *(Località rispetto ai processi) Una classe* Q *di queries gode di data locality se e solo se per ogni host* $H = (\Sigma, V_H, E_H)$ *ed ogni suo nodo* $v \in V_H$*, è possibile decidere se esiste un match per una query di* Q *che utilizzi* v *ispezionando le soluzioni prodotte dai processi in* $\mathbf{P}$*, con* $\mathcal{P}(v) \in \mathbf{P}$*, dove* $|\mathbf{P}|$ *è determinato unicamente dalla query ed è quindi indipendente dalla dimensione di* H*.*

La proprietà di data locality rende quindi la verifica distribuita di queries più semplice in quanto solo un numero di processi dipendente dalla query viene utilizzato per costruire una soluzione [10]. Nonostante questa proprietà non valga per le simulazioni lasche, Proposizione 6.25 garantisce che qualora un guest G produca solo soluzioni di dimensione dipendente unicamente da G, allora anche il numero di processi utilizzati per computare ogni singola soluzione dipenderà unicamente da G. Rispetto ad ogni sottoinsieme di queries delle simulazioni lasche tale da godere della proprietà di data locality — come per

esempio l'isomorfismo di sottografo — l'algoritmo di Amalgamazione Distribuita presentato computerà quindi ogni soluzione interrogando un numero di processi non dipendente dall'host. Nonostante questo importante risultato, Proposizione 6.25 non dice ancora nulla sul numero complessivo di messaggi inviati dai vari processi, ultimo passaggio necessario per completare lo studio dell'algoritmo. Come vedremo a breve, la limitazione superiore trovata per il numero di messaggi è elevata. Tuttavia, è bene notare come questo bound sia relativo a casi la cui occorrenza è, in pratica, difficilmente osservabile. Inoltre, esso è relativo al numero complessivo di messaggi in tutta la rete e non tra due processi.

Teorema 6.28 (Upper bound al numero di interfacce)**.** *Il numero di interfacce generate dai processi in* Proc *appartiene a* $\mathcal{O}(2^{|\text{Proc}|}Z_G^{X+XY+Y+1})$ *dove* Z_G *è un valore che dipende unicamente dal guest mentre* X *e* Y *sono rispettivamente il massimo numero di archi entranti ed uscenti, per ogni sottoinsieme di* Proc*, rispetto a* $\mathcal{P}$*.*

Dimostrazione. Siano $\mathbf{P} \subseteq \text{Proc}$ un sottoinsieme di processi. Iniziamo col trovare il numero massimo di interfacce da loro computabili. Per farlo introduciamo i seguenti due insiemi di archi di H:

$$E_{\mathbf{P}}^{\text{out}} = \{e \in E_H \mid s(e) \in \mathbf{P} \wedge t(e) \notin \mathbf{P}\}$$
$$E_{\mathbf{P}}^{\text{in}} = \{e \in E_H \mid s(e) \notin \mathbf{P} \wedge t(e) \in \mathbf{P}\}$$

$E_{\mathbf{P}}^{\text{out}}$ corrisponde all'insieme degli archi uscenti da processi in $\mathbf{P}$ e sono quindi, in una simulazione lasca parziale $(\phi, \eta) \in p\mathbb{S}_{\mathcal{P},\mathbf{P}}^{G\to H}$, gli unici archi che possono comparire in Y_η. $E_{\mathbf{P}}^{\text{in}}$ corrisponde invece all'insieme degli archi entranti in processi di $\mathbf{P}$, ovvero l'insieme degli archi che possono comparire in X_η.

Ogni interfaccia $I = \mathbf{P} \vdash \langle X, \mu_{\mathcal{M}}, \mu_Y\rangle \xrightarrow{M,\ U} \langle Y, \nu\rangle$ può essere vista — ricordando come ν sia in realtà costante per ogni interfaccia e non venga inviato dai processi — come un sottoinsieme di

$$E_G \times E_{\mathbf{P}}^{\text{out}} \ + \ E_G \times E_{\mathbf{P}}^{\text{in}} \times (E_G \times E_{\mathbf{P}}^{\text{out}} + \mathcal{M} + 1) \ + \ \mathcal{M} \ + \ \mathcal{U}$$

Valgono infatti le seguenti inclusioni:

$$X \subseteq E_G \times E_{\mathbf{P}}^{\text{in}} \qquad M \subseteq \mathcal{M}$$
$$\mu_{\mathcal{M}} \subseteq E_G \times E_{\mathbf{P}}^{\text{in}} \times \mathcal{M} \qquad U \subseteq \mathcal{U}$$
$$\mu_Y \subseteq E_G^2 \times E_{\mathbf{P}}^{\text{in}} \times E_{\mathbf{P}}^{\text{out}} \qquad Y \subseteq E_G \times E_{\mathbf{P}}^{\text{out}}$$

Questo comporta un limite superiore al numero di interfacce computabili dai processi in $\mathbf{P}$ pari a

$$\begin{aligned}
&= 2^{|E_G||E_{\mathbf{P}}^{\text{out}}|} 2^{|E_G||E_{\mathbf{P}}^{\text{in}}|(|E_G||E_{\mathbf{P}}^{\text{out}}|+|\mathcal{M}|+1)} 2^{|\mathcal{M}|} 2^{|\mathcal{U}|} \\
&= (2^{|E_G|})^{|E_{\mathbf{P}}^{\text{out}}|} (2^{|E_G|(|\mathcal{M}|+1)})^{|E_{\mathbf{P}}^{\text{in}}|} (2^{|E_G|^2})^{|E_{\mathbf{P}}^{\text{in}}||E_{\mathbf{P}}^{\text{out}}|} 2^{|\mathcal{M}|} 2^{|\mathcal{U}|} \\
&\leq Z_G^{|E_{\mathbf{P}}^{\text{in}}|+|E_{\mathbf{P}}^{\text{in}}||E_{\mathbf{P}}^{\text{out}}|+|E_{\mathbf{P}}^{\text{out}}|+1}
\end{aligned}$$

dove $Z_G = \max(2^{|E_G|^2},\ 2^{|E_G|(|\mathcal{M}|+1)},\ 2^{|\mathcal{M}|+|\mathcal{U}|})$ è un valore che dipende unicamente dal guest G.

Sia ora $\mathbf{Q}$ l'insieme di processi che massimizza l'esponente di Z_G, ovvero $\mathbf{Q}$ è tale che, per ogni insieme di processi $\mathbf{P}$, si ha

$$|E^{\text{in}}_{\mathbf{P}}| + |E^{\text{in}}_{\mathbf{P}}||E^{\text{out}}_{\mathbf{P}}| + |E^{\text{out}}_{\mathbf{P}}| \leq |E^{\text{in}}_{\mathbf{Q}}| + |E^{\text{in}}_{\mathbf{Q}}||E^{\text{out}}_{\mathbf{Q}}| + |E^{\text{out}}_{\mathbf{Q}}|$$

Siano $X = |E^{\text{out}}_{\mathbf{Q}}|$ e $Y = |E^{\text{in}}_{\mathbf{Q}}|$. Il numero totale di interfacce computate dai processi risulta quindi essere limitato superiormente da

$$\sum_{\mathbf{P} \subseteq \text{Proc}} Z_G^{|E^{\text{in}}_{\mathbf{P}}|+|E^{\text{in}}_{\mathbf{P}}||E^{\text{out}}_{\mathbf{P}}|+|E^{\text{out}}_{\mathbf{P}}|+1} \leq 2^{|\text{Proc}|} Z_G^{X+XY+Y+1}$$

Otteniamo così il risultato $\mathcal{O}(2^{|\text{Proc}|} Z_G^{X+XY+Y+1})$. □

Nota. È importante sottolineare alcune proprietà del limite superiore appena dato. Innanzitutto, come già accennato poc'anzi, su istanze di guest e host significative esso è molto superiore al numero di interfacce realmente calcolato. Questo è dovuto al fatto che esso non prende in considerazione i vincoli di simulazione lasca ai quali sono soggette le soluzioni calcolate localmente dai singoli processi. Inoltre, non tiene nemmeno conto della possibilità, da parte dell'algoritmo di join delle interfacce, di scartare soluzioni. Altra proprietà importante è che esso non dipende direttamente dall'host ma unicamente dagli archi condivisi tra più processi e quindi, in ultima analisi, dalla partizione — oltre a dipendere dal guest, che però generalmente è molto più piccolo dell'host. Diventa dunque fondamentale la scelta della funzione di partizione, in quanto va cercato un trade-off tra numero di soluzioni locali computate dai processi e numero di archi condivisi da questi. Questo problema, non affrontato in questo lavoro, verrà inquadrato più nel dettaglio in Capitolo 7.

Infine, dalla definizione di instradamento, secondo la quale ogni interfaccia viene inviata al massimo una volta, possiamo concludere che il bound sul numero delle interfacce è anche una limitazione al numero di messaggi inviati sulla rete.

Corollario 6.29 (Upper bound al numero di messaggi)**.** *Il numero di messaggi inviato dai processi in* Proc *appartiene a* $\mathcal{O}(2^{|\text{Proc}|} Z_G^{X+XY+Y+1})$.

Con questo risultato si conclude lo studio dell'algoritmo per il calcolo distribuito e parallelo per il problema del vuoto delle simulazioni lasche, una cui implementazione è discussa in Appendice B. È bene ricordare che l'algoritmo di Amalgamazione Distribuita da cui siamo partiti è un algoritmo generale le cui prestazioni possono essere notevolmente inferiori ad un algoritmo ad-hoc per il problema considerato. Nonostante questo, le definizioni date permettono di produrre un numero di messaggi che dipende dall'insieme di archi condivisi tra più processi, la cui cardinalità è nei casi pratici molto inferiore al numero complessivo di archi. Riteniamo che questa proprietà sia piuttosto interessante, in quanto permette, insieme alla definizione di instradamento e alla parallelizzazione delle computazioni, di gestire grafi anche di notevoli dimensioni, a patto che la partizione non generi un numero elevato di archi condivisi.

7

Conclusioni

In questo lavoro sono state introdotte le simulazioni lasche, un tipo di relazione tra grafi introdotta in ambito del pattern matching. Di questa nozione sono state mostrate alcune applicazioni per poi andarne a studiare formalmente la complessità relativa al problema del vuoto, che si è mostrato essere NP-completo. Le simulazioni lasche sono state introdotte allo scopo di verificare in un grafo l'esistenza di strutture corrispondenti simultaneamente a più nozioni di pattern matching presenti in letteratura. È stato infatti mostrato come le simulazioni lasche riescano a sussumere facilmente molti concetti classici del pattern matching su grafi, come per esempio l'isomorfismo di sottografo, ed introducano la possibilità di comporre più guests, grazie alla definizione di opportune operazioni introdotte in Sezione 3.4. Queste due caratteristiche le rendono dunque particolarmente indicate alla descrizione di query *ibride* tra diverse nozioni di pattern matching, come evidenziato in Sezione 3.5 attraverso la ridefinizione in termini di simulazioni lasche di una nozione di isomorfismo di sottografo avente archi etichettati con espressioni regolari.

Accanto alla teoria delle simulazioni lasche è stato introdotto un algoritmo parallelo e distribuito in grado di risolverne il problema del vuoto — Capitolo 6. Questo algoritmo è stato ottenuto tramite istanziazione di un algoritmo generale per il calcolo di predicati in ambiente distribuito. Come studiato in Sezione 6.5, le prestazioni dell'algoritmo, in termini di messaggi trasmessi tra i vari processi, non sono soddisfacenti. Queste tuttavia si riferiscono a casi molto particolari ed improbabili nell'uso pratico dell'algoritmo. Inoltre, il limite superiore al numero di messaggi inviati non dipende direttamente dalla dimensione dell'host ma unicamente dal numero di archi condivisi tra i vari processi, secondo la definizione di partizionamento data in Sezione 6.1. L'algoritmo introdotto, seppur quindi probabilmente meno performante di soluzioni ad-hoc al medesimo problema, ci ha permesso di studiare la possibilità di computare le simulazioni lasche in ambito distribuito, andando così ad approcciare la risoluzione del problema per grafi di notevoli dimensioni, sempre più preminente in ambiti come la profilazione degli utenti di un social network.

7.1 Problemi aperti e lavori futuri

Sia dal punto di vista della teoria delle simulazioni lasche, sia per quanto riguarda l'algoritmo distribuito presentato in questo lavoro, emergono alcuni problemi piuttosto interessanti che cercheremo ora di inquadrare brevemente. Una prima categoria di problemi aperti, riguardante la teoria delle simulazioni

lasche, tratta più nello specifico l'ampliamento di questa in modo da comprendere due risultati piuttosto importanti riguardanti i guests.

Minimizzazione di Guests Guardando al modello di programmazione lineare mista di Capitolo 4 e all'algoritmo distribuito di Capitolo 6 si può evincere come le dimensioni del guest, seppur molto minori rispetto a quelle dell'host, possano incidere negativamente sulle performance degli algoritmi per il calcolo delle simulazioni lasche. Un risultato importante da aggiungere alla teoria sarebbe dunque la definizione di un algoritmo per la minimizzazione di guest, similmente a quanto fatto per gli automi deterministici a stati finiti. Questo problema tuttavia non è altrettanto immediato, principalmente per il non determinismo e la funzione di scelta caratterizzanti i guests. Utilizzando l'operatore di quozientazione di Definizione 3.15 il problema può essere formulato chiedendo di trovare, per un guest G, una relazione d'equivalenza $\sim \subseteq V_G \times V_G$ sui vertici di G tale che $G/_\sim$ sia equivalente a G in termini di simulazioni lasche computate, scritto $G/_\sim \equiv G$, e per ogni altra relazione d'equivalenza $\mathcal{R}$, se $G/_\mathcal{R} \equiv G$ allora il numero di vertici di $G/_\sim$ è minore o uguale al quello di $G/_\mathcal{R}$. In questa formulazione, l'equivalenza tra guests $\equiv$ può essere definita andando a guardare i grafi associati alle simulazioni lasche, introdotti in Definizione 4.2. Dati due guests G_1 e G_2, diremo che $G_1 \equiv G_2$ se per ogni host H e per ogni grafo $\mathcal{G}_1$ relativo ad una simulazione lasca di $\mathbb{S}^{G_1 \to H}$ esiste un grafo $\mathcal{G}_2$ relativo ad una simulazione lasca di $\mathbb{S}^{G_2 \to H}$ rispetto al quale esiste un isomorfismo di multigrafo $f : V_{\mathcal{G}_1} \to V_{\mathcal{G}_2}$ da vertici di $\mathcal{G}_1$ a vertici di $\mathcal{G}_2$ tale da preservare la seconda proiezione dei vertici, *i.e.* $\forall (u, v) \in V_{\mathcal{G}_1} \pi_2 \circ f((u, v)) = v$, e viceversa per ogni grafo $\mathcal{G}_2$ relativo ad una simulazione lasca di $\mathbb{S}^{G_2 \to H}$ esiste un grafo $\mathcal{G}_1$ relativo ad una simulazione lasca di $\mathbb{S}^{G_1 \to H}$ rispetto al quale esiste un isomorfismo di multigrafo $f : V_{\mathcal{G}_2} \to V_{\mathcal{G}_1}$ da vertici di $\mathcal{G}_2$ a vertici di $\mathcal{G}_1$ tale da preservare la seconda proiezione dei vertici.

La formulazione del problema di minimizzazione di guest appena proposta tuttavia non dà alcuna indicazione su come computare la relazione d'equivalenza $\equiv$ tra guests, considerato che la sua definizione richiede l'analisi di un numero infinito di hosts. Esattamente come fatto con gli automi è quindi necessario focalizzarci sui guests e cercare di definire tale equivalenza guardando unicamente alla loro struttura. Inoltre, la stessa formulazione del problema potrebbe ritenersi incompleta o migliorabile in quanto essa induce un ordinamento tra guests che prende unicamente in considerazione il numero dei loro vertici, senza dare alcun peso agli insiemi must, unique ed exclusive e alla funzione di scelta.

Algebra dei Guests In Sezione 3.4 sono state introdotte le operazioni di composizione e quozientazione, le quali permettono la modifica di guests. Un possibile miglioramento alla teoria delle simulazioni lasche sarebbe dunque quello di definire, accanto a queste due operazioni, una teoria algebrica per la generazione di guests. Teorie algebriche di questo tipo sono piuttosto comuni in letteratura, si pensi per esempio a tutti i calcoli relativi all'algebra dei processi — *i.e.* π*-calcolo*, *ambient calcolo* e *sistemi reattivi bigrafici* [9, 32, 33]. Si tratta dunque di definire un insieme minimo di guest *elementari* ed un insieme di operazioni — tra le quali comparirà l'operazione di composizione o una sua variante — in grado di generare tutti i guest, a partire da quelli elementari. Solitamente, accanto a queste definizioni viene inoltre fornita una nozione di *forma normale* (o *canonica*) di un elemento algebrico, nella quale le operazioni vengono svolte seguendo un ordine ben preciso.

L'introduzione di un algebra dei guests fornirebbe quindi un modo formale per costruire queries per simulazioni lasche, fornendo dunque ai guests una semantica in parte veicolata dalle operazioni svolte per generarli.

Ottimizzazione dell'host distribuito Accanto alle estensioni alla teoria delle simulazioni lasche appena enunciate, anche dall'istanziazione dell'algoritmo di Amalgamazione Distribuita emergono due problemi piuttosto interessanti. Il primo di questi riguarda la distribuzione di un host su diversi processi, secondo Definizione 6.1. Dai risultati riguardanti il numero di messaggi lungo la rete — Sezione 6.5 — riscontriamo infatti che questo dipenda dal numero degli archi dell'host condivisi tra più processi. Fissato dunque il numero di processi che si intende utilizzare, vorremmo trovare un partizionamento dell'host che minimizzi il numero di questi archi. Siano Proc un insieme di processi e $H = (\Sigma, V_H, E_H)$ un host. Siano inoltre $A \subset \text{Proc}$ un sottoinsieme non vuoto di Proc e $B \triangleq \text{Proc} \setminus A$ l'insieme complementare di A. Rispetto ad una partizione, sia X l'insieme di archi uscenti da un processo di A verso uno di B e poniamo infine Y come l'insieme di archi entranti in un processo di A ed uscenti da uno di B. Ricordando che il numero di messaggi inviati appartiene a $\mathcal{O}(2^{|\text{Proc}|} Z_G^{X+XY+Y+1})$, si richiede di costruire una partizione $\mathcal{P} : V_H \to \text{Proc}$ che minimizzi, al variare di A, il massimo valore di $X + XY + Y$. Così posto, questo problema — che può essere visto come una variante del problema di taglio minimo [37] — ha come soluzione quella di assegnare l'intero host ad un unico processo, il quale dovrebbe poi computare ogni soluzione localmente. Questo problema infatti si scontra con un secondo problema di bilanciamento del carico dei singoli processi, i quali idealmente dovranno contribuire abbastanza equamente alla computazione delle soluzioni. Si tratta quindi di trovare un trade-off tra numero di archi condivisi tra i vari processi e numero delle soluzioni locali computate da questi. Di questo trade-off è difficile dare il giusto peso alle due parti: questo dipende principalmente dal fatto che non è facile costruire una buona approssimazione al problema di bilanciamento del carico dei singoli processi. Lavorando indipendentemente dai guest, una prima soluzione potrebbe essere quella di richiedere una distribuzione abbastanza equa dei vertici e gli archi dell'host, tra i vari processi. Soluzioni più interessanti tuttavia possono emergere specificando sottostrutture dell'host che è più (o meno) probabile possano formare delle soluzioni — nel caso in cui si possano fare alcune considerazioni sulle caratteristiche dei guests che si intendono utilizzare.

Aggiornamento degli host In [29] viene aggiunta all'algoritmo di Amalgamazione Distribuita la possibilità di eseguire riscritture automatiche sulla porzione di conoscenza rappresentante una soluzione — nell'articolo, un *embedding* di bigrafo. In particolare data una soluzione si effettua una riscrittura nella quale la soluzione viene rimossa dalla conoscenza e rimpiazzata con nuove informazioni. Per permettere le riscritture automatiche di un host utilizzando l'algoritmo di Amalgamazione Distribuita, l'istanziazione di Capitolo 6 va rivista in modo tale da identificare una simulazione lasca e non semplicemente la sua esistenza tramite le interfacce. Questo può essere fatto mantenendo gran parte dei risultavi visti in tale capitolo. Data un'interfaccia $I = \mathbf{P} \vdash \langle X, \mu_{\mathcal{M}}, \mu_Y \rangle \xrightarrow{M,\, U} \langle Y, \nu \rangle$ si aggiunga ad essa una mappa $\xi : \mathbf{P} \to \mathbb{N}$ da processi ad identificativi. L'identificativo associato ad un processo rappresenta una simulazione lasca parziale calcolata da quest'ultimo la cui interfaccia è stata utilizzata per computare I. Si noti inoltre come, essendo l'algoritmo per join di interfacce definito sotto Assunzione 6.16, l'unione di

due mappe di questo tipo può essere fatto tramite somma disgiunta, come definita in 6.4. Per un guest G, un host H ed una partizione $\mathcal{P}$, qualora ogni processo P mantenga una mappa $\mathcal{I}_P : \mathbb{N} \to p\mathbb{S}^{G\to H}_{\mathcal{P},\{P\}}$ da identificatori a simulazioni lasche parziali è possibile per il requester computare una simulazione lasca avente interfaccia I andando a richiedere ai vari processi le simulazioni lasche parziali corrispondenti ai vari identificatori e computando il join $\bigsqcup_{P\in\mathbf{P}} \mathcal{I}_P \circ \xi(P)$. Grazie a questa modifica, l'algoritmo è in grado di computare simulazioni lasche, le quali verranno poi utilizzate per aggiornare gli host. Questi aggiornamenti possono essere implementati attraverso un *linguaggio di scripting* che data la porzione di host rappresentata dalla simulazione lasca ne vada a modificare la struttura, introducendo o rimuovendo nuovi nodi o archi.

Gli aggiornamenti di hosts risultano interessanti anche rispetto al problema riguardante il loro partizionamento, esposto in precedenza. In seguito ad una modifica, infatti, la partizione risultante potrebbe non essere più ottimale. Sono dunque necessarie euristiche che vadano a riconfigurare la porzione di host locale ad un aggiornamento in modo da non distanziarsi troppo dal partizionamento ottimale, evitando quindi la degradazione delle prestazioni dell'algoritmo.

A

Progettazione e implementazione dell'algoritmo di Amalgamazione Distribuita

In Sezione 5 è stato introdotto l'algoritmo per il calcolo distribuito di predicati, denominato Amalgamazione Distribuita. Con lo scopo di fornire *proof of concept*, accanto a questo lavoro prettamente teorico viene presentata una sua implementazione la cui progettazione ed aspetti chiave verranno riportati in questa appendice. I moduli che compongono questo lavoro sono scritti in C++ e Erlang, le cui caratteristiche possono essere studiate in [40] e [2] rispettivamente. Ricordiamo come in Sezione 5.2 siano stati presentati i vari componenti dell'algoritmo di Amalgamazione Distribuita. In questo, ogni processo possiede un modulo denominato solver, in grado di computare soluzioni locali, un modulo joiner, con lo scopo di unire diverse soluzioni parziali, ed un router, per l'instradamento di queste. I processi comunicano tra loro per individuare le soluzioni delle query ricevute dai requester.

Nell'implementazione proposta, i cui moduli sono rappresentati in Figura A.1, la nozione di possesso di un solver da parte di un processo non esiste. I solvers, implementati in C++, sono entità autonome e quindi indipendenti dai processi. Essi mantengono una porzione della conoscenza totale del sistema — per esempio una parte di host, come mostrato in Figura A.1 — ed andranno a generare rispetto ad essa le computazioni locali relative alle query ricevute, indipendentemente da quale processo abbia richiesto tale computazione. Tuttavia, come vedremo in Sezione A.4, essendo l'algoritmo parametrico rispetto alla definizione di sistema e di sua partizione, il solver non ha alcuna informazione sulla conoscenza in suo possesso ma si limita unicamente a mantenerla; come questa venga utilizzata va quindi definito in fase di istanziazione, rispetto ad un problema specifico. Similmente, ogni modulo del framework descritto in questa appendice vede le soluzioni computate da solvers e joiners come semplici sequenze binarie: in fase di istanziazione sarà quindi necessario definire le funzioni di marshalling che ne permettano la corretta trasmissione e traduzione.

D'altro canto, i processi — scritti in Erlang — continueranno a svolgere le funzioni di routing descritte in Sezione 5.2, con un'importante differenza: ricevuta una richiesta, essi eseguiranno un nuovo processo Erlang denominato *handler* — rappresentato in Figura A.1 con un quadrato nero — che provvederà ad interrogare il solver e ad unire le soluzioni ricevute, tramite un processo (joiner) scritto in C++ e

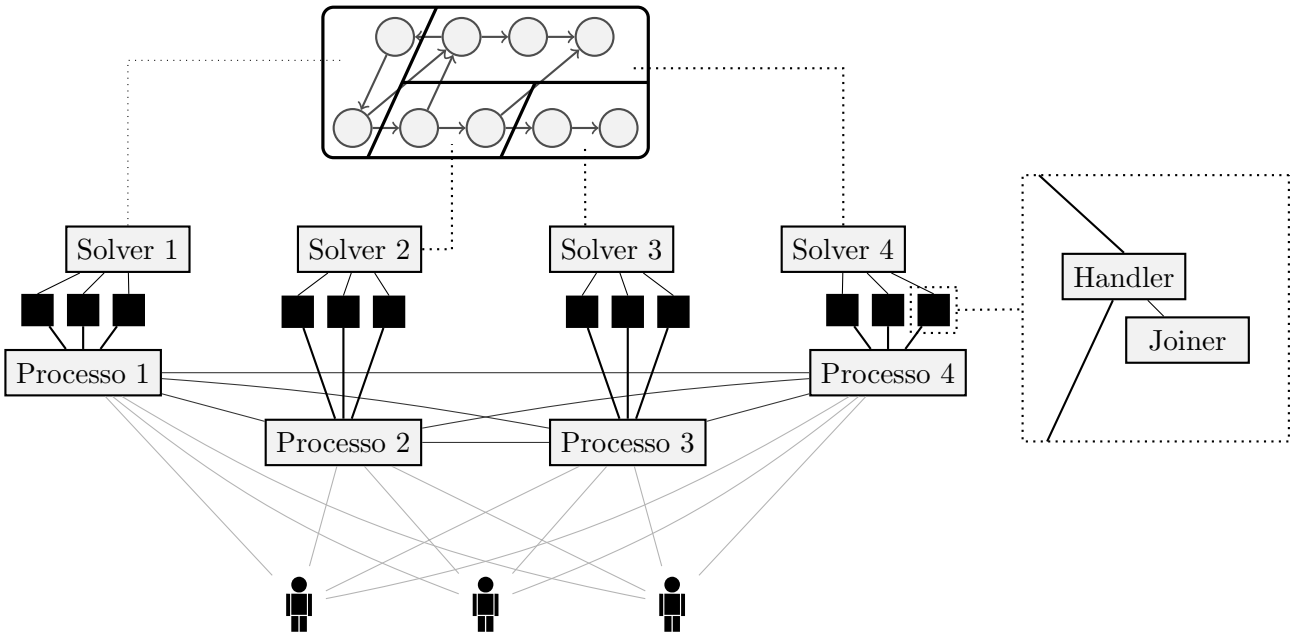


Figura A.1: Moduli dell'implementazione di Amalgamazione Distribuita.

generato dallo stesso handler. I processi dunque avranno anche il compito di inoltrare i messaggi ricevuti da altri nodi agli handlers corretti, a seconda della richiesta alla quale tali messaggi si riferiscono.

Infine, i requester rimangono pressoché identici rispetto a quanto visto in Sezione 5.2. Ogni requester è limitato ad una singola query alla volta — tuttavia non c'è un limite fissato al numero di requester — e devono implementare l'algoritmo di Huang, secondo quanto esposto in Definizione 5.7 e nello pseudocodice di Figura 5.5.

Descriviamo ora più nel dettaglio le caratteristiche di questi moduli.

A.1 Processi

Attraverso le modifiche appena descritte, i processi eseguiranno principalmente funzioni di instradamento rispetto ai messaggi inviategli dagli handlers da lui generati, da altri processi e dai requester. Per essere avviato correttamente, un processo necessita di tre parametri

- Un nome, utilizzato per registrare il processo;
- Un file di specifiche, in formato JSON e seguente la struttura proposta in Figura A.2, contenente gli indirizzi di ogni processo e solver della rete;
- Il nome del programma che implementa le funzioni di joiner.

Rispetto a questi tre parametri, all'avvio, un processo inizierà col esaminare il file di specifiche della rete, utilizzando la funzione built-in `erl_scan:string/1` ed una grammatica per file JSON scritta per il parser messo a disposizione da Erlang.[1] Le informazioni contenute in questo file verranno utilizzate per

[1] Adottando una delle convenzioni di Erlang, con la sintassi `method_name/`N indicheremo che il metodo denominato `method_name` ha arietà N.

```
{ “nodes” : [
        { “node_name” : “node0”, “node_address” : “network0@64.233.167.99” },
        { “node_name” : “node1”, “node_address” : “network1@74.125.224.72” },
        { “node_name” : “node2”, “node_address” : “network2@74.125.86.72” }
    ]
   “solvers” : [
        { “solver_address” : “solver1@209.85.120.236” },
        { “solver_address” : “solver2@74.125.121.111” }
    ]
}
```

Figura A.2: File di specifiche della rete.

contattare correttamente gli altri nodi della rete. Il nodo si preoccuperà poi di registrare il nome passato in input come suo alias, tramite chiamata alla funzione Erlang `register/2` — avente come parametri il nome per l'aliasing ed il *process id*, recuperabile tramite la funzione `self/0`.

Svolti questi passi di configurazione, il processo si metterà in attesa di messaggi, chiamando la funzione ricorsiva `node_loop/5`, avente come parametri:

- `Id`: il nome passato a `register/2`;
- `Solvers`: la lista dei solvers recuperata dal file delle specifiche;
- `Processes`: la lista dei processi, anch'essa recuperata dal file delle specifiche, composta da coppie (*node_name*, *node_address*);
- `HandlersMap`: una mappa, inizialmente vuota, avente come chiavi gli identificatori dei requester e come valori gli handler a loro associati;
- `Join`: il nome del programma per il join di soluzioni parziali, che verrà utilizzato dagli handler.

Lo pseudocodice esposto in Figura A.3 mostra piuttosto bene come i processi fungano principalmente da routers in grado di far comunicare tra loro handlers (di diversi processi) e requesters. Si noti infatti come tutti i messaggi di righe 15-27 vengano semplicemente spediti al corretto destinatario.

Un requester, prima di poter inviare una richiesta ai processi, deve registrarsi a loro tramite un messaggio {*subscribe, SolversMap*}, dove *SolversMap* è una mappa da indirizzi dei solvers a nomi dei processi — rispettivamente *solver_address* e *node_name* in Figura A.2. Questa mappa, che è richiesta essere biunivoca, corrisponde ad un assegnamento tra solvers e processi. Tramite questo assegnamento, il requester indica ai processi quali solvers dovranno utilizzare per risolvere le sue query. Qualora un requester sia già registrato ad un processo, quest'ultimo risponderà con un messaggio *already_subscribed*. In caso contrario, risponderà invece con un *ack* ed avvierà un nuovo handler associato a tale requester, secondo quanto descritto nella prossima sezione. Si noti infine come, nel caso di ricezione di soluzioni parziali da parte di un handler (riga 19) il messaggio contenga anche l'identificatore del processo al quale instradare la soluzione. Infatti, contrariamente a quanto detto in Sezione 5.2, nell'implementazione proposta non sono direttamente i processi ad implementare l'instradamento richiesto da Condizione (CDP 5), ma bensì le implementazioni di solver e joiner.

```
1  node_loop (Id, Solvers, Processes, HandlersMap, Join)
2      receive
3          case {subscribe, SolversMap} from un requester R
4              if Requester ∉ maps:is_key(HandlersMap) then
5                  let  Solver tale che maps:get(Solver, SolversMap) = Id
6                  in H ← spawn(handler_init, self(), R, Solver, SolversMap, Join)
7                  send ack to R
8                  node_loop(Id, Solvers, Processes, maps:put(R, H, HandlersMap), Join)
9                  return
10             else
11                 send already_subscribed to R
12         case unsubscribe from un requester R
13             send unsubscribe to maps:get(R, HandlersMap)
14             node_loop(Id, Solvers, Processes, maps:remove(R, HandlersMap), Join)
15             return
16         case {query, Query} from un requester R
17             send {query, Query} to maps:get(R, HandlersMap)
18         case revoke from un requester R
19             send revoke to maps:get(R, HandlersMap)
20         case {partial_computation, Solution, Requester, Receiver} from un handler
21             let  Address s.t. (Receiver, Addresss) ∈ Processes
22             in  send {partial_computation, Solution, Requester} to Address
23         case {partial_computation, Solution, Requester} from un altro processo
24             send {partial_computation, Solution} to maps:get(Requester, HandlersMap)
25         case {solution, Solution, Requester} from un handler
26             send {solution, Solution} to Requester
27         case {node_off, Requester, HuangNum} from un handler
28             send {node_off, HuangNum} to Requester
29     node_loop(Id, Solvers, Processes, HandlersMap, Join)
```

Figura A.3: Amalgamazione Distribuita - Ricezione messaggi processo.

Nota. Per semplicità, negli pseudocodici proposti in questa appendice poniamo che qualora una chiamata ad un metodo di accesso di una struttura dati fallisca — come per esempio `maps:get/2` nel caso venga utilizzata una chiave non presente nella mappa — il processo si rimetta in attesa di un nuovo messaggio, esattamente come se il messaggio che ha generato il fallimento fosse mal formato. Questo può essere implementato facilmente in Erlang attraverso il costrutto *try-catch*. Si noti come, per la semplicità dei programmi proposti, non sia necessario alcun meccanismo di *rollback*: nel caso in cui sia richiesta una di queste operazioni, essa verrà eseguita prima di ogni altro comando.

A.2 Handler

Per avviare correttamente un handler, i processi dovranno richiamare il metodo `handler_init/5` — riga 6 di Figura A.3 — il cui pseudocodice è proposto in Figura A.4. Questo metodo richiede i seguenti parametri:

- `Process`: indirizzo del processo chiamante;
- `Requester`: indirizzo requester, utilizzato unicamente per indicare ai processi di quale requester un certo handler sta gestendo le query;
- `Solver`: indirizzo del solver da interrogare per processare le query;
- `SolversMap`: mappa da indirizzi di solvers in nomi di processi ricevuta dal requester ed utilizzata per gestire l'instradamento delle computazioni parziali;
- `Join`: nome del programma che implementa le funzioni di joiner.

Una volta eseguito, un handler andrà ad avviare a sua volta il processo Join utilizzando le *porte* di Erlang, le quali permettono la comunicazione tra un processo Erlang ed un programma a lui subordinato attraverso operazioni di lettura e scrittura da standard input e output. L'avvio del processo Join viene dunque fatto attraverso la chiamata a `open_port/3` — riga 2 di Figura A.4 —, dove viene passata l'opzione {`packet, 4`}, ad indicare che ad ogni messaggio verrà aggiunta un'intestazione di 4 byte per specificarne la lunghezza. L'handler si metterà poi in attesa di ricevere una query da processare o in alternativa di un messaggio *unsubscribe* che provocherà la sua chiusura.

Ricevuta una query, questa verrà indicata al processo Join assieme all'identità del solver utilizzato dall'handler per le computazioni locali (riga 5). L'handler si preoccuperà poi di inviare la query anche al solver e proseguirà chiamando la funzione `handler_loop/9`. Di questa funzione — il cui pseudocodice è anch'esso riportato in Figura A.4 —, i primi cinque parametri sono identici a quelli visti per `handler_init/5`, ma viene aggiunta l'identità del joiner e i parametri per implementare l'algoritmo di Huang. In particolare, riprendendo anche quanto visto in Sezione 5.3:

- S è un booleano, inizialmente *true*, rappresentante lo stato del server. Verrà posto a *false* qualora il solver termini la computazione riguardante una query;
- J è un intero positivo, inizialmente pari a 0, incrementato ogniqualvolta una computazione viene inviata dal handler al joiner e decrementato quando quest'ultimo termina di generare le soluzioni per tale computazione;

- *Huang* è un intero, maggiore di 0, rappresentante il valore richiesto per ogni processo dall'algoritmo di Huang — in Definizione 5.7, corrisponde al valore $w_{P,Q}$.

Per implementare l'algoritmo di Huang sono state apportate le seguenti modifiche: al posto di assegnare ad ogni processo un peso iniziale pari a 1 e dividere questo in due parti ad ogni messaggio da inviare, verrà assegnato loro un peso pari a ⌊MAXVALUE/`maps:size`(SolversMap)⌋ che verrà decrementato di 1 ad ogni messaggio. I messaggi hanno dunque un peso a loro associato sempre pari a 1 e che può essere omesso in fase di invio del messaggio. Si noti come `maps:size`(SolversMap) corrisponde al numero di processi coinvolti nella query. Il numero iniziale è dunque pari al numero minimo di messaggi inviabili da un processo ed è scelto in modo tale che anche nel caso in cui tutti i messaggi siano ricevuti da un unico processo, questo non vada in overflow — dove il valore massimo che può raggiungere è dato da *MAXVALUE*. È dunque importante scegliere un valore massimo piuttosto elevato: per esempio, nel caso si utilizzino numeri naturali a 32 bit e la query richieda la cooperazione di 128 processi, un handler può spedire, nel caso pessimo in cui non riceva alcun messaggio, circa 33 milioni di messaggi prima che il suo valore scenda a 0. Queste modifiche all'algoritmo fanno si che la terminazione sia garantita qualora il requester, computando la somma di tutti i valori ricevuti dai vari processi, ottenga il valore `maps:size`(SolversMap)⌊MAXVALUE/`maps:size`(SolversMap)⌋.

Passiamo ora ad analizzare il comportamento dell'handler in risposta ai messaggi ricevuti. Come si può notare, questo corrisponde piuttosto fedelmente allo pseudocodice per il router proposto in Sezione 5.3 e Figura 5.5. La ricezione di un messaggio *revoke* (riga 3), corrispondente alla revoca, da parte del requester, di una query. L'handler inoltrerà dunque la revoca al solver, chiuderà il processo di join e tornerà in fase di inizializzazione andando chiamare la funzione `handler_init/5`. Nel caso in cui il messaggio ricevuto sia invece attinente ad una computazione parziale eseguita da un altro processo (riga 8), l'handler invierà tale computazione al joiner, incrementando le variabili J e *Huang*, esattamente come fatto alla riga 22 di Figura 5.5. Se invece la computazione parziale è stata ricevuta dal solver o dal joiner, essa sarà affiancata all'indirizzo di un solver in grado di continuare la computazione, scelto a seconda della definizione di instradamento. La semantica di questo caso presenta tuttavia una particolarità: a seconda della definizione d'instradamento data, potrebbe essere richiesto di non spedire una soluzione computata ma di mantenerla all'interno del joiner (riga 14). Questo caso viene gestito andando a porre l'indirizzo del destinatario, *ReceiverSolver* nello pseudocodice, a *null*. Se, invece, *ReceiverSolver* sarà pari all'indirizzo di un solver, l'handler lo spedirà a *Process* dopo aver recuperato tramite *SolversMap* l'indirizzo del processo destinatario utilizzante tale solver (riga 17).

Infine, oltre ai messaggi {*solution, Solution*} contenenti soluzioni complete che verranno inoltrate al requester tramite *Process*, l'handler gestirà i messaggi *no_solutions*, relativi all'assenza di soluzioni e necessari per implementare l'algoritmo di Huang. Come visto in Sezione 5.3, qualora tale messaggio sia stato inviato dal solver, allora la variabile S va impostata a *false* (riga 23). Nel caso in cui, invece, il messaggio sia stato spedito dal joiner, sarà la variabile J ad essere decrementata. Al termine di ogni passo di ricezione, ovvero, dopo aver gestito un singolo messaggio, viene controllato il criterio d'arresto distribuito per il singolo handler (riga 26): nel caso in cui il solver abbia terminato il calcolo delle computazioni locali, *i.e.* S = false, ed il joiner abbia gestito tutte le richieste a lui inoltrate, *i.e.* $J = 0$, l'handler invierà un messaggio {*node_off, Requester, Huang*} al processo che l'ha generato, il

```
1  handler_init (Process, Requester, Solver, SolversMap, Join)
2      JoinPort ← open_port(spawn, Join, [{packet, 4}])
3      receive
4          case {query, Query} from Process
5              send {query, Solver, Query} to JoinPort
6              send {query, Query} to Solver
7              Huang ← ⌊MAXVALUE/maps:size(SolversMap)⌋
8              handler_loop(Process, Requester, Solver, SolversMap, Join, JoinPort, true, 0,
               Huang)
9          case unsubscribe from Process
10             send close to JoinPort
```

```
1  handler_loop (Process, Requester, Solver, SolversMap, Join, JoinPort, S, J, Huang)
2      receive
3          case revoke from Process
4              send revoke to Solver
5              send close to JoinPort
6              handler_init(Process, Request, Solver, SolversMap, Join)
7              return
8          case {partial_computation, Solution} from Process
9              send {network_partial_computation, Solution} to JoinPort
10             Huang ← Huang + 1
11             J ← J + 1
12         case {partial_computation, Solution, ReceiverSolver} from Solver or JoinPort
13             if ReceiverSolver = null then
14                 send {local_partial_computation, Solution} to JoinPort
15                 J ← J + 1
16             else
17                 Receiver ← maps:get(ReceiverSolver, SolversMap)
18                 send {partial_computation, Solution, Requester, Receiver} to Process
19                 Huang ← Huang - 1
20         case {solution, Solution} from Solver or JoinPort
21             send {solution, Solution, Requester} to Process
22         case no_solutions from Solver
23             S ← false
24         case no_solutions from JoinPort
25             J ← J - 1
26     if ¬S ∧ J = 0 then
27         send {node_off, Requester, Huang} to Process
28     handler_loop(Process, Requester, Solver, SolversMap, Join, JoinPort, S, J, Huang)
```

Figura A.4: Amalgamazione Distribuita - Handler.

quale si preoccuperà di inoltrarlo al requester. Si noti come questo messaggio contenga anche il numero necessario all'algoritmo di Huang per verificare la terminazione dell'intero algoritmo.

A.3 Joiner

Come già indicato precedentemente, gli handlers comunicano con i joiners attraverso le *porte* Erlang, ovvero utilizzando lo standard input e output. In particolare, i joiners ricevono sullo standard input i messaggi inviatigli dall'handler, mentre per rispondere a quest'ultimo utilizzano lo standard output. Questo meccanismo, per essere implementato correttamente, necessita della definizione di un encoding (e decoding) che traduca strutture Erlang in (e da) stream binari. Queste codifiche, che per semplicità non sono state inserite nello pseudocodice relativo all'handler, vanno eseguite ogniqualvolta i due processi intendono comunicare e possono essere implementate facilmente utilizzando le strutture binarie messe a disposizione da Erlang. Tralasciando i messaggi di tipo *close* (riga 5 di `handler_loop/1`), gestiti direttamente da Erlang, handler e joiner comunicano attraverso sette tipi di messaggi differenti, appartenenti ad una delle tre seguenti categorie:

- atomici, come per esempio messaggi di tipo *revoke*;
- binari, come per esempio {*local_partial_computation, Solution*};
- ternari, come per esempio {*partial_computation, Solution, Receiver*}.

Possiamo quindi codificare i messaggi indicandone il tipo nel primo byte, per poi concatenare il contenuto del messaggio. Per quanto riguarda i messaggi ternari, prima del contenuto è necessario indicare, con quattro byte, la lunghezza del primo elemento del messaggio, in modo da individuare l'inizio del secondo elemento. Otteniamo dunque la seguente codifica, espressa nella *bit syntax* di Erlang:

revoke	$\langle\langle 1 : 8 \rangle\rangle$
no_solutions	$\langle\langle 2 : 8 \rangle\rangle$
{*query, Solver, Query*}	$\langle\langle 3 : 8,$ `byte_size`(Solution) $: 32,$ Solution/`binary`, Query/`binary`$\rangle\rangle$
{*local_partial_computation, Solution*}	$\langle\langle 4 : 8,$ Solution/`binary`$\rangle\rangle$
{*network_partial_computation, Solution*}	$\langle\langle 5 : 8,$ Solution/`binary`$\rangle\rangle$
{*solution, Solution*}	$\langle\langle 6 : 8,$ Solution/`binary`$\rangle\rangle$
{*partial_computation, Solution, Receiver*}	$\langle\langle 7 : 8,$ `byte_size`(Solution) $: 32,$ Solution/`binary`, Receiver/`binary`$\rangle\rangle$

dove con $x : N$ si intende che il dato x verrà rappresentato con esattamente N bit, x/`binary` indica dei dati (binari) di lunghezza arbitraria e `byte_size`(x) ritorna la lunghezza, in byte, di x. I messaggi verranno quindi codificati, inviati via standard input o output ed infine decodificati. Si ricordi come sia handlers che joiners non conoscano la struttura delle computazioni, ma trattino queste come normali sequenze binarie; le operazioni di marshalling e unmarshalling di queste sono infatti delegate alle implementazioni di solver e joiner.

Passiamo ora ad analizzare `joiner_request_handler/1`, metodo centrale del joiner, il cui pseudocodice è presentato in Figura A.5. Esso è parametrico rispetto a due classi: *JoinerInstantiation*, la quale implementa le operazioni di join, e *SolutionsIterator*, che permette l'iterazione delle soluzioni generate da oggetti di tipo *JoinerInstantiation*. Avviato il metodo `joiner_request_handler/1`, che richiede come parametro un oggetto di tipo *JoinerInstantiation*, denominato *Joiner*, il processo si metterà in attesa di un messaggio da parte dell'handler contenente l'identità del solver utilizzato da quest'ultimo — il quale d'ora in avanti denoteremo con il nome di *solver locale* — e la query. L'oggetto Joiner può infatti necessitare di queste informazioni per implementare correttamente l'instradamento e l'operazione di join. Per indicare solver e query al Joiner viene utilizzato il metodo `set_query_infos/2` (riga 5).

Nota. Nello pseudocodice di Figura A.5 con la sintassi

`erl_translate_bin`(Message) $\rightarrow$ {query, Solver, Query}

intendiamo un comando che ritorni *true* se la decodifica di *Message* porta ad un messaggio del tipo {*query, Solver, Query*}, *i.e.* se *Message* era della forma — secondo la codifica introdotta poc'anzi — $\langle\langle 3 : 8,$ `byte_size`(Solution) $: 32,$ Solution/`binary`, Query/`binary`$\rangle\rangle$, e *false* altrimenti.

Una volta ricevuta l'identità del solver e la query, il metodo si metterà in attesa di messaggi sullo standard input. Alla ricezione di un messaggio vuoto, che indica l'invio di *close* da parte dell'handler, il joiner terminerà. In caso di messaggio non vuoto, invece, esso verrà decodificato e gestito:

- se *Message* è decodificato in *revoke* allora il joiner, analogamente a quanto fatto per il messaggio *close*, terminerà (riga 16);
- se *Message* è decodificato in un messaggio del tipo {*local_partial_computation, Solution*} allora *Solution* verrà passato come argomento al metodo `new_local_computation/1` di *Joiner*, il quale tornerà un iteratore alle soluzioni computate eseguendo il join di *Solution* con le altre computazioni mantenute da *Joiner*;
- analogamente, se *Message* è decodificato in un messaggio del tipo {*network_partial_computation, Solution*} allora *Solution* verrà passato come argomento al metodo `new_network_computation/1` di Joiner, il quale tornerà un iteratore alle soluzioni computate eseguendo il join di *Solution* con le altre computazioni mantenute da *Joiner*.

Infine, qualora il messaggio ricevuto abbia prodotto un iteratore di soluzioni — ovvero *SolutionsIterator* diverso da `no_set`, riga 15 — le nuove soluzioni da esso computate verranno spedite all'handler (righe 20 e 22). Si noti come l'esistenza di una soluzione può essere controllata tramite il metodo `next_solution/0` e recuperata tramite `get_solution/0`, la quale indica inoltre se questa è completa e, in caso contrario, identifica il solver in grado di proseguirne la computazione. A questa fase seguirà l'invio di *no_solutions*, informando così l'handler che tutte le soluzioni per tale messaggio sono state spedite. Ricordiamo che, esattamente come per tutti i messaggi di Erlang, anche utilizzando standard input o output i messaggi verranno ricevuti rispettando l'ordine di spedizione. Ciò comporta che ogni soluzione ricevuta dall'handler rispetto ad un singolo *SolutionsIterator* precederà il messaggio *no_solutions* relativo alla medesima computazione di quest'ultimo.

```
1  template ⟨class JoinerInstantiation, class SolutionsIterator⟩
2  joiner_request_handler (JoinerInstantiation Joiner)
3      BinaryData Message ← read_stdin()
4      if erl_translate_bin(Message) → {query, Solver, Query} then
5          Joiner.set_query_infos(Solver, Query)
6      else error
7      SolutionsIterator SolutionsIterator ← no_set
8      while Message ← read_stdin() ∧ Message non è vuoto do
9          if erl_translate_bin(Message) → {local_partial_computation, Solution} then
10             SolutionsIterator ← Joiner.new_local_computation(Solution)
11         if erl_translate_bin(Message) → {network_partial_computation, Solution} then
12             SolutionsIterator ← Joiner.new_network_computation(Solution)
13         if erl_translate_bin(Message) → revoke then
14             return;
15         if SolutionsIterator ≠ no_set then
16             while SolutionsIterator.next_solution() do
17                 tuple⟨Bool, BinaryData, Address⟩
18                     (IsComplete, Solution, Solver) ← SolutionsIterator.get_solution()
19                 if IsComplete then
20                     write_stdout(erl_translate_term({solution, Solution}))
21                 else
22                     write_stdout(erl_translate_term({partial_computation, Solution, Solver}))
23             SolutionsIterator ← no_set
24             write_stdout(erl_translate_term(no_solutions))
```

Figura A.5: Amalgamazione Distribuita - Joiner.

```
1  template ⟨class SolverInstantiation, class Model⟩
2  solver_request_handler (int HandlerFd, const SolverInstantiation& Solver)
3      receive {query, Query} from HandlerFd via erl_receive_msg
4      Model Model ← Solver.create_model(Query)
5      while Model.next_solution() do
6          tuple⟨Bool, BinaryData, Address⟩
7              (IsComplete, Solution, ReceiverSolver) ← Model.get_solution()
8          if IsComplete then
9              send {solution, Solution} to HandlerFd via erl_send
10         else
11             send {partial_computation, Solution, ReceiverSolver} to HandlerFd via erl_send
12         if un messaggio revoke è ricevuto da HandlerFd then
13             return;
14     send no_solutions to HandlerFd via erl_send
```

Figura A.6: Amalgamazione Distribuita - Solver.

Per quanto detto in questa sezione, istanziare correttamente il modulo joiner richiede dunque di fornire una classe che mantenga e componga le soluzioni parziali. Questa classe, dovendo estendere *JoinerInstantiation*, deve implementare i seguenti metodi:

- `set_query_infos/2`, per indicare l'identità del solver locale e la query da questo utilizzata per le computazioni;
- `new_local_computation/1`, per istruire il joiner su una nuova computazione, trovata localmente dal solver locale. Questo metodo tornare un'istanza di *SolutionsIterator*, iterante sui risultati ottenuti componendo la computazione locale con le altre computazioni mantenute dal joiner;
- `new_network_computation/1`, per indicare al joiner una computazione ricevuta dall'handler ma prodotta da un altro processo della rete. Come per `new_local_computation/1`, anche questa funzione deve tornare un'istanza di *SolutionsIterator*, in modo da poter inviare le computazioni ottenute per composizione a partire dalla soluzione passata come parametro.

dove *SolutionsIterator* deve a sua volta implementare le funzioni `next_solution/0`, che indichi qualora esista una soluzione ancora da spedire, e `get_solution/0`, che recuperi tale soluzione. Vedremo come definire questi requisiti per il caso delle simulazioni lasche nella sezione di Appendice B dedicata al joiner.

A.4 Solver

Infine, la libreria proposta fornisce metodi per interfacciare gli handlers con i solver, processi in grado computare le soluzioni locali rispetto ad una porzione del sistema. Così come i joiner, anche i solvers sono implementati in C++, ma sotto forma di *Nodi C*. Come descritto nella documentazione di Erlang [45], i Nodi C sono processi tipicamente scritti in C/C++ che appaiono indistinguibili dai normali nodi Erlang. Essi possono essere implementati utilizzando la libreria Erlang Interface, corrispondente agli header `erl_interface.h` e `ei.h`, inclusi in ogni distribuzione di Erlang. Essa implementa funzioni di codifica e decodifica dei messaggi e metodi per la connessione tra nodi Erlang e Nodi C. In particolare, per avviare correttamente un Nodo C, e quindi un solver, sono necessari i seguenti passaggi:

1. chiamare il metodo `erl_init/2`, il quale inizializza il nodo. I due parametri, nella corrente versione di Erlang Interface (versione 3), non sono più utilizzati e vanno quindi posti a *NULL* e 0.
2. Chiamare il metodo `erl_connect_init/3`, nel caso in cui il nodo debba essere raggiungibile solo da processi nella stessa macchina, oppure `erl_connect_xinit/6` nel caso si voglia renderlo accessibile a tutti i nodi. Nel secondo caso, i parametri da impostare sono:
   - nome della macchina host;
   - nome breve del nodo. Deve essere univoco all'interno della macchina;
   - nome completo del nodo. Solitamente esso è costituito dal nome breve composto al nome della macchina host, separati dal carattere @;
   - indirizzo IP della macchina;

- *secret cookie* utilizzato dalla virtual machine di Erlang alla quale verrà connesso e registrato il solver;
- booleano per il riutilizzo del nome del nodo. Nel caso impostato a *true*, il nodo riceverà anche messaggi destinati a precedenti istanze utilizzanti lo stesso nome.

Invece, nel caso di `erl_connect_init/3`, basta indicare il nome breve del nodo, il *secret cookie* ed il booleano di riutilizzo del nome.

3. Creare un socket ed effettuare il bind di questo rispetto ad una certa porta. Quest'ultima deve infine essere registrata attraverso la funzione `erl_publish/1`, rendendo così il servizio offerto dal Nodo C visibile agli altri nodi Erlang, i quali utilizzeranno tale porta per lo scambio di messaggi.

Una volta avviato, il solver si metterà in attesa di messaggi attraverso la funzione `erl_accept/2`, la quale prende come parametri il file descriptor tornato dalla creazione del socket ed una reference ad un oggetto di tipo *ErlConnect*, che verrà popolato con alcuni dati sulla connessione.

Alla ricezione del primo messaggio inviato da un handler al solver, la funzione `erl_accept/2` tornerà il file descriptor relativo all'handler e il solver avvierà un thread per gestire le comunicazioni con questo. Il thread eseguito dal solver chiamerà il metodo `solver_request_handler/2`, il cui pseudocodice è presentato in Figura A.6. Analogamente a quanto visto per `joiner_request_handler/1`, anche questa procedura è parametrica su due classi: *Model*, i cui oggetti hanno lo scopo di generare computazioni locali rispetto ad una certa query, e *SolverInstantiation*, le cui istanze mantengono una porzione di conoscenza (affidata ai solver) e permettono la generazione di modelli, *i.e.* istanziano oggetti di *Model*. `solver_request_handler/2` prende in input un file descriptor relativo all'handler con il quale il thread deve comunicare ed un riferimento (costante) all'oggetto *Solver* di tipo *SolverInstantiation*, rispetto al quale verranno computate le soluzioni locali. Inizialmente il solver si metterà in attesa di ricevere una query dall'handler (riga 3). Alla sua ricezione seguirà la chiamata alla funzione *create_model/1* di *Solver*, la quale, fornita una query, ritorna un modello in grado di generare tutte le computazioni locali per essa. Generato il modello, il solver andrà a chiamare i metodi `next_solution/0` e `get_solution/0` della classe *Model* per computare e spedire tutte le soluzioni parziali (ciclo while di righe 5-13). Le parametrizzazioni di *Model* devono garantire che il primo metodo — utilizzato nella guardia del ciclo while, riga 5 – ritorni *true* qualora esista un'ulteriore soluzione da inviare, mentre `get_solution/0` deve produrre tale soluzione, indicando inoltre se questa è completa e, in caso non lo sia, proponendo un solver in grado di proseguirne la computazione — oppure *null* nel caso in cui la soluzione vada mantenuta dall'handler, come spiegato in Sezione A.2. Dopo l'invio di ogni soluzione il solver controllerà, con una receive non bloccante, se ha ricevuto un messaggio di *revoke* dall'handler; in caso positivo, il thread terminerà immediatamente. Infine, come visto per il joiner, terminato l'invio delle computazioni parziali, il solver invierà un messaggio *no_solutions* all'handler, ad indicare come l'invio di tutte le soluzioni relative ad una determinata query siano state inviate.

Per istanziare correttamente il modulo solver è dunque necessario fornire una classe implementante il metodo `create_model/1` richiesto da *SolverInstantiation*. Questo metodo deve ritornare un oggetto la cui classe deve essere compatibile con il parametro *Model*. In particolare quindi, esso deve implementare i metodi `next_solution/0` e `get_solution/0`. Nell'appendice B vedremo come queste classi vengano definite per risolvere il problema del vuoto delle simulazioni lasche.

B

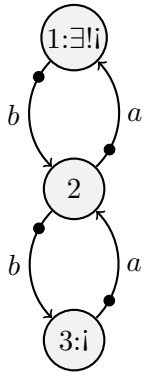
Implementazione Simulazioni Lasche via Amalgamazione Distribuita

In questa appendice vedremo alcuni dettagli progettuali riguardanti l'implementazione di un *proof of concept* per l'algoritmo distribuito del problema del vuoto delle simulazioni lasche presentato in Sezione 6. Essendo l'implementazione proposta un'istanziazione del algoritmo di Amalgamazione Distribuita progettato in Appendice A, invitiamo il lettore a riferirsi a tale appendice per la descrizione dei moduli e la definizione dei requisiti necessari per effettuare l'istanziazione correttamente. L'implementazione proposta è scritta interamente in C++ ed utilizza la libreria per generazione e manipolazione di grafi Boost Graph Library (BGL), la cui documentazione è reperibile in [38, 47], e la suite per problemi combinatori Or-tools, sviluppata da Google [48]. La prima libreria verrà utilizzata per il parsing e l'analisi di host e guest, mentre Or-tools verrà utilizzato per generare e risolvere un modello di programmazione lineare mista in grado di computare le simulazioni lasche parziali.

B.1 Sintassi Guests e Hosts

Nell'implementazione proposta, i solvers dovranno essere in grado di effettuare il parsing dei file corrispondenti alla porzione di host in loro possesso. Analogamente i guest, ricevuti da solvers e joiners in formato testuale, dovranno essere analizzati prima di poter essere utilizzati per il calcolo delle soluzioni. Per rappresentare guests e hosts è stato scelto il linguaggio DOT, formato comunemente utilizzato per la definizione di grafi ed introdotto nel software di visualizzazione Graphviz. DOT permette facilmente di specificare (multi)grafi diretti e non-diretti provvisti di etichette su nodi e archi. Inoltre, grazie alla possibilità di inserire più etichette per ogni singolo elemento del grafo, DOT permette non solo di codificare le etichette degli archi, ma anche di introdurre l'appartenenza dei nodi del guest agli insiemi must, unique ed exclusive, nonché i valori della funzione di scelta. Analogamente, le etichette possono essere utilizzate per rappresentare la partizione di un host, andando ad indicare l'indirizzo del solver associato ad ogni nodo. Per una completa descrizione del linguaggio DOT si consiglia la lettura della documentazione ufficiale [46].

Figura B.1 mostra un esempio di guest rappresentato in sintassi DOT. Ogni grafo DOT deve essere compreso tra parentesi graffe precedute da un identificatore (*guest* in Figura B.1) e una tipologia di grafo



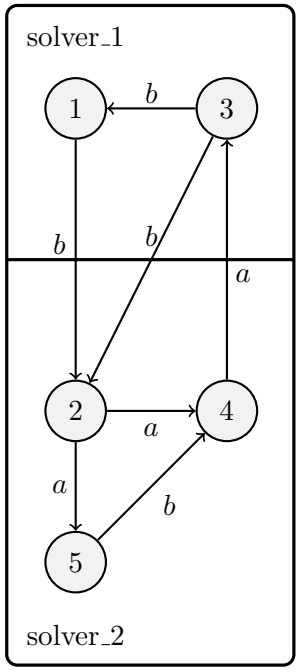

```
digraph guest{
    1 [must="true", unique="true", exclusive="true", choice="{e1}"];
    2 [choice="{e2} {e4}"];
    3 [exclusive="true", choice="{e3}"];
    1 -> 2 [id="e1", label="b"];
    2 -> 3 [id="e2", label="b"];
    3 -> 2 [id="e3", label="a"];
    2 -> 1 [id="e4", label="a"];
}
```

Figura B.1: Un guest e la sua rappresentazione in sintassi DOT.

— nel nostro caso `digraph`, ovvero un grafo diretto. I nodi sono rappresentati con un identificatore (univoco), mentre gli archi sono indicati attraverso coppie di nodi. Queste possono essere ripetute all'interno di un grafo ed ogni coppia rappresenterà un arco differente. Attraverso questa particolarità viene permessa la definizione di multigrafi. Nella codifica proposta per i guests si può notare come l'appartenenza di un nodo agli insiemi must, unique ed exclusive venga rappresentata tramite opportune etichette del nodo stesso. Ogni etichetta di un grafo DOT ha un nome (o tipo) ed un valore rappresentato da una stringa. Nel caso dei tre insiemi sopracitati, le etichette avranno rispettivamente nome `must`, `unique` e `exclusive`, il cui valore rappresenta un booleano; *true* nel caso in cui il nodo appartenga all'insieme indicato da tale etichetta, *false* altrimenti.[1] Per quanto riguarda la funzione di scelta, invece, all'etichetta `choice` di un nodo viene assegnato un valore che rappresenti i possibili insiemi tornati dalla funzione per tale vertice. Si noti come questi insiemi contengano identificatori che, nella definizione di archi, vengono rappresentati tramite l'etichetta `id`. Per le etichette di questo tipo viene richiesto un valore univoco all'interno del guest.

Passiamo ora a dare alcune indicazioni sulla sintassi delle partizioni dell'host. Ricordiamo come, per quanto detto in Appendice A — e diversamente a quanto detto in Sezione 5 —, nell'implementazione le parti di un host distribuito siano assegnate ai solvers e non direttamente ai processi. Figura B.2 mostra la rappresentazione in sintassi DOT di un host distribuito tra due solver, dove ogni sua parte corrisponde ad un diverso grafo DOT. Per ogni solver, alla lista dei nodi ad esso assegnati — privi di alcuna etichetta — segue la lista dei nodi locali ad un altro solver per i quali esiste un arco condiviso con il solver che mantiene il grafo. L'indirizzo del solver che possiede questi nodi è indicato con l'etichetta `solver_addr`, avente come valore l'indirizzo di questo. Viene poi indicata la lista degli archi (locali o condivisi) del solver. Ognuno di questi archi è decorato con la propria etichetta con valore appartenente all'alfabeto Σ, secondo la definizione di host. A tale decorazione tuttavia, per gli archi condivisi tra due solver, viene aggiunta un'ulteriore etichetta, indicata con `id`, rappresentante l'identificatore dell'arco. Questo identificatore è univoco rispetto ad ogni altro arco dell'host e deve quindi apparire unicamente nei due grafi DOT relativi ai solvers che possiedono tale arco.

[1] L'etichetta per gli insiemi must, unique ed exclusive può essere omessa nel caso in cui abbia valore *false*.




```
digraph solver_1_host{
    1
    3
    2 [solver_addr="solver_2"]
    4 [solver_addr="solver_2"]
    3 -> 1 [label="b"];
    1 -> 2 [label="b", id="1"];
    3 -> 2 [label="b", id="2"];
    4 -> 3 [label="a", id="3"];
}
```

```
digraph solver_2_host{
    2
    4
    5
    1 [solver_addr="solver_1"]
    3 [solver_addr="solver_1"]
    2 -> 4 [label="a"];
    2 -> 5 [label="a"];
    5 -> 4 [label="b"];
    1 -> 2 [label="b", id="1"];
    3 -> 2 [label="b", id="2"];
    4 -> 3 [label="a", id="3"];
}
```

Figura B.2: Un host partizionato tra due processi e la rappresentazione di queste partizioni in sintassi DOT.

B.2 Solver

Secondo quanto detto in Sezione A.4, l'implementazione del solver deve essenzialmente fornire una classe che possa generare, per una data query, un modello in grado di computare le soluzioni locali. Questa classe, che d'ora in avanti chiameremo `Driver`, dovrà dunque essere in grado di analizzare la porzione di host assegnata al solver e ogni guest da lui ricevuto. Per fare questo, è stato scelto di utilizzare la Boost Graph Library (BGL), che permette inoltre di effettuare il parsing di grafi in formato DOT tramite il metodo built-in `read_graphviz/4`. La fase di generazione dei grafi verrà quindi realizzata tramite questo metodo, il quale necessita dei seguenti parametri:

- un *filestream* contenente il grafo in formato DOT;
- un riferimento ad un grafo vuoto. BGL permette di costruire grafi in più modi, a seconda delle strutture dati che si intendono utilizzare internamente. Per esempio, l'inizializzazione di un grafo utilizzante liste di adiacenza può essere fatta tramite il template `adjacency_list`;
- un riferimento ad una mappa di proprietà, anch'essa vuota. Questa mappa va costruita a partire dalla classe `dynamic_properties` andando a precisare quali proprietà sono contenute nel grafo — come per esempio la proprietà `solver_addr`, per quanto riguarda l'host. La procedura `read_graphviz/4` modificherà la mappa andando ad aggiungere le proprietà di ogni nodo o arco.

Queste possono poi essere recuperate attraverso il metodo `get/2` che prende l'identificativo di un vertice (o arco) ed il nome della proprietà della quale recuperare il valore;

- una stringa rappresentante il nome con cui recuperare l'identificativo dei nodi — negli esempi di Figura B.1 e B.2 rappresentati tramite valori numerici.

Ogni solver utilizzerà un oggetto della classe `Driver`, il cui costruttore richiede unicamente il file DOT rappresentante la porzione di host assegnata al solver. Qualora un handler si connetta al solver, una vista immutabile — ovvero una reference costante — di questo oggetto verrà passata al metodo `solver_request_handler/2` descritto in Sezione A.4. La classe `Driver` dovrà dunque implementare il metodo `create_model/1`, il quale deve tornare l'istanza di una classe in grado di fornire le funzioni `next_solution/0` e `get_solution/0`. Tutto questo viene implementato, nel caso delle simulazioni lasche, utilizzando Or-tools, suite il cui scopo principale è quello di offrire un'interfaccia di facile utilizzo a solver di programmazione lineare mista come per esempio CPLEX e GLPK. Attraverso i metodi messi a disposizione da Or-tools è possibile generare i vincoli che costituiscono un modello di soddisfacibilità (o ottimizzazione) relativo ad un certo problema, come introdotto brevemente all'inizio di Capitolo 4.

Proporremo ora un modello di programmazione lineare mista, basato su quanto visto in Sezione 4.1, in grado di computare le simulazioni lasche locali ad un solver. Per brevità, indicheremo unicamente le differenze tra i vincoli del modello che stiamo per introdurre rispetto a quello di Sezione 4.1, spiegando principalmente perché i cambiamenti effettuati permettano di generare le soluzioni parziali di Definizione 6.13. Per completare la comprensione del modello si consiglia dunque la lettura di Capitolo 4. In seguito, siano $H = (\Sigma, V_H, E_H)$ e $G = (\Sigma, V_G, E_G, \mathcal{M}, \mathcal{U}, \mathcal{E}, \mathcal{C})$ rispettivamente un host ed un guest. Siano inoltre S un solver e $\mathcal{P}$ una mappa da vertici dell'host in indirizzi di solver. Infine, indicheremo con $\mathcal{P}^{-1}(S) \triangleq \{v \in V_H \mid \mathcal{P}(v) = S\}$ la controimmagine di S rispetto a $\mathcal{P}$. Introdurremo il modello per il calcolo delle simulazioni lasche parziali locali rispetto al solver S. Iniziamo col descrivere le variabili del modello, suddivise nei seguenti cinque insiemi:

$$
\begin{aligned}
X &= \{x_{u,v} \mid u \in V_G \ \wedge \ \mathcal{P}(v) = S\} \\
Y &= \left\{ y_{u,v,\alpha,u',v'} \;\middle|\; \begin{array}{l} \alpha \in \Sigma \ \wedge \ (u,\alpha,u') \in E_G \ \wedge \ (v,\alpha,v') \in E_H, \\ \mathcal{P}(v) = S \ \vee \ \mathcal{P}(v) = S \end{array} \right\} \\
Z &= \{z_{u,v,\gamma} \mid u \in V_G \ \wedge \ \gamma \in \mathcal{C}(u) \ \wedge \ \mathcal{P}(v) = S\} \\
C &= \{c^{e}_{u,v,\gamma} \mid u \in V_G \ \wedge \ \gamma \in \mathcal{C}(u) \ \wedge \ e \in \gamma \ \wedge \ \mathcal{P}(v) = S\} \\
F &= \left\{ f^{g,h,m}_{u,v,\alpha,u',v'} \;\middle|\; \begin{array}{l} \alpha \in \Sigma \ \wedge \ (u,\alpha,u') \in E_G \ \wedge \ (v,\alpha,v') \in E_H \ \wedge \ \mathcal{P}(v) = S, \\ g \in V_G \ \wedge \ \mathcal{P}(h) = S \ \wedge \ m \in \{m \in \mathcal{M} \mid \mathbb{P}_G(g,m) \neq \emptyset\} \end{array} \right\}
\end{aligned}
$$

dove richiediamo che le variabili in X, Y, Z e C abbiano valori interi in $\{0,1\}$ mentre le variabili in F siano a valori reali in $[0,1]$. Come si può facilmente notare, la semantica di questi insiemi di variabili è fondamentalmente inalterata rispetto al modello di Sezione 4.1. L'unica differenza consiste nel restringerci alle variabili corrispondenti a nodi o archi posseduti da un unico solver, *i.e.* S. Andiamo ora ad analizzare i vincoli di Definizione 6.7, tenendo presente tuttavia che vogliamo computare unicamente le simulazioni lasche parziali locali e, dunque, al posto di un sottoinsieme di processi $\mathbf{P}$ lavoreremo con un unico solver. Per quanto riguarda Condizione (SLP 1), essa è garantita dalla definizione degli insiemi

di variabili appena introdotti, dato che in questi vengono esclusi tutti i nodi e archi non appartenenti a S. Le condizioni (SLP 2), (SLP 3) e (SLP 4) sono invece praticamente invariate rispetto a quanto visto in Sezione 4.1 per le condizioni (SL 2), (SL 3) e (SL 4) rispettivamente. L'unica modifica riguarda le variabili utilizzate: ci restringiamo infatti ai vertici e archi di H posseduti da S. Queste condizioni possono quindi essere espresse con i seguenti vincoli:

$$\sum_{h \in \mathcal{P}^{-1}(S)} x_{u,h} \leq 1 \qquad \forall u \in \mathcal{U}$$

$$x_{u,h} + x_{v,h} \leq 1 \qquad \forall u \in \mathcal{E}\ \forall v \in V_G \setminus \{u\}\ \forall h \in \mathcal{P}^{-1}(S)$$

$$y_{u,v,\alpha,u',v'} \leq x_{u,v} \qquad \forall \alpha \in \Sigma\ \forall (u,\alpha,u') \in E_G\ \forall (v,\alpha,v') \in E_H \text{ s.t. } \mathcal{P}(v) = S$$

$$y_{u,v,\alpha,u',v'} \leq x_{u',v'} \qquad \forall \alpha \in \Sigma\ \forall (u,\alpha,u') \in E_G\ \forall (v,\alpha,v') \in E_H \text{ s.t. } \mathcal{P}(v') = S$$

Condizione (SLP 5) invece appare leggermente modificata rispetto a Condizione (SL 5). In questa, infatti, la seconda parte del vincolo richiede che ogni elemento $(e, e') \in \eta$ sia frutto della selezione di un insieme $\gamma \in \mathcal{C}(s(e))$, con $e \in \gamma$ *solo qualora la sorgente di* e' *sia locale a* S, *i.e.* $\mathcal{P} \circ s(e') = S$. Questo controllo di località può essere modellato facilmente a partire dal vincolo corrispondente di Sezione 4.1, producendo quindi il seguente insieme di disequazioni

$$y_{u,v,\alpha,u',v'} \leq \sum_{\substack{\gamma \in C(u) \\ (u,\alpha,u') \in \gamma}} c^{(u,\alpha,u')}_{u,v,\gamma} \qquad \forall \alpha \in \Sigma\ \forall (u,\alpha,u') \in E_G\ \forall (v,\alpha,v') \in E_H \text{ s.t. } \mathcal{P}(v) = S$$

Gli altri vincoli di Condizione (SLP 5) sono invece analoghi a quanto visto per Condizione (SL 5), con la sola riduzione alle variabili corrispondenti a vertici e archi posseduti da S, come fatto per i tre vincoli precedenti. La loro codifica è dunque la seguente:

$$z_{u,v,\gamma} \leq x_{u,v} \qquad \forall u \in V_G\ \forall \gamma \in \mathcal{C}(u)\ \forall v \in \mathcal{P}^{-1}(S)$$

$$x_{u,v} \leq \sum_{\gamma \in C(u)} z_{u,v,\gamma} \forall u \in V_G\ \forall v \in \mathcal{P}^{-1}(S)$$

$$\forall u \in V_G\ \forall \gamma \in \mathcal{C}(u) \setminus \{\emptyset\}\ \forall v \in \mathcal{P}^{-1}(S)$$

$$|\gamma| z_{u,v,\gamma} \leq \sum_{e \in \gamma} c^e_{u,v,\gamma}$$

$$\sum_{e \in \gamma} c^e_{u,v,\gamma} \leq |\gamma|(z_{u,v,\gamma} - 1)$$

$$\forall u \in V_G\ \forall \gamma \in \mathcal{C}(u)\ \forall (u,\alpha,u') \in \gamma\ \forall v \in \mathcal{P}^{-1}(S)$$

$$\sum_{(v,\alpha,v') \in \text{out}_\alpha(v)} y_{u,v,\alpha,u',v'} \leq |\text{out}_\alpha(v)| c^{(u,\alpha,u')}_{u,v,\gamma} \qquad \text{se out}_\alpha(v) \neq \emptyset$$

$$|\text{out}_\alpha(v)| c^{(u,\alpha,u')}_{u,v,\gamma} \leq |\text{out}_\alpha(v)| - 1 + \sum_{(v,\alpha,v') \in \text{out}_\alpha(v)} y_{u,v,\alpha,u',v'} \qquad \text{se out}_\alpha(v) \neq \emptyset$$

$$c^{(u,\alpha,u')}_{u,v,\gamma} = 0 \qquad \text{se out}_\alpha(v) = \emptyset$$

dove $\text{out}_\alpha(v) \triangleq \{e \in \text{out}(v) \mid \sigma(e) = \alpha\}$.

Passiamo infine a considerare Condizione (SLP 6). Per semplificare l'esposizione, similmente a quanto fatto per (SL 6), introduciamo il grafo di una simulazione lasca parziale.

Definizione B.3 (Grafo di una simulazione lasca parziale)**.** *Sia* $(\phi, \eta) \in \mathbb{S}_{\mathcal{P}}^{G \to H}$ *una simulazione lasca parziale e* (X_η, Y_η) *la sua frontiera.* (ϕ, η) *sottintende una quintupla, chiamata* grafo di una simulazione lasca parziale*,* $(\Sigma, \phi, E_\eta, \psi_{in}, \psi_{out})$ *dove*

- Σ *è l'alfabeto di* G *e* H
- $E_\eta \triangleq \{((u,v), \alpha, (u',v')) \mid ((u,\alpha,u'),(v,\alpha,v')) \in \eta\}$*;*
- $\psi_{in} \triangleq \{(u,v) \in V_G \times V_H \mid \exists u' \in V_G \ \exists v' \in V_H \ ((u,\alpha,u'),(v,\alpha,v')) \in X_\eta\}$*;*
- $\psi_{out} \triangleq \{(u,v) \in V_G \times V_H \mid \exists u' \in V_G \ \exists v' \in V_H \ ((u',\alpha,u),(v',\alpha,v)) \in Y_\eta\}$*;*
- $(\Sigma, \phi \cup \psi_{in} \cup \psi_{out}, E_\eta)$ *è un multigrafo diretto etichettato.*

Rispetto a questa definizione, data una simulazione lasca parziale $(\phi, \eta) \in p\mathbb{S}_{\mathcal{P}}^{G \to H}$, Condizione (SLP 6) richiede che, per ogni $(u,v) \in \phi$, dato un $m \in \mathcal{M}$, se $\mathbb{P}_G(u,m) \neq \emptyset$ allora esiste un cammino nel grafo della simulazione lasca parziale $(\Sigma, \phi, E_\eta, \psi_{\text{in}}, \psi_{\text{out}})$ terminante o in un un elemento $j \in \phi$ tale che $\pi_1(j) = m$ oppure in un elemento $k \in \psi_{out}$ tale che $\mathbb{P}_G(\pi_1(k), m) \neq \emptyset$. Da questo deduciamo che il vincolo (SLP 6) possa essere risolto attraverso il modello per il problema multiflusso impostato per (SL 6), dove però, dato un vertice $m \in \mathcal{M}$, oltre all'insieme di coppie $\{(m,x) \mid \mathcal{P}(x) = S\}$ vengono considerate terminazioni anche i nodi $(u',v') \in \psi_{\text{out}}$ — *i.e.* le destinazioni di un archi appartenenti a Y_η — tali che, nel guest, esiste un cammino da u' a w. Per apportare questa modifica al modello basta aggiungere un vincolo che assicuri il flusso complessivo terminante in tali nodi sia pari a quello generato dalla sorgente. Questo può essere fatto facilmente attraverso il seguente insieme di $\mathcal{O}(|V_G|^2|V_H|)$ disequazioni, ciascuna con $\mathcal{O}(|E_G||E_H|)$ variabili.

$$\forall g \in V_G \ \forall m \in \{m \in \mathcal{M} \mid \mathbb{P}(g,m) \neq \emptyset\} \ \forall h \in \mathcal{P}^{-1}(S)$$
$$\sum_{\substack{\alpha \in \Sigma \\ (u,\alpha,m) \in E_G \\ (v,\alpha,v') \in E_H \\ \mathcal{P}(v) \in S}} f^{g,h,m}_{u,v,\alpha,m,v'} + \sum_{\substack{\alpha \in \Sigma \\ (u,\alpha,u') \in E_G \\ (v,\alpha,v') \in E_H \\ \mathbb{P}_G(u',m) \neq \emptyset \\ \mathcal{P}(v') \notin S}} f^{g,h,m}_{u,v,\alpha,u',v'} = x_{g,h}$$

Gli altri vincoli necessari per codificare questa condizione sono invece analoghi a quanto visto per (SL 6), eccezione fatta per i domini delle variabili che, esattamente come visto in precedenza per le altre condizioni, è ristretto a quelle corrispondenti a nodi o archi posseduti da S.

$$\forall g \in V_G \ \forall m \in \{m \in \mathcal{M} \mid \mathbb{P}(g,m) \neq \emptyset\} \ \forall h \in \mathcal{P}^{-1}(S)$$
$$\forall \alpha \in \Sigma \ \forall (u,\alpha,u') \in E_G \ \forall (v,\alpha,v') \in E_H \text{ s.t. } \mathcal{P}(v) \in S$$
$$f^{g,h,m}_{u,v,\alpha,u',v'} \leq y_{u,v,\alpha,u',v'}$$
$$f^{g,h,m}_{u,v,\alpha,u',v'} \leq x_{g,h}$$

$$\forall g \in V_G \; \forall m \in \{m \in \mathcal{M} \mid \mathbb{P}(g,m) \neq \emptyset\} \; \forall h \in \mathcal{P}^{-1}(S)$$

$$\sum_{\substack{\alpha\in\Sigma \\ (u,\alpha,u')\in E_G \\ (v,\alpha,v')\in E_H}} f^{g,h,m}_{u,v,\alpha,u',v'} = \sum_{\substack{\alpha\in\Sigma \\ (u',\alpha,u)\in E_G \\ (v',\alpha,v)\in E_H \\ \mathcal{P}(v')\in S}} f^{g,h,m}_{u',v',\alpha,u,v} \quad \forall (u,v) \in V_G \times \mathcal{P}^{-1}(S) \setminus (\{(g,h)\} \cup \{(m,x) \mid \mathcal{P}(x) = S\})$$

$$\sum_{\substack{\alpha\in\Sigma \\ (g,\alpha,u)\in E_G \\ (h,\alpha,v)\in E_H}} f^{g,h,m}_{g,h,\alpha,u,v} = x_{g,h}$$

$$\sum_{\substack{\alpha\in\Sigma \\ (u,\alpha,g)\in E_G \\ (v,\alpha,h)\in E_H}} f^{g,h,g}_{u,v,\alpha,g,h} = 0 \qquad \text{se } g \neq m$$

Con questi vincoli termina la definizione del modello di programmazione lineare mista per il calcolo delle simulazioni lasche parziali locali. Esattamente come visto per il modello di Capitolo 4, sono state impiegate un numero polinomiale di variabili, disequazioni e variabili per disequazione: anche questo problema risulta essere quindi NP-completo.

Come già detto, questo modello verrà generato dalla classe `Driver` attraverso il metodo `create_model/1`, utilizzando Or-tools. Figura B.4 mostra lo pseudocodice per una parte di questo metodo. In essa viene indicato come accedere ai vertici e alle proprietà di un grafo di Boost Graph Library, come generare le variabili di un modello di Or-tools e come utilizzare queste per aggiungere un vincolo al modello — nello specifico, l'insieme di disequazioni caratterizzanti Condizione (SLP 2). Inizialmente viene generato il grafo per il guest e istanziato l'oggetto solver (righe 2 e 3). La funzione passa poi ad iterare sui vertici di guest e host utilizzando la funzione `vertices/1`. Per ogni coppia di vertici, uno del guest ed uno dell'host, viene aggiunta una variabile avente un identificativo univoco generato a partire dagli identificatori dei due vertici (riga 12). Infine, nel caso in cui il vertice del guest appartenga all'insieme unique, viene aggiunta per questo la disequazione caratterizzante Condizione (SLP 2), utilizzando le variabili del modello salvate nel vettore *variables*, *i.e.* tutte le variabili generate a partire da quel vertice (riga 16).

Una volta aggiunti tutti i vincoli, l'oggetto *solver* rappresentante il modello verrà utilizzato per istanziare un oggetto della classe `Model`, il quale avrà il compito di avviare la procedura di risoluzione del modello e computare, a partire dalle soluzioni di quest'ultimo, le interfacce delle simulazioni lasche. Come già accennato in Sezione A.4 queste operazioni verranno eseguite chiamando i metodi `next_solution/0` e `get_solution/0`. Lo pseudocodice per questi due metodi è riportato in Figura B.5. Il metodo `next_solution/0` inizierà invocando il metodo `NextSolution/0` offerto dall'oggetto *solver* di Or-tools, il quale andrà a computare una nuova soluzione. Qualora questa esista, la funzione tornerà *true* e potrà essere recuperata andando ad analizzare l'istanziazione delle variabili del modello. `next_solution/0` continuerà dunque computando la simulazione lasca parziale, a partire dagli insiemi di variabili X e Y. Questo può essere fatto in virtù del seguente risultato, analogo a quanto visto in Sezione 4.1.

Proposizione B.6 (Da soluzione di modello a simulazione lasca parziale)**.** *Siano* (X, Y, Z, C, F) *le variabili del modello di programmazione lineare mista per il calcolo delle simulazioni lasche parziali locali rispetto ad un solver* S*. Sia* $V = (X_V, Y_V, Z_V, C_V, F_V)$ *una soluzione computata a partire da queste,*

```
1  Driver::create_model (StringStream Guest)
2      Sia Host_graph il grafo dell'host mantenuto da questo oggetto Driver
3      Inizializza Guest_graph (Boost Graph Library)
4      Inizializza Dp (Boost's dinamic_properties)
5      boost::read_graphviz(Guest, Guest_graph, Dp, "node_id")
6      or::Solver Solver("Solver")                          // Inizializza il solver Or-tools
7      foreach Guest_vertex ∈ boost::vertices(Guest_graph) do
8          std::vector Variables
9          foreach Host_vertex ∈ boost::vertices(Host_graph) do
10             if boost::get(solver_addr, Host_vertex) = "" then
11                 String Var_name = "X_" ++ Guest_vertex ++ "_" ++ Host_vertex
12                 or::IntVar* Iv = Solver.MakeIntVar(0, 1, Var_name)  // Aggiungi una variabile {0, 1}
13                 Variables.push_back(Iv)
14         end
15         if boost::get(unique, Guest_vertex) = "true" then
               // Aggiungi le disequazioni per Condizione (SLP 2)
16             Solver.AddConstraint(Solver.MakeLessOrEqual(Solver.MakeSum(Vertices), 1))
17     end
18     Add other constraints
19     return Model(Solver)
```

Figura B.4: Generazione modello Or-tools e codifica Condizione (SLP 2).

```
1  Model::next_solution ()
2      while solver.NextSolution() do
3          let V l'insieme di tutte le variabili del solver, istanziate rispetto alla soluzione corrente in
4              Computa la simulazione lasca parziale ρ equivalente a V
5              Computa l'interfaccia I(ρ)
6              if I(ρ) è una nuova interfaccia then
7                  Salva I(ρ) nell'insieme delle interfacce computate
8                  LastInterface ← I(ρ)   // get_solution/0 leggerà questa variabile per recuperare l'interfaccia
9                  return true
10     return false
```

```
1  Model::get_solution ()
2      Sia LastInterface= {S} ⊢ ⟨X, μ_M, μ_Y⟩ --M, U--> ⟨Y, ν⟩
           l'ultima interfaccia computata da next_solution/0
3      S_min ← min({P(v) | (e, (v, a, v')) ∈ X  ∨  (e, (v', a, v)) ∈ Y})
4      if X = Y = ∅ ∧ M = M then
5          return (true, serialize(LastInterface), null)
6      else if  S ≤ S_min then
7          return (false, serialize(LastInterface), null)
8      else
9          return (true, serialize(LastInterface), S_min)
```

Figura B.5: Metodi `next_solution/0` e `get_solution/0` per gli oggetti di tipo *Model* tornati da `create_model/1`.

con X_V, Y_V, Z_V, C_V e F_V istanziazione delle variabili degli insiemi X, Y, Z, C e F rispettivamente. Allora la coppia di relazioni (ϕ, η) definita come

$$\phi = \{(u, v) \mid X_V \ni x_{u,v} = 1\}$$
$$\eta = \{((u, \alpha, u'), (v, \alpha, v')) \mid Y_V \ni y_{u,v,a,u',v'} = 1\}$$

è una simulazione lasca parziale locale a S.

Data una simulazione lasca parziale, è possibile recuperare la sua interfaccia in tempo polinomiale seguendone semplicemente la definizione data in Sezione 6.3 e ricordando come la raggiungibilità tra due nodi di un grafo sia un problema polinomiale – e quindi le condizioni che richiedono di computare δ_η possono essere risolte attraverso il grafo delle simulazioni lasche parziali. Per la descrizione di un'implementazione delle interfacce, rimandiamo il lettore alle strutture dati definite in Sezione 6.4.

Una volta computata l'interfaccia, il processo verificherà se questa sia già stata computata in precedenza. Le istanze di *Model* possiedono infatti una struttura dati in grado di mantenere queste interfacce. Nel caso in cui l'interfaccia appena computata non appartenga a tale struttura, verrà aggiunta a quest'ultima ed assegnata alla variabile *LastInterface*, in modo tale da poter essere recuperata dal metodo `get_solution/0`. In caso contrario, si procederà con la ricerca di una nuova soluzione, richiamando nuovamente il metodo `NextSolution/0` di *solver*. L'intera funzione termina dunque o quando quest'ultimo metodo ritorna *false* oppure qualora venga computata una nuova interfaccia. Nel secondo caso, per quanto detto in Sezione A.4, verrà chiamato il metodo `get_solution/0`, il quale recupererà l'interfaccia — salvata nella variabile *LastInterface* — e deciderà come questa vada instradata. In particolare, facendo riferimento al codice di questo metodo proposto in Figura B.5, qualora l'interfaccia corrisponda ad una simulazione lasca (completa) — secondo quanto detto in Proposizione 6.21 — `get_solution/0` tornerà la tupla (*true*, `serialize`(*LastInterface*), *null*), indicando così che la soluzione tornata è completa (riga 5). Si noti come il metodo esegua il marshalling dell'interfaccia, rendendone così possibile la trasmissione da parte dell'handler.

Nel caso in cui l'interfaccia non corrisponda ad una simulazione lasca va individuato invece il solver in grado di espandere la soluzione parziale, secondo la definizione di instradamento data in Sezione 6.5. Come ivi spiegato, l'instradamento viene fatto rispetto ad un ordinamento sugli identificativi dei processi — o, nel nostro caso, dei solvers — andando in particolare a verificare se l'identificativo minore tra quelli che possono espandere la soluzione sia o meno maggiore dell'identificativo del solver che l'ha computata. Sia S il solver che ha computato l'interfaccia $\{S\} \vdash \langle X, \mu_\mathcal{M}, \mu_Y \rangle \xrightarrow{M,\ U} \langle Y, \nu \rangle$. Rispetto alla partizione $\mathcal{P}$, l'insieme dei solvers che possono espandere questa interfaccia è dato da tutti i solver che condividono con S archi dell'host nella frontiera (X, Y), *i.e.* $\{\mathcal{P}(v) \mid (e, (v, a, v')) \in X \ \vee \ (e, (v', a, v)) \in Y\}$. Sia $S_{\min}$ il solver con l'identificativo (lessicograficamente) più piccolo tra quelli appartenenti a questo insieme. Qualora $S \leq S_{\min}$, la definizione di instradamento richiede che l'interfaccia sia mantenuta dal joiner associato all'handler che ha interrogato S. Ricordando quanto detto in Sezione A.2 rispetto all'implementazione di handler, questo può essere forzato ponendo a *null* l'indirizzo del destinatario (riga 7). In caso contrario, l'interfaccia dovrà essere inviata all'handler utilizzante $S_{\min}$ (riga 9).

B.3 Joiner

Seguendo quanto descritto in Appendice A, le soluzioni computate dai solvers verranno spedite ai processi joiner eseguiti dagli handlers tramite le *porte* Erlang. Richiamando quanto visto in Sezione A.3, per istanziare correttamente un joiner è necessario fornire una classe, alla quale daremo il nome `Composer`, implementante i metodi

- `set_query_infos/2`, per indicare l'identità del solver locale e il guest da questo utilizzato per le computazioni;
- `new_local_computation/1`, il quale permette il recupero delle soluzioni computate a partire da un'interfaccia di simulazione lasca parziale trovata dal solver locale;
- `new_network_computation/1`, il quale permette il recupero delle soluzioni computate a partire da un'interfaccia ottenuta da un solver diverso da quello locale.

L'implementazione di questi tre metodi è schematizzata attraverso lo pseudocodice di Figura B.7. Per quanto riguarda `set_query_infos/2`, verrà eseguito il parsing del guest — così come descritto per il solver nella sezione precedente — e da questo verrà ricavata la funzione $\widehat{\nu}$ necessaria per computare la funzione ν delle interfacce che, come spiegato in Sezione 6.3, non verrà spedita ed è necessaria per eseguire l'algoritmo di join. L'identificativo del solver locale passato a `set_query_infos/2` non verrà utilizzato.

Passiamo ora ad analizzare i metodi `new_local_computation/1` e `new_network_computation/1`. In entrambi i metodi, si noti come l'interfaccia (di simulazione lasca parziale) passata in input venga deserializzata ed aggiunta ad una struttura dati, rispettivamente *LocalSolutions* per `new_local_computation/1` e *NetworkSolutions* per `new_network_computation/1`. Gli oggetti della classe `Composer`, infatti, manterranno tre strutture dati:

- *LocalSolutions*, contenente tutte le simulazioni lasche parziali ricevute dal joiner e computate dal solver locale;
- *NetworkSolutions*, contenente tutte le simulazioni lasche parziali ricevute dal joiner e computate da solver diversi da quello locale;
- *ComposedSolutions*, contenente tutte le simulazioni lasche parziali computate tramite applicazioni dell'operatore di join a partire da una soluzione di *LocalSolutions* ed almeno una soluzione di *NetworkSolutions*.

Dopo aver aggiunto l'interfaccia passata come input alla relativa struttura dati, i due metodi ritorneranno un oggetto di tipo `ComposerIterator`, il quale permette l'iterazione delle soluzioni computate tramite i metodi `next_solution/0` e `get_solution/0` richiesti in Sezione A.3. Lo pseudocodice relativo al costruttore e ai metodi di questa classe è rappresentato in Figura B.8.

Per istanziare un oggetto `ComposerIterator` vanno forniti tre parametri:

- un interfaccia di una simulazione lasca parziale, utilizzata per effettuare i join;
- una lista di strutture dati sulla quale iterare per computare le nuove soluzioni;

```
1 Composer::set_query_infos (Address Solver, StringStream Guest)
2     Inizializza Guest_graph (Boost Graph Library)
3     Inizializza Dp (Boost's dinamic_properties)
4     boost::read_graphviz(Guest, Guest_graph, Dp, "node_id")
5     Computa ν̂ ≜ λv : E_G.{u ∈ M | v = u ∨ P_G(v, u) ≠ ∅}, utilizzato per effettuare il join
```

```
1 Composer::new_local_computation (BinaryData Interface)
2     Aggiungi deserialize(Interface) all'insieme delle soluzioni locali LocalSolutions
3     return ComposerIterator(deserialize(Interface), [NetworkSolutions], this)
```

```
1 Composer::new_network_computation (BinaryData Interface)
2     Aggiungi deserialize(Interface) all'insieme delle soluzioni non locali NetworkSolutions
3     return ComposerIterator(deserialize(Interface), [LocalSolutions, ComposedSolutions], this)
```

Figura B.7: Metodi `set_query_infos/2`, `new_local_computation/1` e `new_network_computation/1` per gli oggetti di tipo `Composer`.

```
1 ComposerIterator::ComposerIterator (Interface If, List Structures, Composer*
  Composer)
2     Solution ← If
3     Caller ← Composer
4     CurrentStructure ← head(Structures)
5     OtherStructures ← tail(Structures)
6     Iterator ← begin(CurrentStructure)
```

```
1  ComposerIterator::next_solution ()
2      if Iterator = end(CurrentStructure) then
3          if OtherStructures = [] then
4              return false
5          else
6              CurrentStructure ← head(OtherStructures)
7              OtherStructures ← tail(OtherStructures)
8              Iterator ← begin(CurrentStructure)
9              return next_solution()
10     else
11         JoinSolution ← Solution ⊔ *Iterator
12         Iterator++
13         if Join = Error oppure Join appartiene a Caller→ComposedSolutions then
14             return next_solution()
15         else
16             return true
```

```
1 ComposerIterator::get_solution ()
2     Aggiungi JoinSolution all'insieme delle soluzioni computate Caller→ComputedSolutions
3     return JoinSolution
```

Figura B.8: Creazione di un `ComposerIterator` ed i suoi metodi `next_solution/0` e `get_solution/0`.

- un puntatore all'oggetto di tipo `Composer` che effettua la creazione di questo oggetto.

Rispetto alla lista di strutture dati da passare in input, tornando allo pseudocodice relativo al `Composer` — Figura B.7 —, si può notare come questo parametro sia pari a *[NetworkSolutions]* per il metodo `new_local_computation/1`, mentre sia pari a *[LocalSolutions, ComposedSolutions]* nel caso della funzione `new_network_computation/1`. Infatti, per la semantica data alle tre strutture dati mantenute da oggetti `Composer` e ricordando quanto detto in Assunzione 6.16, non è necessario effettuare il join di più interfacce computate dal solver locale. È quindi sufficiente unire queste ultime con le soluzioni in *NetworkSolutions*. D'altro canto, interfacce inviate all'handler da un processo esterno richiedono l'apporto del solver locale per essere estese. Queste verranno quindi composte unicamente con soluzioni di *LocalSolutions* e *ComposedSolutions*.

Il costruttore di `ComposerIterator` dunque, salverà l'interfaccia e il puntatore all'oggetto `Composer` passati in input rispettivamente nelle variabili *Solution* e *Caller*. In seguito preparerà l'iterazione rispetto alla prima struttura dati (righe 4-6), andando a salvare quest'ultima nella variabile *CurrentStructure*, a mantenere le altre strutture dati tramite l'assegnamento di *`tail`(Structures)* a *OtherStructures* e impostando l'iteratore alla prima interfaccia di *CurrentStructure*. Generato un oggetto di tipo `ComposerIterator` esso può quindi essere utilizzato per computare i join ottenuti a partire da una specifica interfaccia, andando ad invocare i metodi `next_solution/0` e `get_solution/0`. Per quanto riguarda il metodo `get_solution/0`, dallo pseudocodice di Figura B.8 si può notare come questo si limiti a salvare l'interfaccia *JoinSolution* trovata tramite `next_solution/0` nella struttura dati *ComputedSolutions* di *Caller*, per poi tornarle tale interfaccia in output. Passiamo dunque ad analizzare il metodo `next_solution/0`. Una parte di questa funzione — la quale, ricordiamo, deve tornare *true* qualora venga trovata un'interfaccia e *false* altrimenti — è adibita all'iterazione delle strutture dati passate all'oggetto in fase di creazione. In particolare, qualora l'iteratore *Iterator* raggiunga il termine della struttura dati attualmente analizzata (riga 2) e nel caso in cui vi siano altre strutture dati da percorrere (riga 5), allora *Iterator* verrà impostato al primo elemento della prossima struttura dati (righe 6-9). Se invece questa non esiste una struttura dati successiva, allora tutte le interfacce computabili a partire da *Solution* sono state trovate e il metodo tornerà *false* (righe 3-4). Infine, qualora l'iteratore non abbia terminato di scandire una struttura dati, l'interfaccia da esso referenziata verrà unita con l'interfaccia *Solution*, utilizzando l'algoritmo per il join di interfacce proposto in Sezione 6.4. Qualora le due interfacce non siano componibili — ovvero l'algoritmo ritorni *Error* — oppure nel caso in cui l'interfaccia risultante sia già presente nella struttura dati *ComputedSolutions* di *Caller*, verrà eseguita una chiamata ricorsiva a `next_solution/0` che andrà ad analizzare la prossima interfaccia — si noti infatti l'incremento dell'iteratore, riga 12. In caso contrario, il risultato dell'algoritmo di join corrisponde ad una nuova interfaccia e quindi il metodo ritornerà *true*.

Bibliografia

[1] Joannis Apostolakis, Robert Körner, e Jörn Marialke. Embedded subgraph isomorphism and its applications in cheminformatics and metabolomics. In *1st German Conference in Chemoinformatics*, 2005.

[2] Joe Armstrong, Robert Virding, e Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall, 1993.

[3] Pablo Barceló, Leonid Libkin, e Juan L. Reutter. Querying regular graph patterns. *J. ACM*, 61(1):8:1–8:54, 2014.

[4] Jiri Barnat, Lubos Brim, e Jitka Stríbrná. Distributed LTL model-checking in SPIN. In *SPIN*, volume 2057 di *Lecture Notes in Computer Science*, pp. 200–216. Springer, 2001.

[5] Robert E. Bixby. Solving real-world linear programs: A decade and more of progress. *Operations Research*, 50(1):3–15, 2002.

[6] Bard Bloom e Robert Paige. Transformational design and implementation of a new efficient solution to the ready simulation problem. *Sci. Comput. Program.*, 24(3):189–220, 1995.

[7] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis E. Shasha, e Alfredo Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics*, 14(S-7):S13, 2013.

[8] Grady Booch, James E. Rumbaugh, e Ivar Jacobson. The unified modeling language user guide. *J. Database Manag.*, 10(4):51–52, 1999.

[9] Luca Cardelli e Andrew D. Gordon. Mobile ambients. In *FoSSaCS*, volume 1378 di *Lecture Notes in Computer Science*, pp. 140–155. Springer, 1998.

[10] Fan R. K. Chung, Ronald L. Graham, Ranjita Bhagwan, Stefan Savage, e Geoffrey M. Voelker. Maximizing data locality in distributed systems. *J. Comput. Syst. Sci.*, 72(8):1309–1316, 2006.

[11] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC*, pp. 151–158. ACM, 1971.

[12] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, e Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[13] Giuseppe De Giacomo e Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI*, pp. 854–860. IJCAI/AAAI, 2013.

[14] Edsger W. Dijkstra e Carel S. Scholten. Termination detection for diffusing computations. *Inf. Process. Lett.*, 11(1):1–4, 1980.

[15] Wenfei Fan. Graph pattern matching revised for social network analysis. In *ICDT*, pp. 8–21. ACM, 2012.

[16] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, e Yinghui Wu. Adding regular expressions to graph reachability and pattern queries. *Frontiers of Computer Science*, 6(3):313–338, 2012.

[17] Wenfei Fan, Xin Wang, Yinghui Wu, e Dong Deng. Distributed graph simulation: Impossibility and possibility. *PVLDB*, 7(12):1083–1094, 2014.

[18] Arash Fard, M. Usman Nisar, Lakshmish Ramaswamy, John A. Miller, e Matthew Saltz. A distributed vertex-centric approach for pattern matching in massive graphs. In *BigData Conference*, pp. 403–411. IEEE, 2013.

[19] Jacques Ferber. *Multi-agent systems - an introduction to distributed artificial intelligence*. Addison-Wesley-Longman, 1999.

[20] Susan L. Graham, Peter B. Kessler, e Marshall K. McKusick. gprof: a call graph execution profiler (with retrospective). In *Best of PLDI*, pp. 49–57. ACM, 1982.

[21] Monika Rauch Henzinger, Thomas A. Henzinger, e Peter W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, pp. 453–462. IEEE Computer Society, 1995.

[22] John E. Hopcroft, Rajeev Motwani, e Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - international edition (2. ed)*. Addison-Wesley, 2003.

[23] Shing-Tsaan Huang, (A cura di). *Detecting Termination of Distributed Computations by External Agents*. Proceedings of the 9th International Conference on Distributed Computing Systems, 1989.

[24] Neil D. Jones. Space-bounded reducibility among combinatorial problems. *J. Comput. Syst. Sci.*, 11(1):68–85, 1975.

[25] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, The IBM Research Symposia Series, pp. 85–103. Plenum Press, New York, 1972.

[26] Orna Kupferman e Moshe Y. Vardi. Verification of fair transition systems. *Chicago J. Theor. Comput. Sci.*, 1998, 1998.

[27] Jens Lischka e Holger Karl. A virtual network mapping algorithm based on subgraph isomorphism detection. In *VISA*, pp. 81–88. ACM, 2009.

[28] Shuai Ma, Yang Cao, Jinpeng Huai, e Tianyu Wo. Distributed graph pattern matching. In *WWW*, pp. 949–958. ACM, 2012.

[29] Alessio Mansutti, Marino Miculan, e Marco Peressotti. Distributed execution of bigraphical reactive systems. *ECEASST*, 71, 2014.

[30] Friedemann Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3):161–175, 1987.

[31] Alberto O. Mendelzon e Peter T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput.*, 24(6):1235–1258, 1995.

[32] Robin Milner. *A Calculus of Communicating Systems*, volume 92 di *Lecture Notes in Computer Science*. Springer, 1980.

[33] Robin Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.

[34] Pavel A. Pevzner. *Computational molecular biology - an algorithmic approach*. MIT Press, 2000.

[35] Grzegorz Rozenberg, (A cura di). *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[36] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2012.

[37] Paolo Serafini. *Ricerca Operativa*. Springer, 2002.

[38] Jeremy G. Siek, Lie-Quan Lee, e Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual, Portable Documents*. Addison-Wesley Professional, 2001.

[39] John F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley, 1984.

[40] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 2013.

[41] Ken Thompson. Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

[42] Julian R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[43] Xifeng Yan e Jiawei Han. gspan: Graph-based substructure pattern mining. In *ICDM*, pp. 721–724. IEEE Computer Society, 2002.

[44] Djelloul Ziadi. Regular expression for a language without empty word. *Theor. Comput. Sci.*, 163(1&2):309–315, 1996.

[45] Erlang Progamming Language. `https://www.erlang.org/docs`.

[46] The DOT Language. `http://www.graphviz.org/content/dot-language`.

[47] The Boost Graph Library. `http://www.boost.org/doc/libs/1_62_0/libs/graph/doc/`.

[48] Google Optimization Tools. `https://developers.google.com/optimization/`.