Università degli Studi di Trieste

Dipartimento di Ingegneria e Architettura

Corso di Studi in Ingegneria Elettronica ed Informatica

Tesi di Laurea Triennale

BisPy

un pacchetto Python per il calcolo della massima bisimulazione di grafi diretti

Studente:
Francesco Andreuzzi

Relatore:
Prof. Alberto Casagrande

Anno accademico 2020-2021

Indice

Pagina

Introduzione

Tramite i *grafi* è possibile formulare modelli informatici di sistemi complessi di varia natura, come *social network* [6], reti fisiche [8] o logistiche [22]. In tali modelli risulta difficoltoso individuare una nozione univoca di equivalenza tra i nodi che compongono il sistema, in quanto diverse definizioni catturano differenti aspetti del problema. Una nozione di equivalenza può essere utilizzata per condurre analisi sul grafo (dividere i nodi in un partizionamento che ne catturi una qualche caratteristica), o per ottimizzare lo spazio occupato dal modello eliminando informazioni ridondanti.

In questo lavoro prenderemo in considerazione una possibile nozione di equivalenza, la *bisimulazione* (secondo la definizione standard); in particolare ci concentreremo sull'aspetto informatico del problema, e studieremo alcuni algoritmi efficienti che consentono di calcolare un partizionamento del grafo in sottoinsiemi di nodi equivalenti. Inoltre sarà presentata la libreria Python `BisPy`, contenente l'implementazione degli algoritmi analizzati. Non è stato possibile trovare altri progetti che trattino il suddetto argomento in modo approfondito, per cui tale software si può considerare inedito nell'ambiente *open source*; la semplicità di utilizzo del pacchetto lo rende uno strumento potenzialmente molto utile per lo studio della massima bisimulazione. Le analisi e le osservazioni riportate in questo documento sono state fondamentali per una corretta implementazione degli algoritmi considerati.

Il lavoro è organizzato in questo modo:

- Nella Sezione 1 forniremo innanzitutto alcune nozioni fondamentali nell'ambito di teoria dei grafi, teoria degli insiemi e relazioni binarie, per poi introdurre le definizioni di bisimulazione e massima bisimulazione, insieme ad alcuni risultati che utilizzeremo per le analisi successive;

- Nella Sezione 2 analizzeremo alcuni algoritmi efficienti per il problema della massima bisimulazione, e ne dimostreremo in modo dettagliato la correttezza e la complessità computazionale;

- Nella Sezione 3 presenteremo il pacchetto Python `BisPy`, e valuteremo il tempo di esecuzione degli algoritmi implementati su alcuni esempi significativi.

1 Nozioni di Base

In questa prima sezione del lavoro intendiamo fornire alcune nozioni di base fondamentali per una piena comprensione dell'argomento; in seguito proporremo la definizione di *bisimulazione*, e ne dedurremo alcune conseguenze immediate che utilizzeremo ampiamente nel seguito della trattazione; proseguiremo illustrando alcune idee che consentono di legare la *bisimulazione* e la teoria degli insiemi; infine concluderemo la sezione introducendo un altro problema, la determinazione della *relational stable coarsest partition*, che come si vedrà è equivalente al problema della bisimulazione. Questa equivalenza avrà un ruolo fondamentale nella Sezione 2.

1.1 Relazioni Binarie

Riportiamo la definizione di *relazione binaria* [18] su uno o due insiemi, che sarà utile per definire formalmente il concetto di *grafo*, fondamentale all'interno di questo elaborato:

Definizione 1.1. Una *relazione binaria* su A, B è un sottoinsieme del prodotto cartesiano $A \times B$.
Una *relazione binaria* su A è un sottoinsieme del prodotto cartesiano $A \times A$.
Se $\mathcal{R}$ mette in relazione u, v, cioè $(u, v) \in \mathcal{R}$, si usa la notazione $u\mathcal{R}v$ o $\mathcal{R}(u, v)$.

Definizione 1.2. L' *insieme immagine* di un elemento x dell'insieme A attraverso la relazione $\mathcal{R}$ è l'insieme $\mathcal{R}(x) = \{y \in B \mid x\mathcal{R}y\}$.

Alcune relazioni binarie dispongono di una o più delle seguenti caratteristiche:

Definizione 1.3. Sia $\mathcal{R}$ una relazione binaria su A. Siano x, y, z qualsiasi appartenenti ad A. Allora $\mathcal{R}$ è:

- *Riflessiva* se $x\mathcal{R}x$;
- *Simmetrica* se $x\mathcal{R}y \implies y\mathcal{R}x$;
- *Transitiva* se $(x\mathcal{R}y \wedge y\mathcal{R}z) \implies x\mathcal{R}z$.

Esempio 1.1. La relazione "$\leq$" su $\mathbb{N}$ è riflessiva e transitiva, ma non simmetrica. La relazione "=" ($a = b \iff$ "a, b sono lo stesso numero") su $\mathbb{N}$ è simmetrica, riflessiva e transitiva.

Definizione 1.4. Una *relazione di equivalenza* su un insieme A è una relazione binaria riflessiva, simmetrica e transitiva. Si vede facilmente che questo genere di relazione partiziona A in *classi di equivalenza*, ovvero sottoinsiemi disgiunti di A all'interno dei quali tutte le coppie di elementi sono in relazione.

Data una relazione di equivalenza $\mathcal{R}$ su un insieme A, si usa la notazione "$[a]_{\mathcal{R}}$" per indicare la classe di equivalenza di $\mathcal{R}$ cui appartiene $\mathcal{R}$. Inoltre si usa la notazione "$A/\mathcal{R}$", letta "*quoziente* di A rispetto a $\mathcal{R}$", per denotare l'insieme delle classi di equivalenza di $\mathcal{R}$ su A.
In alcune situazioni risulta conveniente definire la più piccola relazione (cioè quella che mette in relazione il minor numero possibile di coppie) che dispone di una certa proprietà, e che contiene una relazione binaria di partenza. Una relazione costruita in questo modo è una "*chiusura*":

Definizione 1.5. Sia $\mathcal{R}$ una relazione binaria su A. Le seguenti relazioni sono chiusure di $\mathcal{R}$:

- *Riflessiva*: $\mathcal{R}_r = \mathcal{R} \cup \{(x, x) \mid x \in A\}$;
- *Simmetrica*: $\mathcal{R}_s = \mathcal{R} \cup \{(y, x) \mid x\mathcal{R}y\}$;
- *Transitiva*: $\mathcal{R}_t = \mathcal{R} \cup \{(x, z) \mid \exists y \in A,\, x\mathcal{R}y \wedge y\mathcal{R}z\}$.

Esempio 1.2. La chiusura riflessiva della relazione "$<$" (minore stretto) è la relazione "$\leq$".

Nel seguito useremo ampiamente la definizione seguente:

Definizione 1.6. Sia $\mathcal{R}$ una relazione binaria su $A \times B$. La *contro-immagine* di un elemento $y \in B$ rispetto ad $\mathcal{R}$ è l'insieme $\mathcal{R}^{-1}(y) = \{x \in A \mid x\mathcal{R}y\}$.
Più in generale, la *funzione inversa* di R è la funzione $\mathcal{R}^{-1} : \mathcal{P}(B) \to \mathcal{P}(A)$ (dove "$\mathcal{P}$" denota l'insieme delle parti) che associa ad un sottoinsieme di B tutti gli $x \in A$ tali che vale $x\mathcal{R}y$ per almeno un y del sottoinsieme.

Adottiamo infine la notazione "$|A|$" per indicare la *cardinalità* dell'insieme A. In modo analogo, data una relazione binaria $\mathcal{R}$, $|\mathcal{R}|$ è il numero delle coppie messe in relazione da $\mathcal{R}$.

1.2 Grafi

Il problema che intendiamo affrontare in questo elaborato è descritto naturalmente con il formalismo della teoria dei grafi. In questa sezione presentiamo un sottoinsieme minimale di definizioni e risultati che ci consentirà di introdurre e trattare la questione in modo appropriato.

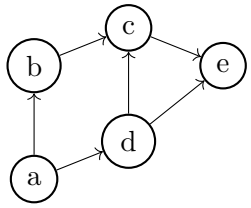


Figura 1: Rappresentazione grafica di un grafo diretto.

1.2.1 Definizione e generalità

Con le premesse viste nella Sezione 1.1 possiamo definire un *grafo* come segue:

Definizione 1.7. Sia V un insieme finito non vuoto. Sia E una relazione binaria su V. La coppia $G = (V, E)$ è un *grafo diretto* (o *orientato*). Con questa notazione:

- V è l'insieme dei *nodi* (o *vertici*);
- E è una relazione binaria (in generale non simmetrica) che mette in relazione alcuni dei nodi di G.

Esempio 1.3. Il grafo di Figura 1 è descritto dalla coppia

- $V = \{a, b, c, d, e\}$
- $E = \{(a, b), (a, d), (b, c), (d, c), (c, e), (d, e)\}$

Nel seguito utilizzeremo ampiamente la seguente terminologia:

Definizione 1.8. Sia $G = (V, E)$ un grafo diretto. Consideriamo un arco $(u, v) \in E$. In questo caso u è la *sorgente* dell'arco, mentre v è la *destinazione*. Se il numero di archi uscenti da un nodo è zero tale nodo è un *pozzo* (dall'inglese *sink*).

Inoltre adotteremo la seguente notazione per indicare che esiste un arco che collega $a, b \in V$ nel grafo $G = (V, E)$: $\langle a, b \rangle$. Se vi è ambiguità specificheremo la relazione binaria E in pedice: $\langle a, b \rangle_E$.

1.2.2 Componenti fortemente connesse

Come abbiamo visto sopra, un grafo diretto è un insieme di elementi (i *nodi*) accoppiato con un insieme di relazioni tra questi elementi (gli *archi*).

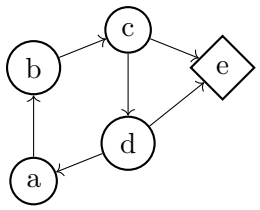


Figura 2: Le SCC del grafo rappresentato sono $\{a, b, c, d\}$ e $\{e\}$.

È naturale associare questo concetto all'idea di percorso: ogni grafo è caratterizzato da un insieme di nodi e da uno di *cammini* che consentono di spostarsi da un nodo ad un altro. La seguente definizione sorge in modo spontaneo da questa interpretazione:

Definizione 1.9. Sia $G = (V, E)$ un grafo diretto. Siano $u, v \in V$. Diciamo che v *è raggiungibile* da u, o in alternativa *esiste un cammino* da u a v, o ancora $\langle u, v \rangle_{E^*}$, se esiste una sequenza finita di nodi $\{x_n\}_{n \in \{0,\dots,K\}}$, tale che $x_0 = u, x_K = v, \langle x_n, x_{n+1} \rangle_E$.

L'esistenza di un cammino tra nodi fornisce un criterio immediato per partizionare un grafo in gruppi di nodi. Diamo innanzitutto la seguente definizione:

Definizione 1.10. Un grafo diretto (V, E) è *fortemente connesso* se per ogni coppia di nodi $v_1, v_2 \in V$ esiste un cammino da v_1 a v_2, cioè $\langle v_1, v_2 \rangle_{E^*}$.

Possiamo individuare i sottografi massimali (cioè quella ripartizione del grafo in sottografi aventi una qualche caratteristica che consente di minimizzare il numero di sottografi) fortemente connessi ([3, Appendice B]):

Definizione 1.11. Le *componenti fortemente connesse* (*strongly connected components*, *SCC*) di un grafo diretto sono i sottografi che compongono la ripartizione (massimale) del grafo in sottografi fortemente connessi.

Esempio 1.4. Nel grafo di Figura 2 le SCC sono rappresentate con forme diverse: $\{a, b, c, d\}, \{e\}$.

Il partizionamento dei nodi in SCC è definito come segue:

Definizione 1.12. Sia $G = (V, E)$ un grafo diretto. Il grafo $G^{\text{SCC}} = (V^{\text{SCC}}, E^{\text{SCC}})$, dove:

– $V^{\text{SCC}} = \{C \mid C \text{ è una SCC}\}$;

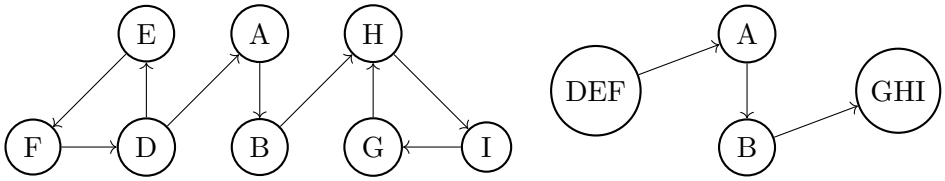
– $E^{\text{SCC}} = \{(A, B) \in V^{\text{SCC}} \times V^{\text{SCC}} \mid A \neq B, \exists m \in A, n \in B, \langle m, n \rangle_E\}$

è il partizionamento del grafo iniziale in SCC.

Riportiamo la seguente proprietà immediata:

Proposizione 1.1. *Sia G^{SCC} il grafo delle SCC di un grafo G generico. Allora G^{SCC} è aciclico.*

Dimostrazione. Supponiamo per assurdo che in G^{SCC} (i cui nodi sono i sottografi massimali che compongono la partizione delle SCC) esista un ciclo. Allora tutti i nodi di V^{SCC} facenti parte del ciclo sono mutuamente raggiungibili (percorrendo il ciclo). Quindi tutti i nodi fanno parte della stessa SCC, ma questo è assurdo. □



(a) Un grafo diretto (b) Grafo delle SCC

Figura 3: Un grafo diretto composto dai nodi $A, B, C, D, E, F, G, H, I$, che possono essere raggruppati nelle SCC $(DEF), (A), (B), (GHI)$.

Esempio 1.5. Nella Figura 3a è riportato un grafo diretto generico, nella Figura 3b si è rappresentato il corrispondente grafo delle componenti fortemente connesse.

Osserviamo infine che la ripartizione in componenti fortemente connesse di un grafo è sempre unica. Per dimostrarlo sfruttiamo il seguente risultato preliminare:

Lemma 1.1. *La relazione binaria $\widehat{E}$ che associa le coppie di nodi mutuamente raggiungibili (si osservi che vale $\widehat{E} \subseteq E^*$) è simmetrica e transitiva.*

Dimostrazione. La simmetria è banale, la transitività discende dal fatto che se $a\widehat{E}b, b\widehat{E}c$, allora è possibile raggiungere c passando per b partendo da a, e facendo lo stesso percorso al contrario si dimostra che $a\widehat{E}c$. □

Corollario 1.1. *La ripartizione in SCC di un grafo è unica.*

Dimostrazione. Supponiamo per assurdo che sia possibile "scegliere" se mettere a, b oppure b, c nella stessa SCC, e che questo generi una ripartizione differente. Ma allora si ha anche $a\widehat{E}c$, e quindi un'unico partizionamento possibile. □

Dato un grafo diretto generico possiamo determinare la partizione delle componenti fortemente connesse utilizzando alcuni algoritmi a complessità lineare [19] [20], che non verranno esposti in questo elaborato.

1.2.3 Visita in profondità

Vi sono due metodologie prevalenti per l'esplorazione di un grafo, *visita in ampiezza* (*Breadth-First-Search*, abbreviato in BFS) e *visita in profondità* (*Depth-First-Search*, abbreviato in DFS) [3]; in questo lavoro utilizzeremo prevalentemente la seconda.

Come dice il nome, questa tecnica consiste nella visita esaustiva di tutti i figli di un nodo, prima di passare agli altri nodi sullo stesso livello.

Riportiamo lo pseudocodice, leggermente modificato:

Algoritmo 1: DFS

```
   Data: G = (V, E)
 1 function dfs-visit(G = (V, E), n, time):
 2 |   n.color = GRAY;
 3 |   forall m | (⟨n, m⟩ ∧ m.color = WHITE) do
 4 |   |   time = dfs-visit(G, m, time);
 5 |   n.finishing-time = time;
 6 |   n.color = BLACK;
 7 |   time = time+1;
 8 |   return time;
 9 begin
10 |   forall n ∈ V do
11 |   |   n.color = WHITE;
12 |   time = 0;
13 |   while ∃n ∈ V | n.color == WHITE do
14 |   |   time = dfs-visit(G, n, time);
```

L'attributo *color* dei nodi del grafo consente di distinguere tra quei nodi che sono già stati visitati in modo esauriente (colore nero), quelli per cui è in corso una visita in profondità (colore grigio) e quelli non ancora esaminati dall'algoritmo (colore bianco). Si osservi che la colorazione dei nodi non è solamente accessoria, ma fornisce un meccanismo per evitare un loop infinito nel caso in cui il grafo contenga un ciclo.

Oltre alla colorazione dei nodi, l'algoritmo imposta l'attributo *finishing-time* per ogni nodo, ovvero l'istante di tempo (a partire da $time = 0$) in cui è stata terminata la visita esaustiva in profondità per tale nodo.

Nello pseudocodice completo (di cui abbiamo fornito un estratto funzionante) vengono considerati altri due attributi: *starting-time* e *parent*. Questi ultimi tre attributi conferiscono alcune importanti proprietà all'algoritmo di visita in profondità, che non saranno approfondite in questo lavoro (si vedano *Parenthesis theorem*, *White-path theorem* [3]).

L'attributo *finishing-time* è sufficiente per alcune applicazioni che saranno esposte nelle sezioni seguenti.

1.3 Insiemi

1.3.1 Cenni di teoria degli insiemi

In generale supporremo validi gli assiomi su cui si fonda la teoria degli insiemi ZFC, ad eccezione dell'Assioma di Fondazione. Diamo innanzitutto una formulazione degli assiomi che interverranno nel seguito del lavoro:

Assioma 1.1 (di estensionalità)**.** Due insiemi sono uguali se e solo se contengono gli stessi elementi.

Assioma 1.2 (di fondazione)**.** Ogni insieme non vuoto contiene un elemento disgiunto dall'insieme stesso.

Del primo faremo un uso esplicito nel seguito. Il secondo, per motivi che saranno evidenti nella sezione seguente, risulta limitante nell'ambito che intendiamo trattare.

Vale la seguente definizione:

Definizione 1.13. Un insieme è *ben-fondato* se non contiene se stesso. Altrimenti è *non-ben-fondato*.

Esempio 1.6. L'insieme $\Omega = \{\Omega\}$ è non-ben-fondato. L'insieme $A = \{1, 2, 3\}$ è ben-fondato.

Riportiamo una formulazione equivalente dell'Assioma 1.2 [12, Chapter III.4]:

Assioma (1.2 bis)**.** $\forall A$ la relazione "$\in$" è ben-fondata su A.

Da questa formulazione risulta evidente l'impossibilità, nel sistema di assiomi ZFC, di costruire insiemi non-ben-fondati.

Rinunciando all'Assioma 1.2 si ottiene un sistema di assiomi che ammette l'esistenza di insiemi non-ben-fondati; tuttavia si può verificare che questo sistema non è più sufficiente a descrivere in modo esaustivo l'aritmetica per mezzo di operazioni su insiemi [1].

Per ovviare a questa mancanza si introduce l'Assioma AFA, che verrà presentato e discusso nella sezione seguente.

1.3.2 Rappresentazione di insiemi tramite grafi diretti

In alcuni casi risulta conveniente fornire un'interpretazione insiemistica della nozione di grafo vista sopra. Introduciamo innanzitutto un concetto fondamentale per il seguito del lavoro:

Definizione 1.14. Sia $G = (V, E)$ un grafo diretto. Supponiamo che esista un $u \in V$ tale che ogni vertice del grafo è raggiungibile da u. Allora la coppia (G, u) è un *accessible pointed graph*, o *APG*.

Per rappresentare un insieme tramite un grafo diretto è necessario passare per un procedimento denominato *decorazione*:

Definizione 1.15. La *decorazione* di un APG è l'assegnazione di un insieme ad ogni suo nodo. In tal caso viene associata la relazione "$\in$" alla relazione E, ovvero $\langle a, b\rangle \iff b \in a$.

Possiamo dare la seguente definizione:

Definizione 1.16. L'*immagine* (o *picture*) di un insieme A è la coppia composta da un APG (G, v) e da una sua decorazione in cui a v è associato lo stesso A [1].

Vale la seguente proposizione [1]:

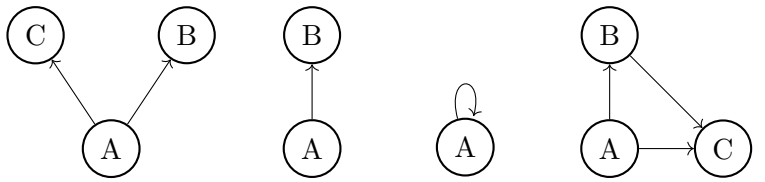
Proposizione 1.2. *Ad un APG aciclico è possibile associare un'unica decorazione.*

Questo risultato non è stato dimostrato nel caso di un APG contenente almeno un ciclo. Per questo motivo viene introdotto il seguente assioma:

Assioma 1.3 (AFA, Anti-Foundation-Axiom)**.** Ogni APG possiede un'unica decorazione.

L'assioma AFA ha un'ovvia conseguenza:

Corollario 1.2. *Ogni APG è immagine di un unico insieme.*



(a) $B = \{\emptyset\}$ (b) $B = \{\emptyset\}$ (c) $A = \{A\}$ (d) $A = \{\emptyset, \{\emptyset\}\}$

Figura 4: Rappresentazione di insiemi tramite grafi.

Esempio 1.7. In Figura 4 sono rappresentati alcuni insiemi sotto forma di APG. Nella Figura 4a B, C non hanno nodi figli, sicchè rappresentano l'insieme vuoto, e dunque A è l'insieme che contiene solamente l'insieme vuoto; analogamente si interpreta la Figura 4b; nella Figura 4c abbiamo un nodo A il cui unico figlio è lo stesso A, sicchè contiene solamente se stesso; infine nella Figura 4d abbiamo un nodo C che rappresenta l'insieme vuoto, un nodo B che contiene solamente l'insieme vuoto, ed un nodo A che contiene l'insieme vuoto è l'insieme rappresentato da B. Associazioni del tipo "B rappresenta l'insieme vuoto" sono *decorazioni*.

1.4 Bisimulazione

In questa sezione introdurremo la definizione di *bisimulazione* ed alcune proprietà. Proseguiremo con la definizione di *massima bisimulazione*, ed infine proporremo alcune osservazioni che legano la bisimulazione e la teoria degli insiemi.

1.4.1 Definizione e risultati preliminari

Definizione 1.17. Siano $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ due grafi diretti. Una relazione binaria $\mathcal{R} : V_1 \times V_2$ è una *bisimulazione* su G_1, G_2 se $\forall a \in V_1, b \in V_2$ valgono congiuntamente le seguenti proprietà:

- $a\mathcal{R}b, \langle a, a' \rangle_{E_1} \implies \exists b' \in V_2 \mid a'\mathcal{R}b' \wedge \langle b, b' \rangle_{E_2}$
- $a\mathcal{R}b, \langle b, b' \rangle_{E_2} \implies \exists a' \in V_1 \mid a'\mathcal{R}b' \wedge \langle a, a' \rangle_{E_1}$

Analogamente si definisce una bisimulazione su un unico grafo diretto G, ponendo $G_1 = G_2 = G$.

Nei prossimi paragrafi osserveremo che l'esistenza di una bisimulazione tra due grafi è un'informazione rilevante se siamo interessati agli insiemi che rappresentano.

Definizione 1.18. Siano $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ due grafi. Essi sono *bisimili* se esiste una bisimulazione su G_1, G_2. Due APG $(G_1, v_1), (G_2, v_2)$ sono *bisimili* se G_1, G_2 sono bisimili e vale $v_1 \mathcal{R} v_2$ per almeno una bisimulazione $\mathcal{R}$ su G_1, G_2.

Osservazione 1.1. Una bisimulazione può non essere riflessiva, simmetrica, nè transitiva.

Esempio 1.8. La relazione $aRb \iff$ "a, b sono lo stesso nodo" su un grafo qualsiasi è una bisimulazione riflessiva, simmetrica e transitiva.

La relazione $R = \emptyset$ è una bisimulazione su un grafo qualsiasi, ma non è riflessiva.

La relazione $R = \{(a,a), (b,b), (c,c), (d,d), (a,b), (b,c), (c,d)\}$ sul grafo $G = (\{a,b,c,d\}, \{(a,b), (b,c), (c,d), (d,d)\})$ è una bisimulazione, ed è solamente riflessiva.

Dalla definizione di bisimulazione possiamo dedurre una proprietà interessante delle chiusure:

Teorema 1.1. *Sia $\mathcal{R}$ una bisimulazione sul grafo diretto G. La sua chiusura riflessiva, simmetrica o transitiva è ancora una bisimulazione su G.*

Dimostrazione. Consideriamo separatamente le tre relazioni $\mathcal{R}_r, \mathcal{R}_s, \mathcal{R}_t$, rispettivamente la chiusura riflessiva, simmetrica e transitiva. Per ogni caso consideriamo solamente le coppie aggiunte dalla chiusura:

- $\mathcal{R}_r$: Per definizione $\mathcal{R} \subseteq \mathcal{R}_r$, quindi è sufficiente dimostrare che $\mathcal{R}_r$ è una bisimulazione quando gli argomenti $u, v \in V$ non sono distinti.

  Sia $u \in V$. Chiaramente per definizione di $\mathcal{R}_r$ si ha $u \mathcal{R}_r u$. Ma $\forall u' \in V \mid \langle u, u' \rangle$ vale (sempre per definizione di $\mathcal{R}_r$) $u' \mathcal{R}_r u'$.

- $\mathcal{R}_s$: Per definizione $\mathcal{R} \subset \mathcal{R}_s$, quindi è sufficiente dimostrare che $\mathcal{R}_s$ è una bisimulazione quando per $u, v \in V$ si ha $u \mathcal{R} v$ ma non $v \mathcal{R} u$.

  Sia $(u, v) \in V \times V$. Allora:

  $$u \mathcal{R}_s v \implies u \mathcal{R} v \vee v \mathcal{R} u$$

  Supponiamo ad esempio che $v \mathcal{R} u$.

  $$\begin{aligned} &\implies \forall v' \in V \mid \langle v, v' \rangle \; \exists u' \in V, \langle u, u' \rangle \wedge v' \mathcal{R} u' \\ &\implies u' \mathcal{R}_s v' \end{aligned}$$

e

$$\implies \forall u' \in V \mid \langle u, u' \rangle \; \exists v' \in V, \langle v, v' \rangle \wedge v' \mathcal{R} u'$$
$$\implies u' \mathcal{R}_s v'$$

cioè sono dimostrate le due condizioni caratteristiche della bisimulazione.
La dimostrazione è analoga se $u \mathcal{R} v$.

– $\mathcal{R}_t$: Per definizione $b \subset \mathcal{R}_t$, quindi è sufficiente dimostrare che $\mathcal{R}_t$ è una bisimulazione quando per gli argomenti $u, v, z \in V$ si ha $u \mathcal{R} v$, $v \mathcal{R} z$ ma non $u \mathcal{R} z$.
Sia $(u, v, z) \in V^3$ una terna con questa proprietà. Allora:

$$\forall u' \in V \mid \langle u, u' \rangle \; \exists v' \in V, \langle v, v' \rangle \wedge u' \mathcal{R} v'$$

Inoltre $\exists z' \mid \langle z, z' \rangle \wedge v' \mathcal{R} z'$.
Riordinando si ha $u' \mathcal{R} v', v' \mathcal{R} z'$. Allora per definizione di b_t, $u' \mathcal{R}_t z'$.
In modo speculare si ottiene la seconda condizione caratteristica della bisimulazione. □

Da questa proposizione si deduce il seguente corollario, che risulta dall'applicazione iterata delle tre chiusure viste in precedenza:

Corollario 1.3. *Ad ogni bisimulazione $\mathcal{R}$ è possibile associare una bisimulazione $\widetilde{\mathcal{R}} \supseteq \mathcal{R}$ che è anche una relazione di equivalenza (Definizione* 1.4*).*

Concludiamo la sezione relativa ai risultati generali sulla bisimulazione con la seguente proposizione, che sarà utile nel seguito:

Proposizione 1.3. *Siano $\mathcal{R}_1, \mathcal{R}_2$ due bisimulazioni su G_1, G_2. Allora $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ è ancora una bisimulazione.*

Dimostrazione. Siano $u, v \mid u \mathcal{R} v$; allora deve valere $u \mathcal{R}_1 v \vee u \mathcal{R}_2 v$. Sia $u' \mid \langle u, u' \rangle$. Ma quindi $\exists v' \mid \langle v, v' \rangle \wedge u' \mathcal{R}_{1|2} v'$. □

1.4.2 Massima bisimulazione

Definiamo ora il concetto di *massima bisimulazione*, che sarà l'argomento principale di questo elaborato:

Definizione 1.19. Una bisimulazione $\mathcal{R}_M$ su G_1, G_2 è la *massima bisimulazione* della coppia di grafi G_1, G_2 se per ogni bisimulazione $\mathcal{R}$ su G_1, G_2 si ha $u\mathcal{R}v \implies u\mathcal{R}_M v$, o equivalentemente $\mathcal{R}_M \supseteq \mathcal{R}$ per ogni bisimulazione $\mathcal{R}$ su G_1, G_2.

Osservazione 1.2. Sia $\mathcal{R}_M$ la massima bisimulazione su un grafo diretto G. Allora per ogni bisimulazione $\mathcal{R}$ su G si ha $|\mathcal{R}_M| \geq |\mathcal{R}|$.

Naturalmente la massima bisimulazione dipende dai due grafi presi in esame. Deduciamo alcune caratteristiche immediate:

Proposizione 1.4. *Valgono le seguenti proprietà:*

1. *La massima bisimulazione su due grafi G_1, G_2 è unica;*

2. *La massima bisimulazione è una relazione di equivalenza.*

Dimostrazione. Le proprietà seguono dal Corollario 1.3 e dall'Osservazione 1.3:

1. Supponiamo per assurdo che esistano due massime bisimulazioni $\mathcal{R}_{M_1}, \mathcal{R}_{M_2}$. La loro unione è ancora una bisimulazione per l'Osservazione 1.3, e vale naturalmente $|\mathcal{R}_{M_1} \cup \mathcal{R}_{M_2}| > |\mathcal{R}_{M_1}|, |\mathcal{R}_{M_2}|$ (se così non fosse le due bisimulazioni coinciderebbero).

2. Se per assurdo la supposta massima bisimulazione non fosse una relazione di equivalenza, potremmo considerare la sua chiusura riflessiva, simmetrica e transitiva. □

Naturalmente il concetto di *massima bisimulazione* può essere definito anche su unico grafo diretto G. Questo caso si rivelerà di grande interesse nel seguito. Per ora dimostriamo il seguente risultato:

Teorema 1.2. *Sia $G = (V, E)$ un grafo diretto finito. Allora esiste la massima bisimulazione su G.*

Dimostrazione. Può esistere solamente un numero finito di relazioni binarie su G, e questo numero fornisce un limite superiore al numero massimo di bisimulazioni su G. Allora possiamo considerare l'unione di questo numero finito di bisimulazioni, che sarà chiaramente la massima bisimulazione.

Abbiamo almeno una bisimulazione non banale (ovvero diversa da $\emptyset$) che possiamo ricavare dalla chiusura riflessiva di $\emptyset$ (Teorema 1.1), ovvero la relazione $\mathcal{R} = \{(u, u) \mid u \in V\}$. □

Con queste premesse possiamo determinare in modo agevole la massima bisimulazione:

Teorema 1.3. *La massima bisimulazione coincide con la relazione binaria che mette in relazione tutte le coppie di nodi bisimili.*

Dimostrazione. Si osservi che la relazione considerata può essere scritta come segue:

$$\mathcal{R} = \bigcup_{\substack{\mathcal{B} \text{ è una} \\ \text{bisimulazione}}} \mathcal{B}$$

Per l'Osservazione 1.3 la relazione $\mathcal{R}$ è ancora una bisimulazione, e chiaramente contiene tutte le possibili bisimulazioni del grafo. Ciò significa che è proprio la massima bisimulazione. □

1.4.3 Interpretazione insiemistica della bisimulazione

Mostriamo ora una conseguenza diretta dell'Assioma di Estensionalità e di AFA (Sezione 1.3.2):

Teorema 1.4. *Due APG rappresentano lo stesso insieme se e solo se sono bisimili.*

Dimostrazione. Siano $G_A = (V_A, E_A), G_B = (V_B, E_B)$. Dimostriamo separatamente le due implicazioni:

($\implies$) Osserviamo innanzitutto che la relazione binaria $\equiv : V_A \times V_B$ definita come:

$a \equiv b \iff$ "le decorazioni di G_A, G_B associano ad a, b lo stesso insieme"

è una bisimulazione sui grafi G_A, G_B.
Chiaramente se $a \equiv b$ e $\langle a, a' \rangle$ si ha:

- $a' \in a$, associando ad a, a' gli insiemi che rappresentano secondo la decorazione (unica per AFA);
- a, b rappresentano lo stesso insieme.

Quindi per l'Assioma di Estensionalità $\exists b' \in b \mid b', a'$ rappresentano lo stesso insieme, cioè $\langle b, b' \rangle$ e $a' \equiv b'$. Si procede specularmente per la seconda condizione caratteristica della bisimulazione.
La relazione "$\equiv$" è quindi una bisimulazione sugli APG G_A, G_B quando si assume per ipotesi che rappresentino lo stesso insieme.

($\Longleftarrow$) Sia $\mathcal{R}$ una bisimulazione su G_A, G_B. Consideriamo la decorazione d_A (l'unica) di G_A. Vogliamo definire una decorazione per G_B, e dimostrare che i due grafi con le rispettive decorazioni sono due immagini dello stesso insieme. Dalla possibilità di operare questo procedimento, dall'Assioma di Estensionalità e da AFA potremo dedurre l'uguaglianza degli insiemi rappresentati.

Definiamo la decorazione d_B come segue:

$$d_B(v) = d_A(u), \text{con } u \text{ nodo di } G_A \mid u\mathcal{R}v$$

Osservazione. Per ogni nodo v di B deve esistere almeno un nodo u di $G_A \mid u\mathcal{R}v$ perchè si suppone che i due APG siano bisimili.

Dimostriamo che d_B è una decorazione di G_B. In altre parole vogliamo dimostrare che per ogni nodo v di G_B l'insieme $d_B(v)$ contiene tutti e soli gli insiemi $d_B(v')$ dove v' è figlio di v.

- Supponiamo per assurdo che tra i figli di v "manchi" il nodo corrispondente ad un elemento $X \in d_B(v)$. Poichè la decorazione di G_A è ben definita, il nodo u di $G_A \mid u\mathcal{R}v$ deve avere un figlio corrispondente a X, che chiameremo u', per cui vale quindi $d_A(u') = X$. Ma $u\mathcal{R}b \wedge \langle u, u' \rangle \implies \exists v' \mid \langle v, v' \rangle, u'\mathcal{R}v'$. Cioè $d_B(v') = d_A(u') = X$. Dunque il nodo mancante è stato trovato.
- Supponiamo per assurdo che tra i figli di v ci sia un nodo "in più", ovvero un nodo $v' \mid \langle v, v' \rangle$ e $d_B(v') = Y$ con $Y \notin d(v)$. Ma allora, considerando un nodo u di $G_A \mid u\mathcal{R}v$, dovrebbe esistere un $u' \mid \langle u, u' \rangle$ e $u'\mathcal{R}v'$, cioè $d_A(u') = d_B(v') = Y$. Ma allora $Y \in d_A(u) = d_B(v)$, deduzione che è chiaramente in contrasto con l'ipotesi.

È possibile che ci siano due nodi di a_1, a_2 di G_A ed un nodo b di G_B per cui vale $a_1\mathcal{R}b \wedge a_2\mathcal{R}b$. In questo caso la decorazione definita è ambigua. Per questo motivo correggiamo la formulazione di d_B come segue:

$$d_B(v) = X, \text{ dove } X \text{ è l'insieme associato al nodo } u \text{ di } A \mid u\mathcal{R}v,$$
$$\text{con } u \text{ preso casualmente tra i nodi di } G_A \text{ bisimili a } v.$$

Per AFA esiste un'unica decorazione di G_B, quindi si deve avere, alternativamente, per ogni nodo v di G_B:

– $\exists! u \in V_A \mid u \mathcal{R} v$;

– $\forall u \in V_A \mid u \mathcal{R} v$, la decorazione di G_A associa a tutti gli u lo stesso insieme.

Dunque l'ambiguità è risolta. □

Tenendo conto di quanto affermato nella sezione 1.3.2, il Teorema 1.4 dimostra che la bisimulazione può sostituire la relazione di uguaglianza tra insiemi quando questi sono rappresentati con APG [4].

Dopo questa considerazone possiamo dare la seguente definizione:

Definizione 1.20. Sia $\mathcal{R}$ una bisimulazione su $G = (V, E)$, dove G è un grafo diretto. Supponiamo che $\mathcal{R}$ sia una relazione di equivalenza. Il grafo, definito a partire dal grafo iniziale, ottenuto con una *contrazione rispetto alla bisimulazione* $\mathcal{R}$ [5], è il grafo diretto $G_{\mathcal{R}} = (V_{\mathcal{R}}, E_{\mathcal{R}})$:

– $V_{\mathcal{R}} = \{[m]_{\mathcal{R}}, m \in V\}$;

– $\langle [m]_{\mathcal{R}}, [n]_{\mathcal{R}} \rangle \in E_{\mathcal{R}} \iff \exists b \in [m]_{\mathcal{R}}, c \in [n]_{\mathcal{R}} \mid \langle b, c \rangle \in E$.

Si chiama *classe* del nodo a rispetto alla bisimulazione $\mathcal{R}$, indicata con la notazione $[a]_{\mathcal{R}}$, il nodo di $V_{\mathcal{R}}$ in cui viene inserito il nodo a.

La Definizione 1.20 è di fondamentale importanza per la seguente osservazione:

Proposizione 1.5. *Sia G un grafo diretto, e sia $G_{\mathcal{R}}$ come nella Definizione 1.20, per una bisimulazione R qualsiasi. Allora $G, G_{\mathcal{R}}$ sono bisimili.*

Dimostrazione. Sia "$\equiv$" la relazione binaria su $V \times V_{\mathcal{R}}$ definita come segue:

$$m \equiv M \iff M = [m]_{\mathcal{R}}$$

Vogliamo dimostrare che tale relazione è una bisimulazione sui grafi $G, G_{\mathcal{R}}$. Supponiamo che $x \equiv X$, e che $\langle x, y \rangle \in E$ per qualche $y \in V$. Chiamiamo $Y \coloneqq [y]_{\mathcal{R}}$. Allora, per la Definizione 1.20, si ha $\langle X, Y \rangle \in E_{\mathcal{R}}$. Inoltre vale banalmente $y \equiv Y$.

Per dimostrare la seconda condizione caratteristica della bisimulazione, supponiamo che $x \equiv X$, e che $\langle X, Y \rangle \in E_{\mathcal{R}}$ per qualche $Y \in V_{\mathcal{R}}$. Sempre per la Definizione 1.20 deve esistere un $y \in Y \mid (y \equiv Y \wedge \langle x, y \rangle \in E)$. □

La Proposizione 1.5 ha una conseguenza ovvia, che risulta evidente dal Teorema 1.4:

Corollario 1.4. *Sia R una bisimulazione che sia anche una relazione di equivalenza. Allora l'APG (G, v) e l'APG $(G_R, [v]_R)$ rappresentano lo stesso insieme.*

Possiamo quindi sfruttare le proprietà della bisimulazione per rappresentare insiemi in una forma minimizzata: è sufficiente ricavare una bisimulazione (che sia anche una relazione di equivalenza) sul grafo che rappresenta l'insieme.

Poichè in generale un grafo ammette più di una bisimulazione (che sia anche una relazione di equivalenza), definiamo una relazione d'ordine sulle rappresentazioni:

Definizione 1.21. La rappresentazione (G_a, v_a) di un insieme è *minore* della rappresentazione equivalente (G_b, v_b) se $|Va| < |Vb|$.

Una rappresentazione è *minima* se non esiste una rappresentazione equivalente minore.

Osservazione 1.3. La *contrazione per bisimulazione* è una rappresentazione minore (o eventualmente uguale) di quella iniziale.

Concludiamo la sezione con il seguente risultato, che stabilisce in modo univoco la bisimulazione prescelta per minimizzare la rappresentazione di un dato insieme:

Teorema 1.5. *Sia (G, v) un APG rappresentante un insieme. Sia $\mathcal{R}_M$ la massima bisimulazione su (G, v). Allora la contrazione per bisimulazione indotta da $\mathcal{R}_M$ su (G, v) fornisce la rappresentazione minima dell'insieme.*

Dimostrazione. Supponiamo per assurdo che esista una bisimulazione $\mathcal{R}_V$ su (G, v) che fornisce una contrazione avente un numero di nodi strettamente inferiore alla contrazione indotta da $\mathcal{R}_M$. Ma questo implica che esistono almeno due nodi di G che sono in relazione secondo $\mathcal{R}_V$ e non secondo $\mathcal{R}_M$. Chiaramente questa deduzione è in contrasto con il fatto che $\mathcal{R}_M$ è la massima bisimulazione.

Supponiamo per assurdo che, dopo la contrazione indotta da $\mathcal{R}_M$, sia possibile trovare una nuova bisimulazione $\mathcal{R}_O$ su $(G_{\mathcal{R}_M}, [v]_{\mathcal{R}_M})$ che induca una contrazione avente un numero di nodi strettamente inferiore a quello di $(G_{\mathcal{R}_M}, [v]_{\mathcal{R}_M})$. Chiaramente $\mathcal{R}_O \subset V_{\mathcal{R}_M} \times V_{\mathcal{R}_M}$.

Definiamo una nuova bisimulazione $\mathcal{R}_{\widetilde{M}} \subset V \times V$ tale che

$$x\mathcal{R}_{\widetilde{M}}y \iff (x\mathcal{R}_M y \vee [x]_{\mathcal{R}_M}\mathcal{R}_O[y]_{\mathcal{R}_M})$$

Per definizione di massima bisimulazione è necessario che valga $\mathcal{R}_{\widetilde{M}} \subset \mathcal{R}_M$, quindi non è possibile che la contrazione indotta da $\mathcal{R}_O$ sia una rappresentazione minore di quella indotta da $\mathcal{R}_M$. □

1.4.4 Applicazioni

Abbiamo esposto le definizioni di base ed introdotto le nozioni di *bisimulazione* e *massima bisimulazione*, per cui riteniamo che sia interessante analizzare alcune applicazioni, prima di entrare nel merito del problema algoritmico.

È risaputo che molti sistemi possono essere modellizzati efficacemente utilizzando grafi di complessità arbitraria. Si pensi ad esempio alla rete di contatti tra utenti di un *social network* [6], di cui i grafi sono la rappresentazione più naturale, o l'insieme delle relazioni all'interno di un qualsiasi tipo di rete logistica [22] o elettronica [8] ("se il componente b è guasto, allora il componente a è guasto").

Si osservi inoltre la potenza del concetto di *etichetta* (in inglese *label*): se ad esempio volessimo classificare gli utenti di un *social network*, potremmo attribuire alle aziende la *label* di "*seller*", agli utenti l'etichetta di "*user*", e ad alcuni utenti particolari (che in gergo si dicono *influencer*), le cui opinioni hanno un'ampia risonanza sulla platea degli *user*, l'etichetta di "*influencer*". I risultati che otterremmo applicando uno qualsiasi degli algoritmi visti su un grafo di questo genere sarebbero sicuramente molto più profondi ed interessanti rispetto al caso *unlabeled*, in quanto abbiamo diversificato il dataset in base ad informazioni che già avevamo, comunicando all'algoritmo il tipo di classificazione a cui eravamo interessati.

La bisimulazione può essere applicata efficacemente per indicizzare dati semi-strutturati, ovvero dati organizzati in gerarchie di oggetti in cui alcuni attributi possono eventualmente essere assenti [14]. In generale saremmo costretti, per risolvere una query, ad esplorare tutti i livelli della gerarchia fino a trovare quella sotto-gerarchia di oggetti che la soddisfa. Utilizzando la bisimulazione possiamo raggruppare i dati in classi di equivalenza, e creare quindi una sorta di indice (chiamato *1-index*); in base alle caratteristiche della query possiamo scegliere la classe di equivalenza in cui sappiamo che devono risidere i possibili risultati della ricerca, e restringere quindi la parte di verifica del "matching" ai soli membri della classe selezionata, riducendo di molto il tempo richiesto dalla procedura.

È possibile applicare la bisimulazione anche nell'ambito della *concurrency theory*, ovvero la teoria che studia l'interazione tra processi software che vengono eseguiti in parallelo. Possiamo costruire un modello semplificato di un sistema appartenente a questa categoria tramite la teoria dei grafi. In questo ambito è vitale ridurre problemi come il *deadlock*, una situazione in cui una risorsa "bloccata" da un processo non viene mai rilasciata, per cui altri processi in attesa della stessa risorsa si bloccano per un tempo

indefinito [16]. Analisi di questo tipo risultano estremamente complesse, soprattutto quando il software diventa molto complesso ed articolato. Per questo motivo esistono degli strumenti, chiamati *concurrency workbench*, per cercare problemi nell'interazione tra processi [2], che sfruttano ampiamente diverse nozioni di equivalenza, come appunto la bisimulazione ed alcune sue varianti [10].

1.5 Relational stable coarsest partition

In questa sezione introdurremo il concetto di *Relational Stable Coarsest Partition* (abbreviato in RSCP). Nel seguito del lavoro evidenzieremo il legame tra questo problema e quello della determinazione della massima bisimulazione, sfruttato ampiamente dagli algoritmi che tratteremo nel seguito in quanto la formulazione del primo lo rende un problema più facilmente approcciabile dal punto di vista algoritmico.

Cominciamo con alcune definizioni di base:

Definizione 1.22. Sia S un insieme finito. Sia $\mathfrak{X} = \{x_1, \ldots, x_n\}$ con $x_i \subseteq S \ \forall i \in \{1, \ldots, n\}$. $\mathfrak{X}$ è una *partizione* di S se:

$$\bigcup_{i=1}^{n} x_i = A \qquad \wedge \qquad x_i \cap x_j = \emptyset \ \forall i, j \in \{1, \ldots, n\} \qquad \wedge \qquad x_i \neq \emptyset \ \forall i$$

Inoltre gli insiemi x_i sono i *blocchi* della partizione $\mathfrak{X}$. Se $a \in x_i$ si usa la notazione $[a]_{\mathfrak{X}} = x_i$.

Osservazione 1.4. Ogni insieme S ha una partizionamento banale, consistente in un unico blocco contenente tutti gli elementi dell'insieme.

Esempio 1.9. Sia $S \coloneqq \{0, 1, 2, 3, 4, 5, 6\}$. Alcuni possibili partizionamenti sono $\{S\}$ (il partizionamento banale), $\mathfrak{S}_1 = \{\{0, 1, 2, 3\}, \{4, 5, 6\}\}, \mathfrak{S}_2 = \{\{0, 1\}, \{2, 3\}, \{4, 5, 6\}\}$.

Definizione 1.23. Siano $\mathfrak{X}_1, \mathfrak{X}_2$ due partizioni dello stesso insieme. $\mathfrak{X}_2$ *rifinisce* $\mathfrak{X}_1$ se $\forall x_2 \in \mathfrak{X}_2, \ \exists x_1 \in \mathfrak{X}_1 \mid x_2 \subseteq x_1$.

Esempio 1.10. Riprendendo l'Esempio 1.9, la partizione $\mathfrak{S}_2$ rifinisce $\mathfrak{S}_1$, che a sua volta rifinisce la partizione banale.

Il partizionamento di insiemi finiti assume interesse nell'ambito trattato in questo lavoro quando viene introdotta la seguente condizione:

Definizione 1.24. Sia S un insieme, ed $\mathcal{R}$ una relazione binaria su S. Sia $\mathfrak{X}$ una partizione di S, e sia $B \subseteq S$.

– $\mathfrak{X}$ è *stabile* rispetto alla coppia $(B, \mathcal{R})$ se $\forall x \in \mathfrak{X}$ vale $x \subseteq \mathcal{R}^{-1}(B) \vee x_i \cap \mathcal{R}^{-1}(B) = \emptyset$;

– La partizione $\mathfrak{X}$ è *stabile* rispetto ad $\mathcal{R}$ se è stabile rispetto a $(x, \mathcal{R})$ $\forall x \in \mathfrak{X}$.

Esempio 1.11. Consideriamo la relazione binaria $\mathcal{R}$ su S che mette in relazione le coppie $\{(0,1),(1,0),(2,3),(3,2),(4,5),(5,6),(0,4),(1,5)\}$. $\mathfrak{S}_2$ è stabile rispetto a $\mathcal{R}$; $\mathfrak{S}_1$ invece non è stabile rispetto a $\mathcal{R}$.

In altre parole per ogni coppia di blocchi di una partizione stabile si hanno due alternative:

– Tutti gli elementi del primo blocco sono in relazione con almeno un elemento del secondo blocco;

– Nessuno degli elementi del primo blocco è in relazione con qualche elemento del secondo blocco.

Vale il seguente risultato:

Proposizione 1.6. *Sia $\mathfrak{S}$ una partizione qualsiasi di un insieme S, e siano $P, Q \subseteq S$. Sia $\mathcal{R}$ una relazione binaria su S. Supponiamo $\mathfrak{S}$ stabile rispetto alle coppie $(\mathcal{R}, P), (\mathcal{R}, Q)$. Allora:*

1. *Qualsiasi rifinitura di $\mathfrak{S}$ resta stabile rispetto alla coppia $(\mathcal{R}, P)$;*
2. *$\mathfrak{S}$ è stabile rispetto alla coppia $(\mathcal{R}, P \cup Q)$.*

Dimostrazione. Dimostriamo separatamente i due enunciati:

1. Chiaramente se per un blocco qualsiasi $x \in \mathfrak{S}$ vale $x \subseteq \mathcal{R}^{-1}(P) \vee x \cap \mathcal{R}^{-1}(P) = \emptyset$, la stessa relazione vale per qualsiasi sottoinsieme di x;
2. Sia $x \in \mathfrak{S}$. Chiaramente vale $\mathcal{R}^{-1}(P), \mathcal{R}^{-1}(Q) \subseteq \mathcal{R}^{-1}(P \cup Q)$. Allora, se almeno uno tra $\mathcal{R}^{-1}(P), \mathcal{R}^{-1}(Q)$ contiene l'immagine di x si avrà $x \subseteq \mathcal{R}^{-1}(P \cup Q)$, altrimenti $x \cap \mathcal{R}^{-1}(P \cup Q)$ dovrà chiaramente essere vuoto. □

È evidente che l'ultimo punto può essere esteso in modo molto semplice all'unione di più sottoinsiemi di S. In altre parole, se $\mathfrak{S}$ è stabile rispetto alle coppie $(\mathcal{R}, S_i), S_i \subset S, i = 1, \ldots, n$, allora è stabile anche rispetto alla coppia $(\mathcal{R}, \bigcup_{i=1}^{n} S_i)$.

Con queste premesse possiamo definire il problema della determinazione della RSCP:

Definizione 1.25. Sia S un insieme, sia R una relazione binaria su S. Sia $\mathfrak{S}$ una partizione di S. La RSCP($\mathfrak{S}, \mathcal{R}$) di S è la partizione di S stabile rispetto ad $\mathcal{R}$, che rifinisce $\mathfrak{S}$ e che contiene il minor numero di blocchi (per questo motivo si dice che è la *più rozza*).

Si usa la notazione "RSCP($\mathcal{R}$)" se la partizione iniziale è quella banale proposta nella Definizione 1.4.

Esempio 1.12. Riprendendo l'Esempio 1.11, è evidente che la RSCP($\mathcal{R}$) deve avere almeno due blocchi: infatti "6" non è in relazione con nessun elemento, dunque deve essere posto in un blocco a parte. La RSCP è quindi $\{\{0,1,2,3,4,5\},\{6\}\}$.

1.5.1 Insiemi *splitter*, e la funzione `split`

La seguente definizione ha grande importanza pratica per il problema della determinazione della RSCP:

Definizione 1.26. Sia $\mathfrak{S} = \{S_1, \ldots, S_n\}$ una partizione qualsiasi di un insieme finito S. Sia $\mathcal{R} : S \to S$, e sia $A \subset S$. A è uno *splitter* di $\mathfrak{S}$ rispetto a $\mathcal{R}$ se

$$\exists S_x \in \mathfrak{S} \mid \; S_x \cap \mathcal{R}^{-1}(A) \neq \emptyset \; \wedge \; S_x \not\subseteq \mathcal{R}^{-1}(A)$$

ovvero se la partizione non è stabile rispetto alla coppia $(\mathcal{R}, A)$.

In tal caso definiamo la seguente funzione:

$$\texttt{split}(A, \mathfrak{S}) = \mathfrak{S} \cup \{S_x \cap \mathcal{R}^{-1}(A), S_x - \mathcal{R}^{-1}(A) \mid S_x \in \mathfrak{S}\} - \{\emptyset\}$$

Si osservi che la definizione di `split` dipende anche dal parametro "$\mathcal{R}$". Siccome solitamente non c'è ambiguità sulla relazione presa in considerazione nella maggior parte dei casi non metteremo in evidenza questa dipendenza.

Valgono i seguenti risultati sulla funzione `split` appena introdotta:

Osservazione 1.5. Se A non è uno *splitter* di $\mathfrak{S}$ si ha $\mathfrak{S} = \texttt{split}(A, \mathfrak{S})$.

Proposizione 1.7. *Se $\mathfrak{S}$ è stabile rispetto ad un insieme A,* `split`$(B, \mathfrak{S})$ *è stabile rispetto ad A, B. In altre parole la funzione* `split` *preserva la stabilità.*

Dimostrazione. `split` può solamente dividere i blocchi, e non mescolarli. □

Dimostriamo il seguente risultato che utilizzeremo nel seguito [4]:

Teorema 1.6. *Consideriamo la funzione* `split` *proposta nella Definizione 1.26:*

1. *La funzione è monotona rispetto al secondo argomento, cioè se* $\mathfrak{S}$ *rifinisce* $\mathfrak{S}_2$*, allora* $\texttt{split}(A, \mathfrak{S})$ *rifinisce* $(A, \mathfrak{S}_2)$*;*

2. *La funzione è commutativa rispetto al primo argomento:*

   $\texttt{split}(A, \texttt{split}(B, \mathfrak{S})) = \texttt{split}(B, \texttt{split}(A, \mathfrak{S}))$.

Dimostrazione. Dimostriamo separatamente gli enunciati:

1. I blocchi di $\texttt{split}(A, \mathfrak{S})$ sono del tipo $x - \mathcal{R}^{-1}(A)$ oppure $x \cap \mathcal{R}^{-1}(A)$, dove x è un blocco di $\mathfrak{S}$. I blocchi di $\texttt{split}(A, \mathfrak{S}_2)$ sono del tipo $y - \mathcal{R}^{-1}(A)$ oppure $y \cap \mathcal{R}^{-1}(A)$, dove y è un blocco di $\mathcal{S}_2$.

   Poichè $\forall x \in \mathfrak{S}\ \exists y \in \mathfrak{S}_2 \mid x \subseteq y$ si ha $x - \mathcal{R}^{-1}(A) \subseteq y - \mathcal{R}^{-1}(A)$ e $x \cap \mathcal{R}^{-1}(A) \subseteq y \cap \mathcal{R}^{-1}(A)$;

2. Conseguenza delle proprietà delle operazioni insiemistiche "$-$" e "$\cap$".

□

1.6 Equivalenza tra RSCP e massima bisimulazione

In quest'ultima sezione della prima parte di questo elaborato esamineremo un legame fondamentale, che fungerà da base per quanto presenteremo nelle sezioni seguenti.

Dimostriamo innanzitutto il seguente risultato preliminare [5]:

Proposizione 1.8. *Sia* $G = (V, E)$*. Sia* $\mathfrak{V}$ *una partizione di* V *stabile rispetto a* E*. Allora la relazione binaria* $\mathcal{R}$ *su* V *definita come:*

$$a \mathcal{R} b \iff [a]_{\mathfrak{V}} = [b]_{\mathfrak{V}}$$

è una bisimulazione su G*.*

Dimostrazione. Siano $a, b \in G \mid a \mathcal{R} b$, e sia $a' \mid \langle a, a' \rangle \in E$. Poichè $\mathfrak{V}$ è stabile, si ha che $[a]_{\mathfrak{V}} \subseteq E^{-1}([a']_{\mathfrak{V}})$. Quindi $\exists b' \in [a']_{\mathfrak{V}} \mid \langle b, b' \rangle \in E$.

L'altra condizione caratteristica della bisimulazione si dimostra in modo speculare. □

In altre parole, una qualsiasi partizione stabile rispetto a E induce su un grafo diretto una bisimulazione che può essere ricavata in modo banale.

Dimostriamo ora il risultato opposto:

Proposizione 1.9. *Sia $\mathcal{R}$ una bisimulazione su G che sia anche una relazione di equivalenza. Allora la partizione $\mathfrak{X}$ i cui blocchi sono le classi di equivalenza di $\mathcal{R}$ è stabile rispetto a E.*

Dimostrazione. Se per assurdo $\mathfrak{X}$ non fosse stabile esisterebbero due blocchi $x_1, x_2 \mid x_1 \cap E^{-1}(x_2)$ non è ne x_1 nè $\emptyset$. Poniamo $A \coloneqq x_1 \cap E^{-1}(x_2)$.

Gli elementi a di A sono i nodi in $x_1 \mid \nexists b \in x_2$ per cui vale $\langle a, b\rangle \in E$. Ma poichè questi a e gli $x \in x_1 - A$ si trovano all'interno dello stesso blocco x_1 deve valere $x\mathcal{R}a$.

Sia $x \in x_1 - A$, ed $y \in x_2 \mid \langle x, y\rangle \in E$. Poichè $\forall a \in A$ vale $x\mathcal{R}a$, allora $\exists a' \mid \langle a, a'\rangle \in E, \langle a', y\rangle \in E$, cioè $[a']_{\mathfrak{X}} = [y]_{\mathfrak{X}} = x_2$. Quindi A deve necessariamente essere vuoto. □

Cioè una bisimulazione induce un partizionamento stabile rispetto a E dei nodi del grafo. Vale il seguente corollario:

Corollario 1.5. *Determinare la massima bisimulazione su un grafo diretto $G = (V, E)$ e trovare la* RSCP(E) *di V sono problemi equivalenti.*

Dimostrazione. Dimostriamo separatamente che la bisimulazione ricavata dalla RSCP(E) è massima, e che la partizione ricavata dalla massima bisimulazione è la RSCP(E).

- Sia $\mathcal{R}_M$ la massima bisimulazione su G. Per la Proposizione 1.4 è una relazione di equivalenza. Per la Proposizione 1.9 è possibile ricavare da $\mathcal{R}_M$ una partizione $\mathfrak{V}_1$ di V stabile rispetto a E.

  Supponiamo per assurdo che $\mathfrak{V}_1 \neq$ RSCP(E) di V, quindi esiste una partizione $\mathfrak{V}_2$ stabile rispetto a E tale che $|\mathfrak{V}_2| < |\mathfrak{V}_1|$.

  Ma per la Proposizione 1.8 da $\mathfrak{V}_2$ è possibile ricavare una bisimulazione $\mathcal{R}_2$ su G. Ma quindi $|\mathcal{R}_2| < |R_M|$, che è assurdo.

- Sia $\mathfrak{V}_1$ la RSCP(E) di V. Supponiamo per assurdo che la bisimulazione $\mathcal{R}$ ricavata da $\mathfrak{V}_1$ come nella Proposizione 1.8 non sia massima. Allora deve esistere un'altra bisimulazione $\mathcal{R}_2$ che sia massima.

  Da questa si può ricavare, come mostrato nella Proposizione 1.9, una partizione $\mathfrak{V}_2$ stabile rispetto a E per cui vale $|\mathfrak{V}_2| \leq |\mathfrak{V}_1|$. Ma questo è assurdo. □

2 Algoritmi per il calcolo della massima bisimulazione

In questa sezione saranno presentati alcuni algoritmi per la risoluzione del problema della determinazione della massima bisimulazione di un grafo diretto. Come abbiamo anticipato verrà sfruttata ampiamente l'equivalenza tra massima bisimulazione e RCSP dimostrata nella Sezione 1.6.

2.1 Algoritmo di Hopcroft

Esaminiamo innanzitutto un algoritmo risolutivo per una versione semplificata del problema, ovvero la minimizzazione di un'automa a stati finiti [9]. La risoluzione di questo caso particolare ha fornito diversi spunti importanti per l'ideazione di algoritmi risolutivi più generali che saranno presentati nel seguito.

2.1.1 Nozioni preliminari

Innanzitutto definiamo il concetto di *automa*, chiaramente centrale nella descrizione del problema:

Definizione 2.1. Consideriamo i seguenti oggetti matematici:

- Un insieme finito $\mathcal{I}$ detto *insieme degli ingressi*;
- Un insieme finito $\mathcal{U}$ detto *insieme delle uscite*;
- Un insieme finito $\mathcal{S}$ detto *insieme degli stati*, ad ogni stato è associata un'unica uscita appartenente ad U;
- Un insieme $\mathcal{F} \subseteq \mathcal{S}$ detto *insieme degli stati finali*;
- Una funzione $\delta : \mathcal{S} \times \mathcal{I} \to \mathcal{S}$ detta *funzione di trasferimento*.

Chiameremo $A = (\mathcal{S}, \mathcal{I}, \mathcal{U}, \delta, \mathcal{F})$ *automa*. Useremo la notazione $Out(x)$ per indicare l'output corrispondente allo stato x.

Possiamo rappresentare un'automa con una tabella degli stati, in cui ad ogni riga corrisponde uno stato, a ogni colonna un ingresso, e ogni cella contiene il nuovo stato quando, nel momento in cui il sistema si trova nello stato corrispondente alla riga, si inserisce l'ingresso corrispondente alla colonna. In alternativa possiamo utilizzare una rappresentazione grafica, in cui ogni stato è descritto da un cerchio contenente il nome dello stato, e le transizioni tra stati sono rappresentate da frecce sulle quali viene specificato l'ingresso che ha innescato la transizione.

	0	1
a[0]	a	b
b[0]	b	c
c[1]	c	c

Tabella 1: Rappresentazione tabellare di un'automa.

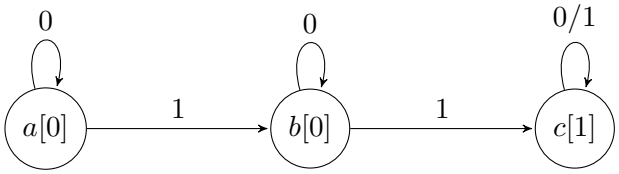


Figura 5: Rappresentazione grafica di un'automa.

Esempio 2.1. Nella Tabella 1 è rappresentato un'automa che cambia stato solamente se l'ingresso è "1", e lo stato non è "c".

Esempio 2.2. Nella Figura 5 è rappresentato lo stesso automa dell'Esempio 2.1.

In alcuni automi è possibile individuare stati che "si comportano in modo simile". Informalmente possiamo dire che due stati $a, b \in \mathcal{S}$ tali che $Out(a) = Out(b)$ si comportano in modo simile quando i due stati in cui ricade il sistema in seguito ad una transizione innescata da uno stesso input i a partire dagli stati a o b (ovvero i due stati $\delta(a, i), \delta(b, i)$) si comportano in modo simile. In Figura 6 illustriamo un esempio di questa situazione: consideriamo gli stati dell'automa rappresentato. Supponiamo di accorpare gli stati b, c in un unico stato, che chiamiamo B'. Se ci interessiamo solamente alla sequenza di output ed all'eventuale raggiungimento di uno stato finale, il nuovo automa risulta indistinguibile dal primo.

Esempio 2.3. In questo esempio gli stati B, C sono equivalenti secondo la

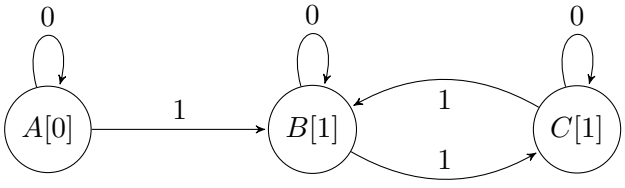


Figura 6: Automa contenente stati equivalenti.

definizione informale proposta sopra.

Proponiamo una possibile definizione formale di equivalenza tra stati. Nel seguito ne dedurremo una equivalente, che consente di stabilire un parallelo con gli argomenti trattati nella Sezione 1.

Definizione 2.2. Sia $\mathcal{I}^*$ l'insieme di tutte le sequenze di input di lunghezza finita. Sia $\delta^* : \mathcal{S} \times \mathcal{I}^*$ la funzione di transizione "iterata". Due stati $x, y \in \mathcal{S}$ sono equivalenti se e soltanto se valgono congiuntamente le seguenti condizioni:

1. $Out(x) = Out(y)$;
2. $\forall i^* \in \mathcal{I}^*,\ \delta^*(x, i^*) \in \mathcal{F} \iff \delta^*(y, i^*) \in \mathcal{F}$.

Se le condizioni valgono utilizzermo la notazione "$x \sim y$".

Osservazione 2.1. La relazione "$\sim$" è una relazione di equivalenza sull'insieme degli stati di un'automa.

La seguente proposizione sarà utile per formulare il problema con la terminologia esposta nella Sezione 1.5:

Proposizione 2.1. *Due stati x, y sono equivalenti nel senso della definizione 2.2 se e solo se*

$$\forall i \in \mathcal{I}, \quad \delta(x, i) \sim \delta(y, i) \tag{2.1}$$

e $Out(x) = Out(y)$.

Dimostrazione. Supponiamo che $\exists i \in \mathcal{I} : \delta(x, i) \not\sim \delta(y, i)$. Allora, ad esempio:

$$\exists i^* \in \mathcal{I}^* : \delta^*(\delta(x, i), i^*) = \delta^*(x, ii^*) \in \mathcal{F},\ \delta^*(\delta(y, i), i^*) = \delta^*(y, ii^*) \notin \mathcal{F}$$

Quindi, per l'esistenza della stringa ii^* si ha $x \not\sim y$. La dimostrazione è speculare se $\exists i^* \in \mathcal{I}^* : \delta^*(x, ii^*) \notin \mathcal{F}, \delta^*(y, ii^*) \in \mathcal{F}$.

Ora supponiamo che valga la 2.1. È chiaro che $\forall i^* \in \mathcal{I}^*$ si ha:

$$\delta^*(x, ii^*) \sim \delta^*(y, ii^*) \forall i \in \mathcal{I}$$

e quindi $x \sim y$. □

Proposizione 2.2. *La relazione "$\sim$" è una bisimulazione sull'insieme $\mathcal{S}$, con la relazione binaria $E := \bigcup_{i \in \mathcal{I}, x \in \mathcal{S}} \{(x, \delta(x, i))\}$.*

Dimostrazione. Supponiamo che $x \sim y$. Sia $\langle x, x' \rangle \in E$, cioè $\exists i \in \mathcal{I} : x' = \delta(x, i)$. Sia $y' = \delta(y, i) \implies \langle y, y' \rangle$. Allora, per la Proposizione 2.1, $x' \sim y'$. Lo stesso argomento vale in modo speculare. □

Osserviamo che nella definizione di E si perde un'informazione importante, cioè il fatto che per $i \in \mathcal{I}$ fissato $\delta_i(x) := \delta(x, i)$ è una funzione, ovvero l'insieme immagine di ogni $x \in \mathcal{S}$ ha cardinalità 1. L'algoritmo di Hopcroft sfrutta questa ipotesi nel procedimento che consente di migliorare l'algoritmo banale che verrà discusso nel seguito del lavoro.

Possiamo ora definire la *minimizzazione per stati equivalenti* di un'automa a stati finiti:

Definizione 2.3. Sia $\mathcal{A} = (\mathcal{S}, \mathcal{I}, \mathcal{U}, \mathcal{F}, \delta)$ un'automa a stati finiti. Chiameremo *minimizzazione per stati equivalenti* l'automa $(\mathfrak{S}, \mathcal{I}, \mathcal{U}, \delta_{\mathfrak{S}}, \mathcal{F}_{\mathfrak{S}})$ dove

- $\mathfrak{S} = \mathcal{S}/\sim$;
- $\delta_{\mathfrak{S}} : (\mathfrak{S} \times \mathcal{I}) \to \mathfrak{S} \mid \delta(i, x) = y \implies \delta_{\mathfrak{S}}(i, [x]_{\mathfrak{S}}) = [y]_{\mathfrak{S}}$;
- $\mathcal{F}_{\mathfrak{S}} = \mathcal{F}/\sim$.

Si rammenta che con la notazione $A/\mathcal{R}$ ci riferiamo all'insieme delle classi di equivalenza indotte da una relazione di equivalenza $\mathcal{R} : A \times A$ su un insieme A, e che con la notazione $[a]_{\mathfrak{X}}$ ci riferiamo al blocco di una partizione $\mathfrak{X}$ in cui viene posto a.

Prima di concludere l'esposizione delle nozioni preliminari è necessario dimostrare un risultato interessante che lega il problema della minimizzazzione di un'automa a stati finiti con quanto è stato presentato nella sezione 1.5. A questo scopo dimostriamo i seguenti lemmi, che consentono di dimostrare tale legame in modo agevole:

Lemma 2.1. *Sia $\mathcal{A} = (\mathcal{S}, \mathcal{I}, \mathcal{U}, \mathcal{F}, \delta)$ un'automa a stati finiti. Sia $\mathfrak{S}$ una partizione di $\mathcal{S}$ stabile rispetto alla famiglia di funzioni $\delta_i, i \in \mathcal{I}$. Allora $\forall (x, y) \in \mathcal{S} \times \mathcal{S}$ tali che $[x]_{\mathfrak{S}} = [y]_{\mathfrak{S}}$ si ha che $\forall i^* \in \mathcal{I}^*$:*

$$[\delta^*(x, i^*)]_{\mathfrak{S}} = [\delta^*(y, i^*)]_{\mathfrak{S}}$$

Dimostrazione. Procediamo per induzione su i^*. Inizialmente i due stati si trovano nello stesso blocco di $\mathfrak{S}$. Ora supponiamo che dopo l'inserimento dell' $(n-1)$-esimo simbolo di i^* i due stati x_{n-1}, y_{n-1} in cui si trova l'automa appartengano ancora allo stesso blocco. La partizione è stabile rispetto alla funzione δ_i, dove i è l' n-esimo simbolo di i^*, quindi tutto il blocco $[x_{n-1}]_{\mathfrak{S}} = [y_{n-1}]_{\mathfrak{S}}$ è contenuto all'interno dell'insieme $\delta_i^{-1}([\delta_i(x_{n-1})]_{\mathfrak{S}}) = \delta_i^{-1}([x_n]_{\mathfrak{S}})$. Quindi $\delta_i(y_{n-1}) \in \delta_i^{-1}([x_n]_{\mathfrak{S}})$, e dunque e si ha anche $[x_n]_{\widehat{S}} = [y_n]_{\mathfrak{S}}$. □

Lemma 2.2. *Sia $\mathcal{A} = (\mathcal{S}, \mathcal{I}, \mathcal{U}, \mathcal{F}, \delta)$ un'automa a stati finiti. Sia $\mathfrak{S}$ una partizione di $\mathcal{S}$ stabile rispetto alla famiglia di funzioni $\delta_i, i \in \mathcal{I}$. Supponiamo inoltre che per $\mathfrak{S}$ valga la segente condizione:*

$$\forall (x, F) \in \mathcal{S} \times \mathcal{F}, \quad [x]_{\mathfrak{S}} = [F]_{\mathfrak{S}} \implies x \in \mathcal{F}$$

Allora $\forall \mathfrak{p} \in \mathfrak{S},\ \forall (x, y) \in \mathfrak{p} \times \mathfrak{p}$ si ha $x \sim y$.

Dimostrazione. Supponiamo per assurdo che in un blocco di $\mathfrak{S}$ esistano due stati x, y non equivalenti. Allora deve esistere una stringa $i^* \in \mathcal{I}^*$ tale che, ad esempio, $\delta^*(x, i^*) \in \mathcal{F} \wedge \delta^*(y, i^*) \notin \mathcal{F}$. Ma per il Lemma 2.1 $[\delta^*(x, i^*)]_{\mathfrak{S}} = [\delta^*(y, i^*)]_{\mathfrak{S}}$. Chiaramente questo è assurdo per la condizione assunta nell'ipotesi, quindi non possono esistere nello stesso blocco due stati non equivalenti. □

A questo punto abbiamo gli ingredienti necessari per dimostrare un risultato interessante su una tipologia particolare di automi così definita:

Definizione 2.4. Un'automa a stati finiti $\mathcal{A} = (\mathcal{S}, \mathcal{I}, \mathcal{U}, \mathcal{F}, \delta)$ è *ben posto* se $x \in \mathcal{F}, y \notin \mathcal{F} \implies Out(x) \neq Out(y)$, ovvero l'output assegnato a stati finali è sempre diverso dall'output corrispondente a stati non finali.

Teorema 2.1. *Sia $\mathcal{A} = (\mathcal{S}, \mathcal{I}, \mathcal{U}, \mathcal{F}, \delta)$ un'automa a stati finiti. La minimizzazione per stati equivalenti $(\mathfrak{S}, \mathcal{I}, \mathcal{U}, \delta_{\mathfrak{S}}, \mathcal{F}_{\mathfrak{S}})$ (Definizione 2.3) è l'automa avente per stati i blocchi della partizione più grossolana stabile rispetto alla famiglia di funzioni $\delta_i, i \in \mathcal{I}$.*

Dimostrazione. Dimostriamo che $\mathfrak{S}$ è una partizione di $\mathcal{S}$ stabile rispetto alle funzioni δ_i. Supponiamo per assurdo che $\exists i \in \mathcal{I} \mid \mathfrak{S}$ non è stabile rispetto a δ_i. Quindi esistono $\mathfrak{s}_1, \mathfrak{s}_2 \in \mathfrak{S}$ tali che:

$$\mathfrak{s}_1 \not\subseteq \delta_i^{-1}(\mathfrak{s}_2) \ \wedge \ \mathfrak{s}_1 \cap \delta_i^{-1}(\mathfrak{s}_2) \neq \emptyset$$

La prima porzione dell'espressione implica che $\exists x \in \mathfrak{s}_1 : \delta_i(x) \notin \mathfrak{s}_2$. Ma questo è chiaramente in contrasto con l'Osservazione 2.1, perchè $\forall (y, Z) \in \mathfrak{s}_1 \times \mathfrak{s}_1$ deve valere $\delta_i(y) \sim \delta_i(Z)$, mentre è evidente che, poichè $\mathfrak{s}_1 \cap \delta_i^{-1}(\mathfrak{s}_2) \neq \emptyset$, c'è almeno una coppia che non soddisfa questa condizione.

Ora supponiamo che la partizione $\mathfrak{S}$ non sia la più grossolana stabile rispetto a tutte le funzioni $\delta_i : i \in \mathcal{I}$. Ne deve esistere quindi una stabile più grossolana, che chiamiamo $\mathfrak{S}_2$. Chiaramente si ha $|\mathfrak{S}_2| < |\mathfrak{S}|$, e quindi devono esistere almeno due blocchi $\mathfrak{s}_1, \mathfrak{s}_2 \in \mathfrak{S}$ ed un blocco $\mathfrak{s}_3 \in \mathfrak{S}_2$ tali che

$$\mathfrak{s}_1 \cap \mathfrak{s}_3 \neq \emptyset \ \wedge \ \mathfrak{s}_2 \cap \mathfrak{s}_3 \neq \emptyset$$

Ma questo è chiaramente in contrasto con il Lemma 2.2 per come è stata costruita la partizione $\mathfrak{S}$, in quanto non possono esistere coppie di stati nello stesso blocco di $\mathcal{S}_2$ ma in un blocco diverso di $\mathcal{S}$, in quanto da quest'ultima condizione segue che $x \not\sim y$. □

Osserviamo che la condizione richiesta nell'ipotesi del Teorema 2.1, che impone l'assegnazione di un output diverso a stati finali e non finali, è estremamente debole: dato un'automa qualsiasi è sempre possibile costruirne un altro che soddisfa tale condizione, in tempo lineare rispetto al numero di stati.

2.1.2 L'algoritmo di Hopcroft

Il problema della minimizzazione di automi a stati finiti è stato risolto nel 1971 da John Hopcroft, che ha proposto un algoritmo efficiente sfruttando un'osservazione che sarà discussa nel seguito. Presenteremo innanzitutto un algoritmo immediato, che sarà utile per un primo approccio al lato algoritmico del problema; dopodichè esamineremo lo pseudocodice dell'algoritmo di Hopcroft, e commenteremo l'intuizione che ha consentito di migliorare l'algoritmo immediato.

Un primo algoritmo triviale Utilizzando i risultati della sezione precedente possiamo progettare il seguente algoritmo, il cui funzionamento è sostanzialmente lineare: per ogni ingresso $i \in \mathcal{I}$ si rifinisce la partizione iniziale (data dagli output assegnati ai vari stati) finchè non diventa stabile rispetto a δ_i; dopodichè si procede con l'ingresso seguente.

La funzione "BlocchiNonStabili" restituisce una coppia $\mathfrak{s}_1, \mathfrak{s}_2$ di blocchi di $\mathfrak{S}$ che trasgredisce la condizione di stabilità rispetto alla funzione δ_i, o "None" quando una tale coppia non esiste. L'algoritmo termina, perchè la condizione del ciclo diventa sicuramente falsa quando la partizione $\mathfrak{S}$ è composta da n blocchi, uno per ogni stato, ed ad ogni iterazione un blocco viene diviso in due blocchi distinti e non vuoti.

Inoltre la risposta fornita è corretta, perchè l'algoritmo si ferma appena viene trovata una partizione stabile. Poichè ad ogni iterazione del ciclo definito nella Riga 4 il numero di blocchi aumenta di 1, il risultato è la partizione stabile più grossolana rispetto alle funzioni δ_i. Osserviamo inoltre che, se la partizione è stabile rispetto a δ_i dopo una certa iterazione del ciclo definito nella Riga 3, allora resta stabile fino alla fine dell'esecuzione per la Proposizione 1.6.

Verifichiamo la complessità dell'algoritmo:

Algoritmo 2: Algoritmo triviale per la minimizzazzione di automi a stati finiti.

Data: $\mathcal{S}, \mathcal{I}, \mathcal{U}, \mathcal{F}, \delta$

`// Partizione iniziale contenente un unico blocco`

1 $\widetilde{S} \coloneqq \{\{s_1, \ldots, s_n\}\}$;

`/* Separiamo gli stati non equivalenti in base all'output */`

2 $\mathfrak{S} = \{\mathfrak{s}_k \in \mathcal{P}(\mathcal{S}) \mid Out(x) = k, \forall x \in \mathfrak{s}_k\}$;

3 **foreach** $i \in I$ **do**

4 | **while** $(\mathfrak{s}_1, \mathfrak{s}_2 = \text{BlocchiNonStabili}(\mathfrak{S}, \delta_i)) \neq \text{None}$ **do**

| | `/* Estraiamo una coppia di blocchi che trasgredisce la condizione di stabilità. */`

5 | | $\widetilde{\mathfrak{s}}_1 \coloneqq \mathfrak{s}_1 - \delta_i^{-1}(\mathfrak{s}_2)$;

6 | | $\widehat{\mathfrak{s}}_1 \coloneqq \mathfrak{s}_1 \cap \delta_i^{-1}(\mathfrak{s}_2)$;

| | `// Aggiorniamo` S

7 | | $\mathfrak{S} = (\mathfrak{S} - \{\mathfrak{s}_1\}) \cup \{\widetilde{\mathfrak{s}}_1, \widehat{\mathfrak{s}}_1\}$;

8 **return** S

- La Riga 2 ha complessità $\Theta(|S|)$;
- Il ciclo alla Riga 3 viene eseguito $\Theta(|I|)$ volte;
- La funzione "BlocchiNonStabili" ha complessità $O(|S|^2)$;
- Il ciclo alla Riga 4 viene eseguito $O(|S|)$ volte;
- Il contenuto del ciclo alla Riga 4 ha complessità $O(|S|)$ (con gli opportuni accorgimenti).

Quindi la complessità dell'algoritmo è

$$T(|S|, |I|) = \Theta|I| \left[O(|S|^2 + O(|S|) * O(|S|))\right] = \Theta(I)O(|S|^2)$$

Se consideriamo costante il numero di ingressi: $T(|S|) = O(|S|^2)$. Seppur di facile comprensione, l'algoritmo triviale è poco efficiente a causa dei numerosi controlli per la stabilità.

Algoritmo di Hopcroft Presentiamo un algoritmo che migliora la procedura triviale presentata nella sezione precedente, portando la complessità relativa alla risoluzione del problema della minimizzazione di automi a stati

Algoritmo 3: Algoritmo di Hopcroft

Data: $\mathcal{S}, \mathcal{I}, \mathcal{U}, \mathcal{F}, \delta$

begin
  foreach $(s, i) \in \mathcal{S} \times \mathcal{I}$ **do**
    $\delta^{-1}(s, i) \coloneqq \{t : \delta(t, i) = s\}$;
  $B(1) \coloneqq \mathcal{F}, B(2) \coloneqq \mathcal{S} - \mathcal{F}$;
  foreach $j \in \{1, 2\}$ **do**
    /* $\forall i \in \mathcal{I}$ costruiamo l'insieme degli stati in $B(j)$ aventi controimmagine non vuota rispetto a δ_i */.
    foreach $i \in \mathcal{I}$ **do**
      $i(j) = \{s : s \in B(j) \wedge \delta^{-1}(s, i) \neq \emptyset\}$;
  /* k è il numero di blocchi della partizione, aumenta in seguito alle rifiniture. */
  $k \coloneqq 2$;
  /* Per ogni ingresso i creiamo un insieme $L(i)$ contenente l'indice che minimizza $|i(\cdot)|$. */
  foreach $i \in \mathcal{I}$ **do**
    $L(i) = \arg\min_j i(j)$;
  while $\exists i \in \mathcal{I} \mid L(i) \neq \emptyset$ **do**
    Seleziona un $j \in L(i)$. $L(i) = L(i) - \{j\}$;
    /* Per ogni blocco $B(m)$ per cui il procedimento ha senso, usiamo $i(j)$ come *splitter*. */
    foreach $m \leq k \mid \exists t \in B(m) : \delta_i(t) \in i(j)$ **do**
      $B^{'}(m) \coloneqq \{u \in B(m) \mid \delta_i(u) \in i(j)\}$;
      $B^{''}(m) \coloneqq B(m) - B^{'}(m)$;
      $B(m) = B^{'}(m)$, $B(k+1) \coloneqq B^{''}(m)$;
      /* Dopo l'aggiunta del blocco $B(k+1)$ aggiorniamo l'indice che minimizza $|a(\cdot)|$. */
      foreach $a \in \mathcal{I}$ **do**
        $a(m) = \{s \mid s \in B(m) \wedge \delta^{-1}(s, a) \neq \emptyset\}$;
        $a(k+1) \coloneqq \{s \mid s \in B(k+1) \wedge \delta^{-1}(s, a) \neq \emptyset\}$;
        $L(a) = \arg\min_j a(j), j \in \{m, k+1\}$;
      // Aggiorniamo il numero di blocchi in B.
      $k = k + 1$;

finiti a *loglineare* [9]. Commenteremo lo pseudocodice in modo da spiegare il procedimento in modo dettagliato. In seguito analizzeremo formalmente l'algoritmo, proporremo la dimostrazione della correttezza e della complessità.

Alla Riga 3 definiamo δ^{-1}, che consente l'accesso in tempo costante all'insieme degli stati che conducono ad un determinato stato conseguentemente ad un determinato ingresso. Inizialmente la partizione consiste di due blocchi, corrispondenti agli insiemi di stati finali e non finali. Di conseguenza le rifiniture successive della partizione manterranno sempre separati stati finali e non finali. Al fine di evitare accessi inutili, alla Riga 7 definiamo $i(j)$ per $i \in \mathcal{I}, j \in \{1,2\}$ come l'insieme degli stati s nel blocco $B(j)$ per cui esiste qualche stato t tale che $\delta(t,i) = s$, ovvero quelli aventi controimmagine non vuota rispetto alla funzione δ_i. All'interno del ciclo della Riga 9 definiamo gli insiemi $L(i)$ $\forall i \in \mathcal{I}$, che contengono gli indici dei blocchi che verranno usati come *splitter* nel seguito del procedimento. In questo insieme avremo sempre e solo gli indici dei blocchi per cui la cardinalità di "$i(\cdot)$" è minima. Questo accorgimento, alla luce del Teorema che enunceremo nelle righe seguenti, consente di ridurre il carico computazionale per l'operazione di `split`.

Si può dimostrare che, limitatamente al caso particolare a cui si applica l'algoritmo di Hopcroft, è possibile scegliere gli *splitter* in modo vantaggioso, allo scopo di ridurre il carico di lavoro e dunque la complessità [9]. Riportiamo e commentiamo la dimostrazione:

Proposizione 2.3. *Sia $\mathcal{A}$ un insieme finito. Sia $f : \mathcal{A} \to \mathcal{A}$ una funzione (cioè $\forall a \in \mathcal{A}, |f(a)| = 1$). Sia $\mathfrak{P}$ una partizione di $\mathcal{A}$. Sia Q l'unione di alcuni blocchi di $\mathfrak{P}$, e sia $\mathfrak{p}$ un blocco di $\mathfrak{P}$, con $\mathfrak{p} \subseteq Q$. Supponiamo inoltre che $\mathfrak{P}$ sia stabile rispetto alla coppia (Q, f).*

Allora vale la seguente espressione:

$$\texttt{split}(\mathfrak{p}, \mathfrak{X}) = \texttt{split}(Q - \mathfrak{p}, \texttt{split}(B, \mathfrak{P})).$$

Dimostrazione. Vogliamo dimostrare che $Q - \mathfrak{p}$ non è uno *splitter* di $\texttt{split}(B, \mathfrak{P})$, cioè che $\forall s_x \in \texttt{split}(B, \mathfrak{P})$ si ha:

$$s_x \subseteq f^{-1}(Q - \mathfrak{p}) \ \vee \ s_x \cap f^{-1}(Q - \mathfrak{p}) = \emptyset.$$

Rammentando che $\mathfrak{P}$ è stabile rispetto a Q, $\mathfrak{P}_2 \coloneqq \texttt{split}(B, \mathfrak{P})$ è stabile rispetto a Q e a B, per l'Osservazione 1.7. Di conseguenza per ogni blocco

$s_x \in \mathfrak{P}_2$ si ha:

$$s_x \subseteq f^{-1}(\mathfrak{p}) \vee s_x \cap f^{-1}(\mathfrak{p}) = \emptyset$$
$$\wedge$$
$$s_x \subseteq f^{-1}(Q) \vee s_x \cap f^{-1}(Q) = \emptyset$$

Se $s_x \subseteq f^{-1}(\mathfrak{p})$ chiaramente $s_x \cap \subseteq f^{-1}(Q - s) = \emptyset$. Se $s_x \cap f^{-1}(Q) = \emptyset$, allora $s_x \cap f^{-1}(Q - \mathfrak{p}) = \emptyset$. Se $s_x \cap f^{-1}(\mathfrak{p}) = \emptyset \wedge s_x \subseteq f^{-1}(Q)$ vale $s_x \subseteq f^{-1}(Q - \mathfrak{p})$. □

Insistiamo sul fatto che le deduzioni della Proposizione 2.3 valgono solamente se f è una funzione. Per questo motivo tale risultato non può essere usato nel contesto più generale della teoria dei grafi. Dal punto di vista computazionale è chiaramente più conveniente usare come *splitter* l'insieme tra B e $Q - B$ avente cardinalità minore. La strategia di Hopcroft "*process the smaller half*" consiste infatti nella selezione degli *splitter* secondo il criterio della cardinalità, a differenza di quanto avviene nell'Algoritmo 2.

Possiamo ora proseguire con l'analisi dello pseudocodice. Dalla Riga 14 alla Riga 17 viene operato lo "Split". Nel ciclo della Riga 17 vengono aggiornate le strutture dati relative ai nuovi blocchi creati. Osserviamo che ad ogni iterazione del ciclo della Riga 17 viene creato un nuovo blocco (posto all'indirizzo $k+1$ della struttura B) ed un blocco già esistente viene "smembrato". Di questi due blocchi se ne sceglie uno da usare come *splitter* seguendo il criterio illustrato sopra. Queste operazioni vengono ripetute finchè in $L(i)$ per qualche $i \in \mathcal{I}$ resta un blocco da usare come *splitter*.

2.1.3 Correttezza e complessità dell'Algoritmo di Hopcroft

Ad ogni iterazione del ciclo della Riga 11 viene rimosso un elemento in $L(i)$, e solamente in seguito alla rifinitura della partizione ne viene inserito un altro. Poichè non è possibile rifinire all'infinito, l'algoritmo termina.

Dimostriamo la correttezza dell'Algoritmo 3 con il seguente Teorema:

Teorema 2.2. *Sia $A = (S, I, \delta, F)$ un'automa. Sia $\mathfrak{S}$ la partizione risultante dall'applicazione dell'Algoritmo 3. Siano $x, y \in \mathcal{S}$. Allora:*

$$x \sim y \iff [x]_{\mathfrak{S}} = [y]_{\mathfrak{S}}.$$

Dimostrazione. Dimostriamo innanzitutto che $[x]_{\mathfrak{S}} \neq [y]_{\mathfrak{S}} \implies x \not\sim y$. Procediamo per induzione:

– La relazione vale prima del ciclo alla Riga 11, infatti stati finali e non finali non possono essere equivalenti;

– Supponiamo che sia vero prima di una certa iterazione. Perchè x, y possano essere posizionati in partizioni diverse di $\mathfrak{S}$ tra le Righe 14 e 17 deve esistere un $i \in \mathcal{I} \mid [\delta(x,i)]_{\mathfrak{S}_n} \neq [\delta(y,i)]_{\mathfrak{S}_n}$, dove $\widetilde{S}_n$ è la partizione costruita dall'algoritmo fino all'iterazione considerata. Ma allora, per l'ipotesi induttiva, $\delta(x,i) \not\sim \delta(y,i)$, e quindi x, y non sono equivalenti (per l'Osservazione 2.1).

Questo ragionamento è valido perchè il fatto che due stati si trovino in partizioni differenti dopo una certa iterazione implica che si troveranno in partizioni differenti anche al termine del procedimento, per come è costruita l'operazione di `split`.

Dimostriamo ora che $[x]_{\widetilde{S}} = [y]_{\widetilde{S}} \implies x \sim y$. Supponiamo per assurdo che per due stati x, y si abbia $[x]_{\mathfrak{S}} = [y]_{\mathfrak{S}} \wedge x \not\sim y$. Allora, ad esempio, $\exists i^* \in \mathcal{I}^* : \delta^*(x, i^*) \in \mathcal{F}, \delta^*(y, i^*) \notin \mathcal{F}$. Ma poichè inizialmente poniamo stati inziali e finali in partizioni diverse, deve valere chiaramente $[\delta^*(x, i^*)]_{\mathfrak{S}} \neq [\delta^*(y, i^*)]_{\mathfrak{S}}$, e quindi procedendo a ritroso sui simboli di i^* si ha:

$$[\delta^*(x, i^*_n)]_{\mathfrak{S}} \neq [\delta^*(y, i^*)_n]_{\mathfrak{S}} \implies [\delta^*(x, i^*_{n-1})]_{\mathfrak{S}} \neq [\delta^*(y, i^*)_{n-1}]_{\mathfrak{S}}.$$

Per cui si ha chiaramente $[x]_{\mathfrak{S}} = [\delta^*(x, i^*_0)]_{\mathfrak{S}} \neq [\delta^*(y, i^*)_0]_{\mathfrak{S}} = [y]_{\mathfrak{S}}$. □

Consideriamo ora la complessità dell'Algoritmo 3. Per costruire δ^{-1} è sufficiente valutare una sola volta ogni stato per ogni ingresso, quindi l'operazione è $\Theta(|\mathcal{S}||\mathcal{I}|)$. La costruzione delle due partizioni iniziali è $\Theta(|S|)$. La costruzione di $L(i)$ $\forall i \in \mathcal{I}$ è chiaramente $\Theta(|\mathcal{I}|)$. Dunque la complessità delle istruzioni precedenti alla Riga 11 è $\Theta(|\mathcal{S}||\mathcal{I}|)$.

Prima di procedere, consideriamo il seguente risultato ausiliario:

Lemma 2.3. *Sia*

$$f_a(x) \coloneqq a \log_2(a) - (a-x)\log_2(a-x) - x\log_2(x) \qquad a > 0,\ 0 < x < a$$

Tale funzione ha massimo in $\frac{a}{2}$, con $f_a(\frac{a}{2}) = a$, ed è strettamente positiva sul dominio considerato.

Dimostrazione. L'osservazione si dimostra con un semplice studio di funzione. □

Il seguente risultato fornisce un *upper-bound* per il contributo al tempo di esecuzione di tutte le iterazioni del ciclo in cui si seleziona un ingresso $i \in \mathcal{I}$, a partire da un'iterazione n-esima fino alla terminazione dell'algoritmo:

Teorema 2.3. *Consideriamo l'iterazione n-esima del ciclo della Riga 11 per un n qualsiasi, descritta dalla seguente configurazione:*

$$\mathfrak{S}_n = \{S_1, \ldots, S_m\}, \quad L(i) = \{i_1, \ldots, i_r\}, \quad \{i_{r+1}, \ldots, i_m\} = \{1, \ldots, m\} - L(i)$$

Il contributo alla complessità dato da tutte le iterazioni in cui si seleziona l'ingresso i, dalla n-esima fino alla terminazione dell'algoritmo, è maggiorato dalla seguente espressione:

$$T_i^n = k \left(\sum_{j=1}^{r} |i(i_j)| \log_2 |i(i_j)| + \sum_{j=r+1}^{m} |i(i_j)| \log_2 \frac{|i(i_j)|}{2} \right)$$

Dimostrazione. Procediamo per induzione "al contrario". Chiaramente la maggiorazione è valida alla terminazione dell'algoritmo, ovvero dopo l'ultima iterazione. Ora supponiamo che la maggiorazione sia valida dopo l'iterazione k-esima (con $k > n$), e dimostriamo che questo implica la validità della stessa prima dell'iterazione k-esima, ovvero al termine dell'iterazione $(k-1)$-esima. In altre parole è necessario dimostrare che la somma tra T_i^{k-1} ed il tempo impiegato per l'esecuzione dell'iterazione k-esima è minore o uguale di T_i^k. La seguente osservazione consente di quantificare il contributo di una singola iterazione:

Osservazione. Se all'inizio dell'iterazione viene selezionato l'indice $x \in L(i)$, la complessità dell'iterazione è $O(|i(x)|)$, cioè il tempo impiegato durante l'esecuzione è maggiorato da $k|i(x)|$, dove k è una qualche costante di propozionalità. Possiamo trarre questa conclusione perchè il ciclo della Riga 13 è in realtà un ciclo sugli stati $s \in i(x)$.

All'inizio di ogni iterazione viene estratto un $a \in \mathcal{I}$. Si presentano due casi:

- $a \neq i$: poichè T_i^k prende in considerazione solamente il tempo impiegato dalle iterazioni in cui viene selezionato i, questa iterazione è esclusa dalla stima. Ciononostante l'iterazione può modificare i blocchi della partizione $\mathfrak{S}_k$, e dobbiamo quindi verificare che $T_i^{k-1} \leq T_i^k$:
  - Se viene modificato un blocco $B(x)$ con $x \in L(i)$, nella stima dobbiamo sostituire un elemento maggiorabile con $b \log_2 b$ con un elemento del tipo $c \log_2 c + (b-c) \log_2 (b-c)$. Per il Lemma 2.3 si ha $T_i^{k-1} < T_i^k$;

– Se invece $x \notin L(i)$, in T_i^{k-1} dobbiamo sostituire un elemento del tipo $b \log_2 \frac{b}{2}$ con un elemento del tipo $c \log_2 c + (b-c) \log_2 \frac{b-c}{2}$ (supponendo che $c \leq b - c$, in caso contrario la dimostrazione è simile). Infatti il blocco avente cardinalità $b - c$ fa parte, alla fine dell'iterazione, dell'insieme dei blocchi il cui indice non appartiene ad $L(i)$, ed al termine del ciclo della Riga 17 si ha $c \in L(i)$. Chiaramente si ha $c \leq \frac{b}{2}$, e quindi:

$$\begin{aligned} c \log_2 c + (b-c) \log_2 \frac{b-c}{2} &\leq c \log_2 \frac{b}{2} + (b-c) \log_2 \frac{b}{2} \\ &= (c + b - c) \log_2 \frac{b}{2} \\ &= b \log_2 \frac{b}{2}. \end{aligned}$$

– $a = i$: come abbiamo osservato sopra, il tempo impiegato all'interno dell'iterazione è $O(|i(x)|)$, dove x è l'indice estratto da $L(i)$. Dunque:

$$T_i^{k-1} + k|i(x)| = k \Bigg(|i(x)| + |i(x)| \log_2 \frac{|i(x)|}{2} + \sum_{\substack{j=1 \\ i_j \neq x}}^{r} |i(i_j)| \log_2 |i(i_j)| + \sum_{j=r+1}^{m} |i(i_j)| \log_2 \frac{|i(i_j)|}{2} \Bigg).$$

abbiamo considerato il tempo impiegato per l'iterazione, ed il fatto che l'indice x al termine dell'iterazione fa parte dell'insieme dei blocchi il cui indice non appartiene ad $L(i)$. Poichè vale:

$$\begin{aligned} |i(x)| + |i(x)| \log_2 \frac{|i(x)|}{2} &= |i(x)| + |i(x)| \log_2 |i(x)| - |i(x)| \log_2 2 \\ &= |i(x)| + |i(x)| \log_2 |i(x)| - |i(x)| \\ &= |i(x)| \log_2 |i(x)| \end{aligned}$$

si deduce facilmente la seguente relazione:

$$T_i^{k-1} + k|i(x)| = T_i^k.$$

□

Prima della prima iterazione del ciclo della Riga 11, per un $i \in \mathcal{I}$ fissato, (supponendo $|\mathcal{S} - \mathcal{F}| \geq |\mathcal{F}|$) si ha

$$T_i = k \left(|\mathcal{S} - \mathcal{F}| \log_2 |\mathcal{S} - \mathcal{F}| + |\mathcal{F}| \log_2 \frac{|\mathcal{F}|}{2} \right)$$

che, per il Lemma 2.3, si maggiora con $k|\mathcal{S}|\log_2|\mathcal{S}|$. Allora la complessità dell'algoritmo è data dalla somma della complessità delle righe precedenti alla 11, ovvero $\Theta(|\mathcal{I}||\mathcal{S}|)$, e $|\mathcal{I}|O(|\mathcal{S}|\log|\mathcal{S}|)$, cioè $|\mathcal{I}||\mathcal{S}|\log|\mathcal{S}|$. Considerando costante la cardinalità di $\mathcal{I}$ otteniamo la complessità dell'algoritmo di Hopcroft:

$$T_{\text{Hopcroft}} = O(|\mathcal{S}|\log|\mathcal{S}|).$$

2.2 Algoritmo di Paige-Tarjan

In questa sezione esamineremo il più celebre procedimento per il calcolo della massima bisimulazione, l'algoritmo di Paige-Tarjan [15]. È stato il primo metodo efficiente per la risoluzione del problema nel caso generale (rammentiamo infatti che l'algoritmo di Hopcroft tratta un caso particolare), e migliora la complessità $O(|E||V|)$ dell'algoritmo di Kanellakis e Smolka, che non adotta particolari accorgimenti per la selezione dei blocchi da usare come *splitter* (si ricordi che E è l'insieme degli archi del grafo, e V l'insieme dei nodi). Innanzitutto presenteremo e commenteremo lo pseudocodice, utilizzando la terminologia introdotta nella Sezione 1; dopodichè dimostreremo la correttezza e la complessità.

2.2.1 L'algoritmo

L'algoritmo di Paige-Tarjan prende in input gli insiemi V, E di nodi e degli archi, e la partizione iniziale $\mathfrak{V}$ da rifinire. Inizialmente vengono definite due partizioni Q, X: la prima coincide con la partizione iniziale $\mathfrak{V}$, mentre la seconda è la partizione banale formata da un unico blocco contenente tutto V.

Algoritmo 4: Algoritmo di Paige-Tarjan

```
   Data: V, E, 𝔙
1  begin
2      Q ≔ 𝔙, X ≔ {V};
       // I blocchi di Q vengono rifiniti rispetto a E^{-1}(V)
3      Q = split(V, Q);
4      while Q ≠ X do
5          Scelgo in modo casuale un blocco S ∈ X | B ∉ Q;
6          Scelgo un blocco B ∈ X | B ⊂ S, |B| ≤ |S|/2;
7          X = (X − {S}) ∪ {B, S − B};
8          Q = split(S − B, split(B, Q));
9      return Q;
```

Alla Riga 3 viene imposta l'invariante che verrà poi mantenuta per tutta l'esecuzione dell'algoritmo: ogni blocco di Q viene diviso in due blocchi (di cui uno eventualmente vuoto): nel primo vengono inseriti i nodi che sono sorgente di almeno un arco; nel secondo vengono inseriti i pozzi (cioè gli elementi dell'insieme $B - E^{-1}(V)$, dove B è un blocco di Q). Dopo questa operazione la partizione Q è stabile rispetto a V.

Il corpo del procedimento è formato da un ciclo che ripete le seguenti istruzioni finchè le partizioni Q, X non coincidono:

1. Viene scelto casualmente un blocco $S \in X$ non appartenente a Q;
2. Viene scelto casualmente un blocco $B \in Q \mid B \subset S$, e la cui cardinalità sia inferiore a $|S|/2$;
3. In X il blocco S viene sostituito da $\{B, S - B\}$;
4. Si sostituisce la partizione Q con $\texttt{split}(S - B, \texttt{split}(B, Q))$.

Quando Q e X coincidono, viene restituita la partizione Q.

Osservazione 2.2. È sempre possibile trovare un blocco in X per portare a termine il passaggio 1. Quando non è possibile l'algoritmo termina, perchè $Q = X$.

2.2.2 Analisi

La correttezza dell'algoritmo di Paige-Tarjan si basa sul mantenimento di un'invariante, che enunciamo e dimostriamo formalmente:

Lemma 2.4. *L'Algoritmo 4 ristabilisce le seguenti relazioni invarianti dopo ogni iterazione del ciclo principale:*

1. *Q è stabile rispetto ad ogni blocco di X;*
2. *Q rifinisce X;*
3. *$\mathrm{RSCP}(\mathfrak{V}, E)$ rifinisce Q.*

Dimostrazione. In entrambi i casi procediamo per induzione sulle iterazioni:

1/2. Prima della prima iterazione Q è stabile rispetto a X per l'operazione eseguita alla Riga 3. Inoltre Q rifinisce X perchè quest'ultima è la partizione banale di V.

Ora supponiamo che l'invariante sia valida prima di un'iterazione qualsiasi: durante l'iterazione successiva in X viene sostituito il blocco

S con i blocchi $S - B, B$, mentre Q viene modificato da sue applicazioni successive della funzione `split`. È evidente che Q deve essere stabile rispetto a $S - B, B$, ed anche a tutti gli altri blocchi di X per l'ipotesi induttiva (si ricordi la Proposizione 1.6). Inoltre, sempre per l'ipotesi induttiva, devono esistere (prima della modifica) dei blocchi $C_1, \ldots, C_n \mid \bigcup_{i=1}^{n} C_i = S$. Dopo la modifica di Q l'unione dei nuovi blocchi generati dalle due chiamate di `split` sui blocchi $\{C_1, \ldots, C_n\} - B$ è $S - B$, mentre l'unione dei due nuovi blocchi (di cui uno eventualmente vuoto) generati dalla divisione di B è chiaramente B. Per cui Q rifinisce ancora X.

3. La proprietà è vera banalmente prima della prima iterazione.

   Supponiamo che l'invariante sia valida prima dell'iterazione i-esima: durante l'iterazione successiva la partizione Q viene rifinita utilizzando come *splitter* i blocchi $B, S - B$. Poichè Q rifinisce X, e S è un blocco di X, S è l'unione di alcuni blocchi di Q. Inoltre, essendo B un blocco di Q (con $B \subset S$), anche $S - B$ è unione di alcuni blocchi di Q. Allora, poichè per l'ipotesi induttiva $\mathrm{RSCP}(\mathfrak{V}, E)$ rifinisce Q:

   $$\mathrm{RSCP}(\mathfrak{V}, E) = \texttt{split}(S - B, \texttt{split}(B, \mathrm{RSCP}(\mathfrak{V}, E)))$$

   Quindi, per il Teorema 1.6 (monotonia di `split`), $\mathrm{RSCP}(\mathfrak{V}, E)$ rifinisce la nuova partizione $Q' = \texttt{split}(S - B, \texttt{split}(B, Q))$. □

Dal Lemma 2.4 possiamo dedurre direttamente il seguente risultato:

Corollario 2.1. *L'Algoritmo 4 termina.*

Dimostrazione. Ad ogni iterazione il numero di blocchi in X aumenta di una unità. Quindi in $O(|V|)$ iterazioni si avrà $X = \{\{v\} \mid v \in V\}$ (se l'algoritmo non termina prima). Ma per il Lemma 2.4 Q rifinisce X prima di ogni iterazione, quindi deve essere necessariamente $Q = \{\{v\} \mid v \in V\} = X$. □

A questo punto possiamo dimostrare in modo molto agevole la correttezza dell'algoritmo di Paige-Tarjan, utilizzando gli ingredienti proposti sopra:

Proposizione 2.4. *Al termine dell'Algoritmo 4 vale l'uguaglianza:*

$$Q = \mathrm{RSCP}(E, \mathfrak{V}).$$

Dimostrazione. Per il Lemma 2.4, la partizione Q è sempre stabile rispetto ad ogni blocco di X. Poichè la condizione di terminazione è $Q = X$, Q è sempre stabile rispetto a X, ed in particolare al termine dell'algoritmo. Ma allora, sempre per il Lemma 2.4 (punto 3), al termine dell'algoritmo si ha $Q = \mathrm{RSCP}(\mathfrak{V}, E)$. □

L'efficienza dell'algoritmo esposto all'inizio della sezione presente è garantita da un dettaglio implementativo che non abbiamo ancora reso esplicito. Introduciamo l'intuizione alla base tramite il seguente risultato [15]:

Lemma 2.5. *Valgono le seguenti uguaglianze insiemistiche per la doppia chiamata alla funzione* `split` *alla Riga 8:*

1. $\texttt{split}(B, Q)$ *divide i blocchi* $D \in Q$ *in due blocchi* $D_1 = D \cap E^{-1}(B), D_2 = D - D_1$. D_1, D_2 *sono entrambi non vuoti se e solo se* $D \cap E^{-1}(B) \neq \emptyset \wedge D - E^{-1}(B) \neq \emptyset$;

2. $\texttt{split}(S - B, \texttt{split}(B, Q))$ *divide i blocchi* D_1 *di* $\texttt{split}(B, Q)$ *in due blocchi* $D_{11} = D_1 \cap E^{-1}(S - B), D_{12} = D_1 - D_{11}$. D_{11}, D_{12} *sono entrambi non vuoti se e solo se* $D_1 \cap E^{-1}(S - B) \neq \emptyset \wedge D_1 - E^{-1}(S - B) \neq \emptyset$;

3. $\texttt{split}(S - B, \texttt{split}(B, Q))$ *non modifica i blocchi di tipo* D_2;

4. $D_{12} = D_1 \cap (E^{-1}(B) - E^{-1}(S - B))$.

Dimostrazione. La dimostrazione è banale per i primi due punti (è una conseguenza diretta della Definizione 1.26).

3. Supponiamo che un blocco $D \in Q$ sia stato diviso in due blocchi non vuoti D_1, D_2 da $\texttt{split}(B, Q)$. Allora, per il punto 1, $D \cap E^{-1}(B) \neq \emptyset$. Quindi, per la stabilità di Q rispetto a (E, S), deve valere $D \subseteq E^{-1}(S)$. Allora $D_2 \subset D \subseteq E^{-1}(S)$. Per definizione D_2 e $E^{-1}(B)$ sono disgiunti, quindi $D_2 \subseteq E^{-1}(S - B)$;

4. Evidente nella Figura 7. □

Di conseguenza abbiamo determinato la tipologia di blocco che può essere generata, a partire da un blocco di Q, dalla Riga 8 dell'algoritmo di Paige-Tarjan. Possiamo sfruttare questa informazione per implementare il procedimento in modo efficiente.

Osservazione 2.3. Con opportuni accorgimenti e strutture dati [15] è possibile implementare una funzione che effettua l'operazione $\texttt{split}(S - B, \texttt{split}(B, Q))$ con complessità $O(|B| + \sum_{y \in B} |E^{-1}(\{y\})|)$.

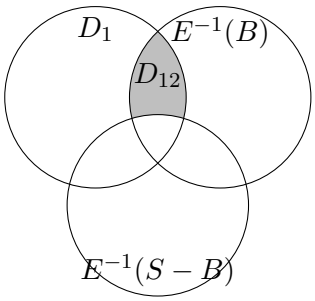


Figura 7: Punto 4. del Lemma 2.5

Da questa implementazione deduciamo il seguente upper-bound sulla complessità dell'Algoritmo di Paige-Tarjan:

Teorema 2.4. *La complessità dell'Algoritmo 4 è* $O(|E|\log|V|)$.

Dimostrazione. Chiaramente la complessità è:

$$\sum_{\substack{B|B \text{ è usato} \\ \text{come } splitter}} O(|B| + \sum_{y\in B} |E^{-1}(\{y\})|)$$
$$= \sum_{\substack{B|B \text{ è usato} \\ \text{come } splitter}} O(|B|) + \sum_{\substack{B|B \text{ è usato} \\ \text{come } splitter}} \sum_{y\in B} |E^{-1}(\{y\})|)$$

Osservazione 2.4. Ogni elemento $x \in S$ può apparire al più $\log_2(|V|+1)$ volte in un blocco usato come *splitter*, visto che ad ogni nuova iterazione un eventuale blocco contentente questo stesso x ha cardinalità dimezzata.

Allora:

$$\sum_{\substack{B|B \text{ è usato} \\ \text{come } splitter}} \sum_{y\in B} |E^{-1}(\{y\})|) \leq \log_2(|V|+1) \sum_{x\in V} |E^{-1}(\{x\})|$$
$$= \log_2(|V|+1)|E|$$
$$= O(|E|\log|V|)$$

Inoltre:

$$\begin{aligned}\sum_{\substack{B|B \text{ è usato}\\ \text{come } splitter}} O(|B|) &\leq \sum_{x \in V} \log_2(|V|+1) \\ &= |V| \log_2(|V|+1) \\ &= O(|V| \log |V|)\end{aligned}$$

È lecito considerare $|V| = O(|E|)$, per cui la complessità è $O(|E| \log |V|)$. □

2.3 Algoritmo di Dovier-Piazza-Policriti

In questa sezione presenteremo l'algoritmo di Dovier-Piazza-Policriti [4], più recente rispetto all'algoritmo di Paige-Tarjan. La differenza fondamentale è l'introduzione di un ordinamento parziale tra i nodi del grafo, il *rango*, che quantifica sostanzialmente la distanza di un nodo dal *pozzo* più lontano raggiungibile percorrendo gli archi (si ricordi che un *pozzo* è un nodo da cui non escono archi). Utilizzando questo ordinamento è possibile rifinire la partizione in modo più oculato, arrivando al risultato con meno chiamate a `split`. La complessità computazionale resta la stessa dell'algoritmo di Paige-Tarjan, ma come vedremo nei risultati sperimentali in molti casi l'algoritmo di Dovier-Piazza-Policriti risulta più economico del suo predecessore.

La sezione è organizzata come segue: innanzitutto daremo alcune nozioni fondamentali per la comprensione dell'algoritmo, e alcuni risultati che risulteranno utili nel seguito; dopodichè esamineremo lo pseudocodice, ed analizzeremo la complessità e la correttezza dell'algoritmo.

2.3.1 Nozioni preliminari

Consideriamo la seguente definizione, che ci consentirà di esprimere in modo più compatto alcune espressioni nel seguito:

Definizione 2.5. Sia $G = (V, E)$ un grafo diretto. Sia $n \in V$. Il grafo $G\big|_n$, letto "*il sottografo di G raggiungibile da n*", è definito come segue:

$$G\big|_n = \left(N\big|_n, E \cap \left(N\big|_n \times N\big|_n\right)\right)$$

dove $N\big|_n$ è il sottoinsieme di V dei nodi raggiungibili da n.

Procediamo con un concetto fondamentale per la definizione del *rango*, l'ordinamento parziale sull'insieme dei nodi menzionato nell'introduzione della sezione:

Definizione 2.6. La parte *ben fondata* (*well founded* in inglese) di un grafo diretto $G = (V, E)$ è:

$$\texttt{WF}(G) = \left\{ n \in V \mid \text{ tale che } G\big|_n \text{ è aciclico} \right\}.$$

La parte *non ben fondata* è definita chiaramente come:

$$\texttt{NWF}(G) = V - \texttt{WF}(G).$$

Vale il seguente risultato immediato (si ricordi che con la notazione $[v], v \in V$ ci riferiamo alla SCC di V cui appartiene il nodo v):

Osservazione 2.5. Condizione necessaria affinchè un vertice n possa appartenere a $WF(G)$ è che $|[n]| = 1$.

Dimostrazione. Supponiamo che $n \in \texttt{WF}(G)$. Questo vuol dire che da tutti i nodi raggiungibili da n non si può ritornare in n (perchè non si hanno cicli). Ma allora non ci può essere un altro nodo nella SCC di n. □

Introduciamo la funzione `rank`, che ricopre un ruolo primario nell'Algoritmo BFA:

Definizione 2.7. Sia $G = (V, E)$ un grafo diretto. La funzione $\texttt{rank} : V \to \mathbb{N} \cup \{0, -\infty\}$ è definita come segue:

$$\texttt{rank}(n) = \begin{cases} 0 & \text{se } n \text{ è un pozzo di } G \\ -\infty & \text{se } [n] \text{ è un pozzo di } G^{\text{SCC}}, \\ & \text{e } n \text{ non è un pozzo di } G \\ \max(\{1 + \texttt{rank}(m) \mid m \in \mathcal{N}_{\text{WF}}(n)\} & \\ \quad \cup \{\texttt{rank}(m) \mid m \in \mathcal{N}_{\text{NWF}}(n)\}) & \text{altrimenti} \end{cases}$$

Abbiamo utilizzato per comodità le funzioni $\mathcal{N}_{\text{WF}}(n), \mathcal{N}_{\text{NWF}}(n)$ definite come segue:

$$\mathcal{N}_{\text{WF}}(n) = \{m \in \texttt{WF}(G) \mid [n] \overset{\text{SCC}}{\to} [m]\}$$
$$\mathcal{N}_{\text{NWF}}(n) = \{m \in V - \texttt{WF}(G) \mid [n] \overset{\text{SCC}}{\to} [m]\}$$

Esse associano ad un nodo $n \in V$ l'insieme delle SCC raggiungibili da $[n]$, rispettivamente contenute in $\texttt{WF}(G), V - \texttt{WF}(G)$.

Osservazione 2.6. Sia $n \in V$, e supponiamo $\texttt{rank}(n) = k$. Allora:

$$\texttt{rank}(m) = k \ \forall m \in [n]$$

Dimostrazione. Se $k = 0$ allora n è una pozzo, per cui $[n]$ ha cardinalità 1, quindi la tesi vale banalmente. Se $k = -\infty$, nessuno dei nodi in $[n]$ può essere un pozzo di G, per cui hanno tutti rango $-\infty$. Infine se $k > 0$ basta osservare che il rango massimo si "propaga" tra nodi della stessa SCC, com'è evidente dalla Definizione 2.7. □

Se un grafo è *ben fondato*, o se restringiamo il dominio ad un sottografo *ben fondato*, possiamo dare una formulazione alternativa della funzione `rank`:

$$\texttt{rank}(n) = \begin{cases} 0 & \text{se } n \text{ è un pozzo di } G \\ 1 + \max\{\texttt{rank}(m) \mid nEm\} & \text{altrimenti} \end{cases}$$

Questa forma sarà utile per semplificare alcune delle dimostrazioni che seguono.

Riportiamo e dimostriamo alcuni risultati [4] che renderanno più agile l'analisi successiva. Il motivo per cui troviamo interessante introdurre il rango nella trattazione algoritmica della massima bisimulazione è sintetizzato dal seguente risultato:

Teorema 2.5. *Sia $G = (V, E)$. Sia "$\equiv$" la massima bisimulazione su G. Siano $m, n \in V$. Allora*

$$m \equiv n \implies \texttt{rank}(m) = \texttt{rank}(n)$$

La dimostrazione del Teorema 2.5 è piuttosto lunga, per cui abbiamo preferito riportarla a pezzi. Cominciamo con la prima parte:

Proposizione 2.5. *Sia $G = (V, E)$. Sia "$\equiv$" la massima bisimulazione su G. Siano $m, n \in \texttt{WF}(G)$. Allora $m \equiv n \implies \texttt{rank}(m) = \texttt{rank}(n)$.*

Dimostrazione. Procediamo per induzione su $\texttt{rank}(m)$. Si ricordi che per il Teorema 1.4 gli APG aventi origine in m, n rappresentano lo stesso insieme. Se $\texttt{rank}(m) = 0, m$ rappresenta l'insieme vuoto. Ma allora lo stesso vale per n, per cui anche $\texttt{rank}(n) = 0$.

Supponiamo (ipotesi induttiva) che due nodi bisimili di rango minore o uguale a $k - 1$ abbiano sempre rango uguale. Sia $\texttt{rank}(m) = k$, e sia $m' \in V$ tale che $\langle m, m' \rangle \in E, \texttt{rank}(m') = k - 1$. Allora deve esistere un nodo n' tale che $\langle n, n' \rangle \in E, m' \equiv n'$, e quindi $\texttt{rank}(m') = \texttt{rank}(n')$. Ma allora $\texttt{rank}(n) \geq k - 1 + 1 = k$, per la formulazione alternativa di `rank`. Analogamente si dimostra che $\texttt{rank}(m) \geq k$, per cui si ha la tesi. □

La seconda parte del Teorema 2.5 richiede un certo numero di sotto-risultati preliminari:

Proposizione 2.6. *Sia $G = (V, E)$. Sia $m \in V$. Allora* $\texttt{rank}(m) = -\infty$ *se e solo se l'APG $G\big|_m$ rappresenta l'insieme Ω.*

Dimostrazione. Supponiamo che $\texttt{rank}(m) = -\infty$. Si ricordi la caratterizzazione dell'insieme Ω (cioè l'insieme che contiene solamente se stesso [1]): un APG rappresenta Ω se e solo se ogni nodo ha almeno un arco uscente. Dalle ipotesi fatte si deduce che $[m]$ non contiene solamente m (altrimenti m sarebbe un pozzo di G). Possiamo dimostrare che ogni nodo in $[m]$ deve avere almeno un arco uscente: se così non fosse il nodo non avrebbe modo di tornare all'interno della SCC $[m]$, per cui non potrebbe stare nella stessa SCC di m.

Supponiamo ora che l'APG $G\big|_m$ rappresenti l'insieme Ω. Chiaramente m non può essere un pozzo di G, altrimenti non rispetterebbe la caratterizzazione. Supponiamo per assurdo che $[m]$ non sia un pozzo di G^{SCC}, cioè da un nodo di $[m]$ parte un arco verso un nodo $x \notin [m]$. Si presentano quattro possibilità:

1. Da x non esce alcun nodo: impossibile per la caratterizzazione di Ω;
2. Da x esce un unico nodo che ha per destinazione x stesso: $[x]$ contiene solamente x, ed è un pozzo di G^{SCC}, per cui ha rango $-\infty$;
3. Da x esce un nodo che ha per destinazione un nodo di $[m]$: impossibile perchè si è supposto $x \notin [m]$;
4. Da x esce un nodo che ha per destinazione un nodo $x' \notin [m]$.

Per cui è evidente che le uniche opzioni valide sono la 2. e la 4. Possiamo dimostrare che il blocco di G^{SCC} contenente una sequenza di nodi connessi secondo queste due regole è necessariamente un pozzo di G^{SCC}: è possibile aggiungere un numero arbitrario di nodi che le rispettino, ma l'ultimo nodo dovrà avere un *self loop*, oppure tornare in un nodo precedente della sequenza. In entrambi i casi i nodi coinvolti ottengono rango $-\infty$, per cui a cascata (per il terzo caso nella definizione di `rank`) i nodi precedenti della sequenza hanno tutti rango $-\infty$. □

La seguente osservazione è una riformulazione di una deduzione proposta nella Sezione 1:

Osservazione 2.7. Sia $G = (V, E)$. Siano $m, n \in V$. Allora:

$$\texttt{rank}(m) = \texttt{rank}(n) = 0 \implies m \equiv n$$

Dimostrazione. m, n sono pozzi di G, per cui rappresentano l'insieme $\emptyset$. Allora m, n sono bisimili per il Teorema 1.4. □

Possiamo dunque enunciare e dimostrare la parte mancante della dimostrazione:

Proposizione 2.7. *Con le stesse ipotesi della Proposizione 2.5, supponiamo che* $m, n \in V - \texttt{WF}(G)$. *Allora* $m \equiv n \implies \texttt{rank}(m) = \texttt{rank}(n)$.

Dimostrazione. Sempre per il Teorema 1.4, m, n rappresentano lo stesso insieme. Se $\texttt{rank}(m) = -\infty$, per la Proposizione 2.6 m rappresenta Ω, per cui anche n rappresenta Ω. Ma allora $\texttt{rank}(n) = -\infty$.

Se $\texttt{rank}(m) = h > 0$, per come è stato definita la funzione `rank` deve esistere un nodo *ben fondato* m' raggiungibile da m, non necessariamente in modo diretto, tale che $\texttt{rank}(m') = h - 1$ (il rango aumenta solo in corrispondenza di archi verso nodi *ben fondati*). Poichè $m \equiv n$ deve esistere un nodo *ben fondato* n' raggiungibile da n tale che $m' \equiv n'$. Ma allora $(m', n' \in \texttt{WF}(G))$ $\texttt{rank}(n') = \texttt{rank}(m') = h - 1$, e quindi $\texttt{rank}(n) \geq \texttt{rank}(m) = h$. Analogamente si dimostra che $\texttt{rank}(m) \geq \texttt{rank}(n)$, per cui si ha la tesi. □

Dimostrazione. [Teorema 2.5] Due nodi bisimili devono essere entrambi *ben fondati* oppure entrambi *non ben fondati*. La tesi segue dunque dalle Proposizioni 2.5, 2.7. □

Dal Teorema 2.5 deduciamo che due nodi aventi rango differente sono sicuramente **non** bisimili. Inoltre nodi a basso rango non sono influenzati da quanto avviene ai nodi ad alto rango; per cui, invece di procedere in modo randomico, possiamo utilizzare la funzione `split` sui blocchi in ordine crescente di rango. Ci aspettiamo con molta probabilità un risparmio di qualche tipo in termini di tempo di esecuzione rispetto all'algoritmo di Paige-Tarjan, in particolare su grafi che mostrano una struttura di qualche tipo; grafi generati in modo randomico, invece, hanno più probabilità di mostrare una varianza del rango poco significativa, per cui è possibile che l'algoritmo di Dovier-Piazza-Policriti impieghi più tempo. Come sarà possibile osservare sulla base dello pseudocodice che proporremo nel prossimo paragrafo, l'algoritmo di Dovier-Piazza-Policriti parte da una partizione iniziale data dai blocchi di nodi aventi lo stesso rango. Meglio la relazione binaria "stesso rango" approssima la massima bisimulazione, meno cambiamenti sarà necessario operare su questa partizione iniziale, o equivalentemente, più l'algoritmo sarà performante.

Algoritmo 5: Compute-Rank

```
   Data: G = (V, E)
 1 function dfs-rank-visit(G = (V, E), n):
 2 |   n.color = GRAY;
 3 |   if |E(n)| == 0 then
 4 |   |   n.rank = 0;
 5 |   else
 6 |   |   max-rank = −∞;
   |   |   // Visitiamo E(n) in ordine decrescente di ft.
 7 |   |   foreach v ∈ E(n) do
 8 |   |   |   if v.color == WHITE or v.color == GRAY or !v.wf
   |   |   |    then
 9 |   |   |   |   n.wf = false;
10 |   |   |   if v.color == WHITE then
11 |   |   |   |   dfs-rank-visit(G, v);
12 |   |   |   if v.rank != None then
13 |   |   |   |   if v.wf then
14 |   |   |   |   |   n.rank = max(n.rank, v.rank + 1);
15 |   |   |   |   else
16 |   |   |   |   |   n.rank = max(n.rank, v.rank);
17 |   n.color = BLACK;
18 begin
19 |   DFS(G^{-1});
   |   // Resettiamo i colori prima della seconda DFS.
20 |   foreach n ∈ V do
21 |   |   n.color = WHITE;
   |   // Visitiamo V in ordine decrescente di ft.
22 |   foreach n ∈ V do
23 |   |   if n.color == WHITE then
24 |   |   |   dfs-rank-visit(G, n);
```

Riportiamo un algoritmo per il calcolo del rango dei nodi di un grafo diretto [4]. Ne analizziamo brevemente le caratteristiche discutendo alcuni semplici risultati:

Osservazione 2.8. L'Algoritmo 5 termina, essendo una DFS. Inoltre, per lo stesso motivo, l'algoritmo ha complessità $O(|V| + |E|)$.

Nell'analisi dell'Algoritmo 5 indicheremo con "v.rank" il rango del nodo v

impostato dall'algoritmo, e con "$\texttt{rank}(v)$" il rango corretto. Inoltre useremo la notazione "*v.ft*" per indicare il *finishing-time* del nodo v relativo alla DFS su G^{-1} (Riga 19).

Teorema 2.6. *Dopo l'esecuzione dell'Algoritmo 5 il rango di ogni nodo di G è corretto, cioè $\forall v \in V$ vale* $\texttt{rank}(v) = v.\text{rank}$.

Proponiamo innanzitutto alcuni risultati preliminari che consentono di semplificare la trattazione:

Osservazione 2.9. Se $\langle v, u\rangle \in E$ e $v.ft > u.ft$, allora $u \in [v]$.

Dimostrazione. Chiaramente da v è possibile raggiungere u per ipotesi. Consideriamo la DFS su G^{-1}: ovviamente si avrà $\langle u, v\rangle \in E^{-1}$. Ma se v fosse stato raggiunto per la prima volta durante la visita in profondità di u, avremmo $u.ft > v.ft$. Allora v viene raggiunto *prima* di u, e l'unica possibilità per cui si possa avere $v.\text{ft} > u.\text{ft}$ è che esista un percorso da v a u (in G^{-1}). Procedendo al contrario lungo questo cammino si verifica che da u è possibile raggiungere v. □

L'osservazione può essere riformulata in modo più esplitito:

Corollario 2.2. *Se durante la visita dell'immagine v si incontra un vertice u di colore bianco, v è raggiungibile da u e viceversa.*

E generalizzata:

Corollario 2.3. *Se $v.ft > u.ft$, e u è raggiungibile da v, allora $u \in [v]$.*

Dall'Osservazione 2.9 possiamo dedurre ancora alcuni risultati immediati:

Corollario 2.4. *Se $\langle v, u\rangle \in E$ e $v.ft > u.ft$, allora $u, v \in \texttt{NWF}(G)$.*

Si osservi che da quest'ultimo corollario si deduce che il criterio usato dall'Algoritmo 5 per impostare il campo "*wf*" dei nodi è corretto.

Corollario 2.5. *Il* finishing-time *dei nodi di una SCC è un intervallo di interi senza buchi.*

Dimostrazione. Supponiamo per assurdo che esista un nodo x che trasgredisca questa proprietà ($x.ft = i$). Supponiamo che $v_1, \ldots, v_n \in [v]$ (in ordine decrescente di *finishing-time*), e che $v_n.ft = i - 1$. Inoltre supponiamo che ci siano altri nodi in $[v]$, tutti con *finishing-time* minore di $v_n.ft$. Chiaramente deve valere $\langle v_n, x\rangle \in E^{-1}$, o equivalentemente $\langle x, v_n\rangle \in E$. Allora $x \in [v]$ per l'Osservazione 2.9. □

Per comodità, nel seguito adotteremo la seguente notazione:

$$[v].ft^* := \max_{x \in [v]} x.ft$$
$$[v].ft_* := \min_{x \in [v]} x.ft$$

Deduciamo il seguente risultato:

Proposizione 2.8. `rank`(v) *dipende esclusivamente dal rango dei nodi* u *che soddisfano la seguente proprietà:*

$$u \in [v] \vee u.ft > [v].ft^* \tag{2.2}$$

Dimostrazione. Sia x un nodo per cui non vale nessuna delle due condizioni, quindi $u.ft < [v].ft_*$. Allora dai nodi di $[v]$ non è possibile raggiungere u (nemmeno con cammini più lunghi di un arco): se per assurdo fosse possibile, per il Corollario 2.3 si avrebbe $u \in [v]$; allora, per la definizione di `rank`, si ha la tesi. □

Abbiamo spiegato perchè risulta conveniente operare la DFS su G in ordine decrescente di *finishing-time*: ad ogni passo saranno già disponibili le uniche informazioni necessarie, cioè il rango dei nodi con *finishing-time* maggiore.

A questo punto abbiamo tutti gli ingredienti necessari per dimostrare il teorema che avevamo lasciato in sospeso, relativo alla correttezza dell'Algoritmo 5:

Dimostrazione. [Teorema 2.6] Sia v un nodo. Se $v.rank = 0$, allora v è un pozzo.

Se $v.rank \neq 0$, poichè il rango viene calcolato come da definizione, è sufficiente dimostrare che al momento della visita di $E(v)$ si ha $u.rank$ = `rank`(u), $\forall u \in E(v)$. Chiaramente gli unici nodi problematici sono quelli di colore grigio al momento della visita, in quanto l'esplorazione della loro immagine non è ancora terminata. Questi nodi stanno in $[v]$, e per il Corollario 2.5 non ci sono nodi esterni alla SCC nell'intervallo, se si considera il *finishing-time*. Ma allora tutti i nodi il cui rango è rilevante sono già stati visitati (Proposizione 2.8). □

2.3.2 L'algoritmo

Per comodità di notazione poniamo $\mathbb{N}^* \coloneqq \mathbb{N} \cup \{-\infty, 0\}$. Presentiamo e commentiamo lo pseudocodice dell'Algoritmo di Dovier-Piazza-Policriti:

Algoritmo 6: Algoritmo di Dovier-Piazza-Policriti

```
   Data: G = (V, E)
 1 function collapse(G = (V, E), B ⊆ V):
 2 |   Sia u ∈ B scelto casualmente;
 3 |   foreach v ∈ B − {u} do
 4 |   |   foreach z ∈ E(v) do
 5 |   |   |   E = (E − {⟨v, z⟩}) ∪ {⟨u, z⟩};
 6 |   |   foreach z ∈ E^{-1}(v) do
 7 |   |   |   E = (E − {⟨z, v⟩}) ∪ {⟨z, u⟩};
 8 |   |   V = V − {v};
 9 |   return u;
10 function split2(G = (V, E), P, u, B̂):
11 |   foreach B ∈ P | B ∈ B̂ do
12 |   |   P = P − {B} ∪ {{v ∈ B | ⟨v, u⟩ ∈ E},
13 |   |   |       {v ∈ B | ⟨v, u⟩ ∉ E}};
14 begin
15 |   compute-rank(G);
16 |   ρ ≔ max{rank(n) | n ∈ V};
17 |   B_k ≔ {n ∈ V | rank(n) = k},  k ∈ ℕ*;
18 |   P ≔ {B_i | i ∈ ℕ*};
   |   // u è il nodo di B_{-∞} preservato da collapse.
19 |   u ≔ collapse(G, B_{-∞});
20 |   split2(G, P, u, ⋃_{i=0}^{ρ} B_i);
21 |   foreach i = 0, ..., ρ do
   |   |   // Selezioniamo i blocchi di P aventi rango i.
22 |   |   D_i ≔ {B ∈ P | B ⊆ B_i};
   |   |   // Il sottografo di G dei nodi di rango i.
23 |   |   G_i = (B_i, (B_i × B_i) ∩ E);
   |   |   // D_i è una partizione di G_i.
24 |   |   D_i = Paige-Tarjan(G_i, D_i);
25 |   |   foreach B ∈ D_i do
26 |   |   |   u ≔ collapse(G, B);
27 |   |   |   split2(G, P, u, ⋃_{j=i+1}^{ρ} B_j);
```

La funzione `collapse` rimuove dal grafo tutti i nodi all'interno di un blocco, ad eccezione di un nodo scelto in modo casuale; dopodichè sostituisce

il nodo mantenuto in ogni arco incidente ad uno dei nodi rimossi. Viene utilizzata per ridurre ad un solo vertice un blocco per cui si è già stabilito che tutti i nodi sono bisimili.

La funzione `split2` è analoga alla funzione `split`, ma consente di specificare i blocchi che possono essere modificati, solo quelli in $\widehat{B}$, e di ignorare quelli non appartenenti a $\widehat{B}$; inoltre lo `split` viene effettuato rispetto ad un unico vertice, infatti ogni $B \in \widehat{B}$ viene rimpiazzato in P da due blocchi $\{v \in B \mid \langle v, u\rangle \in E\}$ e $\{v \in B \mid \langle v, u\rangle \notin E\}\}$, che sono rispettivamente la notazione non insiemistica degli insiemi $B \cap E^{-1}(\{u\}), B - E^{-1}(\{u\})$. Ci si aspetta che u sia il nodo "soppravvissuto" da una chiamata a `collapse`.

L'algoritmo inizia con il calcolo del rango dei nodi del grafo. Alla Riga 18 viene creata una partizione iniziale, da rifinire, i cui blocchi sono composti dai nodi aventi lo stesso rango. Per il Teorema 2.5 la RSCP sarà sicuramente una rifinitura di questa partizione.

Per la Proposizione 2.6 possiamo impunemente considerare bisimili tutti i nodi di rango $-\infty$. Per questo motivo alla Riga 19 viene collassato il blocco B_∞. Si noti che un'assunzione del genere non è valida per altri valori del rango, in quanto nodi non bisimili possono avere lo stesso rango. A questo punto è necessario aggiornare la partizione e dividere ogni blocco che non rispetta la condizione di stabilità (Riga 20).

In seguito, per ogni rango a partire 0, si considerano i blocchi di rango i e si isola il sottografo contenente solamente nodi composti dai nodi che vi sono contenuti. Si applica l'algoritmo di Paige-Tarjan a questo sottografo, al fine di calcolarne la RSCP. I blocchi di questa nuova partizione vengono collassati, ed analogamente a quanto fatto in precedenza si impone ai blocchi di rango superiore la condizione di stabilità. Si nodi che ad ogni iterazione il numero di blocchi e nodi influenzati dalla chiamata a `split2` si riduce, coerentemente con quanto abbiamo anticipato nella Sezione 2.3.1.

2.3.3 Analisi

Come abbiamo fatto per gli algoritmi di Hopcroft e Paige-Tarjan, verifichiamo la correttezza e la complessità dell'Algoritmo 6.

Teorema 2.7. *Due nodi m, n vengono collassati in un unico nodo durante l'esecuzione dell'Algoritmo 6 se e solo se sono bisimili.*

Dimostrazione. Siano $m, n \in V \mid m \equiv n$. Per il Teorema 2.5 m, n devono necessariamente stare all'interno dello stesso blocco nella partizione iniziale creata alla Riga 18. Dimostriamo ora che durante l'esecuzione dell'algoritmo m, n verranno collassati in un unico nodo.

Se $\texttt{rank}(m) = \texttt{rank}(n) = -\infty$ il blocco contenente questi due nodi viene collassato alla Riga 19.

Per i valori positivi del rango, procediamo per induzione. Se $\texttt{rank}(m) = \texttt{rank}(n) = 0$ inizialmente appartengono entrambi a B_0. Sicuramente non vengono divisi da `split2` alla Riga 20, e l'intero blocco B_0 viene collassato alla Riga 26. Se $\texttt{rank}(m) = \texttt{rank}(n) = k > 0$, poichè abbiamo supposto che $m \equiv n$, si ha che $\langle m, m' \rangle \in E \implies \exists n' \mid \langle n, n' \rangle \in E \wedge \langle m', n' \rangle \in E$ (e lo stesso vale partendo da n). Questo significa che l'esistenza di un arco uscente da uno dei due nodi implica l'esistenza di un altro arco, uscente dall'altro nodo, e che le due destinazioni sono bisimili. Ma allora m, n non possono essere stati divisi nei passaggi precedenti, perchè per l'ipotesi induttiva i nodi bisimili tra loro di rango inferiore a k appartengono allo stesso blocco all'inizio dell'iterazione i-esima del ciclo alla Riga 21. Non c'è ancora stata alcuna divisione di blocchi indotta da nodi di rango uguale a k (gli unici verso cui un nodo di rango k può avere un arco, oltre ai nodi di rango strettamente minore di k), per cui m, n sono necessariamente nello stesso blocco all'inizio dell'iterazione i-esima. Per la correttezza dell'Algoritmo di Paige-Tarjan, m, n appartengono ancora allo stesso blocco quando questo viene collassato alla Riga 26. In modo analogo si dimostra che se due nodi appartengono allo stesso blocco quando questo viene collassato, allora sono necessariamente bisimili. □

Quest'ultimo risultato dimostra chiaramente la correttezza dell'algoritmo di Dovier-Piazza-Policriti. Prima di valutare la complessità, discutiamo il costo delle funzioni ausiliarie:

Osservazione 2.10. Un'implementazione efficiente congiunta delle funzioni `collapse` e `split2` ha complessità $\Theta(|E^{-1}(B)|)$, dove B è il blocco su cui sono chiamate una dopo l'altra.

Dimostrazione. È sufficiente creare inizialmente l'insieme $E^{-1}(\{v\})\, \forall v \in V$, cioè la contro-immagine di v rispetto alla relazione binaria E. Per collassare il blocco B e ricalcolare la partizione, cioè per eseguire le due seguenti righe di pseudocodice:

$$u \coloneqq \texttt{collapse}(G, B);$$
$$\texttt{split2}(G, P, u, \bigcup_{j=i+1}^{\rho} B_j);$$

è sufficiente visitare in modo esaustivo l'insieme $E^{-1}(B)$ e per ogni blocco C della partizione iniziale (tra quelli contenuti in $\bigcup_{j=i+1}^{\rho} B_j$) raggiunto da

questa visita rimuovere da C i nodi che rientrano nella contro-immagine di B, ed inserirli in un nuovo blocco. □

Teorema 2.8. *L'Algoritmo 6 ha complessità* $O(|E| \log |V|)$.

Dimostrazione. Il rango dei nodi del grafo si calcola con l'Algoritmo 5, che ha complessità $O(|V| + |E|)$. Per l'osservazione precedente le operazioni alle Righe 19, 20 hanno un'implementazione lineare.

Il ciclo alla Riga 21 è composto da tre elementi:

1. La creazione dell'insieme D_i e del sottografo $G\big|_i$;
2. La chiamata all'Algoritmo PTA;
3. Il ciclo sui blocchi appartenenti all'insieme D_i.

Gli elementi di tipo 1 sono chiaramente dominati dagli elementi di tipo 2, in quanto determinare il sottografo $G\big|_i$ ha complessità $O(|B_i| + |E \cap (B_i \times B_i)|)$, e lo stesso vale per gli elementi di tipo 3. La complessità degli elementi di tipo 2 invece è $O(|E \cap (B_i \times B_i)| \log |B_i|)$, come abbiamo già dimostrato (Teorema 2.4).

Ne segue che la complessità dell'algoritmo è:

$$\begin{aligned} T(V,E) &= O(|V| + |E|) + \sum_{i=0}^{\rho} O(|E \cap (B_i \times B_i)| \log |B_i|) \\ &= O(|V| + |E|) + O(\log |V|) \sum_{i=0}^{\rho} O(|E \cap (B_i \times B_i)|) \\ &= O(|V| + |E|) + O(\log |V||E|) \\ &= O(|E| \log |V|) \end{aligned}$$

□

Come abbiamo anticipato sopra, la complessità identica all'algoritmo di Paige-Tarjan non rende l'algoritmo di Dovier-Piazza-Policriti poco interessante: quest'ultimo infatti dovrebbe risultare molto più performante qualora la partizione iniziale $P = \{B_k \mid k \in \mathbb{N}\}$ contenga molti blocchi di piccola dimensione. È quello che succede ad esempio con gli alberi bilanciati. È necessario comunque tenere in considerazione il calcolo preliminare del rango, che potrebbe rendere l'algoritmo più performante solo asintoticamente.

2.4 Algoritmo incrementale di Saha

Gli algoritmi trattati finora consentono di determinare la RSCP (e quindi la massima bisimulazione) a partire da un grafo e da una eventuale partizione iniziale. Una modifica di tale grafo, ad esempio l'aggiunta di un arco, comporta necessariamente una nuova esecuzione dell'algoritmo al fine di calcolare la nuova RSCP, senza che venga utilizzata in qualche modo la conoscenza della RSCP prima della modifica. L'algoritmo di Saha [17] consente di ridurre (in alcuni casi) la complessità del ricalcolo della RSCP, qualora sia nota la RSCP del grafo prima della modifica.

Nel seguito useremo la seguente notazione: posponendo un apice singolo ad un simbolo $(G', X', \dots)$ si intende designare l'oggetto matematico rappresentato dal simbolo senza apice, dopo l'applicazione della modifica. Ad esempio, G' è il grafo G con l'aggiunta di un nuovo arco. Inoltre, per mantenere la notazione proposta nell'articolo, indicheremo con X la RSCP di G (e dunque con X' la RSCP del grafo in seguito alla modifica).

Scriveremo inoltre "$A \implies B$" se A, B sono blocchi di una stessa partizione e $\exists a \in A, b \in B \mid \langle a, b\rangle \in E$; in questo caso diremo che A è un *predecessore* di B, e che B è un *successore* di A. Usiamo la notazione $\texttt{Pre}_\Sigma(A)$ per denotare l'insieme dei *predecessori* di A appartenenti alla partizione Σ. Qualora la partizione cui questi blocchi appartengono possa essere omessa senza ambiguità non aggiungeremo il subscritto per alleggerire la notazione. In modo analogo useremo la notazione $\texttt{Succ}_\Sigma(A)$.

L'algoritmo si compone di tre fasi. Indicheremo con X_i la partizione ottenuta dopo la fase i-esima:

1. `split`: Ha lo scopo di dividere i nodi appartenenti ad uno stesso blocco di X che non sono più bisimili dopo la modifica;

2. `merge`: Si tenta di unire in un unico blocco i blocchi di X_1 che contengono nodi diventati bisimili in seguito alla modifica tentando di minimizzare l'errore. Come vedremo infatti il metodo utilizzato per operare questa fase non garantisce risultati corretti, e dunque potremmo avere nodi non bisimili che vengono a trovarsi nello stesso blocco;

3. `split`: Si rifinisce la partizione X_2 rimuovendo l'errore introdotto nella seconda fase.

Nella sezione seguente esporremo nel dettaglio il funzionamento delle singole fasi.

2.4.1 Risultati preliminari e idea fondante

Innanzitutto motiviamo le varie fasi dell'algoritmo, evidenziando i problemi risolti da ognuna e chiarendo le necessarie differenze nell'approccio in base a come si integra il nuovo arco all'interno del grafo iniziale. Possiamo individuare due casi, che vanno gestiti in modo sostanzialmente diverso. Nel seguito chiameremo $u, v \in V$ i due nodi coinvolti dall'inserimento in $G = (V, E)$ del nuovo arco $\langle u, v\rangle \in E'$.

La prima fase, comune ad entrambi i casi, consiste nell applicazione di una variante dell'algoritmo di Paige-Tarjan. La differenza rispetto all'algoritmo originale consiste nella scelta dei blocchi *splitter*, che vengono scelti in ordine crescente di rango. Inoltre prima dell'avvio dell'algoritmo la partizione X viene rifinita con l'applicazione della procedura `split`$([v]_X, X)$, dove v è il nodo destinazione del nuovo arco. Come evidenziamo nello pseudocodice che proporremo tra alcune pagine, a questo punto il nuovo arco $\langle u, v\rangle$ è già stato inserito, sicchè la chiamata a `split`$([v]_X, X)$ in generale può modificare la partizione X (ad esempio, $[u] \cap E^{-1}([v])$ è non vuoto). Nel seguito ci riferiremo alla versione modificata dell'algoritmo di Paige-Tarjan che abbiamo appena caratterizzato con il nome "`ranked_split`(B, P)", dove P è la partizione rifinita e B il blocco utilizzato come *splitter*.

Vale la seguente caratterizzazione di X_1:

Proposizione 2.9. *Siano $u, v \in V'$ due nodi non bisimili di G'. Allora u, v appartengono a blocchi diversi di X_1.*

Ricordiamo che X_1 è la partizione ottenuta al termine della prima fase di *Split*.

Dimostrazione. Segue dalla correttezza dell'Algoritmo di Paige-Tarjan (Teorema 2.4). □

Non possiamo fermarci ad X_1 per ovvi motivi: essa è la RSCP di G' con partizione iniziale X, e quest'ultima è una rifinitura della "vera" partizione iniziale. In altre parole, X_1 è la RSCP per una partizione iniziale più "restrittiva": per questo motivo possiamo sperare che la soluzione effettiva sia in realtà *più grossolana* rispetto ad X_1.

Durante la seconda fase cerchiamo appunto di riformare questa partizione più grossolana unendo alcuni dei blocchi di X_1. Dimostreremo il seguente risultato:

Proposizione 2.10. *Se $a, b \in V$ sono bisimili, allora $[a]_{X_2} = [b]_{X_2}$.*

Come abbiamo anticipato sopra, a partire da questo punto l'algoritmo prevede due casi:

1. Il nuovo arco non genera un nuovo ciclo in G';

2. Il nuovo arco genera un nuovo ciclo.

Li affrontiamo separatamente, evidenziando le differenze e le motivazioni alla base degli approcci adottati per attaccare, algoritmicamente parlando, il problema. Si noti che per determinare l'esistenza di un nuovo ciclo è sufficiente una DFS su G^{-1} a partire da u. Nel caso in cui questa visita raggiunga v possiamo concludere che è stato formato un nuovo ciclo. Per motivi che esamineremo nel seguito risulta conveniente lasciar proseguire la visita finchè non si arresta spontaneamente, e non fermare l'esecuzione appena v viene raggiunto come sarebbe più logico.

Supponiamo dapprima che il nuovo arco non generi un nuovo ciclo. In questo caso utilizziamo per la seconda fase l'algoritmo implementato nella funzione `merge_phase`, il cui pseudo-codice verrà proposto tra alcune pagine. Ne spieghiamo brevemente il funzionamento: supponiamo che durante il primo calcolo della RSCP, a causa dell'assenza dell'arco $\langle u, v\rangle$, i blocchi U, U_1 siano stati divisi durante un'applicazione di `split`; l'aggiunta dell'arco sana questa mancanza, per cui possiamo riformare il blocco originale. Si noti che questo potrebbe portare all'unione a cascata di altre coppie di blocchi, che sono state divise in X perchè U, U_1 erano divisi.

L'algoritmo inizia dunque con una visita di tutti i *predecessori* di $[v]_{X_1}$: se in questo insieme è presente un blocco che possa essere unito a $[u]_{X_1}$, i due blocchi vengono uniti; ricorsivamente poi si verifica se per una coppia di *predecessori* della coppia unita l'unione è diventata possibile.

Si osservi che in questo modo vengono presi in considerazione tutti i blocchi la cui unione diventa possibile in seguito all'aggiunta del nuovo arco: se per assurdo esistesse un blocco che teoricamente dovrebbe essere unito ad un altro, ma che non viene raggiunto dall'algoritmo che implementa la fase di *merging*, esso non sarebbe *predecessore* (nemmeno indiretto) di $[u]_{X_1}, [v]_{X_1}$, nè potrebbe essere unito a un *predecessore* di uno di quei due blocchi. È chiaro che una tale perturbazione nella RSCP non può, dunque, essere stata causata dall'aggiunta dell'arco $\langle u, v\rangle$. Da questa considerazione segue la validità della Proposizione 2.10 per questo primo caso.

A questo punto necessitiamo di una condizione operativa per verificare se due blocchi possano o meno essere uniti. Omettiamo due condizioni banali, che è comunque bene tenere presenti, ovvero l'appartenenza allo stesso

blocco della partizione iniziale, e l'avere lo stesso rango. Diamo innanzitutto la seguente definizione, che come si vedrà nel seguito avrà un ruolo fondamentale all'interno dell'algoritmo:

Definizione 2.8. Siano $B, B' \in X_1$. Un blocco $C \in X_1$ è un *causal splitter* di B_1, B_2 se valgono le seguenti condizioni:

1. $C \in X'$;
2. $B \implies C \wedge B' \not\Rightarrow C$ oppure $B' \implies C \wedge B \not\Rightarrow C$.

Un *causal splitter* di due blocchi B_1, B_2 è intuitivamente uno dei blocchi che potrebbe aver determinato la divisione di B_1, B_2. Dimostriamo alcune proprietà che consentono una comprensione migliore di questo concetto, e che saranno usate tacitamente nel seguito della trattazione:

Osservazione 2.11. Siano B_1, B_2, B_3 tre blocchi, e sia C un blocco che *non è* un *causal splitter* di B_1, B_2 nè di B_2, B_3. Allora:

- C non è un *causal splitter* di B_1, B_3;
- Sia $B_4 \coloneqq B_1 \cup B_2$. C non è un *causal splitter* di B_3, B_4;
- Consideriamo i blocchi B_1, B_2, C_1, C_2. Supponiamo che C_1 non sia un *causal splitter* di B_1, B_2. Allora $C_1 \cup C_2$ è un *causal splitter* di B_1, B_2 se e solo se C_2 è un *causal splitter* di B_1, B_2.

Dimostrazione. I tre punti sono conseguenze elementari della definizione. □

Corollario 2.6. *Siano $B, B' \in X_1$. Se esiste un* causal splitter*, allora i nodi di B e B' non sono bisimili.*

La prima richiesta nella Definizione 2.8 è un punto estremamente critico: considerare come *causal splitter* un blocco che non siamo sicuri faccia effettivamente parte di X' (che, ricordiamo, è la RSCP del grafo dopo l'aggiunta del nuovo arco) significa rinunciare ad unire due blocchi. Se in un passaggio successivo della seconda fase tale *causal splitter* fosse unito ad un altro blocco, la condizione 2. potrebbe non essere più valida.

Per questo motivo, nel caso in cui non fossimo sicuri dell'effettiva appartenenza di un blocco ad X' questo verrà escluso dall' insieme dei *causal splitter*, ragion per cui si rende necessaria la terza fase: ignorare un *causal splitter* equivale a considerare possibile l'unione di due blocchi senza essere sicuri che essi soddisfino effettivamente il criterio fornito dal Corollario 2.6.

Dimostriamo ora che, nel caso in cui il nuovo arco non generi un nuovo ciclo, la terza fase di *splitting* non è necessaria. Come vedremo nel seguito, ciò equivale a dire che nella seconda fase (soltanto in questo caso particolare) non viene introdotta alcuna approssimazione [17].

Proposizione 2.11. *Supponiamo che durante un'iterazione della fase di* merging *si renda necessario valutare la possibilità di unire due blocchi A, B. Se il nuovo arco non genera un nuovo ciclo all'interno del grafo, tutti i blocchi* successori *di A, B appartengono ad X'.*

Abbiamo allora il seguente risultato fondamentale:

Proposizione 2.12. *Nel caso in cui il nuovo arco non generi un nuovo ciclo all'interno del grafo, si ha $X_2 = X'$.*

Dimostrazione. Per la Proposizione 2.11 per ogni coppia $(A, B) \in P$ di blocchi incontrati durante la fase di *merging* tutti i successori di A, B sono blocchi di X' (ed in generale non di X, in quanto potrebbero essere stati modificati). Ciò significa che non viene introdotta alcuna approssimazione nella seconda fase, e dunque se due blocchi vengono uniti (secondo il criterio esposto sopra) ciò avviene perchè niente impedisce (o impedirà) questa operazione. Poichè i blocchi vengono uniti in modo esaustivo la partizione risultante è la RSCP. □

Chiaramente da questo segue la Proposizione 2.10, ed anche il seguente corollario che chiude la trattazione del primo caso:

Corollario 2.7. *Nel caso in cui il nuovo arco non generi un nuovo ciclo la terza fase non è necessaria.*

Procediamo ora con alcune considerazioni sul caso, che potremmo definire "non semplice", in cui il nuovo arco generi un nuovo ciclo in G'. Come vedremo si rende effettivamente necessaria la terza fase, in quanto saremo costretti in generale ad ignorare alcuni possibili *causal splitter* perchè non appartenenti ad X'.

Rimanendo però ancora nella seconda fase, è chiara la necessità di modificare l'algoritmo che implementa il *merging*. Si osservi infatti che, mantenendo invariato l'algoritmo, potremmo incorrere nel seguente caso spiacevole: eseguendo *merge_phase* uniamo $[u]_{X_1}$ ed un *predecessore* di $[v]$. A catena percorriamo i *predecessori* dei blocchi uniti (facendo dunque una sorta di visita di G^{-1}), finchè non raggiungiamo, sfortunatamente, il blocco $[v]_{X_1}$. Da qui, supponendo che venga effettuata una nuova unione, possiamo ritornare a $[u]_{X_1^*}$ (l'asterisco è stato aggiunto perchè alcuni blocchi di

X_1 sono stati uniti, sicchè $X_1^* \neq X_1$), e da qui ripetere la visita del grafo. Naturalmente questa è una situazione indesiderabile e va evitata: la visita potrebbe in generale ripetersi numerose volte, finchè vi sono blocchi che è possibile unire lungo tutto il tragitto.

Così come intendiamo modificare l'algoritmo per il *merging*, aggiornaremo anche la nozione di *causal splitter*. O meglio, aggiorneremo il criterio operativo per verificare la condizione 1. (appartenenza del blocco ad X'): intendiamo ricavare un criterio che lavori meglio con il nuovo algoritmo di *merging*. Illustriamo innanzitutto l'idea per la modifica dell'algoritmo che implementa la seconda fase, di cui riporteremo lo pseudocodice tra alcune pagine.

Operiamo una DFS su G' in ordine crescente di *finishing time* (segnato dalla prima DFS, quella che abbiamo sfruttato per verificare l'esistenza di un nuovo ciclo). Ad ogni nuovo blocco incontrato durante questa nuova visita verifichiamo se è possibile unirlo con uno dei blocchi già incontrati, e in caso affermativo propaghiamo l'unione ai *predecessori* dei due blocchi con le modalità viste sopra. È chiaro che la natura esaustiva di questo procedimento e la validità delle stesse considerazioni fatte sopra anche per questo secondo caso ci garantiscono la fondatezza della Proposizione 2.10 se utilizziamo l'algoritmo per la fase di *merging* appena presentato.

A questo punto possiamo occuparci di individuare una caratteristica che ci consenta di stabilire quando un blocco può essere considerato un *causal splitter*. Vale il seguente risultato:

Osservazione 2.12. Supponiamo che una DFS su G^{-1} dei *predecessori* di $[u]_{X_1}$ non visiti un blocco $B \in X_1$. Allora $B \in X_2$, e $B \in X'$.

Abbiamo mantenuto la notazione introdotta sopra: $\langle u, v\rangle$ è il nuovo arco, X_1 la partizione dopo la prima fase. Il significato dell'osservazione è il seguente: se una tale visita (effettuata subito dopo il termine della prima fase) non raggiunge il blocco B, ne segue che esso non verrà influenzato da una qualsiasi delle unioni che verranno effettuate durante la fase di *merging*. Questo significa che esso soddisfa automaticamente la condizione 1. della Definizione 2.8, e siamo quindi autorizzati a considerarlo un *causal splitter* se per qualche blocco vale congiuntamente anche la 2.

Dimostrazione. Evidente per la formulazione dell'algoritmo che implementa la fase di *merging* nel caso 2. □

Nel caso in cui l'ipotesi dell'Osservazione 2.12 non sia verificata per un dato blocco non possiamo dire nulla in generale, e dunque non considereremo

il blocco in questione un possibile *causal splitter*. Così facendo stiamo introducendo un'approssimazione nella soluzione parziale X_2 che dovremo sanare durante la terza fase. In generale, infatti, potremmo avere unito dei blocchi per cui abbiamo ignorato un *causal splitter* che non rispettava la condizione proposta nell'Osservazione 2.12.

La fase di *splitting* è delegata ancora una volta alla procedura `ranked_split` presentata sopra. Tutti i blocchi visitati durante la seconda fase vengono inseriti in una partizione Y; questo sotto-grafo di G' viene rifinito tramite l'algoritmo di Paige-Tarjan. Teniamo traccia dei blocchi di X_2 rifiniti durante l'esecuzione, e poi propaghiamo ogni modifica al grafo intero utilizzando `ranked_split`. Per la correttezza dell'algoritmo di Paige-Tarjan, e per la natura esaustiva del procedimento adottato, abbiamo $X_3 = X'$.

2.4.2 L'algoritmo

L'algoritmo incrementale di Saha riceve in ingresso il grafo originale, i nodi u, v agli estremi del nuovo arco e la RSCP S del grafo originale. Poichè in questo algoritmo sono presenti numerosi accorgimenti tecnici si è preferito uno pseudocodice più naturale rispetto a quello usato per gli algoritmi presentati precedentemente. Per i dettagli si rimanda all'implementazione in Python.

Innanzitutto (Riga 2) verifichiamo se $[u]_X$ era già *predecessore* di $[v]_X$ prima dell'inserimento del nuovo arco: in questo caso non è necessario fare nulla, in quanto la RSCP non risulta modificata dall'aggiunta. Se il controllo fallisce eseguiamo la prima fase con le modalità descritte sopra.

A questo punto è necessario distinguere diversi sotto-casi, in quanto ognuno va trattato in modo leggermente diverso. Oltre alla distinzione relativa all'introduzione del nuovo ciclo, che determina quale tipo di seconda fase adoperare e se eseguire la terza fase, è necessario anche considerare se u, v sono *well-founded*. Questa verifica si rende necessaria per decidere quale procedura utilizzare per il ri-calcolo del rango, un aspetto che abbiamo omesso nella trattazione della sezione precedente, ma che in realtà risulta fondamentale per tutte le procedure che intervengono nell'algoritmo di Saha. Utilizzeremo due procedure differenti per propagare un cambiamento di rango in un nodo a seconda che esso sia o meno *well-founded*, ovvero `propagate_wf` e `propagate_nwf`.

La prima (Riga 38) consiste nella visita della contro-immagine del nodo il cui rango è stato aggiornato, in ordine crescente di rango. Il rango dei nodi visitati viene aggiornato secondo la definizione. Inoltre, per la stessa definizione, siamo certi che quando raccogliamo le informazioni necessarie

Algoritmo 7: Algoritmo incrementale di Saha

```
   Data: G = (V, E), ⟨u, v⟩, S
 1 begin
 2     if [u]_S ⟹ [v]_S then
           // La RSCP non cambia
 3         return S;
       // Prima fase:  split
 4     ranked-split(S, [v]_S);
 5     if !u.WF and v.WF then
 6         if rank(v) + 1 > rank(u) then
 7             rank(u) = rank(v) + 1;
 8             propagate-nwf(u);
 9         merge-phase([u]_S, [v]_S);
10     else
11         if rank(u) > rank(v) then
12             merge-phase([u]_S, [v]_S);
13         else
14             DFS su G^{-1} da u. Salva i nodi in ordine crescente di
                finishing-time. Imposta a True la flag visited dei
                blocchi raggiunti;
15             if v.visited then
                   // È stato formato un ciclo
16                 u.WF = False;
17                 rank(u) = rank(v);
18                 propagate-nwf(u);
19                 merge-split-phase(finishing_time);
20             else
21                 if u.WF then
22                     if v.WF then
23                         rank(u) = rank(v) + 1;
24                         propagate-wf(u);
25                     else
26                         if rank(u) < rank(v) then
27                             rank(u) = rank(v);
28                             propagate-nwf(u);
29                 else
30                     if rank(u) ≠ rank(v) then
31                         rank(u) = rank(v);
32                         propagate-nwf(u);
33                 merge-phase([u]_S, [v]_S);
```

ad aggiornare il rango di un nodo (ovvero il rango dei nodi nella sua immagine) non utilizziamo informazioni provvisorie che possono variare nel seguito dell'esecuzione. Le modifiche vengono poi propagate con `propagate_wf` o `propagate_nwf` a seconda della *well-foundedness* del nodo.

Per quanto riguarda l'implementazione di `propagate_nwf` (Riga 41), viene utilizzato l'Algoritmo di Sharir (o di Kosaraju) [19] per la determinazione delle SCC, di cui forniremo solamente una trattazione operativa. Tale algoritmo, avente complessità $O(|V| + |E|)$, consente di determinare le SCC di un grafo. Lo applicheremo al sotto-grafo *non-well-founded*, per cui avremo una complessità solo $O(|V_{\texttt{NWF}}| + |E_{\texttt{NWF}}|)$, e per evitare computazioni inutili salveremo il risultato per eventuali chiamate successive di

---

34 **function** `ranked-split`*($P, B \in P$)*:
35 | $X = Q = P$;
36 | `split` (B, P);
37 | Applichiamo l'Algoritmo 4, *splitter* scelti in ordine crescente di rango.
38 **function** `propagate-wf`*($a \in V$)*:
39 | Costruiamo la contro-immagine di a in ordine crescente di rango;
40 | Aggiorniamo il rango dei nodi secondo l'ordine indotto dalla lista.
41 **function** `propagate-nwf`*($a \in V$)*:
42 | SCCs = `kosaraju`$(G_{\texttt{NWF}})$;
43 | **foreach** $b \in$ SCC$[a]$ **do**
44 | | b.`rank` = a.`rank`;
45 | | `propagate-nwf`(b);
46 **function** `merge-condition`*(A, B, check_visited)*:
47 | **if** A.`initial_partition` $\neq B$.`initial_partition` **then**
48 | | **return** `False`;
49 | **if** $A == B$ **then**
50 | | **return** `False`;
51 | **if** A.`rank` $\neq B$.`rank` **then**
52 | | **return** `False`;
53 | CS = `causal-splitters`(A, B);
54 | **if** *check_visited* **then**
55 | | **return** $|\{C \in CS \mid C.\texttt{visited}\}| == 0$;
56 | **else**
57 | | **return** $|CS| == 0$;

```
34 function recursive-merge(B_1, B_2 ∈ P):
35 |  Unisci B_1, B_2;
36 |  foreach (A, B) ∈ Pre(B_1) × Pre(B_2) | merge-condition(A, B)
   |   do
37 |  |  recursive-merge(A, B);
38 function merge-phase(U, V ∈ P):
39 |  foreach U_1 ∈ Pre(V) | merge-condition(U_1, U) do
40 |  |  recursive-merge(U_1, U);
41 function merge-split-phase(finishing_time):
42 |  DFS su G usando l'ordine indotto da finishing_time;
43 |  foreach (A, B) ∈ P × P | A, B raggiunti dalla DFS do
44 |  |  if merge-condition(A, B, True) then
45 |  |  |  recursive-merge(A, B);
46 |  X = {C ∈ P | C raggiunto dalla DFS};
47 |  Applica l'Algoritmo 4 su X, e propaga gli split a tutto G con
   |   ranked-split.
```

`propagate_nwf`. Chiamando a il nodo su cui è stata chiamata la funzione, percorriamo la SCC cui appartiene a e impostiamo il rango di tutti i nodi a a.`rank` (il rango infatti resta costante nelle SCC). Dopodichè propaghiamo i cambiamenti con una chiamata ricorsiva a `propagate_nwf` su tutti i nodi della SCC.

Descriviamo ora i vari sotto-casi (mutuamente esclusivi) seguendo l'ordine in cui sono trattati nello pseudocodice dell'Algoritmo 7:

– Se $!u$.`WF` e v.`WF` (Riga 5) aggiorniamo il rango di u come da definizione. Se il rango viene modificato è necessario propagare il cambiamento con `propagate_nwf`. Vale il seguente risultato:

Osservazione 2.13. Se $!u$.`WF` e v.`WF`, l'aggiunta di un nuovo arco $\langle u, v\rangle$ non può generare un nuovo ciclo.

Dimostrazione. Perchè sia possibile generare un ciclo dovrebbe essere possibile, tornare ad u partendo da v. Ma se così fosse anche v sarebbe *non-well-founded*. □

Concludiamo dunque che ci troviamo nel caso 1. (si veda la trattazione della sotto-sezione precedente per i dettagli).

– Se $u.\texttt{rank} > v.\texttt{rank}$ (Riga 11) non è necessario modificare il rango di u. Vale inoltre questo semplice risultato:

Osservazione 2.14. Se $u.\texttt{rank} > v.\texttt{rank}$, l'aggiunta di un nuovo arco $\langle u, v\rangle$ non può generare un nuovo ciclo.

Dimostrazione. Come sopra, dovrebbe essere possibile raggiungere u partendo da v. Ma se così fosse avremmo $v.\texttt{rank} \geq u.\texttt{rank}$. □

Sicchè valgono ancora le considerazioni fatte sopra per il caso 1.

Se non ci troviamo in una di queste due situazioni, l'algoritmo verifica se è stato generato un nuovo ciclo (Riga 14). Se il responso è positivo aggiorniamo la flag `WF` ed il rango di u, propaghiamo i cambiamenti con `propagate_nwf` ed eseguiamo la seconda e terza fase del caso 2. descritti nella sotto-sezione precedente, ed implementate dalla funzione `merge-split-phase`.

In caso contrario (Riga 20) verifichiamo le varie combinazioni di rango e *well-foundedness*, propaghiamo opportunamente le modifiche e utilizziamo la procedura `merge_phase` per la seconda fase, poichè evidentemente ci troviamo nel caso 1.

2.4.3 Complessità

La complessità dell'algoritmo incrementale di Saha è data chiaramente dalla somma dei contributi delle tre fasi:

1. `split`: La complessità della versione modificata dell'Algoritmo 4 è $O(|E_1| \log |V_1|)$, dove $G_1 = (E_1, V_1)$ è il sotto-grafo di G che viene modificato dalla prima chiamata a `split` alla Riga 36;

1b. `rank`: Per tutti i casi possibili, un upper-bound per il ri-calcolo è $O(|\{\langle x, y\rangle \in E \mid x \in \Delta_{\texttt{WF}}\}| + |E_{\texttt{NWF}}| + |V_{\texttt{NWF}}|)$, dove $\Delta_{\texttt{WF}}$ è l'insieme dei nodi *well-founded* il cui rango cambia in seguito all'aggiunta del nuovo arco, e $E_{\texttt{NWF}}, V_{\texttt{NWF}}$ sono rispettivamente gli archi e i nodi appartenenti al sottografo *non-well-founded* di G. Il secondo termine è dato dalla complessità dell'Algoritmo di Sharir [19] (supponiamo di avere già pronto il sotto-grafo *non-well-founded*);

2. `merge`: Per tutti i casi possibili possiamo maggiorare la complessità di questo step con $O(|E_2||V_2|)$, dove $G_2 = (V_2, E_2)$ è il sottografo dei nodi che si trovano in blocchi modificati durante la fase di *merging* (questo termine tiene conto anche della ricerca dei *causal splitter* per tutte le possibili coppie di blocchi);

3. `split`: La complessità dell'Algoritmo di Paige-Tarjan sul sotto-grafo G_2 a cui viene applicato durante l'esecuzione della terza fase, ovvero $O(|E_2| \log |V_2|)$.

La complessità dell'algoritmo incrementale di Saha ha dunque la seguente forma:

$$\begin{aligned} T_{\text{Saha}} = O(|E_1| \log |V_1|) + O(|\Delta_{\texttt{WF}} \log |\Delta_{\texttt{WF}}|) + O(|E_{\texttt{NWF}}| + |V_{\texttt{NWF}}|) \\ + O(|E_2||V_2|) + O(|E_2| \log |V_2|) \end{aligned}$$

Da questa formula si evince che non è per nulla scontato che l'algoritmo incrementale risulti più efficiente degli algoritmi di Paige-Tarjan o Dovier-Piazza-Policriti, nonostante questi ultimi non richiedano il dato relativo alla massima bisimulazione prima dell'aggiunta del nuovo arco. Si rende dunque necessaria un analisi di tipo probabilistico sul tipo di archi aggiunti e di come questi perturbano la massima bisimulazione. Nel caso in cui gli archi inseriti abbiano spesso come sorgente un nodo a basso rango (cioè nodi molto vicini ai pozzi del grafo) e come destinazione un nodo ad alto rango, l' aggiunta, oltre a turbare notevolmente il rango di un gran numero di nodi, causerà sicuramente un gran numero di `split` e `merge`. In questa situazione l'utilizzo dell'algoritmo incrementale è fortemente sconsigliato, in favore di uno degli altri algoritmi presentati nella Sezione 2. Nel caso opposto in cui caso gli archi inseriti abbiano spesso come sorgente un nodo ad alto rango e come destinazione un nodo a basso rango l'algoritmo incrementale è un buon candidato per l'aggiornamento della massima bisimulazione, in quanto ci aspettiamo che tutti i termini della complessità T_{saha} siano abbastanza controllati. Nella Sezione 3.3 presenteremo alcuni risultati sperimentali relativi alla nostra implementazione degli algoritmi proposti, e confronteremo il tempo di esecuzione dell'algoritmo incrementale con quello di altri algoritmi non incrementali in diversi casi.

3 BisPy

Abbiamo implementato gli algoritmi presentati nella Sezione 2 nel pacchetto Python `BisPy`, il cui codice sorgente è disponibile su GitHub con licenza MIT all'indirizzo https://github.com/fAndreuzzi/BisPy.

Esaminiamo un brevissimo caso di utilizzo della libreria per la determinazione della massima bisimulazione. Creiamo il grafo rappresentato in Figura 8, ovvero un albero bilanciato con *branching factor* 2 e *profondità* 3. Utilizzeremo `BisPy` per calcolarne la massima bisimulazione, utilizzando come partizione iniziale la partizione banale di V:

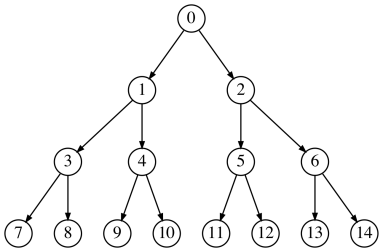


Figura 8

```
import networkx as nx
from bispy import paige_tarjan

# creazione del grafo
graph = nx.balanced_tree(2,3, create_using=nx.DiGraph)

print(paige_tarjan(graph))
```

Lo script restituisce il seguente output:

```
>>> [(7, 8, 9, 10, 11, 12, 13, 14), (3, 4, 5, 6),
    (1, 2), (0,)]
```

Chiaramente tutti i nodi sullo stesso livello
risultano bisimili, per verificarlo è sufficiente applicare la definizione. Come si può osservare l'interfaccia di `BisPy` è molto semplice, i grafi in ingresso possono essere rappresentati utilizzando uno strumento molto diffuso come la libreria `NetworkX`, e per calcolare una massima bisimulazione sono sufficienti poche righe di codice.

Nel resto di questa sezione forniremo alcuni dettagli sull'implementazione di `BisPy`, sugli strumenti utilizzati durante lo sviluppo, e presenteremo alcuni risultati sperimentali.

3.1 Implementazione

Grafi e partizioni sono rappresentati con classi apposite all'interno del file `graph_entities.py`, di cui forniamo un breve glossario:

– `_Vertex`: un nodo del grafo;

– `_Edge`: un arco del grafo;

– `_QBlock`, `_XBlock`: rappresentano rispettivamente i blocchi di tipo `Q`,`X` dell'algoritmo di Paige-Tarjan, negli algoritmi successivi solo la prima classe viene utilizzata, in quanto è più flessibile e contiene diverse funzioni utili (come `_QBlock.mitosis(_Vertex[])`, che consente di dividere il blocco in due in tempo lineare, data una lista di nodi da estrarre).

Le classi contengono attributi utilizzati ampiamente dagli algoritmi per depositare temporaneamente informazioni che verranno recuperate in seguito. Oltre agli algoritmi per il calcolo della massima bisimulazione, nel pacchetto sono stati implementati da zero alcune *subroutine* che abbiamo menzionato durante l'analisi degli pseudocodici nella Sezione 2. Riportiamo un elenco degli algoritmi implementati:

– Algoritmo di Paige-Tarjan 4;

– Algoritmo di Dovier-Piazza-Policriti 6;

– Algoritmo incrementale di Saha 7;

– Depth first search (varie versioni);

– Algoritmo di Kosaraju/Sharir [19].

Infine, elenchiamo le librerie open source non standard utilizzate all'interno del pacchetto:

– `NetworkX`, per la rappresentazione dei grafi presi in input dall'utente;

– `llist`, per un'implementazione delle *doubly linked list*, fondamentali per una corretta stesura dell'algoritmo di Paige-Tarjan.

3.2 Strumenti per lo sviluppo

Abbiamo utilizzato *git* come *Version Control System*, sfruttando ampiamente diverse feature come le branch per lavorare contemporaneamente su più algoritmi ancora incompleti, ed il comando `git bisect` per verificare in quale *commit* è stato introdotto un certo errore. Come abbiamo anticipato nell'introduzione della Sezione 3, il codice sorgente è caricato su GitHub in un repository pubblico.

Il pacchetto è ampiamente testato, sia in termini di risultati finali, con grafi aventi caratteristiche e dimensioni varie di cui abbiamo generato con un approccio *brute force* la massima bisimulazione, sia per quanto riguarda le funzioni intermedie. Per eseguire i test abbiamo utilizzato la libreria `PyTest` [11]. Nel momento in cui stiamo scrivendo il pacchetto ha *code coverage* (che a grandi linee può essere definita come la percentuale di righe di codice coperte da almeno un test) pari a 97%. Quest'ultima è stata calcolata tramite il tool *coveralls*, che "esamina" il codice a richiesta e fornisce la *code coverage*.

Per automatizzare le interazioni con le API di *coveralls* abbiamo utilizzato *GitHub Actions*, un framework che consente di specificare alcune azioni da compiere (la richiesta di revisione a *coveralls*, ad esempio) in risposta ad eventi di vario genere (nel nostro caso il "push" di una o più commit sul repository GitHub).

Per ottenere informazioni circa la performance del pacchetto è stato utilizzato il tool *cProfile*, che analizza l'esecuzione di una funzione (nel nostro caso uno degli algoritmi che abbiamo implementato) e fornisce per ogni funzione (tra gli altri) i seguenti dati:

- Numero di chiamate;
- Numero di chiamate ricorsive e non ricorsive da cui è stata raggiunta;
- Durata complessiva dell'esecuzione (comprese le chiamate ad altre funzioni);
- Durata complessiva dell'esecuzione (senza considerare le chiamate ad altre funzioni);

Sfruttando questo strumento siamo riusciti ad identificare diversi problemi di implementazione degli algoritmi che abbiamo considerato, spesso guardando solamente il dato sul numero di chiamate.

Infine, abbiamo utilizzato ampiamente il debugger dell'IDE *VSCode*.

3.3 Risultati sperimentali

Proponiamo alcuni risultati sperimentali ottenuti dal package Python contenente gli algoritmi per il calcolo della massima bisimulazione che abbiamo presentato nelle sezioni precedenti. Innanzitutto illustreremo brevemente l'ambiente e gli strumenti con cui sono state prese le misurazioni. Dopodichè presenteremo i risultati in forma di grafici, e tenteremo di evidenziare le differenze tra i vari algoritmi.

3.3.1 Hardware e strumenti per le misura

I risultati sono stati misurati su un computer con sistema operativo *CentOS Linux*, architettura x86_64, processore *Intel(R) Core(TM) i7-4790 CPU* (4 cores, 3.60GHz), e memoria RAM da 16 GB. Per ottenere le misurazioni è stato utilizzato il package Python *timeit*, che presenta alcune caratteristiche che lo rendono un valido strumento per la misurazione del tempo di esecuzione [21]:

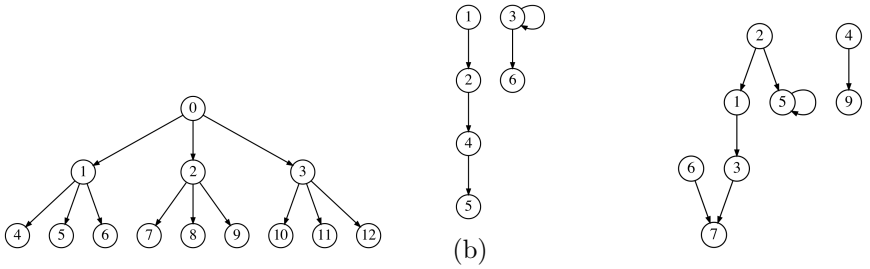
- Utilizza la più precisa funzione disponibile sulla piattaforma per la misurazione del tempo trascorso;
- Disabilita il *garbage collector*, che potrebbe intervenire in un momento casuale della misurazione introducendo rumore nei risultati;
- Esegue lo script preso in esame molte volte in modo da ridurre l'imprecisione dovuta a temporanei sovraccarichi della CPU o della RAM.

I dataset su cui abbiamo effettuato le misurazioni sono stati generati utilizzando alcune funzioni del package Python `NetworkX` [7].

3.3.2 Performance

Consideriamo alcune tipologie differenti di grafi, e valutiamo il tempo necessario per l'esecuzione degli algoritmi che abbiamo implementato nel pacchetto `BisPy`. Chiaramente i risultati che presenteremo non determinano in modo definitivo che un algoritmo sia più o meno veloce rispetto ad un altro: ad esempio, per grafi molto strutturati in cui la relazione "stesso rango" approssima bene la massima bisimulazione ci aspettiamo che, asintoticamente, l'algoritmo di Dovier-Piazza-Policriti risulti molto più conveniente dell'algoritmo di Paige-Tarjan. La scelta dell'algoritmo più opportuno per il calcolo della massima bisimulazione va quindi ponderata in base ai dati di cui si è in possesso sulla struttura del grafo. Come vedremo non è nemmeno garantito che l'algoritmo incrementale risulti sempre il più conveniente in seguito all'aggiunta di un arco, in particolare nei casi in cui la massima bisimulazione cambia molto. Questo avviene ad esempio quando ad una catena di nodi (tutti aventi un unico arco uscente ed un unico arco entrante, eccetto il primo e l'ultimo che hanno rispettivamente solo un arco uscente ed uno entrante) si aggiunge un arco dall'ultimo al primo nodo. In questo caso la massima bisimulazione collassa da $|V|$ blocchi ad uno solo.

Il primo confronto che considereremo riguarda gli algoritmi di Paige-Tarjan e Dovier-Piazza-Policriti: verificheremo che vi sia effettivamente una



(a) Un albero bilanciato, *depth*=2, *branching factor*=3.

(b) Hopcroft, $n = 2$

(c) Hopcroft, $n = 3$

Figura 9: Tipologie di grafi su cui abbiamo testato gli algoritmi di Dovier-Piazza-Policriti e Paige-Tarjan.

convenienza (almeno asintotica) nell'utilizzo di quest'ultimo per grafi strutturati. Il secondo confronto coinvolge tutti gli algoritmi che abbiamo considerato, ed il suo scopo è mostrare che l'algoritmo incrementale può risultare molto vantaggioso rispetto per l'aggiornamento della massima bisimulazione rispetto al ricalcolo da zero, ma che talvolta questa convenienza può venire meno. Nei grafici adotteremo una scala data dal valore $|E| \log |V|$ lungo l'asse delle ascisse, che ci consente di visualizzare in modo più regolare l'andamento asintotico delle quantità che considereremo durante gli esperimenti.

Paige-Tarjan, Dovier-Piazza-Policriti Ci aspettiamo che l'algoritmo di Dovier-Piazza-Policriti risulti asintoticamente più conveniente per grafi che mostrano una qualche struttura. Per grafi piccoli invece dovrebbe risultare vincitore l'algoritmo di Paige-Tarjan in quanto non richiede l'elaborazione preliminare del rango (che ricordiamo aggiunge un termine $O(|V| + |E|)$).

Cominciamo l'esposizione dei risultati con la categoria degli "alberi bilanciati", ovvero grafi che possono essere rappresentati nella forma esposta nella Figura 9a. Essi sono caratterizzati da due parametri:

- *depth*: il numero di livelli (senza contare la *root*);
- *branching factor*: il numero di successori di ogni nodo (ad eccezione dei nodi nell'ultimo livello).

Generiamo grafi di questo tipo tramite la funzione `balanced_tree` di `NetworkX`, che appunto prende in input questi due parametri. Dal grafico rappresentato nella parte sinistra della Figura 10 possiamo osservare che l'algoritmo di

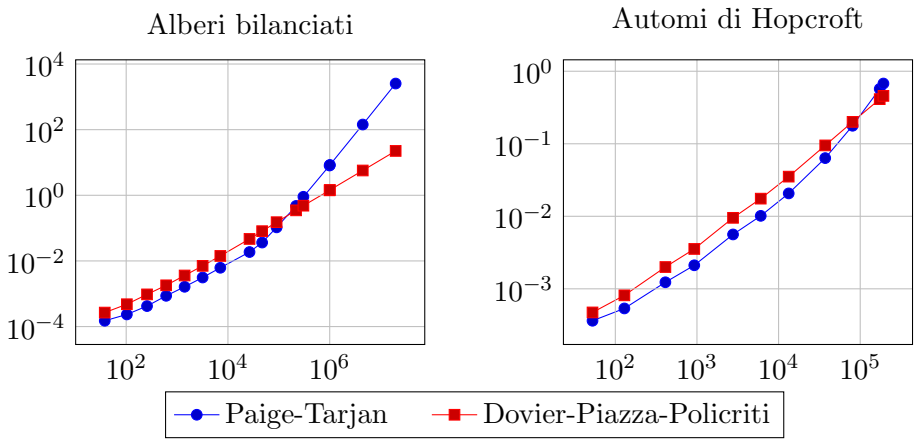


Figura 10: Tempo di esecuzione degli algoritmi di Dovier-Piazza-Policriti e Paige-Tarjan su alberi bilanciati (destra) e automi proposti da Hopcroft (sinistra). Lungo l'asse delle ascisse (logaritmico) varia la quantità $|E| \log |V|$, mentre sull'asse delle ordinate (logaritmico) è riportato il tempo di esecuzione dell'algoritmo in secondi.

Dovier-Piazza-Policriti risulta asintoticamente più efficiente, a partire da un certo valore di $|E| \log |V|$. Sebbene infatti la metodologia sia più "rifinita" dell'approccio di Paige-Tarjan, lo sforzo iniziale per il calcolo del rango è apprezzabile, soprattutto se consideriamo il linguaggio in cui l'algoritmo è stato implementato. Abbiamo però la conferma che per valori alti di $|E| \log |V|$, per questa tipologia di grafo, si ottengono i risultati previsti.

La seconda tiplogia di grafi che consideriamo è tratta dall'articolo in cui viene presentato l'algoritmo di Dovier-Piazza-Policriti, ed è un esempio di automa proposto originariamente da Hopcroft [9]. Nelle Figure 9b e 9c ne visualizziamo alcune realizzazioni. Nella parte destra della Figura 10 sono riportati i risultati ottenuti dagli algoritmi. Osserviamo nuovamente che l'algoritmo di Dovier-Piazza-Policriti risulta asintoticamente più conveniente.

Algoritmo incrementale Proseguiamo l'esposizione dei risultati sperimentali ottenuti utilizzando il pacchetto `BisPy` con un confronto tra gli algoritmi di Paige-Tarjan e Dovier-Piazza-Policriti (cui ci riferiremo in questa sezione con "algoritmi *from scratch*", in quanto ricalcolano da zero la massima bisimulazione) e l'algoritmo incrementale di Saha. L'obiettivo di questo confronto è verificare di quanto e in quali casi l'algoritmo incrementale risulti

più efficiente nel ricalcolare la massima bisimulazione in seguito all'aggiunta di un nuovo arco, rispetto ad un algoritmo che non sfrutti le informazioni relative alla massima bisimulazione prima della modifica. Rammentiamo la complessità dell'algoritmo di Saha, ottenuta nella Sezione 2.4.3 come la somma delle varie fasi di cui si compone il procedimento:

$$T_{\text{Saha}} = O(|E_1| \log |V_1|) + O(|\Delta_{\texttt{WF}} \log |\Delta_{\texttt{WF}}|) + O(|E_{\texttt{NWF}}| + |V_{\texttt{NWF}}|) \tag{3.1}$$

$$+ O(|E_2||V_2|) + O(|E_2| \log |V_2|) \tag{3.2}$$

$\Delta_{\texttt{NWF}}$ è l'insieme di nodi *non-well-founded* il cui rango è cambiato in seguito all'inserimento del nuovo arco; E_1, V_1 e E_2, V_2 sono rispettivamente nodi e archi appartenenti al sottografo di G composto dai nodi modificati durante le fasi 1. (`split`) e 2. (`merge`); $E_{\texttt{NWF}}$, $V_{\texttt{NWF}}$ sono rispettivamente archi e nodi del sottografo *non-well-founded* di G. Si osservi che numerosi termini possono diventare molto facilmente "problematici", e rendere quindi l'algoritmo incrementale meno efficiente del ricalcolo da zero della massima bisimulazione. In questa breve analisi presenteremo i risultati su alcune tipologie di grafo, e ne verificheremo la coerenza con l'analisi proposta nelle sezioni precedenti.

Per il primo esperimento consideriamo grafi randomici di Erdős-Rényi (o *grafi binomiali*), ovvero grafi contenenti un numero arbitrario `n` di nodi, per

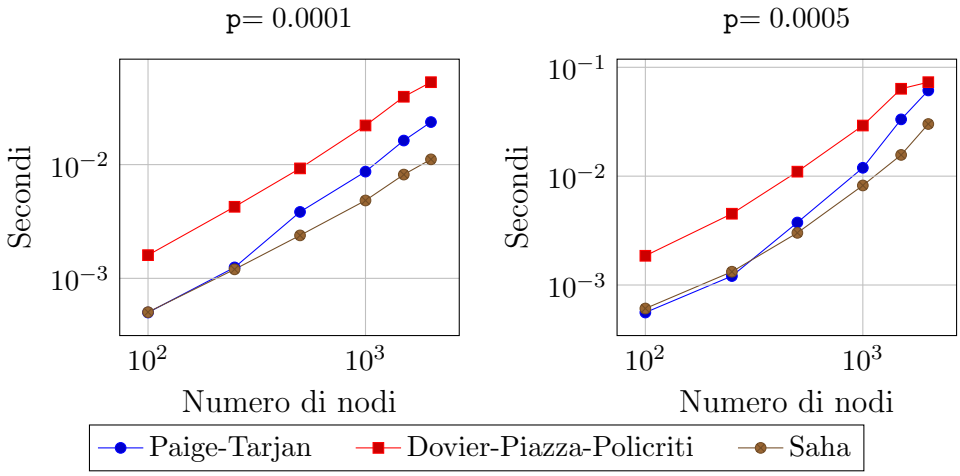


Figura 11: Tempo medio in secondi (su un campione di 100 grafi) impiegato per l'aggiornamento della massima bisimulazione in seguito all'aggiunta di un arco generato in modo randomico. Lungo l'asse delle ascisse varia il numero di nodi del grafo binomiale.

cui presa una coppia qualsiasi $u, v \in V$ si ha $\langle u, v \rangle \in E$ con probabilità $\mathtt{p} \leq 1$. Sono stati generati 100 grafi tramite la funzione `fast_gnp_random_graph` di `NetworkX`, che prende in ingresso i parametri `n,p`, e per ognuno è stato aggiunto un arco scelto randomicamente tra quelli non esistenti. Infine si è verificato il tempo medio di esecuzione sul campione ottenuto, per gli algoritmi di Paige-Tarjan e Saha (Figura 11). Naturalmente per l'algoritmo incrementale si è supposto di avere già a disposizione la massima bisimulazione precedente all'aggiunta dell'arco, mentre l'algoritmo di Paige-Tarjan è stato eseguito direttamente sul grafo modifcato. L'algoritmo di Saha risulta mediamente più conveniente per entrambi i valori di `p` considerati, ma è evidente

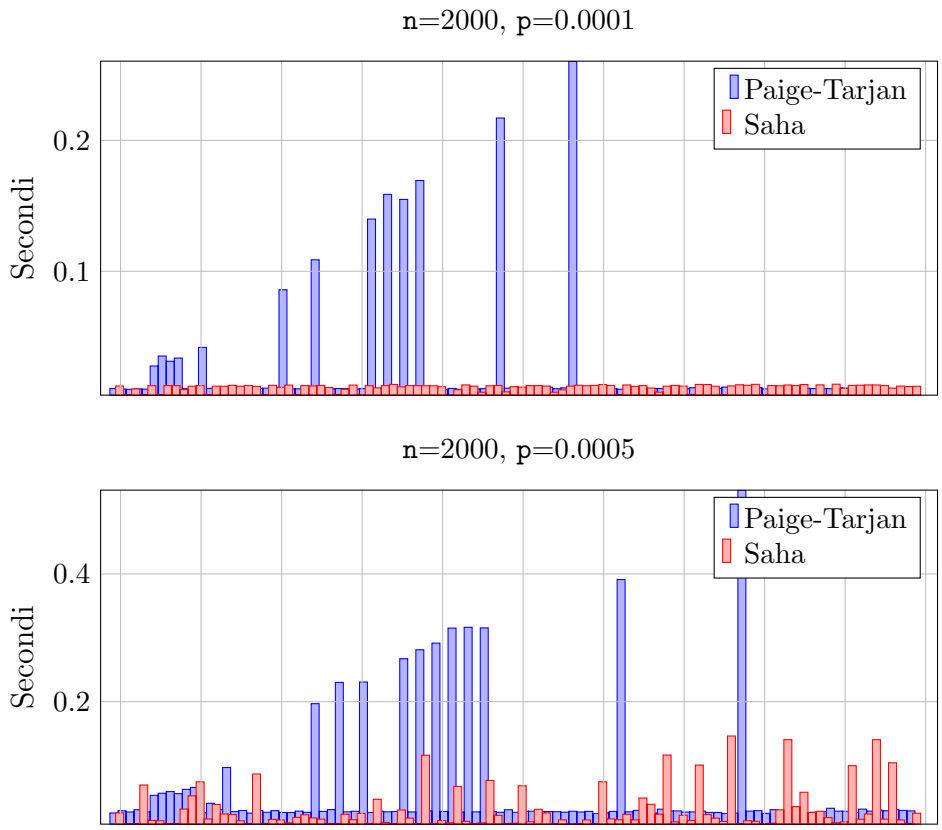


Figura 12: Tempo di esecuzione degli algoritmi di Saha e Paige-Tarjan sui 100 grafi generati in modo randomico menzionati nel primo esperimento, per due coppie (`n,p`) = (2000,0.0001) (sopra), (2000,0.0005) (sotto).

che con l'aumento di questo parametri la convenienza si riduce in modo significativo. Tale comportamento è coerente con l'Equazione (3.1). Nella Figura 12 sono riportati i 100 risultati per due coppie (`n,p`). Questa visualizzazzione ci consente di osservare che non sempre l'algoritmo incrementale è il più conveniente: per i valori considerati viene saltuariamente superato dall'algoritmo di Paige-Tarjan.

Per il secondo esperimento abbiamo ritenuto interessante esaminare i risultati ottenuti su grafi con una certa struttura. Per questo motivo sono stati considerati degli *alberi bilanciati*, una tipologia di grafi che abbiamo già descritto sommariamente nel paragrafo precedente. Presi alcuni alberi bilanciati di dimensioni varie, per ognuno abbiamo generato 100 archi in modo randomico (escludendo quelli già presenti); in seguito sono stati eseguiti gli algoritmi di Paige-Tarjan, Dovier-Piazza-Policriti e Saha per aggiornare la massima bisimulazione (i primi due sul grafo modificato, l'ultimo supponendo di avere a disposizione la massima bisimulazione prima dell'aggiunta dell'arco). I risultati sono rappresentati nella parte sinistra della Figura 13. Ad una prima analisi del grafico sembra che non vi sia una particolare convenienza nell'utilizzo dell'algoritmo di Saha; questa impressione è

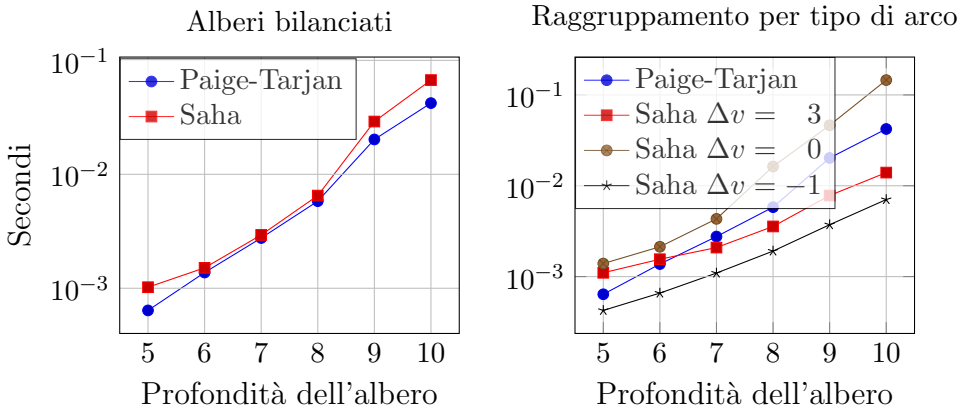


Figura 13: Tempo medio in secondi (su un campione di 100 archi) impiegato per l'aggiornamento della massima bisimulazione in seguito all'aggiunta di un arco generato in modo randomico, su alberi bilanciati con *depth* variabile e *branching factor*=2. Nell'immagine a sinistra abbiamo rappresentato i risultati complessivi, nell'immagine a destra abbiamo diviso gli archi in categorie a seconda della distanza (in livelli) tra sorgente e destinazione dell'arco aggiunto, ed abbiamo poi preso il tempo medio di ognuno di questi "cluster".

però in parte sbagliata, è necessario ricordate che gli archi sono generati in modo randomico e che il tempo di esecuzione è estremamente suscettibile alla tipologia di arco in cui consiste la modifica. La situazione è rappresentata meglio nella parte destra della Figura 13, in cui abbiamo diviso gli archi in categorie in base alla seguente funzione:

$$\Delta v(\langle x, y \rangle) = \texttt{Depth}(x) - \texttt{Depth}(y)$$

dove $\texttt{Depth}(x)$ è il livello dell'albero in cui si trova il nodo x. Per chiarezza abbiamo rappresentato solamente l'andamento temporale dell'algoritmo di Paige-Tarjan, e dell'algoritmo di Saha per gli archi con $\Delta v \in \{-1, 0, 3\}$. Si osservi che per $\Delta v = -1$ il tempo di esecuzione è molto basso in quanto non viene introdotto alcun cambiamento. Per $\Delta v = 0$ invece il tempo medio di esecuzione è addirittura peggiore di quello registrato dall'algoritmo di Paige-Tarjan, in quanto la massima bisimulazione viene cambiata in modo molto significativo. Questo esperimento mostra come una valutazione probabilistica della tipologia di arco in cui consiste la modifica del grafo sia fondamentale per decidere quale algoritmo utilizzare nelle applicazioni pratiche.

3.3.3 Dimensione della massima bisimulazione

Terminiamo la sezione relativa ai risultati sperimentali con un'ultima analisi che potrebbe essere di qualche interesse nella prospettiva di ciò che abbiamo

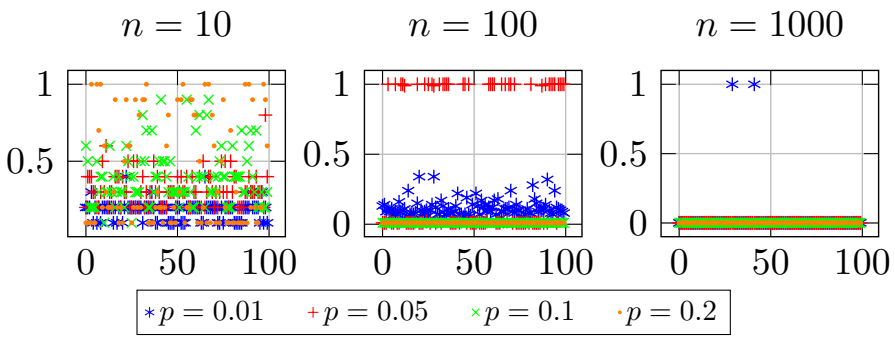


Figura 14: Numero di classi di equivalenza della massima bisimulazione per grafi generati in modo casuale, normalizzato rispetto al numero di nodi nel grafo. Ad ogni *tick* sull'asse delle ascisse corrisponde un grafo generato in modo casuale. Lungo l'asse delle ordinate è riportato il numero di blocchi della massima bisimulazione.

presentato nella Sezione 1.4.4. Verificheremo l'andamento del numero di classi di equivalenza nella massima bisimulazione per i grafi randomici di Erdős-Rényi considerati poco sopra. Considereremo valori piccoli di `p`, in quanto avvicinandoci ad 1 otteniamo grafi sempre più "completi" di interesse molto basso nella pratica. Nella Figura 14 abbiamo considerato tre diversi valori per il parametro `n`, e quattro valori per il parametro `p`.

Per ogni combinazione di `n`,`p` abbiamo generato 100 grafi binomiali con la funzione `fast_gnp_random_graph` del pacchetto `NetworkX`, e ne abbiamo calcolata la massima bisimulazione con l'algoritmo di *Dovier-Piazza-Policriti*. Nei grafici abbiamo visualizzato la quantità $\frac{|V/\equiv|}{|V|}$, ovvero il rapporto tra il numero di classi di equivalenza nella massima bisimulazione ed il numero di nodi del grafo: questa quantità potrebbe essere considerata come il fattore di riduzione che otteniamo quanto sostuitiamo al grafo originale la sua contrazione secondo la massima bisimulazione.

Per `n=10` la quantità che stiamo valutando varia molto tra 1 (10 classi di equivalenza, nessuna riduzione) e 0.1 (tutti i nodi sono bisimili, riduzione massima). Per valori più grandi la variabilità si riduce, e per `n=1000` il fattore si schiaccia decisamente verso 0.001 (un'unica partizione).

Conclusioni

In questo elaborato abbiamo studiato il problema della *massima bisimulazione*, ed alcuni algoritmi che utilizzano metodi diversi per la risoluzione del problema. Inoltre abbiamo presentato il pacchetto `BisPy`, parte integrante di questo lavoro. Durante il processo di implementazione del software è risultata fondamentale l'analisi condotta nella Sezione 2, che ha consentito di acquisire una comprensione profonda del tema e delle metodologie utilizzate per la risoluzione. Il pacchetto è stato utilizzato per valutare il tempo di esecuzione degli algoritmi considerati su varie tipologie di grafo, e per verificare la coerenza dei risultati ottenuti con quanto è emerso dall'analisi.

Intendiamo continuare lo sviluppo del pacchetto come progetto *open source*, in quanto sono numerose le funzionalità che sarebbe auspicabile aggiungere al fine di rendere `BisPy` più versatile ed applicabile in contesti pratici. Per quello che abbiamo potuto osservare non ci sono altri progetti Python che trattino la bisimulazione in modo approfondito, per cui riteniamo che vi sia una certa componente di originalità nel lavoro. Di seguito sono elencate alcune migliorie e funzionalità che potrebbero essere aggiunte al progetto in futuro:

- *Labeled edges*: La possibilità di assegnare etichette agli archi consente di formulare modelli più espressivi e risolvere problemi più complessi ed interessanti, infatti le applicazioni che abbiamo considerato nella Sezione 1.4.4 beneficiano notevolmente di questa variante della bisimulazione. Sono sufficienti alcuni cambiamenti di lieve entità agli algoritmi originali per supportare questa generalizzazione ([4], Sezione 7), che aumenterebbe molto l'applicabilità del pacchetto in contesti pratici;

- *k-bisimulazione*: In alcuni casi la bisimulazione fornisce un partizionamento del grafo troppo approfondito, inutilizzabile all'atto pratico, poichè ciò che succede "lontano" da un nodo talvolta è di poca importanza per quel nodo; per questo motivo si introduce una variante, la k-bisimulazione, in cui le caratteristiche topologiche di un nodo (gli archi nella sua immagine, gli archi nell'immagine dei successori del nodo, ...) vengono considerate solo in un "intorno" di raggio $k \geq 0$ [13]; la k-bisimulazione peraltro è piuttosto economica da calcolare, e trova quindi diverse applicazioni pratiche;

- Migliorare la compatibilità con altri package Python nell'ambito della teoria dei grafi, come ad esempio `NetworkX`, in modo da poter sfruttare

altre librerie per aggiungere nuove funzionalità, o per poter ingrandire l'insieme di formati di input supportati;

– Scrivere alcune porzioni di codice critiche utilizzando `Cython` (ad esempio, la ricerca dei *causal splitter* nell'Algoritmo di Saha).

Bibliografia

[1] Peter Aczel. *Non-well-founded sets*, volume 14 of *CSLI lecture notes series*. CSLI, 1988.

[2] Jesper R Andersen, Nicklas Andersen, Søren Enevoldsen, Mathias M Hansen, Kim G Larsen, Simon R Olesen, Jirí Srba, and Jacob K Wortmann. CAAL: concurrency workbench. In *International Colloquium on Theoretical Aspects of Computing*, pages 573–582. Springer, 2015.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[4] Agostino Dovier, Carla Piazza, and Alberto Policriti. A fast bisimulation algorithm. In *International Conference on Computer Aided Verification*, pages 79–90. Springer, 2001.

[5] Raffaella Gentilini, Carla Piazza, and Alberto Policriti. From bisimulation to simulation: Coarsest partition problems. *Journal of Automated Reasoning*, 31(1):73–103, 2003.

[6] Martin Grandjean. A social network analysis of twitter: Mapping the digital humanities community. *Cogent Arts & Humanities*, 3(1):1171458, 2016.

[7] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using NetworkX. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[8] Åke J Holmgren. Using graph models to analyze the vulnerability of electric power networks. *Risk analysis*, 26(4):955–969, 2006.

[9] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*, pages 189–196. Elsevier, 1971.

[10] Paris C Kanellakis and Scott A Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and computation*, 86(1):43–68, 1990.

[11] Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, Floris Bruynooghe, Brianna Laugher, and Florian Bruhin. PyTest 6.2.2, 2004.

[12] Kenneth Kunen. *Set theory, an introduction to independence proofs*. Elsevier, 2014.

[13] Yongming Luo, George HL Fletcher, Jan Hidders, Yuqing Wu, and Paul De Bra. External memory k-bisimulation reduction of big graphs. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 919–928, 2013.

[14] Tova Milo and Dan Suciu. Index structures for path expressions. In *International Conference on Database Theory*, pages 277–295. Springer, 1999.

[15] Robert Paige and Robert E Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[16] Bill Roscoe. The theory and practice of concurrency. 1998.

[17] Diptikalyan Saha. An incremental bisimulation algorithm. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 204–215. Springer, 2007.

[18] Gunther Schmidt and Thomas Ströhlein. *Relations and graphs: discrete mathematics for computer scientists*. Springer Science & Business Media, 2012.

[19] Micha Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.

[20] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[21] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[22] Stephan M Wagner and Nikrouz Neshat. Assessing the vulnerability of supply chains using graph theory. *International Journal of Production Economics*, 126(1):121–129, 2010.