Università di Pisa

Dipartimento di Matematica

Corso di Laurea Triennale in Matematica

Tesi di Laurea

Ricerca veloce di pattern comuni a due grafi

Relatore:
Prof. *Roberto Grossi*

Candidato:
Luca Versari

Anno Accademico 2014–2015

Indice

Capitolo 1

Introduzione

I grafi sono tra gli oggetti matematici più studiati in informatica. Il motivo principale per questo grande interesse è da ricercarsi nel fatto che i grafi possono essere usati per modellare moltissime situazioni reali.

Storicamente, uno dei primi problemi a esser modellato con l'utilizzo di un grafo è il problema dei ponti di Königsberg. Oggi gli oggetti che vengono rappresentate mediante grafi sono innumerevoli. Alcuni esempi sono:

- le mappe stradali: i vertici corrispondono agli incroci, gli archi alle strade che li congiungono
- i social network: i vertici corrispondono agli utenti, gli archi alle relazioni fra essi
- il web: i vertici corrispondono alle singole pagine, gli archi ai collegamenti fra esse

1.1 Definizioni di base e notazioni

Definizione 1.1. Si dice **grafo** una coppia ordinata (V, E), con V insieme dei vertici (o nodi) e $E \subset V \times V$ insieme degli archi.

Se due vertici u, v sono collegati da un arco si dicono estremi dell'arco. In tal caso l'arco viene spesso indicato con la coppia (u, v).

Se $\nexists u : (u, u) \in E$, il grafo si dice *semplice*. Se $(u, v) \in E \Leftrightarrow (v, u) \in E$ il grafo si dice *non orientato*. Se $|V| < \infty$, il grafo si dice *finito*. Qui considereremo solo grafi finiti, semplici e non orientati, a meno che non venga specificato altrimenti.

Si dicono *vicini* di un vertice v gli elementi dell'insieme $N(v) = \{u : (v,u) \in E\}$. Il numero dei vicini di v si chiama *grado* di v e si indica con $\deg v$. Indichiamo il massimo grado dei vertici di un grafo con Δ.

Una sequenza di vertici $v_1, \ldots, v_k$ si dice un *cammino* se $(v_i, v_{i+1}) \in E \ \forall i = 1 \ldots k-1$; v_1 e v_k si dicono *estremi* del cammino. Il numero $k-1$ si dice *lunghezza* del cammino. Un grafo si dice *connesso* se esiste un cammino tra ogni coppia di suoi vertici (ovvero che ha quei due vertici come estremi). Un cammino che ha lunghezza positiva, non ripercorre due volte lo stesso arco e ha gli estremi coincidenti si definisce *ciclo*.

Definizione 1.2. Un grafo $G' = (V', E')$ si dice **sottografo** di $G = (V, E)$ se $V' \subset V$ e $E' \subset E$. Un sottografo si dice **indotto** se $E' = (V' \times V') \cap E$.

Scrivendo $G' \subset G$ indichiamo che G' è un sottografo di G. $G' < G$ indica invece che G' è un sottografo indotto di G.

Definizione 1.3. Dati due grafi $G = (V, E)$ e $G' = (V', E')$, una funzione bigettiva $I : V \to V'$ si dice **isomorfismo di grafi** se $(u, v) \in E \Leftrightarrow (I(u), I(v)) \in E'$.

Con la scrittura $I : G \to G'$ si intende che I è un isomorfismo tra i due grafi G e G'. Se H è un sottografo indotto di G, con un leggero abuso di notazione indichiamo con $I_{|H}$ l'isomorfismo I ristretto ai vertici del grafo che appartengono a H. Diciamo inoltre che due grafi sono isomorfi se esiste un isomorfismo tra di essi.

Definizione 1.4. Si dice **etichettatura** di un grafo $G = (V, E)$ una funzione $l : V \to L$. Gli elementi di L sono detti etichette.

Dati due grafi etichettati $G = (V, E)$ e $G' = (V', E')$, aventi come funzioni di etichettatura $l : V \to L$ e $l' : V' \to L$, si dice che un isomorfismo $I : G \to G'$ *preserva l'etichettatura* se vale $l(v) = l'(I(v)) \ \forall v \in V$.

Definizione 1.5. Si definisce **cricca** in un grafo G un sottografo di G completo (ovvero che ha tutti gli archi possibili).

Dicendo che l'insieme $\{v_1, \ldots, v_k\}$ forma una cricca intenderemo che il sottografo indotto che ha per vertici $\{v_1, \ldots, v_k\}$ è una cricca. Indichiamo con K_n il grafo completo con n vertici. Inoltre, indichiamo con $K_{a_1,\ldots,a_k}$ un grafo *k-partito completo*, ovvero che ha per vertici $\bigcup_{i=1}^{k} A_k$, con $|A_k| = a_k$, e tale che $\forall u \in A_i, \forall v \in A_j : (u, v) \in E \Leftrightarrow i \neq j$.

1.2 Il problema del sottografo comune

Dati due grafi, è spesso utile conoscere somiglianze strutturali tra essi. Questa necessità è stata formalizzata con il concetto di *sottografo comune*. Formalmente, si dice che due grafi G e G' hanno un sottografo H comune se esistono due sottografi, rispettivamente di G e di G', isomorfi a H.

Elencare tutti i sottografi comuni a due grafi è poco pratico, in quanto il loro numero può essere molto elevato. Inoltre verrebbero generate molte informazioni superflue: per esempio, se H è un sottografo comune, anche un qualsiasi $H' \subset H$ lo è, ma sapere ciò non fornisce alcuna informazione in più.

Risulta dunque naturale voler elencare i soli sottografi comuni *massimali* (rispetto alla relazione di inclusione) di due grafi. D'altra parte, questa informazione non è sempre sufficiente: spesso torna utile avere esplicitamente tutti gli isomorfismi tra i sottografi comuni. Il problema che cerchiamo di risolvere è dunque il seguente:

Sottografi comuni massimali (MCS). Dati due grafi G e H, elencare tutti gli isomorfismi $I : G' \to H', G' \subset G, H' \subset H$, con G' e H' massimali per inclusione nella famiglia dei sottografi comuni.

In alcune applicazioni, è sufficiente considerare i soli sottografi indotti. Si ha allora il seguente problema, del tutto analogo al precedente:

Sottografi indotti comuni massimali (MCIS). Dati due grafi G e H, elencare tutti gli isomorfismi $I : G' \to H', G' < G, H' < H$, con G' e H' massimali per inclusione nella famiglia dei sottografi indotti comuni.

È in realtà possibile risolvere tutti i problemi MCS risolvendo un problema MCIS, come dimostra il seguente lemma.

Lemma 1.1. *Dato un algoritmo per enumerare i sottografi massimali indotti comuni a due grafi, ne esiste uno per enumerare tutti i sottografi massimali comuni.*

Dimostrazione. Sia infatti G un grafo. Definiamo $D(G)$ il grafo *duale* di G, ovvero il grafo che ha per vertici gli archi di G e che ha un arco tra due vertici se e solo se gli archi a essi corrispondenti hanno esattamente un estremo in comune. È evidente che, se $H \subset G$, allora $D(H) < D(G)$, come rappresentato in figura 1.1. Inoltre, il teorema di isomorfismo di Whitney [1] dimostra che H è isomorfo a H' se e solo se $D(H)$ è isomorfo a $D(H')$, eccetto che per la coppia K_3 e $K_{1,3}$[1]. Da ciò si deduce che per enumerare i sottografi comuni massimali tra due grafi G e G' è sufficiente enumerare i sottografi indotti comuni massimali tra $D(G)$ e

[1] Infatti $D(K_3) = K_3$ e $D(K_{1,3}) = K_3$.

Figura 1.1: Un sottografo di G con il rispettivo sottografo indotto di $D(G)$

$D(G')$: per ogni coppia ottenuta i sottografi di G e G' sono isomorfi a meno che il sottografo trovato non sia K_3, nel qual caso è comunque immediato trovare i tre sottografi massimali comuni corrispondenti. □

È quindi sufficiente limitarsi a trattare i problemi MCIS.

Nelle applicazioni torna spesso utile considerare grafi etichettati e imporre che tra i due sottografi comuni vi sia un isomorfismo che preserva l'etichettatura. Questo problema è evidentemente una generalizzazione di MCIS (basta considerare un'etichettatura costante sui vertici di entrambi i grafi) e sarà indicato con MCLIS.

Non sono noti né algoritmi polinomiali né algoritmi efficienti in pratica per risolvere "direttamente" MCLIS. Tutti gli algoritmi noti (Koch [8], McGregor [9], Ullman[10]) risolvono invece il seguente problema:

Isomorfismi massimali che conservano l'etichettatura (MLI). Dati due grafi G e H, con due etichettature l_G e l_H rispettivamente, elencare tutti gli isomorfismi I massimali (secondo l'ordinamento standard tra funzioni) nella famiglia degli isomorfismi

$$\mathcal{I}_{G,H} = \{I : G' \to H', G' < G, H' < H, l_H(I(v)) = l_G(v)\ \forall v \in G'\}$$

È evidente che è facile ottenere l'insieme dei sottografi comuni a partire dall'insieme degli isomorfismi. D'altra parte, quest'ultimo è potenzialmente molto più grande. Infatti, consideriamo ad esempio la coppia di grafi $G, H = K_n$ (etichettati con un'etichettatura uguale su tutti i vertici). È evidente che esiste un solo sottografo comune massimale (l'intero K_n), mentre il numero di isomorfismi massimali è pari a $n!$. Analogamente avviene se abbiamo due grafi G e H che non contengono archi.

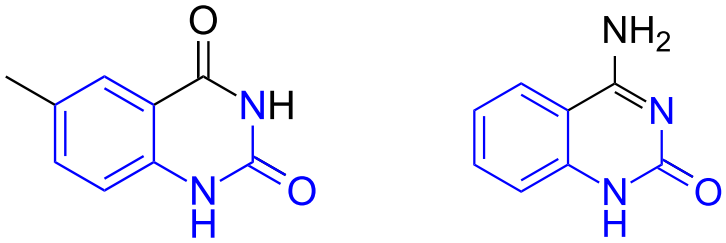

Figura 1.2: Due basi azotate rappresentate dalla loro *formula di struttura* (in cui è sottinteso che agli angoli tra due legami sia presente un atomo di carbonio e sono omessi gli atomi di idrogeno a essi collegati). La più grande sottomolecola comune è evidenziata in blu.

In pratica, la maggior parte dei grafi usati nelle applicazioni sono grafi *sparsi*, ovvero con un numero di archi che è minore di una (piccola) costante moltiplicata per il numero di vertici. Dato che un grafo sparso contiene cricche relativamente piccole, il primo problema non si pone, ma rimane da risolvere il secondo: infatti in generale un grafo sparso contiene molti sottografi indotti senza archi. Una possibile soluzione consiste nel richiedere che i sottografi indotti che sono presi in considerazione da MLI siano anche *connessi*: fare ciò riduce molto il numero di isomorfismi considerati, ma non le informazioni ottenute (tutti gli isomorfismi possono essere ottenuti unendo isomorfismi tra sottografi connessi). Il problema corrispondente sarà indicato con MCLI.

1.3 Applicazioni

Una delle applicazioni più conosciute di questo problema è alla chimica e alla biologia computazionale: date due molecole, spesso ci si chiede quali parti esse abbiano in comune.

Una molecola può essere modellizzata come un grafo etichettato in modo abbastanza naturale. A ogni atomo corrisponde un vertice; l'etichetta di un vertice è pari all'elemento del corrispondente atomo; infine c'è un arco tra due vertici se e solo se le molecole corrispondenti formano un legame.

Data questa modellizzazione di una molecola, è evidente che il problema di trovare strutture comuni in due molecole è equivalente a un problema MCLIS. Inoltre, i grafi ottenuti in questo modo hanno la proprietà ulteriore di avere il grado limitato, dato che tipicamente gli atomi formano pochi legami.

Una variante di questa applicazione si ottiene considerando come vertici le *strutture secondarie* di una proteina invece che i suoi atomi: questo procedimento ha il vantaggio di ridurre di molto il numero di vertici del grafo risultante.

Capitolo 2

Algoritmi di risoluzione

2.1 Riduzione di Levi

Gli algoritmi noti più efficienti si basano su un teorema noto come "riduzione di Levi" [2], che permette di risolvere un problema MLI con gli algoritmi usati per trovare le cricche massimali in un grafo.

Definizione 2.1. Si dice **prodotto modulare** di due grafi $G = (V, E)$ e $H = (V', E')$, con etichettature rispettivamente l_G e l_H, il grafo $G \cdot H$ che ha come insieme dei vertici l'insieme

$$\{(v, v') \in V \times V' : l_G(v) = l_H(v')\}$$

e che ha un arco tra i due vertici (u, u'), (v, v') se e solo se vale una di queste due condizioni:

- $(u, v) \in E$ e $(u', v') \in E'$ (chiameremo tali archi "archi neri")
- $u \neq u'$, $v \neq v'$, $(u, v) \notin E$ e $(u', v') \notin E'$ (chiameremo tali archi "archi bianchi")

Teorema 2.1. *Esiste una bigezione tra l'insieme $\mathcal{I}_{G,H}$ e l'insieme delle cricche del grafo $G \cdot H$. Inoltre, questa bigezione fa corrispondere isomorfismi massimali a cricche massimali e viceversa.*

Dimostrazione. ($\Rightarrow$) Sia $I \in \mathcal{I}$ e G' il sottografo corrispondente in G. Consideriamo ora l'insieme $C = \{(v, I(v)) : v \in G'\}$. Dato che I conserva l'etichettatura, sicuramente vale $l_G(v) = l_H(I(v))$ e quindi C è contenuto nell'insieme dei vertici del grafo $G \cdot H$. Se ora consideriamo due elementi distinti $(u, I(u))$ e $(v, I(v))$ di C, per definizione di isomorfismo vi è un arco tra u e v se e solo se vi è un arco tra $I(u)$ e $I(v)$; inoltre, essendo I bigettivo,

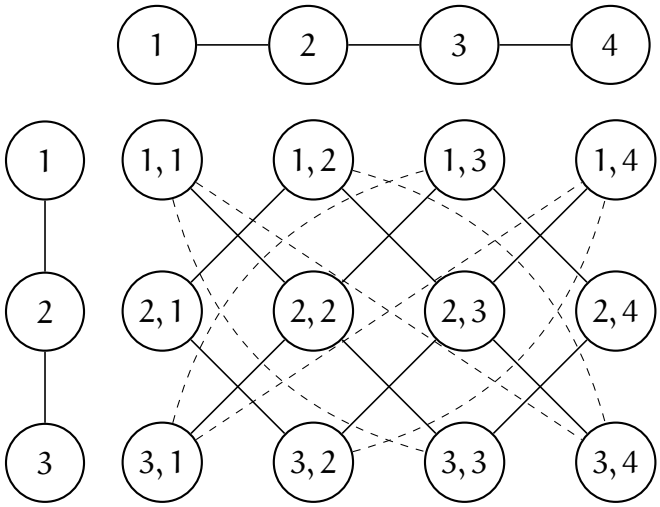

Figura 2.1: Prodotto modulare di due grafi. Gli archi tratteggiati rappresentano gli archi bianchi, mentre gli altri rappresentano quelli neri.

$(u, I(u)) \neq (v, I(v)) \Rightarrow u \neq v \wedge I(u) \neq I(v)$ e, quindi, per la definizione di prodotto modulare, esiste un arco tra $(u, I(u))$ e $(v, I(v))$. Di conseguenza C è una cricca.

($\Leftarrow$) Sia ora C una cricca in $G \cdot H$. Siano

$$G' = \{u \in G : \exists v \in H, (u, v) \in C\}$$

$$H' = \{v \in H : \exists u \in G, (u, v) \in C\}$$

Per la definizione di prodotto modulare, $(u, v) \in C \Rightarrow (u, v') \notin C$ se $v \neq v'$ dato che non esistono archi tra coppie con la prima componente uguale; analogamente $(u, v) \in C \Rightarrow (u', v) \notin C$ se $u \neq u'$. Segue quindi che posso definire una funzione $I : G' \to H'$ che manda u in quell'unico v tale che $(u, v) \in C$ e che tale funzione è bigettiva. Essendo C contenuto nell'insieme dei vertici di $G \cdot H$ e dato che un elemento (u, v) è un vertice di $G \cdot H$ se e solo se $l_G(u) = l_H(v)$, sicuramente I conserva l'etichettatura. Rimane da verificare che c'è un arco tra (u, v) se e solo se c'è tra $(I(u), I(v))$, ma questo segue direttamente dalla definizione di I e di prodotto modulare tra grafi.

È evidente che queste operazioni sono una l'inversa dell'altra e che dunque si tratta di una bigezione. È altrettanto evidente che $C' \supset C$ se e solo se $I' \supset I$, e dunque isomorfismi massimali corrispondono a cricche massimali e viceversa. □

Il seguente lemma mostra invece la relazione tra il prodotto modulare di grafi e i problemi MCLI.

Lemma 2.2. *Un isomorfismo* $I : G' \to H'$ *è tale che* G' *è connesso se e solo se il sottografo formato dai soli archi neri nella cricca* C *a esso corrispondente in* $G \cdot H$ *è connesso.*

Dimostrazione. ($\Rightarrow$) Siano $(u, I(u))$ e $(v, I(v))$ due vertici nella cricca C di $G \cdot H$. Poiché $u, v \in G'$ e G' è connesso, esiste una sequenza di vertici $u = v_1, \ldots, v_k = v$ con v_i e v_{i+1} connessi da un arco in G'. Allora anche $I(v_i)$ e $I(v_{i+1})$ sono connessi da un arco, e quindi vi è un arco nero tra $(v_i, I(v_i))$ e $(v_{i+1}, I(v_{i+1}))$. Di conseguenza $(u, I(u)) = (v_1, I(v_1)), \ldots, (v_k, I(v_k)) = (v, I(v))$ è un cammino composto di soli archi neri che congiunge $(u, I(u))$ e $(v, I(v))$.

($\Leftarrow$) Siano $u, v \in G'$. Poiché il sottografo di C formato dai soli archi neri è connesso, esiste un cammino tra $(u, I(u))$ e $(v, I(v))$ formato da soli archi neri, diciamo $(u, I(u)) = (v_1, I(v_1)), \ldots, (v_k, I(v_k)) = (v, I(v))$. Allora i vertici $u = v_1, \ldots, v_k = v$ formano un cammino in G'. □

2.2 Algoritmi per le cricche massimali

Esistono due famiglie di algoritmi per elencare le cricche massimali di un grafo. La prima è derivata dall'algoritmo di Bron-Kerbosch [3] e la seconda da quello di Tsukiyama [4].

2.2.1 Bron-Kerbosch

È possibile costruire un grafo orientato $\mathcal{C}_G$ che descrive le cricche[1] di un grafo G nel seguente modo: l'insieme dei vertici è formato da tutte le cricche del grafo G, ed è presente un arco da una cricca C a C' se e solo se esiste un vertice $v \in G$ per cui $C' = C \cup \{v\}$. È evidente che una cricca C è massimale se e solo se non esistono archi che hanno C come primo estremo.

Dimostriamo ora che l'algoritmo 1 (eseguito con parametri iniziali $C = \varnothing, P = V, X = \varnothing$) è corretto, ovvero che trova tutte le cricche massimali del grafo $G = (V, E)$. Per farlo, sarà fondamentale il seguente lemma.

Lemma 2.3. *Quando una chiamata a* BronKerbosch(C, P, X) *termina, tutte le cricche raggiungibili da* C *con archi di* $\mathcal{C}_G$ *e che non contengono elementi di* X *sono state esaminate esattamente una volta. Inoltre, nessuna cricca contenente elementi di* X *viene esaminata.*

Dimostrazione. Definiamo *altezza* di una cricca C (indicata con $h(C)$) la massima lunghezza di un cammino in $\mathcal{C}_G$ che inizia con C. Questa è una buona definizione, perché lungo un percorso di $\mathcal{C}_G$ la dimensione di una cricca aumenta: di conseguenza tutti i percorsi sono lunghi al massimo $|V|$.

[1] In questa sezione considereremo che l'insieme vuoto sia una cricca.

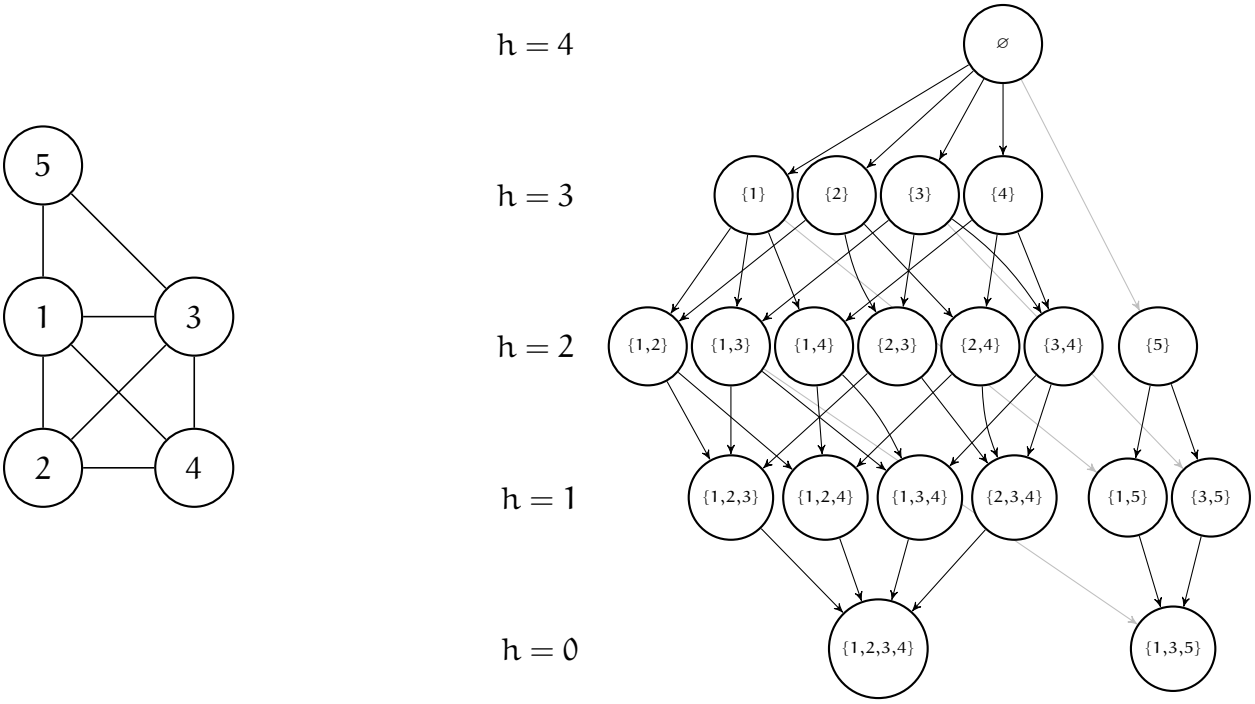

Figura 2.2: Un grafo G con il rispettivo grafo delle cricche $\mathcal{C}_G$ (in grigio gli archi lungo cui l'altezza cala di più di uno)

Algoritmo 1 L'algoritmo di Bron-Kerbosch. $P \cup X$ è l'insieme dei vertici che sono vicini di tutti gli elementi di C, mentre C è la cricca attualmente in esame. Inoltre $P \cap X = \varnothing$.

function BronKerbosch(C, P, X)
  if $P = \varnothing \wedge X = \varnothing$ **then**
    C è una cricca massimale
  end if.
  for all $v \in P$ **do**
    BronKerbosch($C \cup \{v\}, P \cap N(v), X \cap N(v)$)
    $P \leftarrow P \setminus \{v\}$
    $X \leftarrow X \cup \{v\}$
  end for.
end function.

Dimostriamo ora la tesi per induzione estesa sull'altezza di C.

Passo base: $h(C) = 0$. Se l'unico cammino che inizia in C è quello banale, allora C è una cricca massimale. Questo vuol dire che sia P che X sono vuoti, in quanto se esistesse $v \in P \cup X$ allora $C \cup \{v\}$ sarebbe una cricca più grande di C che la contiene. Quindi non vengono effettuate altre chiamate a BronKerbosch e la cricca C, l'unica raggiungibile da C, è stata esaminata esattamente una volta.

Passo induttivo. Consideriamo le cricche raggiungibili direttamente da C. Alcune di queste si otterranno da C aggiungendo un vertice in X. Poiché P e X sono disgiunti, nessuna di queste viene esplorata direttamente dalla visita di C; inoltre tutte le cricche raggiungibili da queste contengono elementi di X, e quindi non è necessario visitarle. Consideriamo ora l'ordine $v_1, \ldots, v_k$ in cui vengono esaminati i vertici in P dal ciclo for. La i-esima chiamata di BronKerbosch visiterà la cricca $C' = C \cup \{v_i\}$, con $P' = (P \setminus \{v_1, \ldots, v_{i-1}\}) \cap N(v_i)$ e $X' = (X \cup \{v_1, \ldots, v_{i-1}\}) \cap N(v_i)$: di conseguenza, per ipotesi induttiva (essendo $h(C') < h(C)$), visita tutte le cricche raggiungibili da $C \cup \{v_i\}$ che non contengono i vertici in X o i vertici $v_1, \ldots, v_{i-1}$. Poiché tutte le cricche raggiungibili da C che non contengono vertici di X (eccetto C stessa) contengono almeno un vertice in P, e poiché l'i-esima chiamata non visita cricche contenenti vertici v_j con $j < i$, ogni cricca raggiungibile da C è visitata esattamente una volta. Infine, tutte le cricche raggiungibili da $C \cup \{v_i\}$ e contenenti un vertice di X contengono un vertice di $X \cap N(v_i)$ e quindi, sempre per ipotesi induttiva, non sono visitate dall'i-esima chiamata di BronKerbosch (che è l'unica che potrebbe visitarle).

□

Teorema 2.4. *La funzione* BronKerbosch *chiamata con* $C = \varnothing, P = V, X = \varnothing$ *visita una e una sola volta tutte le cricche di* G.

Dimostrazione. È evidente che tutte le cricche C di G sono raggiungibili da $\varnothing$ con il cammino $\varnothing, \{v_1\}, \{v_1, v_2\}, \ldots, \{v_1, \ldots, v_k\} = C$. Essendo X vuoto, la tesi segue banalmente dal lemma precedente. □

Corollario 2.5. *Il tempo di esecuzione dell'algoritmo 1 è* $O(\Delta|\mathcal{C}|)$*, dove* $\mathcal{C}$ *indica l'insieme di tutte le cricche di* G*, e la memoria utilizzata è* $O(\Delta^2)$.

Dimostrazione. È evidente che tutte le operazioni sono eseguite una volta per cricca. Poiché possono essere eseguite tutte in tempo $O(1)$ tranne che le intersezioni con $N(v)$, che hanno un costo in tempo fino a $O(\Delta)$, il tempo di esecuzione complessivo è $O(\Delta|\mathcal{C}|)$. Inoltre, ogni chiamata ricorsiva ha bisogno di al massimo $O(\Delta)$ memoria e la profondità di ricorsione è al massimo la dimensione della massima cricca, e quindi sicuramente minore o uguale a Δ. Di conseguenza la memoria totale utilizzata è $O(\Delta^2)$. □

Esistono altre varianti di questo algoritmo (come quelle descritte in [5, 6]), ma non sono particolarmente utili per risolvere il problema considerato e pertanto non verranno esposte qui.

2.2.2 Tsukiyama

L'algoritmo di Tsukiyama si basa su una caratterizzazione induttiva delle cricche massimali di un grafo. Infatti, scelto un vertice v, esse possono essere formate da una cricca massimale C di $G \setminus \{v\}$ in uno dei due seguenti modi:

- $C \cup \{v\}$ se $N(v) \supset C$
- C e $(C \cap N(v)) \cup \{v\}$ se $N(v) \not\supset C$

D'altra parte, estendere la lista di cricche massimali di $G \setminus \{v\}$ in quella di G può inserire duplicati nella lista. L'algoritmo di Tsukiyama inserisce quindi una nuova cricca nella lista delle cricche di G solo quando la cricca di partenza è lessicograficamente massima tra tutte quelle che possono generarla.

In questo modo si ottiene un algoritmo che genera tutte le cricche di un grafo G in tempo $O(nm|\bar{C}|)$ (dove $\bar{C}$ indica l'insieme delle cricche massimali di G), successivamente migliorato (vedi [7]) in $O(\Delta^4|\bar{C}|)$ o $O(M(n)|\bar{C}|)$[2].

Per quanto l'algoritmo di Tsukiyama abbia proprietà teoriche migliori, esso risulta in pratica più lento dell'algoritmo di Bron-Kerbosch. Inoltre le idee dell'algoritmo non si adattano altrettanto facilmente al problema di generare gli isomorfismi tra sottografi *connessi*.

2.3 L'algoritmo di Koch

Da quanto detto precedentemente, dunque, per risolvere un problema MLI è sufficiente costruire il prodotto modulare dei due grafi e applicare un algoritmo di ricerca delle cricche massimali su questo.

Per risolvere un problema MCLI, invece, la riduzione di Levi non è sufficiente e si rende necessario sfruttare le informazioni sul colore degli archi nel grafo prodotto. Ciò è stato fatto inizialmente nell'algoritmo 2, dovuto a Ina Koch [8].

Da qui in poi, scrivendo "cricca connessa" intenderemo "cricca connessa con i soli archi neri". Inoltre, $N_b(v)$ e $N_w(v)$ indicheranno rispettivamente i vertici di G vicini a v con un arco nero o bianco, rispettivamente. L'intero n indicherà il numero di vertici del grafo G, indicati con $v_1, \ldots, v_n$, e V_i indicherà l'insieme (eventualmente vuoto) $\{v_1, \ldots, v_i\}$.

[2] $M(n)$ indica il tempo necessario a moltiplicare tra loro due matrici quadrate $n \times n$.

Algoritmo 2 L'algoritmo di Koch. $\mathsf{RP} \cup \mathsf{RX}$ è l'insieme dei vertici che sono vicini di tutti gli elementi di C e vicini ad almeno uno di essi con un arco nero, $\mathsf{UP} \cup \mathsf{UX}$ è l'insieme dei vertici vicini ad elementi di C solamente con archi bianchi, mentre C è la cricca connessa attualmente in esame. Inoltre $\mathsf{RP} \cap \mathsf{UX} = \varnothing$, $\mathsf{UP} \cap \mathsf{UX} = \varnothing$.

function KOCH($\mathsf{C}, \mathsf{RP}, \mathsf{UP}, \mathsf{RX}, \mathsf{UX}$)
  if $\mathsf{RP} = \varnothing \wedge \mathsf{RX} = \varnothing$ **then**
    C è una cricca connessa massimale
  end if.
  for all $v \in \mathsf{RP}$ **do**
    $\mathsf{RP}' \leftarrow (\mathsf{RP} \cup (\mathsf{UP} \cap \mathsf{N}_b(v))) \cap \mathsf{N}(v)$
    $\mathsf{RX}' \leftarrow (\mathsf{RX} \cup (\mathsf{UX} \cap \mathsf{N}_b(v))) \cap \mathsf{N}(v)$
    $\mathsf{UP}' \leftarrow \mathsf{UP} \cap \mathsf{N}_w(v)$
    $\mathsf{UX}' \leftarrow \mathsf{UX} \cap \mathsf{N}_w(v)$
    KOCH($\mathsf{C} \cup \{v\}, \mathsf{RP}', \mathsf{UP}', \mathsf{RX}', \mathsf{UX}'$)
    $\mathsf{RP} \leftarrow \mathsf{RP} \setminus \{v\}$
    $\mathsf{RX} \leftarrow \mathsf{RX} \cup \{v\}$
  end for.
end function.

La dimostrazione del Lemma 2.3 è sostanzialmente identica, a patto di considerare al posto dell'insieme $\mathcal{C}$ di tutte le cricche l'insieme $\mathcal{C}^c$ delle cricche connesse: è evidente, infatti, che se una cricca si ottiene da un'altra aggiungendo un vertice v tale vertice è collegato con un arco nero ad almeno un vertice della cricca iniziale e sta quindi in $\mathsf{RP} \cup \mathsf{RX}$.

Teorema 2.6. *La funzione* KOCH*, chiamata* n *volte con* $\mathsf{C} = \{v_i\}$, $\mathsf{RP} = \mathsf{N}_b(v_i) \setminus \mathsf{V}_{i-1}$, $\mathsf{UP} = \mathsf{N}_w(v_i) \setminus \mathsf{V}_{i-1}$, $\mathsf{RX} = \mathsf{V}_{i-1} \cap \mathsf{N}_b(v_i)$, $\mathsf{UX} = \mathsf{V}_{i-1} \cap \mathsf{N}_w(v_i)$ *rispettivamente, visita una e una sola volta tutte le cricche connesse del grafo* G.

Dimostrazione. Notiamo che basta dimostrare che l'i-esima chiamata visita tutte e sole le cricche connesse che hanno v_i come vertice di indice minimo. Una cricca connessa visitata nell'i-esima chiamata contiene sicuramente v_i e nessuno dei vertici precedenti per come sono stati scelti RX e UX.

Viceversa, supponiamo che una cricca connessa C contenga v_i e sia C_j una cricca connessa tale che $\{v_i\} \subset \mathsf{C}_j \subsetneq \mathsf{C}$ e che $|\mathsf{C}_j| = j$. Ci basta dimostrare che esiste un'analoga cricca $\mathsf{C}_j \subset \mathsf{C}_{j+1} \subset \mathsf{C}$ raggiungibile da C_j per avere un cammino da $\{v_i\}$ a C nel grafo $\mathcal{C}^c_{\mathsf{G}}$. Da ciò segue per il Lemma 2.3 che C viene visitata nell'i-esima chiamata.

Sia $w \in \mathsf{C} \setminus \mathsf{C}_j$. Essendo C connessa con archi neri, esiste un cammino $w_1, \ldots, w_k$ da v_i a w che percorre solo archi neri e composto di vertici in C. Inoltre $w_1 \in \mathsf{C}_j$ e $w_k \notin \mathsf{C}_j$. Sia

l il minimo indice per cui $w_l \notin C_j$. Allora sicuramente $1 < l \leqslant k$ ed esiste un arco nero tra $w_{l-1} \in C_j$ e $w_l \notin C_j$. Dunque $C_{j+1} = C_j \cup \{w_l\}$ è raggiungibile da C_j e ha un vertice in più di essa. □

Analogamente al Corollario 2.5, si dimostra il seguente

Corollario 2.7. *Il tempo di esecuzione dell'algoritmo 2 è* $O(\Delta|\mathcal{C}^c|)$*, e la memoria utilizzata è* $O(\Delta^2)$*.*

Consideriamo ora di applicare l'algoritmo di Koch per risolvere un problema MCLI e analizziamone il costo computazionale. D'ora in avanti indicheremo con $\mathcal{I}$ l'insieme degli isomorfismi tra i sottografi indotti di G e H che conservano l'etichettatura, con I un suo elemento, con I_G e I_H il rispettivo dominio ed immagine e con $N_G(u)$ e $N_H(v)$ rispettivamente i vicini di u e v in G e H. Con leggero abuso di notazione, $|I|$ indicherà la dimensione dell'insieme I_G e Δ e n rispettivamente il massimo grado di un qualsiasi vertice e il massimo numero di vertici nei due grafi. Assumeremo inoltre che Δ sia piccolo e, quindi, che entrambi i grafi siano sparsi e che le etichettature dei grafi abbiano come insieme immagine l'insieme $\{1, \ldots, k\}$ e indicheremo con L la quantità $\sum_{i=1}^{k} |l_G^{-1}(i)||l_H^{-1}(i)|$, ovvero il numero di vertici di $G \cdot H$. Infine supporremo che[3] $L > \frac{n^2}{k}$ e che k sia piccolo (in generale, il problema diventa più facile al crescere di k).

Lemma 2.8. *In questa situazione, il grado di un vertice di* $G \cdot H$ *è almeno* $L/2$*.*

Dimostrazione. È sufficiente contare i vertici (u', v') a cui il vertice (u, v) non può essere vicino e dimostrare che sono meno di $L/2$.

Un vertice (u', v') non è vicino di (u, v) se e solo se $u' \in N_G(u) \wedge v' \notin N_H(v)$ o viceversa. Quindi, detto $i_G = l_G^{-1}(i)$ e analogamente i_H, il numero di non vicini è

$$\sum_{i=1}^{k} |i_G \cap N_G(u)||i_H \setminus N_H(v)| + |i_G \setminus N_G(u)||i_H \cap N_H(v)| \leqslant 2\Delta n \leqslant \frac{n^2}{2k} < L/2$$

□

Lemma 2.9. *Non vi sono cricche di cardinalità maggiore di* n *nel grafo prodotto.*

Dimostrazione. Ovvio nel momento in cui si considera la bigezione tra cricche e isomorfismi. □

[3]Nella maggior parte dei casi reali, i vertici dei due grafi rappresentano oggetti dello stesso tipo. Di conseguenza $|l_G^{-1}(i)| \approx |l_H^{-1}(i)|$ e dunque il numero di vertici del grafo prodotto è circa $\sum_{i=1}^{k} |l_G^{-1}(i)|^2 \geqslant \sum_{i=1}^{k} \left(\frac{n}{k}\right)^2 = \frac{n^2}{k}$.

Costruire il grafo prodotto porta di conseguenza ad occupare $O(L^2)$ memoria, che rende l'algoritmo improponibile anche per grafi relativamente piccoli. È in realtà piuttosto facile modificare l'algoritmo in modo che ciò non si renda necessario, riducendo il costo in memoria a $O(Ln)$. In ogni caso, il costo in tempo è di $O(L|\mathcal{J}|)$.

2.4 Miglioramenti

L'algoritmo di Koch è migliorabile, infatti non usa alcuna informazione sulla struttura del grafo prodotto. Utilizzando queste informazioni, il miglioramento è in molti casi piuttosto rilevante; dato che vengono usate spesso informazioni sulla struttura del grafo, l'algoritmo sarà espresso riferendosi direttamente agli isomorfismi e ai due grafi originali.

Algoritmo 3 L'algoritmo di Koch implicito. N è l'insieme dei vertici che sono vicini ad almeno un elemento di I_G, mentre I è l'isomorfismo attualmente in esame. Inoltre X è l'insieme dei vertici di G da non considerare.

```
function KochImplicit(I, N, X)
    maximal ← true
    for all v ∈ N do
        C_v ← ⋂_{n∈N_G(v)∩I_G} N_H(I(n)) \ I_H
        C_v ← {c ∈ C_v : I^{-1}(N_H(c) ∩ I_H) = N_G(v) ∩ I_G}
        if C_v ≠ ∅ then
            maximal ← false
        end if.
        if v ∉ X then
            for all c ∈ C_v do
                KochImplicit(I ∪ {(v, c)}, N ∪ N_G(v) \ {v}, X)
            end for.
        end if.
        X ← X ∪ {v}
    end for.
    if maximal then
        I è un isomorfismo massimale
    end if.
end function.
```

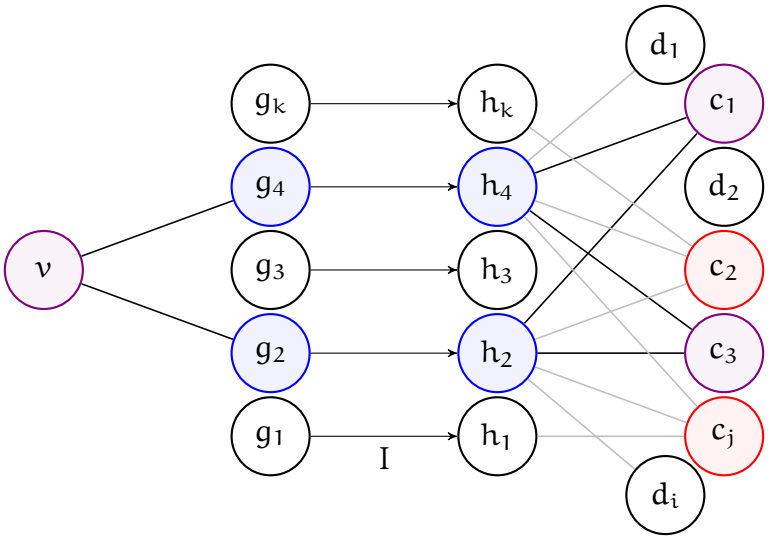

Figura 2.3: Rappresentazione dell'insieme C_v: i vertici in blu sono i vicini di v, i vertici in rosso sono i vertici che sono scartati da C_v perché hanno più vicini in I_H del necessario e i vertici in viola sono quelli parte di C_v.

Teorema 2.10. *La funzione nell'algoritmo 3 esegue le stesse chiamate che eseguirebbe l'algoritmo 2 sul grafo prodotto e, di conseguenza, è chiamato una e una sola volta per isomorfismo.*

Dimostrazione. Notiamo innanzitutto che il vertice (v, c) del grafo prodotto è collegato a tutti i vertici della cricca corrispondente all'isomorfismo se e solo se $I^{-1}(N_H(c) \cap I_H) = N_G(v) \cap I_G$ (chiameremo S l'insieme delle coppie (v, c) con questa proprietà). Inoltre tra essi vi è almeno un arco nero se e solo se questo insieme non è vuoto. Dato che l'insieme C_v contiene tutti e soli i vertici che verificano questa condizione quando sono abbinati con v (vedi figura 2.3), e dato che se (v, c) è collegato con almeno un arco nero allora $N_G(v) \cap I_G \neq \varnothing \Rightarrow v \in N$, l'insieme $N' = \{(v, c) : v \in N, c \in C_v\}$ contiene tutti e soli gli elementi di $RP \cup RX$ nell'algoritmo di Koch.

Vogliamo ora dimostrare che una chiamata a KochImplicit è equivalente a una chiamata a Koch con i parametri modificati nel seguente modo:

$$\begin{aligned} C &= \text{la cricca corrispondente a I} \\ RP &= \{(v, c) \in N' \quad : v \notin X\} \\ UP &= \{(v, c) \in S \setminus N' \; : v \notin X\} \\ RX &= \{(v, c) \in N' \quad : v \in X\} \\ UX &= \{(v, c) \in S \setminus N' \; : v \in X\} \end{aligned}$$

Innanzitutto è evidente che maximal = **true** alla fine dell'iterazione sui vertici in N se e solo se $C_v = \varnothing\ \forall v \in N$, ovvero se e solo se $RP = RX = \varnothing$.

La tesi sarebbe evidente se l'insieme X fosse l'insieme delle "coppie vietate" e non solo dei "vertici vietati", e se fosse aggiornato dopo ogni iterazione sui vertici in C_v. A causa dell'ordine in cui vengono eseguite le chiamate la cosa è però equivalente. Infatti in una chiamata più interna C_v non può ingrandirsi e di conseguenza è sufficiente escludere tutte le coppie con un dato primo membro invece che solo quelle già considerate: durante la chiamata ricorsiva sugli elementi di C_v sicuramente v non verranno prese in considerazione coppie con v come primo membro, mentre nelle chiamate successive tutte le coppie (v, c) saranno già state considerate precedentemente e andranno dunque escluse.

Di conseguenza è sufficiente mantenere l'insieme dei "vertici vietati", evitando di memorizzare tutte le coppie e risparmiando dunque notevole spazio in memoria. □

Corollario 2.11. *Il tempo di esecuzione dell'algoritmo 3 è* $O\left(\Delta^3 \sum_{I \in \mathcal{I}} |I|\right)$ *e la memoria occupata è* $O(n\Delta)$.

Dimostrazione. La funzione viene chiamata una volta per isomorfismo. L'unica cosa non banale è calcolare il numero di volte che viene eseguito il ciclo più esterno, che è pari a $N \leqslant |I| \cdot \Delta$. A questo punto tutte le operazioni interne al ciclo si eseguono in $O(\Delta^2)$ e quindi il tempo per isomorfismo è $O(\Delta^3|I|)$, da cui la tesi.

Per quanto riguarda la memoria, ogni chiamata ricorsiva occupa $O(\Delta)$ memoria e la profondità di ricorsione è al massimo n, dunque la memoria occupata è $O(n\Delta)$. □

Capitolo 3

Risultati sperimentali

In questa sezione vengono confrontati i risultati sperimentali ottenuti dagli algoritmi Koch e KochImplicit su alcuni grafi.

3.1 Implementazione

Il codice di entrambi gli algoritmi è stato scritto in C++ ed è disponibile in appendice A. In entrambi i casi i grafi vengono rappresentati in memoria mantenendo l'insieme dei vicini di ogni vertice in una struttura dati di tipo `unordered_set`.

Per l'algoritmo Koch non sono state necessarie particolari accortezze implementative, eccetto che evitare di costruire esplicitamente in memoria il grafo (cosa che avrebbe portato l'algoritmo originale a esaurire la memoria presente sulla macchina nella maggior parte dei casi).

Nell'algoritmo KochImplicit invece è presente una struttura dati apposita per rappresentare l'isomorfismo correntemente in esame, che consiste in un `vector` che contiene l'insieme dei vertici del primo grafo presenti nell'isomorfismo (e che consente di iterare sugli elementi dell'isomorfismo, di aggiungere un elemento e di rimuovere l'ultimo elemento in tempo costante) e due array in cui a ogni vertice di un grafo è associato il vertice a lui corrispondente nell'altro (o -1 se tale vertice non esiste).

3.2 Input

I grafi di input su cui sono stati provati gli algoritmi appartengono a una delle seguenti categorie:

- `path`: il grafo è un cammino, ovvero un grafo per cui esiste una permutazione dei vertici tale che due vertici sono connessi da un arco se e solo se sono vicini nella permutazione.
- `clique`: il grafo è una cricca.
- `random`: il grafo è stato generato casualmente, fissando il numero di vertici e di archi e assegnando gli archi in maniera equiprobabile, scegliendo M elementi in modo uniformemente casuale tra le $\binom{N}{2}$ coppie di vertici.

I grafi `long_path` e `very_long_path` sono della prima categoria. `clique` e `big_clique` sono della seconda, mentre tutti gli altri sono della terza. Le loro dimensioni sono riportate nella tabella 3.1. Il codice sorgente dei generatori di tali grafi è disponibile in appendice B.

3.3 Risultati

I test sono stati eseguiti su un computer con processore Intel i7-3770K, 16GB di RAM e sistema Arch Linux con kernel 4.2. Ogni algoritmo è stato eseguito ricevendo in input due differenti grafi generati con gli stessi parametri.

In tabella 3.1 sono riportati i parametri con cui sono stati generati i grafi (N indica il numero di vertici, M il numero di archi e k il numero di etichette distinte), il massimo grado in entrambi i grafi generati e i tempi ottenuti dai due algoritmi. È evidente che i tempi di esecuzione di KochImplicit sono minori su grafi sparsi e maggiori su quelli densi.

Questi risultati sono d'altra parte in linea con l'analisi. Per grafi densi si ha $\Delta^3|I| \approx N^4$, mentre per quelli sparsi Δ è piccolo e in genere minore di 20; invece in entrambi i casi il numero L di vertici nel grafo prodotto è circa N^2.

Inoltre, abbiamo eseguito un test su due grafi reali che rappresentano proteine su un'altra macchina, con processore Intel Xeon E5-2620, 128GB di RAM e sistema Ubuntu 14.04.1 con kernel 3.16. I grafi sono stati scaricati dal sito `http://www.eecs.wsu.edu/mgd/gdb.html` e contengono rispettivamente 4384 e 2871 vertici e 4500 e 2948 archi. Visto il basso numero di etichette (3, corrispondenti a atomi di carbonio, ossigeno e azoto), vi è un numero molto alto di isomorfismi e dunque le esecuzioni richiedono molto tempo; di conseguenza gli algoritmi sono stati terminati dopo 1000 minuti. L'algoritmo Koch ha trovato 87 isomorfismi massimali, occupando fino a 99GB di RAM, mentre l'algoritmo KochImplicit ha trovato 527 isomorfismi massimali, con un consumo di memoria inferiore a 15MB.

	N	M	k	Δ	Koch	KochI
long_path	100	99	5	2	0.7s	0.01s
very_long_path	300	299	10	2	12.6s	0.01s
small_random	50	50	2	5	1.8s	0.04s
random	100	100	4	7	11.6s	0.10s
mb_random	200	200	5	7	30s	0.07s
big_random	500	500	15	7	41s	0.03s
huge_random	5000	10000	40	14	46h	0.77s
few_colors	500	500	5	7	2h	1.99s
medium_colors	500	500	50	7	4.8s	0.01s
medium_colors_huge	5000	10000	500	14	40m	0.09s
many_colors	500	500	100	7	1.1s	0.01s
many_colors_huge	5000	5000	1000	9	10m	0.06s
many_colors_very_huge	40000	100000	9999	17	$>$ 100h	1.5s
clique	13	78	4	12	0.8s	0.8s
small_clique	8	28	1	7	1.9s	1.5s
small_dense	80	1200	80	58	0.03s	0.1s
dense	100	2000	100	70	0.05s	0.4s
big_dense	150	4500	150	114	0.53s	14s

Figura 3.1: Parametri per generare i casi di prova e risultati (N: numero di vertici, M: numero di archi, k: numero di etichette distinte).

Capitolo 4

Conclusioni e sviluppi futuri

Abbiamo analizzato alcuni degli algoritmi esistenti per risolvere il problema dell'isomorfismo tra sottografi e da uno di questi ne abbiamo derivato l'algoritmo KochImplicit, che presenta una maggiore efficienza sia dal punto di vista del tempo impiegato che della memoria su grafi sparsi. Abbiamo anche confermato che tale miglioramento non è solo teorico con vari esperimenti su grafi sintetici.

Una possibile direzione per future indagini riguarda la famiglia di algoritmi di Tsukiyama: tali algoritmi infatti hanno un tempo di esecuzione polinomiale moltiplicato per il numero di cricche *massimali* del grafo, al contrario dell'algoritmo di Bron-Kerbosch che invece ha tempo polinomiale moltiplicato per il numero di *tutte* le cricche del grafo. Tuttavia non è semplice adattare la definizione ricorsiva delle cricche data da Tsukiyama al caso di cricche connesse e dunque si rende necessario sviluppare nuove tecniche algoritmiche che estendano l'algoritmo di Tsukiyama.

Bibliografia

[1] Harary, F. (1972) “8. Line Graphs”, Graph Theory, Massachusetts: Addison-Wesley, pp. 71–83.

[2] Levi, G. (1973), “A note on the derivation of maximal common subgraphs of two directed or undirected graphs”, Calcolo 9 (4): 341–352, doi:10.1007/BF02575586.

[3] Bron, C.; Kerbosch, J. (1973), “Algorithm 457: finding all cliques of an undirected graph”, Communications of the ACM 16 (9): 575–577, doi:10.1145/362342.362367.

[4] Tsukiyama, S.; Ide, M.; Ariyoshi, I.; Shirakawa, I. (1977), “A new algorithm for generating all the maximal independent sets”, SIAM Journal on Computing 6 (3): 505–517, doi:10.1137/0206036.

[5] Eppstein, David; Löffler, Maarten; Strash, Darren (2010), “Listing all maximal cliques in sparse graphs in near-optimal time”, in Cheong, Otfried; Chwa, Kyung-Yong; Park, Kunsoo, 21st International Symposium on Algorithms and Computation (ISAAC 2010), Jeju, Korea, Lecture Notes in Computer Science 6506, Springer-Verlag, pp. 403–414, arXiv:1006.5440, doi:10.1007/978–3–642–17517–6_36.

[6] Tomita, Etsuji; Tanaka, Akira; Takahashi, Haruhisa (2006), “The worst-case time complexity for generating all maximal cliques and computational experiments”, Theoretical Computer Science 363 (1): 28–42, doi:10.1016/j.tcs.2006.06.015.

[7] Makino, K.; Uno, T. (2004), “New algorithms for enumerating all maximal cliques”, Algorithm Theory: SWAT 2004, Lecture Notes in Computer Science 3111, Springer-Verlag, pp. 260–272.

[8] Koch, I. (2001), “Enumerating all connected maximal common subgraphs in two graphs”, Theoretical Computer Science 250 (2001)

[9] McGregor, James J. “Backtrack search algorithms and the maximal common subgraph problem.”, Software: Practice and Experience 12.1 (1982): 23-34.

[10] Ullmann, Julian R. “An algorithm for subgraph isomorphism.”, Journal of the ACM (JACM) 23.1 (1976): 31-42.

Appendice A

Codice sorgente

A.1 Codice sorgente di Koch

```
#include <cstdio>
#include <unordered_set>
using namespace std;

int N1, N2, M1, M2;
unordered_set<int> *g1;
int* c1;
unordered_set<int> *g2;
int* c2;

int Np;
bool *fake;

inline int p2i(int a, int b) {
    return a*N2+b;
}

inline int p2i(pair<int, int> p) {
    return p2i(p.first, p.second);
}

inline pair<int, int> i2p(int i) {
    int f = i / N2;
    return {f, i - N2*f};
}

void print_clique(unordered_set<int>& clique) {
    printf("{");
    for (auto x: clique) {
        auto p = i2p(x);
        printf("%d, ", p.first);
    }
    printf("\b\b} -> {");
    for (auto x: clique) {
        auto p = i2p(x);
```

```
        printf("%d, ", p.second);
    }
    printf("\b\b}\n");
}

inline bool check_ok(pair<int, int> p1, pair<int, int> p2) {
    if (p1.first == p2.first) return false;
    if (p1.second == p2.second) return false;
    if (g1[p1.first].count(p2.first) != g2[p1.second].count(p2.second)) return false;
    return true;
}

void bk(unordered_set<int>& clique,
        unordered_set<int>& c_cand,
        unordered_set<int>& d_cand,
        unordered_set<int>& c_excl,
        unordered_set<int>& d_excl) {
    if (c_cand.empty() && c_excl.empty())
        return print_clique(clique);
    unordered_set<int> C_cand = c_cand;
    for (auto x: C_cand) {
        auto p1 = i2p(x);
        c_cand.erase(x);
        unordered_set<int> new_c_cand = c_cand;
        unordered_set<int> new_d_cand = d_cand;
        unordered_set<int> new_c_excl = c_excl;
        unordered_set<int> new_d_excl = d_excl;
        for (auto y: d_cand) {
            auto p2 = i2p(y);
            if (!check_ok(p1, p2)) {
                new_d_cand.erase(y);
                continue;
            }
            if (g1[p1.first].count(p2.first)) {
                new_d_cand.erase(y);
                new_c_cand.insert(y);
            }
        }
        for (auto y: d_excl) {
            auto p2 = i2p(y);
            if (!check_ok(p1, p2)) {
                new_d_excl.erase(y);
                continue;
            }
            if (g1[p1.first].count(p2.first)) {
                new_d_excl.erase(y);
                new_c_excl.insert(y);
            }
        }
        unordered_set<int> new_clique = clique;
        new_clique.insert(x);
        for (auto y: c_cand) {
            auto p2 = i2p(y);
            if (!check_ok(p1, p2)) {
                new_c_cand.erase(y);
            }
        }
```

```
        for (auto y: c_excl) {
            auto p2 = i2p(y);
            if (!check_ok(p1, p2)) {
                new_c_excl.erase(y);
            }
        }
        bk(new_clique, new_c_cand, new_d_cand, new_c_excl, new_d_excl);
        c_excl.insert(x);
    }
}

void bk_start() {
    unordered_set<int> used;
    unordered_set<int> c_cand;
    unordered_set<int> d_cand;
    unordered_set<int> c_excl;
    unordered_set<int> d_excl;
    unordered_set<int> clique;
    for (int i=0; i<Np; i++) {
        if (fake[i]) continue;
        auto p1 = i2p(i);
        c_cand.clear();
        d_cand.clear();
        c_excl.clear();
        d_excl.clear();
        clique.clear();
        for (int x=0; x<Np; x++) {
            if (fake[x]) continue;
            auto p2 = i2p(x);
            if (!check_ok(p1, p2)) continue;
            if (g1[p1.first].count(p2.first)) {
                if (!used.count(x)) {
                    c_cand.insert(x);
                } else {
                    c_excl.insert(x);
                }
            } else {
                if (!used.count(x)) {
                    d_cand.insert(x);
                } else {
                    d_excl.insert(x);
                }
            }
        }
        clique.insert(i);
        bk(clique, c_cand, d_cand, c_excl, d_excl);
        used.insert(i);
    }
}

int main(int argc, char** argv) {
    if (argc < 3) {
        fprintf(stderr, "Usage: %s g1 g2\n", argv[0]);
        return 1;
    }
    FILE* f1 = fopen(argv[1], "r");
    fscanf(f1, "%d%d", &N1, &M1);
```

```
    g1 = new unordered_set<int>[N1];
    c1 = new int[N1];
    for (int i=0; i<N1; i++) fscanf(f1, "%d", c1+i);
    for (int i=0; i<M1; i++) {
        int a, b;
        fscanf(f1, "%d%d", &a, &b);
        g1[a].insert(b);
        g1[b].insert(a);
    }
    FILE* f2 = fopen(argv[2], "r");
    fscanf(f2, "%d%d", &N2, &M2);
    g2 = new unordered_set<int>[N2];
    c2 = new int[N2];
    for (int i=0; i<N2; i++) fscanf(f2, "%d", c2+i);
    for (int i=0; i<M2; i++) {
        int a, b;
        fscanf(f2, "%d%d", &a, &b);
        g2[a].insert(b);
        g2[b].insert(a);
    }

    Np = N1*N2;
    fake = new bool[Np];
    for (int i=0; i<Np; i++) {
        auto p1 = i2p(i);
        if (c1[p1.first] != c2[p1.second])
            fake[i] = true;
    }
    bk_start();
}
```

A.2 Codice sorgente di KochImplicit

```
#include <cstdio>
#include <unordered_set>
#include <vector>

using namespace std;

int N1, N2, M1, M2;
unordered_set<int> *g1;
int *c1;
unordered_set<int> *g2;
int *c2;

class iso_t {
    private:
        vector<int> present;
        int* direct;
        int* inverse;
    public:
        iso_t(size_t g1_size, size_t g2_size) {
            direct = new int[g1_size];
            inverse = new int[g2_size];
            for (size_t i=0; i<g1_size; i++)
                direct[i] = -1;
```

```
            for (size_t i=0; i<g2_size; i++)
                inverse[i] = -1;
        }
        ~iso_t() {
            delete[] direct;
            delete[] inverse;
        }
        void append(int a, int b) {
            present.push_back(a);
            direct[a] = b;
            inverse[b] = a;
        }
        void print() {
            printf("{");
            for (auto x: present) {
                printf("%d, ", x);
            }
            printf("\b\b} -> {");
            for (auto x: present) {
                printf("%d, ", direct[x]);
            }
            printf("\b\b}\n");
        }
        void pop() {
            int to_erase = present.back();
            int other = direct[to_erase];
            direct[to_erase] = -1;
            inverse[other] = -1;
            present.pop_back();
        }
        bool direct_has(int d) {
            return direct[d] != -1;
        }
        bool inverse_has(int i) {
            return inverse[i] != -1;
        }
        int get_direct(int d) {
            return direct[d];
        }
        int get_inverse(int i) {
            return inverse[i];
        }
};

int rts = 0;

void bk(iso_t& iso,
        unordered_set<int>& neigh,
        vector<bool>& excl) {
    rts++;
    bool maximal = true;
    vector<int> friends;
    vector<int> candidates;
    vector<int> excl_added;
    vector<int> diff;
    vector<int> Neigh(neigh.begin(), neigh.end());
    for (auto n: Neigh) {
```

```
        if (!maximal && excl[n]) continue;
        friends.clear();
        candidates.clear();
        for (auto d: g1[n]) {
            if (iso.direct_has(d))
                friends.push_back(iso.get_direct(d));
        }
        for (auto c: g2[*friends.begin()]) {
            if (c1[n] != c2[c]) continue;
            bool ok = true;
            for (auto f: friends)
                if (!g2[f].count(c)) {
                    ok = false;
                    break;
                }
            if (iso.inverse_has(c)) ok = false;
            if (ok) candidates.push_back(c);
        }
        for (auto c: candidates) {
            bool ok = true;
            for (auto v: g2[c])
                if (iso.inverse_has(v) && !g1[n].count(iso.get_inverse(v))) {
                    ok = false;
                    break;
                }
            if (!ok) continue;
            maximal = false;
            if (excl[n]) break;
            diff.clear();
            iso.append(n, c);
            for (auto v: g1[n])
                if (!neigh.count(v) && !iso.direct_has(v))
                    diff.push_back(v);
            for (auto v: diff)
                neigh.insert(v);
            neigh.erase(n);
            bk(iso, neigh, excl);
            neigh.insert(n);
            for (auto v: diff)
                neigh.erase(v);
            iso.pop();
        }
        if (!excl[n]) {
            excl_added.push_back(n);
            excl[n] = 1;
        }
    }
    if (maximal)
        iso.print();
    for (auto e: excl_added)
        excl[e] = 0;
}

void bk_start() {
    vector<bool> excl(N1);
    iso_t iso(N1, N2);
    for (int i=0; i<N1; i++) {
```

```
        unordered_set<int> neigh = g1[i];
        for (int j=0; j<N2; j++) {
            if (c1[i] != c2[j]) continue;
            iso.append(i, j);
            bk(iso, neigh, excl);
            iso.pop();
            excl[i] = 1;
        }
    }
}

int main(int argc, char** argv) {
    if (argc < 3) {
        fprintf(stderr, "Usage: %s g1 g2\n", argv[0]);
        return 1;
    }
    FILE* f1 = fopen(argv[1], "r");
    fscanf(f1, "%d%d", &N1, &M1);
    g1 = new unordered_set<int>[N1];
    c1 = new int[N1];
    for (int i=0; i<N1; i++) fscanf(f1, "%d", c1+i);
    for (int i=0; i<M1; i++) {
        int a, b;
        fscanf(f1, "%d%d", &a, &b);
        g1[a].insert(b);
        g1[b].insert(a);
    }
    FILE* f2 = fopen(argv[2], "r");
    fscanf(f2, "%d%d", &N2, &M2);
    g2 = new unordered_set<int>[N2];
    c2 = new int[N2];
    for (int i=0; i<N2; i++) fscanf(f2, "%d", c2+i);
    for (int i=0; i<M2; i++) {
        int a, b;
        fscanf(f2, "%d%d", &a, &b);
        g2[a].insert(b);
        g2[b].insert(a);
    }
    bk_start();
}
```

Appendice B

Codice sorgente dei generatori

B.1 Codice sorgente di `gen_clique`

```
#!/usr/bin/env python2

import sys
import random

sz = int(sys.argv[1])
nc = int(sys.argv[2])
if len(sys.argv) > 3:
    random.seed(int(sys.argv[3]))

print sz, sz*(sz-1)/2
for i in xrange(sz):
    print random.randint(0, nc-1),
print
for i in xrange(sz):
    for j in xrange(1+i, sz):
        print i, j
```

B.2 Codice sorgente di `gen_path`

```
#!/usr/bin/env python2

import sys
import random

sz = int(sys.argv[1])
nc = int(sys.argv[2])
if len(sys.argv) > 3:
    random.seed(int(sys.argv[3]))

print sz, sz-1
for i in xrange(sz):
    print random.randint(0, nc-1),
print
```

```python
for i in xrange(sz-1):
    print i, i+1
```

B.3 Codice sorgente di `gen_random`

```cpp
#include "graph.hpp"
#include <cstdio>
#include <cstdlib>
#include <string>
#include <iostream>
using namespace std;

int main(int argc, char** argv){
    if (argc < 4) {
        fprintf(stderr, "Usage: %s N M color_number [seed]\n", argv[0]);
        return 1;
    }
    if (argc > 4) {
        Random::srand(atoi(argv[4]));
    }

    Graph::Graph<Graph::Undirected> g(atoi(argv[1]));
    g.add_edges(atoi(argv[2]), "rnd");
    g.shuffle();
    g.connect();
    cout << g.n_vertices() << " " << g.n_edges() << endl;
    for (int i=0; i<g.n_vertices(); i++)
        cout << Random::randint(atoi(argv[3])) << " ";
    cout << endl;
    for (auto edg: g)
        cout << edg << endl;
}
```

B.4 Codice sorgente di `graph.hpp`

Questo header contiene una libreria per la generazione di grafi, scritta principalmente per la preparazione dei problemi delle Olimpiadi di Informatica. Tale libreria è in grado di aggiungere in maniera uniformemente casuale un numero fissato di archi a un grafo, generare grafi con opportune strutture combinatorie, aggiungere il numero minimo di archi a un grafo per renderlo connesso ed eseguire varie operazioni tra grafi (differenza, differenza simmetrica, unione, unione disgiunta, prodotto, complementare). Inoltre può gestire sia grafi orientati che non orientati.

È stata scritta con particolare attenzione alla velocità di esecuzione. Nel namespace `Random` è implementato un generatore di numeri casuali a 64 bit, mentre nel namespace `SpeedHacks` è presente una funzione per convertire più velocemente un numero in stringa. Il namespace `Graph` contiene la parte principale del codice, che consiste nelle classi `DSU` (che implementa una struttura dati union-find), `Graph` (che contiene le funzioni principali

della libreria) e `Edge` (che fornisce le funzioni per gestire gli archi e per convertirli tra la rappresentazione come coppia e la rappresentazione come singolo intero).

```
#ifndef _Graph_HPP_
#define _Graph_HPP_
#include <vector>
#include "google_btree/btree_set.h"
#include <string>
#include <stdint.h>
#include <algorithm>
#include <cmath>
using namespace std;
using namespace btree;

namespace Random {

    uint64_t x = 8867512362436069LL;
    uint64_t w;

    /**
     *  Simple 64-bit variant of the XorShift random number algorithm.
     */
    inline uint64_t xor128() {
        uint64_t t;
        t = x ^ (x << 11);
        x = w;
        return w = w ^ (w >> 19) ^ (t ^ (t >> 8));
    }


    /**
     *  Random functions seed.
     *
     *  @param S seed
     */
    void srand(int S) {
        w = S;
    }

    /**
     *  Returns a random integer in the range [0, N).
     *
     *  @param N the maximum number to return
     */
    inline uint64_t randint(uint64_t N) {
        return xor128() % N;
    }

    /**
     *  Returns K distinct random integers in the range [0, N).
     *  Assumes K <= N
     *
     *  @param K the number of integers to return.
     *  @param N the maxium number to return
     */
    vector<uint64_t> randints(uint64_t K, uint64_t N){
        vector<uint64_t> r;
        for(unsigned i=0; i<K; i++) r.push_back(randint(N-K+1));
```

```
        sort(r.begin(), r.end());
        for(unsigned i=0; i<K; i++) r[i] += i;
        return r;
    }
}

namespace SpeedHacks {
    const char digit_pairs[201] = {
      "00010203040506070809"
      "10111213141516171819"
      "20212223242526272829"
      "30313233343536373839"
      "40414243444546474849"
      "50515253545556575859"
      "60616263646566676869"
      "70717273747576777879"
      "80818283848586878889"
      "90919293949596979899"
    };

    /**
     *  Dirty (but fast!) way to write a integer in a string. It cuts the time
     *  to print ~4 million integers from ~0.3s (with stringstream) to 0.1s.
     *
     *  Note: it probably won't work on ARM systems.
     *
     *  @param n the integer to write
     *  @param c pointer to the beginning of the string
     */
    char* itos(int n, char* c) {
        if(n==0) {
            *c = '0';
            return c+1;
        }

        int sign = -(n<0);
        unsigned int val = (n^sign)-sign;

        int size;
        if(val>=10000) {
            if(val>=10000000) {
                if(val>=1000000000) size=10;
                else if(val>=100000000) size=9;
                else size=8;
            } else {
                if(val>=1000000) size=7;
                else if(val>=100000) size=6;
                else size=5;
            }
        }
        else {
            if(val>=100) {
                if(val>=1000) size=4;
                else size=3;
            } else {
                if(val>=10) size=2;
                else size=1;
            }
```

```
        }
        size -= sign;
        if(sign) *c='-';
        c += size-1;
        while(val>=100) {
           int pos = val % 100;
           val /= 100;
           *(short*)(c-1)=*(short*)(digit_pairs+2*pos);
           c-=2;
        }
        while(val>0) {
            *c--='0' + (val % 10);
            val /= 10;
        }
        return c + size + 1;
    }
}

namespace Graph {

    /**
     *  Class that implements a Disjoint Set Union data structure.
     */
    class DSU {
    public:

        vector<int> dad;
        vector<int> rank;

        /**
         *  Creates N disjoint sets.
         *
         *  @param N the number of sets to create
         */
        DSU(int N) {
            dad.resize(N);
            rank.resize(N);
            for(int i=0; i<N; i++) dad[i] = i;
        }

        /**
         *  Finds the representative of x.
         *
         *  @param x what to find the representative of
         */
        inline int find(int x) {
            if (x == dad[x]) return x;
            return dad[x] = find(dad[x]);
        }

        /**
         *  Merges the set that contains x with the one that contains y
         *
         *  @param x an element in one of the sets to merge
         *  @param y an element in one of the sets to merge
         */
        inline void join(int x, int y) {
            int rx = find(x);
```

```
            int ry = find(y);
            if (rx == ry) return;
            if (rank[rx] < rank[ry]) {
                dad[rx] = ry;
                rank[ry] += rank[rx];
            } else {
                dad[ry] = rx;
                rank[rx] += rank[ry];
            }
        }
    };


    /**
     *  Template class for a graph. It takes a class that represents an edge
     *  type as a template parameter. See EdgeType, Undirected, Directed and
     *  DAG.
     *
     *  @tparam T type of the graph.
     */
    template<class Edge>
    class Graph {
        btree_set<Edge> edges;
        vector<int> labels;
        int vertices;

    public:

        /**
         *  Builds a graph with N vertices, labeled from 0 to N-1, and no edges.
         *
         *  @param N number of edges
         */
        Graph<Edge>(int N): vertices(N) {
            for(unsigned i=0; i<N; i++) labels.push_back(i);
        }

        /**
         *  Builds a graph from a given number of vertices and a given vector
         *  of edges.
         *
         *  @param V number of vertices
         *  @param E vector that contains the edges
         */
        Graph<Edge>(const int& V, const vector<Edge>& E) {
            vertices = V;
            insert_edges(E);
        }

        /**
         *  Builds a graph from a given number of vertices and given labels.
         *
         *  @param lbl the labels of the vertices
         *  @param E vector that contains the edges
         */
        Graph<Edge>(const vector<int>& lbl, const vector<Edge>& E) {
            vertices = lbl.size();
            labels = lbl;
```

```
    insert_edges(E);
}

/**
 *  Disjoint union with another graph.
 *
 *  @param other
 */
Graph<Edge>& operator+=(const Graph<Edge>& other) {
    vector<Edge> tmp;
    for(auto e: other.edges) {
        pair<int, int> p = e;
        p.first = other.labels[p.first] + vertices;
        p.second = other.labels[p.second] + vertices;
        tmp.push_back(Edge(p));
    }
    vertices += other.vertices;
    for(unsigned i=0; i<other.vertices; i++) {
        labels.push_back(other.labels[i]);
    }
    insert_edges(tmp);
    return *this;
}

/**
 *  Cartesian product with another graph. Not yet implemented.
 *
 *  @param other
 */
Graph<Edge>& operator*=(const Graph<Edge>& other) {
    return *this;
}

/**
 *  Union with another graph. Assumes that the graphs have the same
 *  number of vertices.
 *
 *  @param other
 */
Graph<Edge>& operator|=(const Graph<Edge>& other) {
    vector<Edge> tmp;
    tmp.resize(other.edges.size() + edges.size());
    auto it = set_union(edges.begin(), edges.end(),
                        other.edges.begin(), other.edges.end(),
                        tmp.begin());
    replace_edges(tmp.begin(), it);
    return *this;
}

/**
 *  Intersection with another graph. Assumes that the graphs have the
 *  same number of vertices.
 *
 *  @param other
 */
Graph<Edge>& operator&=(const Graph<Edge>& other) {
    vector<Edge> tmp;
    tmp.resize(other.edges.size() + edges.size());
```

```
    auto it = set_intersection(edges.begin(), edges.end(),
                               other.edges.begin(), other.edges.end(),
                               tmp.begin());
    replace_edges(tmp.begin(), it);
    return *this;
}

/**
 *  Subtraction of another graph. Assumes that the graphs have the same
 *  number of vertices.
 *
 *  @param other
 */
Graph<Edge>& operator-=(const Graph<Edge>& other) {
    vector<Edge> tmp;
    tmp.resize(other.edges.size() + edges.size());
    auto it = set_difference(edges.begin(), edges.end(),
                             other.edges.begin(), other.edges.end(),
                             tmp.begin());
    replace_edges(tmp.begin(), it);
    return *this;
}

/**
 *  Symmetric difference with another graph. Assumes that the graphs
 *  have the same number of vertices.
 *
 *  @param other
 */
Graph<Edge>& operator^=(const Graph<Edge>& other) {
    vector<Edge> tmp;
    tmp.resize(other.edges.size() + edges.size());
    auto it = set_symmetric_difference(edges.begin(),
                                       edges.end(),
                                       other.edges.begin(),
                                       other.edges.end(),
                                       tmp.begin());
    replace_edges(tmp.begin(), it);
    return *this;
}

/**
 *  Disjoint union of two graphs. Assumes that the first graph's labels
 *  are consecutive integers starting from 0, and the second graph's
 *   labels are non-negative integers.
 *
 *  @param other
 */
Graph<Edge> operator+(const Graph<Edge>& other) {
    Graph<Edge> tmp = *this;
    tmp += other;
    return tmp;
}

/**
 *  Cartesian product of two graphs. Not yet implemented.
 *
 *  @param other
```

```
 */
Graph<Edge> operator*(const Graph<Edge>& other) {
    Graph<Edge> tmp = *this;
    tmp *= other;
    return tmp;
}

/**
 *  Union of two graphs. Assumes that the graphs have the same number
 *  of vertices.
 *
 *  @param other
 */
Graph<Edge> operator|(const Graph<Edge>& other) {
    Graph<Edge> tmp = *this;
    tmp |= other;
    return tmp;
}

/**
 *  Intersection of two graphs. Assumes that the graphs have the same
 *  number of vertices.
 *
 *  @param other
 */
Graph<Edge> operator&(const Graph<Edge>& other) {
    Graph<Edge> tmp = *this;
    tmp &= other;
    return tmp;
}

/**
 *  Difference of two graphs. Assumes that the graphs have the same
 *  number of vertices.
 *
 *  @param other
 */
Graph<Edge> operator-(const Graph<Edge>& other) {
    Graph<Edge> tmp = *this;
    tmp -= other;
    return tmp;
}

/**
 *  Symmetric difference of two graphs. Assumes that the graphs have
 *  the same number of vertices.
 *
 *  @param other
 */
Graph<Edge> operator^(const Graph<Edge>& other) {
    Graph<Edge> tmp = *this;
    tmp ^= other;
    return tmp;
}

/**
 *  Complement of the graph.
 *
 *  @todo decide what should happen for directed graphs
```

```
 */
Graph<Edge> operator-() {
    Graph<Edge> all(vertices);
    all.add_edges(0, "clique");
    all -= *this;
    return all;
}

/**
 *  Check if two graph are equal.
 *
 *  @param other
 */
bool operator==(const Graph<Edge>& other) const {
    if(vertices != other.vertices) return false;
    auto me = edges.begin();
    auto ot = other.edges.begin();
    while (me != edges.end() && ot != other.edges.end()) {
        if (*me != *ot) return false;
        me++;
        ot++;
    }
    return true;
}

/**
 *  Check if two graph are different.
 *
 *  @param other
 */
bool operator!=(const Graph<Edge>& other) const {
    return !(*this == other);
}

/**
 *  Check if this graph is a proper subset of the second one.
 *
 *  @param other
 */
bool operator<(const Graph<Edge>& other) const {
    if(vertices != other.vertices) return false;
    return includes(other.edges.begin(), other.edges.end(),
                    edges.begin(), edges.end());
}

/**
 *  Check if this graph is a subset of the second one.
 *
 *  @param other
 */
bool operator<=(const Graph<Edge>& other) const {
    return *this == other || *this < other;
}

/**
 *  Check if this graph is a proper superset of the second one.
 *
 *  @param other
```

```
 */
bool operator>(const Graph<Edge>& other) const {
    return other < *this;
}

/**
 *  Check if this graph is a superset of the second one.
 *
 *  @param other
 */
bool operator>=(const Graph<Edge>& other) const {
    return other <= *this;
}

/**
 *  Adds an edge, if it is not already present in the graph.
 *
 *  @param e the edge to add
 */
void add_edge(Edge e) {
    edges.insert(e);
}

/**
 *  Removes an edge, if it is present in the graph.
 *
 *  @param e the edge to remove
 */
void del_edge(Edge e) {
    edges.erase(e);
}

/**
 *  Check if an edge is present in the graph.
 *
 *  @param e the edge to check
 */
bool has_edge(Edge e) {
    return edges.count(e);
}

/**
 *  Performs a permutation of the vertices of the graphs.
 *
 *  @param first first vertex to shuffle (default 0)
 *  @param last last vertex to shuffle (default -1, meaning last)
 */
void shuffle(int first = 0, int last = -1) {
    random_shuffle(labels.begin() + first,
                   labels.begin() + (last==-1 ? vertices : last),
                   Random::randint);
}

/**
 *  Replaces the current edges of the graph with the new range.
 *
 *  @param start iterator to the first edge
 *  @param end iterator to the next-after-last edge
```

```
 */
void replace_edges(typename vector<Edge>::iterator start,
                   typename vector<Edge>::iterator end) {
    btree_set<Edge> new_edges(start, end);
    edges.swap(new_edges);
}

/**
 *  Inserts the given range of edges in the graph.
 *
 *  @param start iterator to the first edge
 *  @param end iterator to the next-after-last edge
 */
 void insert_edges(typename vector<Edge>::const_iterator start,
                   typename vector<Edge>::const_iterator end) {
    vector<Edge> tmp;
    tmp.resize(end - start + edges.size());
    auto it = set_union(edges.begin(), edges.end(),
                        start, end,
                        tmp.begin());
    replace_edges(tmp.begin(), it);
 }

/**
 *  Inserts the given edges in the graph, in linear time.
 *
 *  @param edg vector of the (encoded) edges to add
 */
void insert_edges(const vector<Edge>& edg) {
    insert_edges(edg.begin(), edg.end());
}

/**
 *  Adds m edges to the graph. If type is 'rnd', it generates m edges
 *  that aren't already in the graph and adds them. Otherwise, it
 *  generates the desired structure of the graph and adds the edges
 *  if they aren't already in the graph.
 *
 *  @param m the number of edges to add
 *  @param type one of rnd, cycle, path, tree, forest, clique, star
 *  @todo add wheel, gearcaterpillar, lobster
 */
void add_edges(int m, string type = "rnd") {
    vector<Edge> tmp;
    tmp.reserve(m);
    if(type == "cycle") {
        for(int i=1; i<vertices; i++) {
            tmp.push_back(Edge(i-1, i));
        }
        tmp.push_back(Edge(vertices-1, 0));
    } else if (type == "path") {
        for(int i=1; i<vertices; i++) {
            tmp.push_back(Edge(i-1, i));
        }
    } else if (type == "tree") {
        for(int i=1; i<vertices; i++) {
            tmp.push_back(Edge(Random::randint(i), i));
```

```
            }
        } else if (type == "forest") {
            if (m > vertices-1) m = vertices-1;
            for(auto i: Random::randints(m, vertices-1)) {
                tmp.push_back(Edge(Random::randint(i+1), i+1));
            }
        } else if (type == "clique") {
            for(int i=0; i<vertices; i++) {
                for(int j=i+1; j<vertices; j++) {
                    tmp.push_back(Edge(i, j));
                }
            }
        } else if (type == "star") {
            for(int i=1; i<vertices; i++)
                tmp.push_back(Edge(0, i));
        } else {
            uint64_t big = Edge(vertices-1, vertices-1)+1;
            if(Edge(big) == big-1) big++;
            vector<Edge> selflinks;
            for(int i=0; i<vertices; i++) {
                uint64_t x = Edge(i, i);
                selflinks.push_back(Edge(x));
                if(Edge(x+1) == x) selflinks.push_back(Edge(x+1));
            }
            vector<Edge> forbidden(selflinks.size() + edges.size());
            auto it = set_union(edges.begin(), edges.end(),
                                selflinks.begin(), selflinks.end(),
                                forbidden.begin());
            uint64_t nfree = big - (it - forbidden.begin());
            if(nfree < m) m = nfree;
            vector<uint64_t> t = Random::randints(m, nfree);
            for(unsigned i=0; i<t.size(); i++) tmp.push_back(Edge(t[i]));
            int skipped = 0;
            int cur = 0;
            for(auto used: forbidden) {
                while(cur != tmp.size() && tmp[cur] + skipped < used) {
                    tmp[cur++] += skipped;
                }
                if(cur == tmp.size()) break;
                if(tmp[cur] + skipped == used) skipped++;
            }
        }
        insert_edges(tmp);
    }

    /**
     *  Adds the minimum number of edges required to make the graph
     *  (weakly) connected.
     */
    void connect() {
        DSU u(vertices);
        vector<Edge> tmp;
        for(auto e: edges) {
            pair<int, int> p = e;
            u.join(p.first, p.second);
        }
        int li = 0;
```

```
        for(int i=0; i<vertices; i++) {
            if(u.find(i) != u.find(li)) {
                u.join(li, i);
                tmp.push_back(Edge(li, i));
                li = i;
            }
        }
        insert_edges(tmp);
    }

    /**
     *  Generates a string representation of the graph, in the format that
     *  is most used at the Informatics Olympiads: first line contains the
     *  number of vertices and edges, and the following lines contain two
     *  integers a, b that represent an edge from a to b.
     *
     *  @param start the first number of a vertex
     */
    string to_string(int start = 0) const {
        // 22 is the maximum number of bytes that a pair of ints can take
        string s;
        s.resize(22 * (edges.size() + vertices + 1));
        char* c = &s[0];
        c = SpeedHacks::itos(vertices, c);
        *c++ = ' ';
        c = SpeedHacks::itos(edges.size(), c);
        *c++ = '\n';
        for(auto e: edges) {
            pair<int, int> p = e;
            c = SpeedHacks::itos(labels[p.first], c);
            *c++ = ' ';
            c = SpeedHacks::itos(labels[p.second], c);
            *c++ = '\n';
        }
        s.resize(c - &s[0] - 1);
        return s;
    }

    typedef typename btree_set<Edge>::iterator iterator;

    /**
     *  Returns edges.begin()
     */
    typename btree_set<Edge>::iterator begin() {
        return edges.begin();
    }

    /**
     *  Returns edges.end()
     */
    typename btree_set<Edge>::iterator end() {
        return edges.end();
    }

    /**
     *  Returns the number of vertices.
     */
```

```
    int n_vertices() {
        return vertices;
    }

    /**
     *  Returns the number of edges.
     */
    int n_edges() {
        return edges.size();
    }

    /**
     *  Writes a graph to an ostream. Used by boost_python to implement
     *  __str__.
     *
     *  @param os the ostream
     *  @param g the graph
     */
    friend ostream& operator<<(ostream& os, const Graph<Edge>& g) {
        return os << g.to_string();
    }
};

/**
 *  Template for edges.
 */
template <uint64_t       (_enc)(pair<int, int>),
          pair<int, int> (_dec)(uint64_t)>
class Edge {
    uint64_t x;
public:

    /**
     *  Construct an edge from its integer representation.
     */
    Edge<_enc, _dec>(uint64_t x): x(x) {}

    /**
     *  Returns the integer representation.
     */
    operator uint64_t() {
        return x;
    }

    /**
     *  Returns the vertices that the edge connects.
     */
    operator pair<int, int>() {
        return _dec(x);
    }

    /**
     *  Creates an edge from the vertices it connects.
     *
     *  @param first starting edge
     *  @param second ending edge
     */
    Edge<_enc, _dec>(int first, int second) {
```

```
    x = _enc(make_pair(first, second));
}

/**
 *  Creates an edge from the vertices it connects.
 *
 *  @param first starting edge
 *  @param second ending edge
 */
Edge<_enc, _dec>(pair<int, int> p) {
    x = _enc(p);
}

/**
 *  Default constructor.
 */
Edge<_enc, _dec>() {}

/**
 *  Writes an edge to an ostream. Used by boost_python to implement
 *  __str__.
 *
 *  @param os the ostream
 *  @param e the edge
 */
friend ostream& operator<<(ostream& os, const Edge<_enc, _dec>& e) {
    pair<int, int> p = _dec(e.x);
    return os << p.first << " " << p.second;
}

/**
 *  Checks if the edge is less then the other edge.
 *
 *  @param other
 */
inline bool operator<(const Edge<_enc, _dec>& other) const {
    return x < other.x;
}

/**
 *  Checks if the edge is equal to the other edge.
 *
 *  @param other
 */
inline bool operator==(const Edge<_enc, _dec>& other) const {
    return x == other.x;
}

/**
 *  Checks if the edge is different from the other edge.
 *
 *  @param other
 */
inline bool operator!=(const Edge<_enc, _dec>& other) const {
    return !(*this == other);
}

/**
```

```
        *  Checks if a certain representation corresponds to this edge.
        *
        *  @param other
        */
       inline bool operator==(const uint64_t& r) const {
           return _dec(x) == _dec(r);
       }

       /**
        *  Increments the edge representation.
        */
       Edge<_enc, _dec>& operator+=(const uint64_t& inc) {
           x += inc;
           return *this;
       }
   };

   /**
    *  Encodes an undirected edge (a, b), where we suppose a >= b, to
    *  integer a*(a+1)/2 + b.
    *
    *  @param e the pair representing the edge
    */
   inline uint64_t undirected_enc(pair<int, int> e) {
       uint64_t a = e.first < e.second ? e.second : e.first;
       uint64_t b = e.first < e.second ? e.first : e.second;
       return a*(a+1)/2 + b;
   }

   /**
    *  Decodes an undirected edge. The first vertex will be
    *  round(sqrt(2*(x+1))-1) and the second one will be x - a*(a+1)/2,
    *  given in a random order.
    *
    *  @param x encoded value of the edge
    */
   inline pair<int, int> undirected_dec(uint64_t x) {
       uint64_t a = round(sqrt(2*(x+1)))-1;
       uint64_t b = x - a*(a+1)/2;
       if(Random::randint(2)) return make_pair(b, a);
       else return make_pair(a, b);
   }

   /**
    *  Encodes a directed edge (a, b), where we suppose a >= b, to integer
    *  a*(a+1) + 2*b (+1 if the edge goes from a lower vertex to a
    *  greater one).
    *
    *  @param e the pair representing the edge
    */
   inline uint64_t directed_enc(pair<int, int> e) {
       uint64_t a = e.first < e.second ? e.second : e.first;
       uint64_t b = e.first < e.second ? e.first : e.second;
       return a*(a+1) + 2*b + (e.first<e.second);
   }

   /**
    *  Decodes a directed edge. The firstvertex will be
```

```
     *  round(sqrt(2*(x/2+1))-1) and the second one will be (x - a*(a+1))/2,
     *  given in an order decided by the parity of x.
     *
     *  @param x encoded value of the edge
     */
    inline pair<int, int> directed_dec(uint64_t x) {
        uint64_t a = round(sqrt(2*(x/2+1)))-1;
        uint64_t b = (x - a*(a+1))/2;
        if(x%2) return make_pair(b, a);
        else return make_pair(a, b);
    }

    /**
     *  Encodes a DAG edge (a, b), where we suppose a >= b, to integer
     *  a*(a+1)/2 + b.
     *
     *  @param e the pair representing the edge
     */
    inline uint64_t dag_enc(pair<int, int> e) {
        uint64_t a = e.first < e.second ? e.second : e.first;
        uint64_t b = e.first < e.second ? e.first : e.second;
        return a*(a+1)/2 + b;
    }

    /**
     *  Decodes a DAG edge. The first vertex will be round(sqrt(2*(x+1))-1)
     *  and the second one will be x - a*(a+1)/2.
     *
     *  @param x encoded value of the edge
     */
    inline pair<int, int> dag_dec(uint64_t x) {
        uint64_t a = round(sqrt(2*(x+1)))-1;
        uint64_t b = x - a*(a+1)/2;
        return make_pair(b, a);
    }

    typedef Edge<undirected_enc, undirected_dec> Undirected;
    typedef Edge<directed_enc, directed_dec> Directed;
    typedef Edge<dag_enc, dag_dec> DAG;

    /**
     *  Transposes a directed graph.
     *
     *  @param g the graph to transpose
     */

    Graph<Directed> transpose(Graph<Directed> g) {
        vector<Directed> tmp;
        for(auto e: g) {
            pair<int, int> p = e;
            swap(p.first, p.second);
            tmp.push_back(Directed(e));
        }
        return Graph<Directed>(g.n_vertices(), tmp);
    }
}
#endif
```