Algoritmi deterministici per il mantenimento di proprietà su grafi dinamici

Moreno Marzolla

16 marzo 1998

Ai miei genitori.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco degli Algoritmi

Sommario

Scopo della tesi è il progetto e l'analisi di algoritmi *deterministici* per il mantenimento di proprietà su grafi non orientati soggetti ad inserimenti e rimozioni di archi. Il contributo di questa tesi è quello di fornire algoritmi deterministici efficienti per la risoluzione dei seguenti problemi:

- mantenere una foresta ricoprente di peso minimo di un grafo G con n vertici, i cui archi possano avere al più k pesi distinti, in tempo ammortizzato $O(k \log^2 n)$ per inserimento o cancellazione di archi;
- mantenere una $1+\epsilon$-minima foresta ricoprente di G con archi di peso compreso tra 1 e U in tempo ammortizzato $O(\log^2 n \log(U(1+\epsilon))/\log(1+\epsilon))$ per operazione;
- testare se G è bipartito in tempo $O(1)$ nel caso pessimo, supportando modifiche in tempo ammortizzato $O(\log^2 n)$ per operazione;
- testare se la rimozione di k archi dati rende il grafo sconnesso in tempo ammortizzato $O(k \log^2 n)$, supportando aggiornamenti in tempo ammortizzato $O(\log^2 n)$ per operazione;
- mantenere una decomposizione ricoprente massimale di ordine k di G in tempo ammortizzato $O(k \log^2 n)$ per operazione;

Questi algoritmi rappresentano un notevole miglioramento rispetto alle soluzioni deterministiche esistenti per la risoluzione dei medesimi problemi [23], ed eguagliano le prestazioni dei più efficienti algoritmi randomizzati esistenti [21].

Capitolo 1

Introduzione

In questo capitolo viene introdotto il problema del mantenimento di proprietà su strutture dati dinamiche in generale, e su grafi dinamici in particolare. Il capitolo si conclude illustrando lo scopo della tesi e i risultati conseguiti, con particolare riferimento alle applicazioni pratiche degli stessi.

1.1 Generalità sugli algoritmi dinamici

Molti problemi legati all'informatica, alla logistica, alla gestione di reti di comunicazione, trovano nei grafi una rappresentazione astratta naturale. La risoluzione del problema si riduce quindi al calcolo di una "proprietà" associata al grafo. Se la struttura del grafo è soggetta a variazioni, la proprietà deve essere ricalcolata nel modo più efficiente possibile.

Un algoritmo *statico* lavora su una istanza dell'input x per calcolare una qualche funzione $f(x)$; i dati di input x possono essere costituiti da strutture dati complesse, e il risultato della computazione $f(x)$ può rappresentare, oltre che un valore, una qualche "annotazione" dell'input. Se i dati in input sono soggetti a variazioni, e in seguito a ciascuna variazione Δx si vuole determinare il nuovo risultato $f(x + \Delta x)$, gli algoritmi statici non traggono vantaggio dalla conoscenza dei risultati precedenti, ma ricalcolano di volta in volta il valore della funzione f iniziando da zero. In molti casi, però, a "piccole" variazioni dell'input Δx corrispondono

minime differenze tra il risultato precedente $f(x)$ ed il nuovo risultato $f(x + \Delta x)$. Ad esempio, se x rappresenta un grafo non orientato con archi di peso non negativo, e $f(x)$ indica una foresta ricoprente di peso minimo di x, per ogni modifica dell'input Δx corrispondente all'inserimento o alla cancellazione di un singolo arco, $f(x+\Delta x)$ differirà da $f(x)$ al più in due archi (vedi paragrafo 2.5.2).

Algoritmi che fanno uso di soluzioni precedenti, e quindi risultano più efficienti rispetto a quelli che le ricalcolano di volta in volta, sono detti *algoritmi dinamici*. Un algoritmo dinamico viene inizializzato con un blocco di input x e con il corrispondente output $f(x)$. Al suo interno, l'algoritmo può mantenere delle informazioni ausiliarie per agevolare le computazioni successive; per ogni cambiamento nel blocco di input Δx, vengono utilizzate le informazioni ausiliarie e il precedente risultato $f(x)$ per calcolare il nuovo output $f(x + \Delta x)$, ove $x + \Delta x$ rappresenta l'input aggiornato con le variazioni descritte in Δx. Oltre a ciò, le informazioni interne dell'algoritmo vengono modificate di conseguenza (vedi figura 1.1).

1.2 Impieghi degli algoritmi dinamici

Gli algoritmi dinamici sono potenzialmente utili in tutti i contesti nei quali l'utente costruisce qualche struttura in modo graduale, e tale struttura deve essere ripetutamente processata durante le fasi della costruzione; ad esempio situazioni in cui l'utente crea e modifica oggetti in modo interattivo per elaborarli subito, effettuando eventualmente nuove modifiche prima di una ulteriore elaborazione (vedi figura 1.2).

Spesso le strutture dati non vengono modificate direttamente dall'utente, ma da altri programmi; ad esempio, i programmi compilatori, nella fase di ottimizzazione del codice, riordinano e cambiano le istruzioni allo scopo di migliorare le prestazioni del codice generato in termini di velocità di esecuzione o di memoria occupata. Per scegliere quali istruzioni modificare, il compilatore di solito si avvale di una rappresentazione interna (spesso sotto forma di grafo) del codice sorgente, sulla quale mantiene determinate informazioni (ad esempio, i blocchi di istruzioni irraggiun-

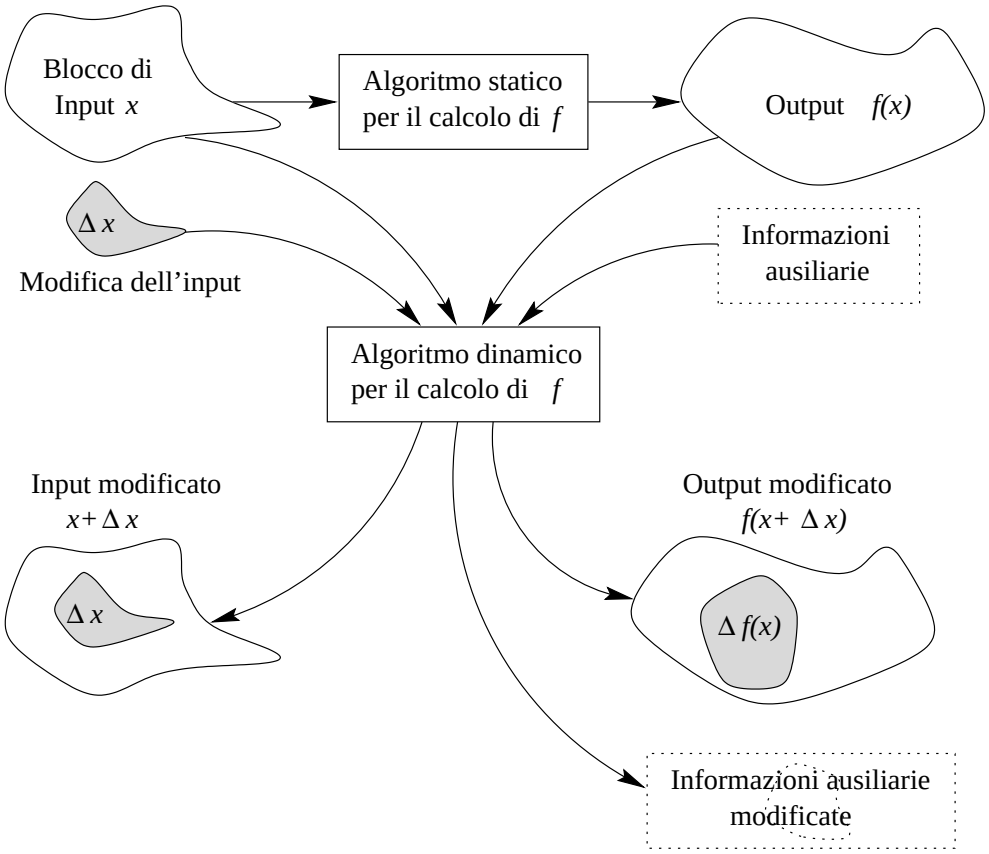


Figura 1.1: Lo schema illustra il principio di funzionamento di un generico algoritmo dinamico. Le parti scure rappresentano le modifiche dell'input e dell'output.

gibili, oppure quali registri del microprocessore sono disponibili in un certo punto, ecc.). Tutte queste informazioni devono essere aggiornate efficientemente mano a mano che il codice è soggetto a modifiche.

Vengono riportate in seguito alcune applicazioni degli algoritmi dinamici in generale, e di algoritmi su grafi dinamici in particolare.

Sviluppo di Software. Il tipico ciclo di sviluppo di un programma prevede i seguenti passi:

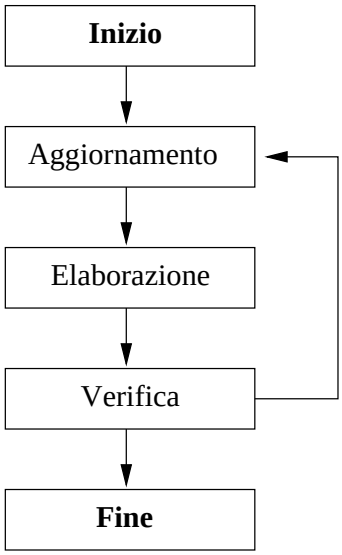


Figura 1.2: In figura è rappresentata la situazione tipica in cui è utile disporre di algoritmi dinamici. Un determinato oggetto viene modificato, quindi elaborato con un opportuno programma; dopo un eventuale controllo, il ciclo può essere ripetuto più volte.

1. Editare il codice sorgente;
2. Compilare il codice sorgente utilizzando un compilatore;
3. Individuare mediante un *debugger* eventuali errori logici, e correggerli tornando al punto 1.

Nel caso l'utente apporti ad ogni iterazione dei piccoli cambiamenti al programma sorgente, l'impiego di algoritmi dinamici durante le varie fasi della compilazione (analisi sintattica, semantica, analisi del flusso dei dati, compilazione, generazione del codice eseguibile) può ridurre drasticamente il tempo richiesto per l'esecuzione delle stesse.

Molto spesso, poi, la scrittura del codice sorgente avviene mediante *editor* in grado di riconoscere la sintassi dello specifico linguaggio di programmazione adottato, in modo da segnalare all'utente errori sintattici

(o anche semantici) durante la stesura del codice. Dato che tali editor devono effettuare una analisi sintattica mentre l'utente interviene sul sorgente, l'uso di algoritmi dinamici si è diffuso rapidamente in questo campo.

Elaborazione di Documenti. I passi necessari per completare la stesura di un documento, utilizzando strumenti di editoria elettronica, sono molto simili a quelli appena visti nel caso dello sviluppo di un programma. Il ciclo diviene in questo caso:

1. Editare il documento;
2. Formattare il documento con un programma apposito, per generarne una versione stampabile;
3. Stampare il documento
4. Leggere il documento per individuare gli errori; tornare al punto 1 per apportare le correzioni.

Anche in questo caso tutti i passaggi necessari per l'elaborazione del documento possono essere velocizzati impiegando algoritmi dinamici.

Disegno di Circuiti Integrati. Si supponga di volere disporre una serie di connettori su una piastra per circuiti integrati. Ciascun connettore ha una posizione ben definita sulla piastra, e tutti i connettori devono essere collegati insieme utilizzando fili di rame. È richiesta la determinazione dello schema di collegamento che minimizzi la lunghezza complessiva dei fili utilizzati.

Questo semplice problema può essere modellato mediante un grafo, i cui vertici rappresentano i connettori, e ogni coppia di vertici viene collegata da un arco di peso uguale alla distanza tra i connettori corrispondenti. Il problema è risolto calcolando un albero ricoprente di peso minimo del grafo. Se il calcolo deve essere effettuato mentre l'utente apporta modifiche al circuito, l'adozione di algoritmi dinamici permette di ridurre notevolmente i tempi di ricalcolo.

Linguaggi formali e Compilatori. Molti algoritmi su grafi trovano applicazione nel campo dello sviluppo di compilatori. I problemi su grafi che si devono affrontare in quest'area includono: l'individuazione dei vertici raggiungibili da un dato punto, la determinazione delle componenti connesse di un grafo, l'ordinamento topologico di un grafo diretto aciclico, la costruzione dell'*albero dominatore* di un grafo, la colorazione di un grafo [2], e l'individuazione degli archi equivalenti rispetto ai cicli. Molti sforzi sono stati compiuti recentemente nello sviluppo di compilatori incrementali, per i quali sono indispensabili algoritmi dinamici per la risoluzione di questi ed altri problemi.

Gestione di reti di comunicazione La diffusione di reti di comunicazione planetarie (Internet) che collegano milioni di utenti, o di calcolatori a elevato parallelismo in cui migliaia di processori devono comunicare in maniera efficiente, hanno dato notevole impulso allo studio di algoritmi dinamici su grafi relativamente ai problemi della connettività, del bipartitismo, della determinazione di foreste ricoprenti di peso minimo e di alberi dei cammini minimi.

Infatti, ogni rete di comunicazione può essere rappresentata da un grafo, eventualmente pesato, i cui vertici rappresentano gli *host* (o le CPU nel caso di calcolatori paralleli), e gli archi rappresentano linee fisiche di collegamento tra di essi. Di questo grafo occorre mantenere alcune proprietà, ad esempio una foresta ricoprente di peso minimo, oppure un albero di cammini minimi, allo scopo di instradare i messaggi nel percorso più conveniente. La struttura del grafo può cambiare nel tempo, perché interruzioni di collegamenti o di host, e loro successive riattivazioni, comportano inserimenti e cancellazioni di archi e vertici nel grafo rappresentante la rete. L'uso di algoritmi dinamici è in questi casi indispensabile per ricalcolare le proprietà associate al grafo nella maniera più efficiente possibile, aumentando così le prestazioni del sistema.

È curioso constatare che, nonostante gli indubbi vantaggi che comportano, gli algoritmi dinamici non siano ancora molto diffusi in questi campi, preferendo ancora le versioni statiche.

In un recente lavoro, Frigioni, Ioffreda, Nanni e Pasqualone [14] hanno

messo in luce la potenza degli algoritmi dinamici per il calcolo di un albero di cammini minimi (*single-source shortest path*) su un grafo pesato soggetto a inserimenti e rimozioni di archi, rispetto ad una implementazione statica efficiente dell'algoritmo di Dijkstra [9] per la risoluzione del medesimo problema. Gli algoritmi dinamici considerati sono quelli proposti da Ramalingam e Reps [42] e da Frigioni, Marchetti-Spaccamela e Nanni [16].

Uno dei grafi utilizzati nei test modellava una porzione della rete Internet, costituita da 1259 vertici e 5104 archi; a ciascun arco è stato assegnato peso 1. Su tale grafo sono state effettuate 10,000 operazioni di inserimento e rimozione di archi, ricalcolando dopo ciascuna l'albero dei cammini minimi. Gli algoritmi dinamici hanno impiegato un tempo complessivo corrispondente al 3-5% del tempo richiesto dall'algoritmo di Dijkstra, esaminando meno dello 0.5% degli archi totali (contro il 100% richiesto dall'algoritmo statico); i risultati della simulazione sono riportati nella tabella 1.1.

	Tempo totale (in secondi)			Tempo totale/Tempo(Dij)		
Sorgente	`Dij`	`RR`	`FMN`	`Dij`	`RR`	`FMN`
`AS1755`	12.615026	0.653482	0.462232	100.00%	5.18%	3.66%
`AS3561`	12.563434	0.651382	0.470504	100.00%	5.18%	3.75%
	Archi esaminati			Archi esaminati/Archi(Dij)		
Sorgente	`Dij`	`RR`	`FMN`	`Dij`	`RR`	`FMN`
`AS1755`	6,725,010	20,466	15,401	100.00%	0.30%	0.23%
`AS3561`	6,706,948	21,073	15,829	100.00%	0.31%	0.24%

Tabella 1.1: Confronto sperimentale tra l'algoritmo di Dijkstra (`Dij`) statico e gli algoritmi dinamici di Ramalingam e Reps (`RR`) e di Frigioni, Marchetti-Spaccamela e Nanni (`FMN`) relativamente al problema della determinazione di un albero di cammini minimi (*single source shortest paths*). Gli esperimenti sono stati compiuti su un grafo che modella una porzione della rete Internet, costituito da 1259 vertici e 5104 archi di peso 1, considerando come vertice sorgente i sistemi autonomi AS1755 e AS3561. Le tabella riportano il tempo totale e il numero di archi esaminati durante l'elaborazione di una sequenza casuale di 10,000 operazioni di inserimento e rimozione di archi.

Altri studi sperimentali condotti da Amato, Cattaneo e Italiano [4] sulle prestazioni di algoritmi dinamici per il mantenimento di una foresta ricoprente di peso minimo, e da Alberts, Cattaneo e Italiano [3] su algoritmi dinamici per la risoluzione del problema della connettività, hanno evidenziato l'utilità pratica e l'efficienza delle soluzioni dinamiche. Relativamente al problema della connettività, ad esempio, su una sequenza di 500 operazioni di aggiornamento in un grafo con 500 nodi per diverse densità, gli esperimenti hanno messo in evidenza che l'algoritmo dinamico ottenuto dall'applicazione della tecnica della *sparsificazione* [11] ad un algoritmo statico ha ottenuto prestazioni migliori del 30%, per grafi sparsi, fino al 6000% per grafi molto densi rispetto al solo algoritmo statico. Dal grafico riportato in figura 1.3 è possibile constatare l'incremento di efficienza dell'algoritmo dinamico rispetto a quello statico già su grafi sparsi.

1.3 Classificazione degli algoritmi su grafi dinamici

In questa tesi verranno presi in esame algoritmi su grafi dinamici, cioè algoritmi che mantengono determinate proprietà riguardanti un grafo soggetto a cambiamenti nel tempo. In letteratura, gli algoritmi su grafi dinamici vengono classificati in base all'entità oggetto di variazione in:

- **Algoritmi dinamici su Archi**, detti anche *edge-dynamic*, se è consentito inserire o cancellare solamente archi. In altre parole, dato un grafo di partenza $G = (V, E)$, gli algoritmi dinamici su archi consentono di modificare l'insieme E, lasciando V invariato.
- **Algoritmi dinamici su Vertici**, detti anche *vertex-dynamic*, se è consentita la rimozione di vertici, con tutti gli archi ad essi incidenti; questa categoria di problemi non è stata studiata a fondo, e uno dei primi risultati è riportato in [15].

Una ulteriore classificazione è possibile in base al tipo di operazioni di aggiornamento consentite:

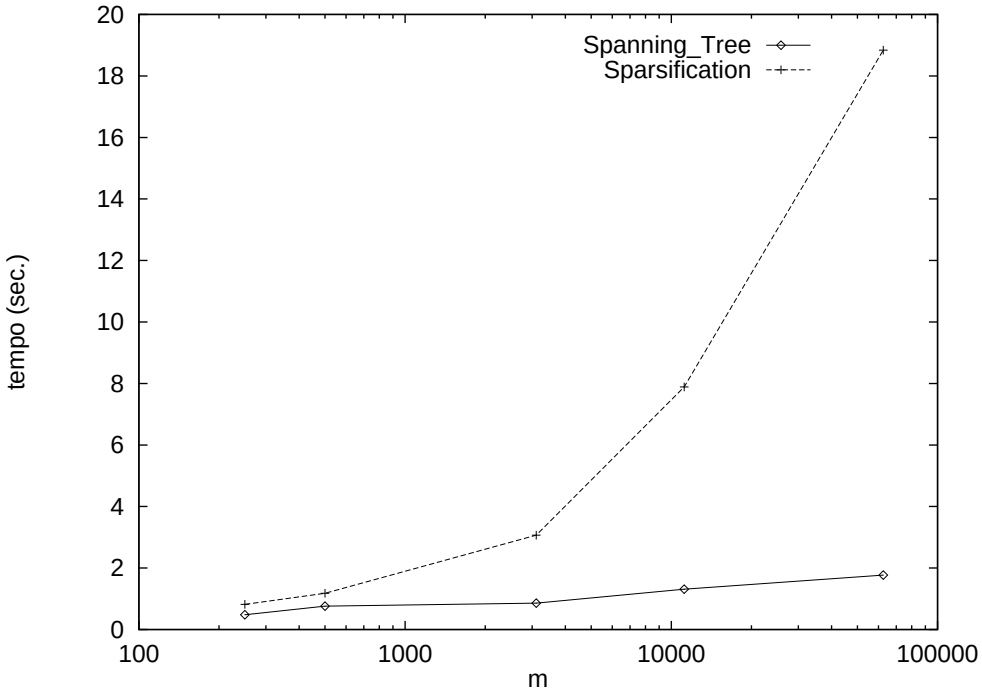


Figura 1.3: La figura illustra i tempo di esecuzione degli algoritmi `Spanning_Tree`, della libreria `LEDA` [36], e sparsificazione su `Spanning_Tree`, per il mantenimento di una foresta ricoprente su un grafo con $n = 500$ vertici al variare del numero m di archi, su una sequenza di 500 operazioni di aggiornamento. La figura è basata su dati forniti in [3].

- **Algoritmi totalmente dinamici** (*fully dynamic*), in cui sono consentiti sia inserimenti che rimozioni di archi (o vertici).
- **Algoritmi parzialmente dinamici**, se sono ammessi solo inserimenti (algoritmi *incrementali*) oppure cancellazioni (algoritmi *decrementali*).

Nella figura 1.4 è riportato uno schema riepilogativo sulla classificazione degli algoritmi su grafi dinamici.

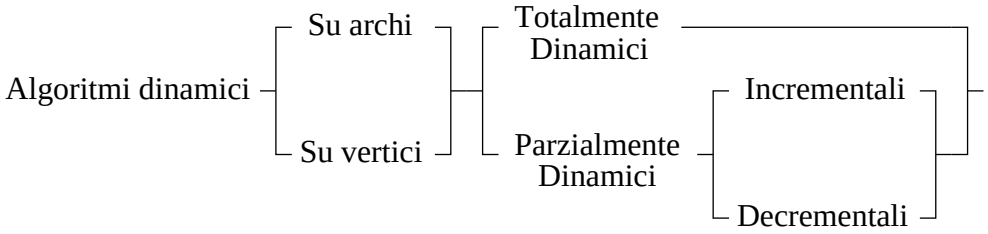


Figura 1.4: Riepilogo delle classificazioni degli algoritmi su grafi dinamici

Gli algoritmo presentati nei capitoli 3 e 4 sono *totalmente dinamici su archi*, in cui il grafo può essere modificato mediante inserimenti e rimozioni di archi.

1.4 Proprietà su grafi dinamici

Dato un grafo $G = (V, E)$, una proprietà $\mathcal{P}$ è una funzione che soddisfa una delle seguenti definizioni

(a) mappa G nell'insieme $\{\text{vero}, \text{falso}\}$, ossia $\mathcal{P}(G) =$ vero se e solo se la proprietà vale sul grafo G.

(b) mappa ogni tripla (G, v, w), con $v, w \in V$, nell'insieme $\{\text{vero}, \text{falso}\}$, ossia $\mathcal{P}(G, v, w) =$ vero se e solo se la proprietà vale nel grafo G relativamente ai vertici v e w.

In base a questa definizione, una foresta ricoprente di G non è una proprietà, dato che è un sottografo e non una funzione booleana. Visto l'importantissimo ruolo che le foreste ricoprenti svolgono nell'ambito delle applicazioni degli algoritmi su grafi dinamici, è conveniente estendere la definizione di proprietà per includere anche il punto seguente:

(c) mappa G in un suo sottografo G'.

Un esempio del tipo (a) è una funzione che mappa ogni grafo bipartito su vero, e ogni grafo non bipartito su falso. Un esempio per il tipo di proprietà (b) è una funzione che mappa (G, v, w) su vero se e solo se i

vertici v e w sono conessi nel grafo G. Infine, un esempio di proprietà di tipo (c) è una funzione che mappa ogni grafo G, ai cui archi è associato un peso, in una foresta ricoprente di peso minimo di G.

1.5 Algoritmi deterministici o randomizzati?

Relativamente agli algoritmi presentati in questa tesi, le migliori soluzioni precedenti sono di tipo *randomizzato*. Gli algoritmi randomizzati sono sempre stati considerati più pratici, semplici da capire e da programmare rispetto a quelli deterministici; esistono molti problemi lei cui soluzioni deterministiche hanno complessità esponenziale, mentre le controparti randomizzate hanno complessità subesponenziale, se non addirittura polinomiale [5, 8, 17, 29, 33]. Relativamente al problema della determinazione di una foresta ricoprente di peso minimo di un grafo pesato con n vertici ed m nodi, esiste un semplice algoritmo randomizzato avente complessità attesa $O(m + n)$ [30], mentre il più efficiente algoritmo deterministico ha complessità $O(m\alpha(m, n) \log \alpha(m, n))$, ove $\alpha(m, n)$ è l'inversa funzionale della funzione di Ackermann, ma è estremamente complesso e come tale poco adeguato ad un utilizzo in situazioni reali [7].

Nell'informatica moderna la randomizzazione è considerata sempre più come una risorsa, al pari del tempo di esecuzione o dello spazio di memoria richiesto da un algoritmo. Sebbene le ultime due siano state individuate già in origine come fattori importanti, solo recentemente si è riconosciuto che la generazione di bit *realmente* casuali è una impresa difficile [40]. Per molto tempo, pochi hanno obiettato sull'uso di generatori di numeri pseudo-casuali, e ancora meno si sono posti il problema di verificare se tali generatori soddisfano effettivamente le distribuzioni di probabilità assunte per l'analisi di algoritmi (nella realtà molti esperimenti hanno dimostrato che tali assunzioni non sono giustificate).

Inoltre, si tenga presente che non esiste alcuna sorgente di "casualità" all'interno dei moderni calcolatori, salvo forse dispositivi meccanici i cui tempi di risposta sono di diversi ordini di magnitudine più lenti dei calcolatori con cui devono comunicare, e le cui distribuzioni di pro-

babilità sono sconosciute. Anche se simili dispositivi fossero disponibili, la mancata *riproducibilità* delle computazioni renderebbe la fase di debugging estremamente difficile. Una alternativa generalmente accettata consiste nell'utilizzare generatori di numeri pseudo-casuali ma, come già detto, poco si conosce in genere delle loro distribuzioni di probabilità (fa eccezione il generatore a congruenza polinomiale [38]).

Oltre al problema legato alla difficoltà di ottenere buoni generatori di numeri casuali, gli algoritmi randomizzati non potrebbero ugualmente essere impiegati nelle applicazioni in cui è richiesta una risposta entro un tempo prestabilito, come ad esempio in applicazioni *real-time*.

Questi motivi hanno spinto la ricerca sugli algoritmi in due direzioni: la prima consiste nell'applicare procedimenti di *derandomizzazione* [18] per derivare, a partire da un algoritmo randomizzato, un algoritmo deterministico avente circa la stessa complessità computazionale; la seconda consiste nella ricerca di nuovi algoritmi intrinsecamente deterministici. Vari procedimenti di derandomizzazione sono stati applicati con successo a problemi legati alla geometria computazionale [6, 35], ad algoritmi paralleli [31] e alla risoluzione approssimata di problemi NP-completi su grafi [34]. Purtroppo le tecniche di derandomizzazione note non sono di tipo generale, e si possono applicare solo a ristrette classi di problemi, per cui in generale l'unica strada percorribile per risolvere un problema in modo deterministico consiste nell'ideare un algoritmo risolutivo che sia *intrinsecamente* deterministico; questa è la strada seguita in questa tesi.

1.6 Scopo della tesi e risultati conseguiti

Fra i numerosi problemi riguardanti i grafi, i più investigati sono probabilmente quelli relativi al mantenimento di una minima foresta ricoprente di un grafo pesato [11, 13, 23] e delle componenti connesse [13, 21, 24, 25].

Nel 1995 Henzinger e King [21] fornirono i primi algoritmi randomizzati di tipo Las-Vegas per la risoluzione dei problemi della connettività, del bipartitismo, dell'equivalenza rispetto ai cicli e del mantenimento di una minima foresta ricoprente approssimata in tempo atteso polilogarit-

mico per operazione; l'algoritmo per la connettività, ad esempio, è in grado di mantenere una foresta ricoprente di un grafo G con n nodi, soggetto a inserimenti e rimozioni di archi, in tempo atteso $O(\log^3 n)$[1] per operazione. Tali algoritmi fanno uso di semplici strutture dati, e pertanto si prestano particolarmente bene ad essere utilizzati in pratica. Nel 1996, il tempo di esecuzione è stato migliorato di un fattore $\log n$ da Henzinger e Thorup [24] utilizzando un nuovo e più sofisticato schema di selezione casuale.

Relativamente al problema della connettività, è esistita una notevole differenza nelle prestazioni degli algoritmi randomizzati e deterministici (l'algoritmo deterministico più efficiente per il mantenimento di una foresta ricoprente, esistente nel 1995, supportava inserimenti e rimozioni di archi in tempo $O(\sqrt{n})$ per operazione [11]); tale differenza è stata recentemente colmata da Holm, de Lichtenberg e Thorup [25] i quali, sfruttando una nuova tecnica, hanno sviluppato un algoritmo in grado di mantenere le componenti connesse di un grafo non orientato con n vertici, supportando inserimenti e cancellazioni di archi in tempo ammortizzato $O(\log^2 n)$ per operazione, e di rispondere a interrogazioni sulla connettività ("i vertici u e v sono connessi?") in tempo $O(\log n / \log\log n)$ nel caso pessimo, eguagliando quindi le prestazioni del migliore algoritmo randomizzato conosciuto. La stessa tecnica è stata applicata anche alla risoluzione del problema del mantenimento di una foresta ricoprente di peso minimo, e delle componenti *2-archi* e *2-vertici connesse* di un grafo in tempo ammortizzato $O(\log^4 n)$ per operazione [26].

Per ciò che riguarda il problema della connettività, dunque, il divario tra le prestazioni delle soluzioni deterministiche e randomizzate è stato colmato. Rimanevano senza risposta le domande: esistono algoritmi deterministici in grado di mantenere anche altre proprietà su grafi dinamici in tempo polilogaritmico per operazione? Può essere colmata anche in questi casi la differenza tra le prestazioni degli algoritmi randomizzati e deterministici?

Questo lavoro da una risposta affermativa a entrambe le domande.

[1] Si pone $\log^n x = (\log x)^n$; ove non espressamente indicato, i logaritmi si assumono in base 2, e si pone $\log x = \log \max\{1, x\}$, in modo tale che il logaritmo di x sia sempre una quantità non negativa.

Parte dei risultati di Henzinger e King [21] e alcune delle recenti idee di Holm ed altri [25] vengono combinate con risultati originali per ottenere nuovi algoritmi deterministici in grado di mantenere altre proprietà su grafi dinamici in tempo ammortizzato polilogaritmico per operazione; gli algoritmi proposti si basano sull'idea comune di organizzare le informazioni sulla proprietà da mantenere attorno ad una foresta ricoprente del grafo. Tali algoritmi non sono di interesse puramente teorico, perché utilizzando semplici strutture dati si prestano ad essere implementati efficientemente in pratica. Questo non è sempre scontato, dato che gli algoritmi dinamici tendono ad essere penalizzati nella loro complessità computazionale da elevati fattori costanti "nascosti" nelle notazioni asintotiche, come messo in evidenza da Eppstein ed altri [11]. I risultati conseguiti rappresentano sensibili miglioramenti rispetto ai precedenti algoritmi deterministici relativi alla soluzione dei medesimi problemi. Inoltre, l'algoritmo presentato per il mantenimento di una $1+\epsilon$-minima foresta ricoprente rappresenta, in base alla nostra migliore conoscenza, il primo algoritmo deterministico per la soluzione del problema, se si trascurano gli algoritmi che banalmente calcolano una foresta ricoprente di peso minimo.

Gli algoritmi proposti sono *totalmente dinamici su archi*, cioè implementano strutture dati che mantengono un grafo non orientato G e una proprietà $\mathcal{P}$ sotto una sequenza arbitraria delle seguenti primitive:

- **Insert**(u, v): Aggiunge l'arco $\{u, v\}$ a G;
- **Delete**(u, v): Rimuove l'arco $\{u, v\}$ da G;
- **Query**$(\mathcal{P})$: Ritorna vero se e solo se la proprietà $\mathcal{P}$ vale per il grafo G, falso altrimenti.

È importante sottolineare che ogni operazione deve essere effettuata *online*, senza sapere nulla riguardo le operazioni che seguono.

In dettaglio, in questa tesi verranno presentati algoritmi deterministici per il mantenimento delle seguenti proprietà su grafi non orientati dinamici:

- Mantenere una foresta ricoprente di peso minimo di un grafo i cui archi possano avere pesi appartenenti all'insieme $\{1, 2, \ldots, k\}$, per

k fissato, supportando inserimenti e rimozioni di archi in tempo ammortizzato $O(k \log^2 n)$ per operazione (si veda la tabella 1.2). L'algoritmo è facilmente estendibile al caso più generale in cui in ogni istante i pesi degli archi del grafo appartengano ad un insieme di al più k valori arbitrari distinti, per k fissato inizialmente. L'algoritmo dinamico più efficiente in grado di mantenere una minima foresta ricoprente nel caso generale supporta inserimenti e cancellazioni di archi in tempo ammortizzato $O(\log^4 n)$ per operazione [25]; tuttavia, in molte applicazioni pratiche il peso degli archi varia in un piccolo insieme discreto di valori, per cui l'algoritmo proposto può risultare assai più conveniente.

Autore	*Anno*	*Update time*
Henzinger e King [21] *R*	1995	$O(k \log^3 n)$
Henzinger e Thorup [24] *R*	1996	$O(k \log^2 n)$
Henzinger e King [23] *D*	1997	$O(kn^{1/3} \log n)$
Algoritmo presentato		$O(k \log^2 n)$

Tabella 1.2: Risultati relativi al problema della minima foresta ricoprente con k pesi. Accanto al nome dell'autore, una R indica che si tratta di un algoritmo randomizzato, mentre una D indica che si tratta di un algoritmo deterministico.

- Mantenere una $1+\epsilon$-minima foresta ricoprente di un grafo pesato in tempo ammortizzato $O(\log^2 n \log(U(1+\epsilon))/\log(1+\epsilon))$ per operazione, ove i pesi degli archi siano compresi tra 1 e U (si veda la tabella 1.3). Una foresta ricoprente F di un grafo pesato G è detta $1+\epsilon$-minima se il suo peso si discosta al più per un fattore ϵ da quello di una minima foresta ricoprente dello stesso grafo. L'algoritmo proposto è indubbiamente vantaggioso nelle applicazioni in cui il tempo di risposta è un fattore critico e in cui non è essenziale calcolare una foresta ricoprente di peso minimo, ma è sufficiente una

soluzione approssimata. Al meglio della nostra conoscenza, l'algoritmo costituisce la prima soluzione deterministica per il calcolo di una minima foresta ricoprente approssimata, se si escludono gli algoritmi che calcolano la soluzione esatta.

Autore	*Anno*	*Update time*
Henzinger e King [21] R	1995	$O\left(\log^3 n \dfrac{\log(U(1+\epsilon))}{\log(1+\epsilon)}\right)$
Henzinger e Thorup [24] R	1996	$O\left(\log^2 n \dfrac{\log(U(1+\epsilon))}{\log(1+\epsilon)}\right)$
Algoritmo presentato		$O\left(\log^2 n \dfrac{\log(U(1+\epsilon))}{\log(1+\epsilon)}\right)$

Tabella 1.3: Risultati relativi al problema della $1+\epsilon$-minima foresta ricoprente. In [21] e [24] la complessità è stata riportata erroneamente con un fattore $\log U/\epsilon$ al posto del valore corretto $\log(U(1+\epsilon))/\log(1+\epsilon)$.

- Testare se un grafo è bipartito in tempo $O(1)$ nel caso pessimo, supportando inserimenti e rimozioni di archi in tempo ammortizzato $O(\log^2 n)$ per operazione (si veda la tabella 1.4). L'algoritmo può essere immediatamente impiegato per decidere in tempo costante se il grafo è 2-colorabile (dato che un grafo è bipartito se e solo se è 2-colorabile), e con una semplice estensione è in grado di decidere in tempo $O(\log n)$ se esiste una 2-colorazione del grafo che assegna a due vertici arbitrari u, v i colori c_u, c_v. Si tenga presente che per $k > 2$, il problema di decidere se un grafo $G = (V, E)$ è k-colorabile è NP-completo.

Autore		*Anno*	*Update time*	*Query time*
Eppstein e altri [11]	*D*	1992	$O(\sqrt{n})$	$O(1)$
Eppstein e altri [11]	*D*	1992	$O(\alpha(n))$ ins. $O(n)$ canc.	$O(1)$
Henzinger e King [21]	*R*	1995	$O(\log^3 n)$	$O(1)$
Henzinger e Thorup [24]	*R*	1996	$O(\log^2 n)$	$O(1)$
Henzinger e King [23]	*D*	1997	$O(n^{1/3} \log n)$	$O(1)$
Algoritmo presentato			$O(\log^2 n)$	$O(1)$

Tabella 1.4: Risultati relativi al problema del bipartitismo

- Testare se la rimozione di k archi dati $e_1, e_2, \ldots, e_k$ rende i vertici u e v sconnessi in tempo $O(k \log^2 n)$, per ogni coppia di vertici u e v, supportando inserimenti e rimozioni di archi in tempo ammortizzato $O(\log^2 n)$ per operazione (vedi tabella 1.5). Si noti che questo problema è strettamente legato al problema della k-archi connettività (*k-edge connectivity*) nel modo seguente: due vertici u e v sono k-archi connessi, per $k \geq 2$, se la rimozione di $k-1$ archi qualsiasi lascia u e v connessi. Sebbene esista un algoritmo dinamico in grado di stabilire se un grafo è k-archi connesso, per $k > 3$, in tempo $O(n \log n)$ per operazione [11] (risultati più efficienti per i casi $k = 2$ sono descritti in [22, 26], e per $k = 3$ sono descritti in [11]), la soluzione presentata è in grado di risolvere il problema del *testimone alla $k+1$-archi connettività* in tempo $O(k \log^2 n)$: dati due vertici u e v, la rimozione dei k archi $e_1, e_2, \ldots, e_k$ lascia u e v connessi? L'algoritmo presentato risulta più efficiente nella soluzione di questo problema rispetto agli algoritmi esistenti per la soluzione del problema più generale della k-archi connettività.

Autore	*Anno*	*Update time*	*Query time*
Henzinger e King [21] R	1995	$O(\log^3 n)$	$O(k \log^3 n)$
Henzinger e Thorup [24] R	1996	$O(\log^2 n)$	$O(k \log^2 n)$
Henzinger e King [23] D	1997	$O(n^{1/3} \log n)$	$O(kn^{1/3} \log n)$
Algoritmo presentato		$O(\log^2 n)$	$O(k \log^2 n)$

Tabella 1.5: Risultati relativi al problema del tesimone alla $k+1$-archi connettività

Come immediata applicazione, si ricava un algoritmo per testare l'equivalenza di due archi rispetto ai cicli in tempo $O(\log^2 n)$ per interrogazione e tempo ammortizzato $O(\log^2 n)$ per modifica (si veda la tabella 1.6). Determinare quando due archi sono equivalenti rispetto ai cicli è essenziale nello sviluppo di compilatori, in quanto la relazione di equivalenza di *control-dependence* di un programma è la relazione di equivalenza rispetto ai cicli della versione non orientata del grafo di flusso [28]. In particolare, algoritmi di ottimizzazione del codice come *static single-assignment form construction*, e algoritmi di analisi del flusso come la determinazione della *subexpression availability*, possono essere velocizzati se sono note le classi di equivalenza rispetto ai cicli del grafo di flusso [27]. Una ulteriore applicazione dell'algoritmo riguarda l'assegnazione globale delle istruzioni per macchine pipelined [19].

Autore	*Anno*	*Update time*	*Query time*
Henzinger [20] D	1994	$O(\sqrt{n}\log n)$	$O(\log^2 n)$
(grafi piani) D		$O(\log n)$	$O(\log n)$
(grafi planari) D		$O(\log n)$ ins.	$O(\log^2 n)$
		$O(\log^2 n)$ canc.	
Henzinger e King [21] R	1995	$O(\log^3 n)$	$O(\log^3 n)$
Henzinger e Thorup [24] R	1996	$O(\log^2 n)$	$O(\log^2 n)$
Henzinger e King [23] D	1997	$O(n^{1/3}\log n)$	$O(n^{1/3}\log n)$
Algoritmo presentato		$O(\log^2 n)$	$O(\log^2 n)$

Tabella 1.6: Risultati relativi al problema dell'equivalenza rispetto ai cicli

- Mantenere una decomposizione ricoprente massimale di ordine k in tempo ammortizzato $O(k\log^2 n)$ per operazione (si veda la tabella 1.7). Una decomposizione massimale di ordine k di un grafo non orientato G è una sequenza di foreste $F_1, F_2, \ldots, F_k$ tali che F_i è una foresta ricoprente di $G_i = G \setminus \cup_{j<i} F_j$. Tale decomposizione è interessante poiché il grafo $G^* = \cup_{i=1}^{k} F_i$ ha $O(kn)$ archi, ed ha le stesse componenti k-archi connesse di G [39], ossia è un *certificato* per la proprietà di k connettività rispetto agli archi. La possibilità di mantenere certificati per determinate proprietà efficientemente e in maniera dinamica è il fulcro su cui si basano diverse tecniche algoritmiche, tra cui quella della *sparsificazione* [11].

Autore	*Anno*	*Update time*
Henzinger e King [21] *R*	1995	$O(k \log^3 n)$
Henzinger e Thorup [24] *R*	1996	$O(k \log^2 n)$
Henzinger e King [23] *D*	1997	$O(kn^{1/3} \log n)$
Algoritmo presentato		$O(k \log^2 n)$

Tabella 1.7: Risultati precedenti relativi al problema della decomposizione massimale di ordine k

1.7 Struttura della tesi

La tesi è strutturata nel modo seguente: nel capitolo 2 vengono introdotti brevemente i concetti fondamentali e la terminologia riguardanti gli algoritmi e le strutture dati. Nel capitolo 3 viene descritto l'algoritmo deterministico per la connettività dinamica di Holm, de Lichtenberg e Thorup [25], i cui dettagli implementativi sono presentati nell'appendice A, e l'algoritmo randomizzato di Henzinger e King [21]. Nel successivo capitolo 4 vengono presentati gli algoritmi che costituiscono il risultato originale della tesi; ciascuno di essi viene inizialmente descritto con linguaggio naturale, quindi presentato sotto forma di pseudocodice e successivamente analizzato nella complessità computazionale. La tesi si conclude col capitolo 5 in cui si riassumono i risultati conseguiti e si suggeriscono eventuali sviluppi futuri.

Capitolo 2

Preliminari

In questo capitolo vengono introdotti i concetti fondamentali riguardanti gli algoritmi, le strutture dati, l'analisi della complessità computazionale, le notazioni asintotiche ed altro. Il materiale presentato serve come base per i capitoli successivi.

2.1 Algoritmi

Informalmente, si definisce *algoritmo* qualsiasi procedura computazionale ben definita che accetta dati o insiemi di dati come *input* e produce altri dati, o insiemi di dati, come *output*. Pertanto un algoritmo è una procedura computazionale che trasforma ogni input nell'output corrispondente [9].

È possibile vedere un algoritmo anche come strumento per risolvere un *problema computazionale* ben definito. La definizione del problema descrive senza ambiguità la relazione di input/output voluta, e l'algoritmo implementa una procedura computazionale per ottenere tale relazione di input/output.

Un algoritmo è detto *corretto* se, qualunque sia l'istanza dell'input, termina fornendo l'output corretto. Diremo in questo caso che l'algoritmo risolve il problema computazionale dato. Per i nostri scopi, qualunque algoritmo che non soddisfi questi requisiti è considerato scorretto.

Esistono molti modi per descrivere un algoritmo, ad esempio sotto

forma di programma in un linguaggio di programmazione opportuno, o mediante frasi in linguaggio "naturale", o ancora sotto forma di pseudocodice; in ogni caso deve essere fornita una descrizione precisa e non ambigua della procedura computazionale da seguire. Nel seguito tutti gli algoritmi verranno prima presentati con linguaggio naturale, quindi descritti mediante pseudocodice.

2.2 Analisi degli algoritmi

L'analisi degli algoritmi ha per oggetto la determinazione della quantità di risorse necessarie per terminare la computazione sui dati passati in input. Talvolta risorse come memoria centrale, memoria di massa, o larghezza di banda sono le limitazioni maggiormente vincolanti, ma in generale si è interessati a stimare il tempo necessario all'algoritmo per terminare, detto *tempo di esecuzione*. Conseguentemente, la complessità computazionale di un algoritmo fornisce una misura del tempo necessario per terminare la computazione, assumendo che ciascun algoritmo venga implementato su una macchina astratta a processore singolo, basata sul modello computazionale della macchina ad accesso casuale (RAM). In tale modello, le istruzioni sono eseguite sequenzialmente una dopo l'altra, e non sono permesse operazioni concorrenti (per una descrizione più formale dell'architettura RAM si veda [1]).

Il tempo di esecuzione di un algoritmo dipende dalla *dimensione dell'input* su cui esso opera. La nozione di "dimensione dell'input" è legata a sua volta allo specifico problema preso in esame. In alcuni casi, come ad esempio nell'ordinamento di un vettore di numeri interi, la misura più naturale è il numero di elementi del vettore. Se invece l'algoritmo opera su alberi, la complessità può essere valutata in funzione del numero di vertici o della profondità dell'albero. In altri casi, la dimensione dell'input deve essere specificata mediante più di un parametro: ad esempio, se l'algoritmo opera su grafi, la dimensione dell'input può essere funzione del numero di vertici e di archi che lo compongono.

Il tempo di esecuzione di un algoritmo su un particolare input è il numero di *operazioni primitive* eseguite. È opportuno definire la nozione di

operazione primitiva in modo tale che sia il più indipendente possibile da ogni implementazione dell'algoritmo. Una definizione generalmente adottata è la seguente [1, 9]: si suppone che ogni linea di pseudocodice richieda tempo costante per essere eseguita, sebbene ciascuna possa impiegare un tempo diverso da ogni altra[1]. Si definisce quindi la complessità computazionale di un algoritmo, con riferimento ad una istanza dell'input di dimensione n, come il numero di operazioni primitive eseguite per fornire il risultato. Il tempo di esecuzione viene a volte indicato con il termine *complessità temporale*, o semplicemente *complessità* dell'algoritmo.

Accanto al tempo di esecuzione, è spesso utile considerare anche le richieste in termini di occupazione di memoria necessaria per mantenere le strutture dati di cui l'algoritmo si serve durante la sua esecuzione. La *complessità spaziale* è una stima dello spazio di memoria richiesto da un particolare algoritmo in funzione della dimensione dell'input.

La definizione di complessità può essere estesa in modo naturale anche ad operatori che agiscono su strutture dati. Una *struttura dati* è costituita da un insieme di dati e di operatori per agire su di essi [32]. Definiamo *costo computazionale*, o semplicemente *costo* di un operatore, la complessità temporale dello stesso. Vedremo in seguito una tecnica (l'analisi ammortizzata) in grado di semplificare l'analisi della complessità di operatori su strutture dati.

2.2.1 Analisi nel caso pessimo, ottimo e medio

Si è visto nel paragrafo precedente che l'analisi della complessità di un algoritmo consiste nel trovare una funzione T che ad ogni input di dimensione n associ il numero di operazioni elementari $T(n)$ eseguite per terminare. Molte volte la complessità di un algoritmo non può essere caratterizzata da una sola funzione di complessità. Infatti a parità di dimensione dell'input, il tempo di esecuzione può dipendere dalla spe-

[1]Questa visione semplicistica non è immune da problemi. Infatti molte righe di pseudocodice espresse in linguaggio naturale possono essere varianti di procedura che richiedono più di un tempo costante per essere portate a termine. È richiesta quindi la massima attenzione in fase di analisi

cifica configurazione dei dati in input; non è escluso in generale che a configurazioni diverse corrispondano tempi di esecuzione diversi.

Si prendono in esame tre diversi tipi di complessità: la complessità nel *caso pessimo*, nel *caso ottimo* e quella nel *caso medio*. La prima si ottiene considerando quella particolare configurazione dell'input che, a parità di dimensione dei dati, dà luogo al massimo tempo di esecuzione; la complessità nel caso ottimo si ottiene considerando la configurazione che dà luogo al minimo tempo di esecuzione, e la complessità nel caso medio si riferisce al tempo di esecuzione mediato su tutte le possibili configurazioni dei dati, sempre per una determinata dimensione dell'input.

Sia I_n l'insieme di tutti i possibili input di dimensione n su cui possa operare un certo algoritmo. Per ogni $I \in I_n$ sia $T(I)$ il tempo impiegato dall'algoritmo per terminare sull'input I; indichiamo con $P(I)$ la probabilità che il particolare input $I \in I_n$ si possa presentare. Allora è possibile definire la complessità nel caso pessimo $T_{\text{worst}}(n)$, nel caso ottimo $T_{\text{best}}(n)$ e nel caso medio $T_{\text{average}}(n)$ come segue:

$$\begin{aligned} T_{\text{worst}}(n) &= \max_{I \in I_n} T(I) \\ T_{\text{best}}(n) &= \min_{I \in I_n} T(I) \\ T_{\text{average}}(n) &= \sum_{I \in I_n} P(I)T(I) \end{aligned}$$

Tra queste misure di complessità, quella nel caso pessimo è generalmente più utilizzata, in quanto stabilisce un limite superiore al tempo di esecuzione di un algoritmo indipendentemente dalla configurazione dei dati in input. Tuttavia è possibile che la configurazione dell'input che dà origine al massimo tempo di esecuzione si possa presentare solo raramente. Vedremo in seguito ulteriori tecniche di analisi per tenere conto di situazioni del genere.

2.2.2 Algoritmi randomizzati

Un algoritmo randomizzato è un algoritmo che effettua scelte casuali durante la sua esecuzione. Di conseguenza, un algoritmo randomizzato applicato più volte alla medesima istanza dell'input può agire di volta in volta in maniera diversa. Esistono due tipi di algoritmi randomizzati:

Las-Vegas che forniscono sempre la soluzione corretta. L'unica differenza tra una esecuzione dell'algoritmo e l'altra è il tempo di esecuzione, di cui si è interessati a studiare il *valore atteso*, che definisce quindi la *complessità attesa* dell'algoritmo.

Monte Carlo che possono occasionalmente fornire una risposta errata, sebbene si possa fissare un limite alla probabilità che questo accada. Per problemi di decisione (problemi nei quali le soluzioni possibili sono "sì" oppure "no"), si distinguono due tipi di algoritmi Monte Carlo: quelli con *one-side error*, e quelli con *two-side error*. Un algoritmo Monte Carlo è *two-side error* se esiste una probabilità non nulla che possa fornire un risultato errato sia quando fornisce la risposta "sì", sia quando fornisce la risposta "no". È detto *one-side error* se la probabilità di risposta errata è nulla in almeno uno dei casi.

In realtà la distinzione è abbastanza labile, in quanto esiste spesso la possibilità di ottenere un algoritmo Las-Vegas a partire da uno Monte Carlo. Vale infatti il seguente

Teorema 2.1 *Dato un algoritmo randomizzato* Π *di tipo Monte Carlo, avente complessità al più* $T(n)$ *su qualsiasi input di dimensione* n*, che produce una soluzione corretta con probabilità* $\gamma(n)$ *e tale che sia possibile verificare la correttezza della soluzione in tempo* $t(n)$*, esiste un algoritmo Las-Vegas che fornisce sempre la risposta corretta in tempo atteso* $(T(n)+t(n))/\gamma(n)$.

Dimostrazione. L'algoritmo Las-Vegas è definito nel modo seguente:

1. Calcola una soluzione usando Π in tempo $T(n)$;
2. Verifica la soluzione in tempo $t(n)$;
3. Se la soluzione è corretta, termina; altrimenti ripete dal punto 1.

Sia X la variabile aleatoria discreta definita come

$$X = k \equiv \text{l'algoritmo Las-Vegas termina dopo } k \text{ iterazioni}$$

La probabilità che l'algoritmo termini esattamente alla k-esima iterazione, per $k \geq 1$, vale

$$\Pr(X = k) = (1 - \gamma(n))^{k-1}\gamma(n)$$

ossia le prime $k-1$ iterazioni dell'algoritmo Π devono fornire una risposta errata, e solo la k-esima iterazione deve dare la risposta corretta. Il valore atteso di X è

$$\begin{aligned} E[X] &= \sum_{j=1}^{+\infty} j\Pr(X = j) \\ &= \sum_{j=1}^{+\infty} j(1 - \gamma(n))^{j-1}\gamma(n) \\ &= \gamma(n) \sum_{j=1}^{+\infty} j(1 - \gamma(n))^{j-1} \\ &= \gamma(n)\frac{1}{\gamma(n)^2} \\ &= \frac{1}{\gamma(n)} \end{aligned}$$

Poiché ciascuna iterazione ha complessità $T(n) + t(n)$, la tesi segue. □

Recentemente gli algoritmi randomizzati, originariamente impiegati per risolvere problemi di teoria dei numeri, sono stati applicati con successo a quasi ogni tipo di problema (si veda [37] per una raccolta di esempi e per una vasta bibliografia sull'argomento), grazie alla loro generale efficienza e alla semplicità di implementazione. Lo svantaggio maggiore degli algoritmi randomizzati, comunque, consiste proprio nella impossibilità di prevedere con certezza il loro comportamento. Ad esempio, l'algoritmo Las-Vegas visto nella dimostrazione del lemma 2.1 ha probabilità non nulla di terminare alla n-esima iterazione, per ogni $n > 0$, per cui risulta impossibile fissare a priori un limite superiore alla complessità. In applicazioni particolarmente critiche, questo è ovviamente inaccettabile.

2.2.3 Le notazioni asintotiche

Definire la complessità temporale di un algoritmo equivale a determinare una funzione T che associa alla dimensione n dell'input il numero di operazioni primitive $T(n)$ eseguite. Normalmente si assume $T : \mathbb{N} \to \mathbb{R}$, sebbene in certi casi risulti conveniente estendere la definizione per includere funzioni del tipo $T : \mathbb{R} \to \mathbb{R}$.

Piuttosto che determinare esattamente la funzione di complessità T, è importante conoscerne l'*ordine di crescita*, trascurando cioè le costanti moltiplicative e i termini di ordine inferiore, essendo il loro contributo sempre più trascurabile mano a mano che la dimensione dell'input aumenta. Si osservi, infatti, che nell'analisi degli algoritmi la dimensione dell'input viene considerata senza limiti, per cui ciò che interessa è la complessità *asintotica*, ossia la complessità $T(n)$ per n tendente all'infinito. Avendo due funzioni di complessità $f(n) = n^2 + 40n + 80$ e $g(n) = 8000n + 1000$, quello che è importante osservare è che f è *quadratica*, mentre g è *lineare*; questo implica che, sebbene il fattore del termine lineare di g sia elevato rispetto al fattore del termine quadratico di f, per valori di n sufficientemente elevati, sicuramente $g(n) \leq f(n)$. In altri termini, ciò significa che un algoritmo avente complessità $g(n)$ è sicuramente da preferire ad uno di complessità $f(n)$, tranne al più per input di piccole dimensioni.

Notazione Θ

Data una funzione $g(n)$, denotiamo con $\Theta(g(n))$ l'insieme di funzioni:

$$\Theta(g(n)) = \{f(n) \mid \text{Esistono costanti positive } c_1, c_2, n_0 \text{ tali che } \forall n \geq n_0,\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

Sebbene $\Theta(g(n))$ sia un insieme, d'ora in avanti scriveremo $f(n) = \Theta(g(n))$ per indicare che $f(n)$ è un elemento di $\Theta(g(n))$.

Se $f(n) = \Theta(g(n))$, diremo che $g(n)$ è un *tight bound* asintotico per $f(n)$ (vedi figura 2.1(a)).

Notazione O

Data una funzione $g(n)$, denotiamo con $O(g(n))$ l'insieme di funzioni:

$$O(g(n)) = \{f(n) \mid \text{Esistono costanti positive } c, n_0 \text{ tali che } \forall n \geq n_0,\ 0 \leq f(n) \leq cg(n)\}$$

Scriveremo $f(n) = O(g(n))$ per indicare che $f(n)$ è un elemento di $O(g(n))$. Si noti che $f(n) = \Theta(g(n))$ implica $f(n) = O(g(n))$; in notazione insiemistica, si ha $O(g(n)) \subseteq \Theta(g(n))$.

Se $f(n) = O(g(n))$, diremo che $g(n)$ è un *upper bound* asintotico per $f(n)$ (vedi figura 2.1(b)).

Notazione Ω

Data una funzione $g(n)$, denotiamo con $\Omega(g(n))$ l'insieme di funzioni:

$$\Omega(g(n)) = \{f(n) \mid \text{Esistono costanti positive } c, n_0 \text{ tali che } \forall n \geq n_0,\ 0 \leq cg(n) \leq f(n)\}$$

Se $f(n) = \Omega(g(n))$, diremo che $g(n)$ è un *lower bound* asintotico per $f(n)$ (vedi figura 2.1(c)).

Dalle definizioni delle notazioni asintotiche viste fino ad ora, deriva immediatamente il seguente lemma:

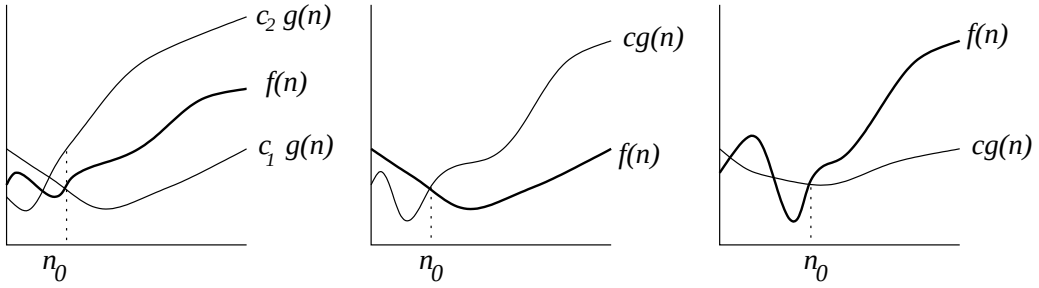
Lemma 2.1 *Per ogni coppia di funzioni $f(n)$ e $g(n)$, $f(n) = \Theta(g(n))$ se e solo se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$*

Confronto tra funzioni

Supponiamo che $f(n)$ e $g(n)$ siano asintoticamente positive; allora valgono le relazioni seguenti:

Transitività:

$$\begin{aligned}
f(n) = \Theta(g(n)) \text{ e } g(n) = \Theta(h(n)) &\quad \text{implica} \quad f(n) = \Theta(h(n)) \\
f(n) = O(g(n)) \text{ e } g(n) = O(h(n)) &\quad \text{implica} \quad f(n) = O(h(n)) \\
f(n) = \Omega(g(n)) \text{ e } g(n) = \Omega(h(n)) &\quad \text{implica} \quad f(n) = \Omega(h(n))
\end{aligned}$$



(a) $f(n) = \Theta(g(n))$ (b) $f(n) = O(g(n))$ (c) $f(n) = \Omega(g(n))$

Figura 2.1: La figura illustra graficamente la relazione asintotica tra due funzioni f e g, nel caso in cui: (a) $f(n) = \Theta(g(n))$; (b) $f(n) = O(g(n))$; (c) $f(n) = \Omega(g(n))$.

Riflessività:

$$\begin{aligned} f(n) &= \Theta(f(n)) \\ f(n) &= O(f(n)) \\ f(n) &= \Omega(f(n)) \end{aligned}$$

Simmetria:

$$f(n) = \Theta(g(n)) \quad \text{se e solo se} \quad g(n) = \Theta(f(n))$$

Simmetria trasposta:

$$f(n) = O(g(n)) \quad \text{se e solo se} \quad g(n) = \Omega(f(n))$$

2.3 Analisi ammortizzata

Si consideri una struttura dati $\mathcal{D}$ sulla quale si possa agire mediante opportuni operatori. Nell'*analisi ammortizzata*, il tempo richiesto per eseguire una sequenza di operazioni sulla struttura dati $\mathcal{D}$ è mediato

sul numero di operazioni effettuate; pertanto l'analisi ammortizzata può essere utile per dimostrare che il costo di una operazione è piccolo, se mediato su una sequenza, sebbene una singola operazione possa essere costosa. Si noti che l'analisi ammortizzata differisce dall'analisi nel caso medio in quanto non entrano in gioco le probabilità; viene invece fornita una stima della complessità media di ciascuna operazione nel caso pessimo. L'analisi ammortizzata verrà impiegata nei capitoli 3 e 4 per analizzare il tempo di esecuzione degli algoritmi dinamici illustrati.

Per determinare il costo ammortizzato di operatori su strutture dati è possibile utilizzare uno dei seguenti metodi: il *metodo di aggregazione*, il *metodo dell'addebito* e quello del *potenziale*.

2.3.1 Il metodo di aggregazione

Nel metodo di aggregazione dell'analisi ammortizzata è necessario dimostrare che per ogni n, qualunque sequenza di n operazioni sulla struttura dati $\mathcal{D}$ può essere completata in tempo totale $T(n)$ nel caso pessimo; il costo ammortizzato di una operazione è quindi $T(n)/n$. Si noti che questo costo ammortizzato viene riferito ad *ogni* operazione, sebbene possano esserci diversi tipi di operazioni nella sequenza, ciascuno con un costo computazionale diverso. I due metodi di analisi successivi, il metodo dell'addebito e quello del potenziale, possono assegnare costi ammortizzati diversi a diversi tipi di operazioni.

2.3.2 Il metodo dell'addebito

Nel *metodo dell'addebito* dell'analisi ammortizzata è possibile assegnare costi diversi ad operazioni diverse, ove alcune operazioni possono essere addebitate per un costo superiore al loro costo computazionale effettivo. L'addebito assegnato a ciascuna operazione è il suo costo ammortizzato; quando il costo ammortizzato supera il costo effettivo, la differenza è assegnata all'oggetto specifico della struttura dati come *credito*. Il credito può essere utilizzato successivamente per pagare operazioni il cui costo ammortizzato è inferiore a quello effettivo. Si noti che il credito associato ad un oggetto della struttura dati deve essere in ogni istante non negativo,

dato che rappresenta l'ammontare per cui il costo totale ammortizzato incorso fino a quell'istante supera il costo effettivo.

2.3.3 Il metodo del potenziale

Nel *metodo del potenziale* dell'analisi ammortizzata, all'intera struttura dati è assegnato un credito che costituisce una sorta di *energia potenziale*, o semplicemente *potenziale*. Tale energia potenziale può essere rilasciata per pagare il costo computazionale di future operazioni.

Il metodo del potenziale si applica come segue: si inizia con una struttura dati $\mathcal{D}_0$ sulla quale vengono eseguite n operazioni. Per ogni $i = 1, 2, \ldots, n$, sia c_i il costo effettivo della i-esima operazione, e sia $\mathcal{D}_i$ la struttura dati ottenuta applicando la i-esima operazione alla struttura $\mathcal{D}_{i-1}$. La funzione potenziale Φ associa a ciascuna struttura $\mathcal{D}_i$ un numero reale non negativo $\Phi(\mathcal{D}_i)$ detto *potenziale*. Il costo ammortizzato $\hat{c}_i$ assegnato alla i-esima operazione, rispetto alla funzione potenziale Φ, è definito da

$$\hat{c}_i = c_i + \Phi(\mathcal{D}_i) - \Phi(\mathcal{D}_{i-1}) \tag{2.1}$$

dunque il costo ammortizzato di ciascuna operazione è dato dal suo costo effettivo più l'incremento di potenziale causato dall'operazione. Dall' equazione (2.1), il costo complessivo ammortizzato delle n operazioni è

$$\begin{aligned} \sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(\mathcal{D}_i) - \Phi(\mathcal{D}_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(\mathcal{D}_n) - \Phi(\mathcal{D}_0) \end{aligned} \tag{2.2}$$

Se la funzione potenziale è tale che $\Phi(\mathcal{D}_n) \geq \Phi(\mathcal{D}_0)$, come generalmente si richiede, il costo totale ammortizzato fornisce una stima per eccesso del costo totale effettivo della sequenza di operazioni.

Il costo ammortizzato definito dalle equazioni (2.1) e (2.2) dipende dalla scelta della funzione potenziale Φ; funzioni diverse possono dar luogo a costi diversi, sebbene tali costi rappresentino sempre un limite superiore del costo effettivo.

2.4 Altri metodi di analisi

Sono stati proposti altri metodi per l'analisi della complessità di algoritmi dinamici su grafi, che si prestano in modo particolare ad essere utilizzati nel caso di grafi orientati. Tra questi, quello che ha riscosso particolare successo è il metodo proposto da Ramalingam [41]: sia $G = (V, E)$ un grafo con n vertici ed m archi, sia $\mathcal{P}$ una proprietà da mantenere su G e δ l'insieme delle modifiche (inserimenti e rimozioni di archi o cambiamento di pesi) da apportare a G. Il parametro che definisce il costo computazionale delle modifiche è dato dalla *dimensione estesa* degli aggiornamenti all'output, denotato come $\|\delta\|$, che rappresenta il numero di vertici che modificano il loro valore di output rispetto alla proprietà $\mathcal{P}$, dopo il cambiamento dell'input δ, più il numero di archi avente almeno un estremo affetto dal cambiamento.

Questo metodo di analisi viene applicato prevalentemente ad algoritmi che operano su *grafi orientati*, cioè su grafi i cui archi sono costituiti da coppie ordinate di vertici. In questi casi, infatti, il numero di aggiornamenti da apportare alle strutture dati a seguito dell'inserimento o cancellazione di un singolo arco può non essere limitato a priori da una costante, rendendo poco significative le altre tecniche di analisi.

2.5 Grafi

Un *grafo non orientato* è una coppia $G = (V, E)$, ove V è un insieme finito di *vertici*, ed E è composto da coppie non ordinate di vertici $\{u, v\}$, con $u, v \in V$ e $u \neq v$ (si veda la figura 2.2 per una rappresentazione visiva). Per comodità, d'ora in avanti useremo il termine grafo per indicare un grafo non orientato.

L'insieme dei vertici di un grafo G verrà denotato con $V(G)$, mentre $E(G)$ indicherà l'insieme degli archi. Dato un grafo $G = (V, E)$, l'arco $\{u, v\}$ ha i vertici u e v come *estremi*, è *incidente* ai vertici u e v, e il vertice u è *adiacente* al vertice v (e viceversa). In un grafo non orientato, la relazione di adiacenza è necessariamente simmetrica. Dato un vertice $v \in V$, definiamo *grado di* v il numero di archi incidenti ad esso.

Un *cammino* (o *percorso*) di lunghezza k dal vertice u al vertice v è una

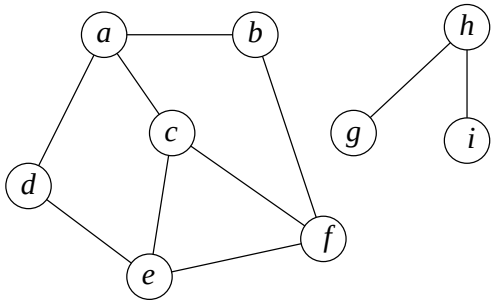


Figura 2.2: Esempio di grafo non orientato $G = (V, E)$; l'insieme dei vertici è $V = \{a, b, c, d, e, f, g, h, i\}$, e l'insieme degli archi è $E = \{\{a, b\}, \{a, c\}, \{a, d\}, \{b, f\}, \{c, e\}, \{c, f\}, \{d, e\}, \{e, f\}, \{g, h\}, \{h, i\}\}$.

sequenza di vertici $\langle v_0, v_1, \ldots, v_k \rangle$, tali che $v_0 = u$, $v_k = v$ e $\{v_{i-1}, v_i\} \in E$ per $i = 1, 2, \ldots, k$. La lunghezza di un cammino è il numero di archi presenti. Diremo che il cammino contiene i vertici $v_0, v_1, \ldots, v_k$ e gli archi $\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{k-1}, v_k\}$. Un cammino è *semplice* se tutti i vertici in esso contenuti sono distinti. Diremo che il vertice v è raggiungibile dal vertice u se esiste un cammino P da u a v, e scriveremo in tal caso $u \overset{P}{\rightsquigarrow} v$ (chiaramente $u \rightsquigarrow v$ se e solo se $v \rightsquigarrow u$).

Un cammino $\langle v_0, v_1, \ldots, v_k \rangle$ è un *ciclo* se $v_0 = v_k$; un ciclo è semplice se in più $v_1, v_2, \ldots v_k$ sono distinti. Un grafo senza cicli è detto *aciclico*.

Talvolta è conveniente identificare un cammino, o un ciclo, come un insieme di archi, piuttosto che come una sequenza di vertici; quindi al cammino $\langle v_0, v_1, \ldots v_{k-1}, v_k \rangle$ corrisponde l'insieme di k archi $\{\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{k-1}, v_k\}\}$. La possibilità di identificare un cammino con un insieme di archi permette di utilizzare i familiari operatori insiemistici per modificare e combinare cammini.

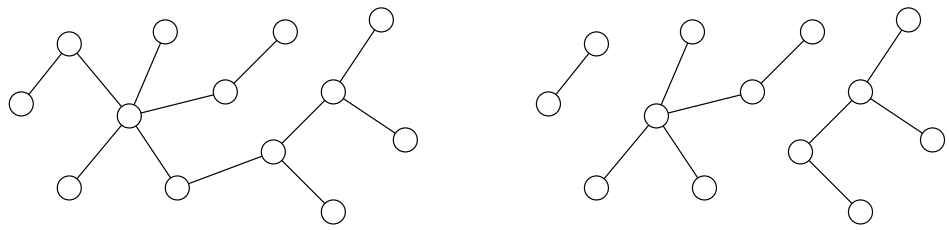
Diremo che un grafo G' è un *sottografo* di G se $V(G') \subseteq V(G)$ ed $E(G') \subseteq E(G)$; in tal caso, estendendo la notazione insiemistica, scriveremo $G' \subseteq G$. Dato l'insieme $V' \subseteq V$, il *sottografo indotto* da V' è il grafo $G' = (V', E')$, dove $E' = \{\{u, v\} \in E \mid u, v \in V'\}$. Dato un sottografo $G' \subseteq G$, un arco $e \in E(G) \setminus E(G')$ è *incidente* a G' se almeno un estremo di e appartiene a $V(G')$.

Un grafo non orientato è *connesso* se ogni coppia di vertici è connes-

sa da un cammino. Le *componenti connesse* di un grafo sono le classi di equivalenza dei vertici indotte dalla relazione simmetrica $u\mathcal{R}v \equiv$ "u è raggiungibile da v". Il grafo in figura 2.2 ha due componenti connesse.

Dato un grafo non orientato $G = (V, E)$ e un intero $k \geq 2$, due vertici u, v sono *k-archi connessi* (*k-edge connected*) se e solo se sono connessi, e la rimozione di $k - 1$ archi qualsiasi lascia u e v connessi. Si tratta di una relazione di equivalenza che partiziona l'insieme V in classi di equivalenza, dette *componenti k-archi connesse*. G è k-archi connesso se e solo se è connesso, e la rimozione di $k - 1$ archi qualsiasi lascia G connesso.

Un grafo non orientato aciclico è detto *foresta*, mentre un grafo non orientato connesso aciclico è detto *albero* (vedi figura 2.3).



(a) (b)

Figura 2.3: Un esempio di albero (a) e di foresta (b)

Un *albero con radice* è un albero libero in cui uno dei vertici è etichettato come *radice*; i vertici di un albero con radice verranno indicati come *nodi*. In figura 2.4 è riportato un albero con radice.

Si consideri un nodo x di un albero T avente r come radice. Ogni nodo y che appartenga all'unico cammino che collega r con x è chiamato *antenato* di x. Se y è un antenato di x, allora x è un *discendente* di y (ogni nodo è sia antenato che discendente di se stesso). Il *sottoalbero con radice* x è l'albero indotto da tutti i discendenti di x (compreso), considerando x come radice. Ad esempio, il sottoalbero di radice 5 dell'albero in figura 2.4 è composto dai nodi 5, 9, 10, 13.

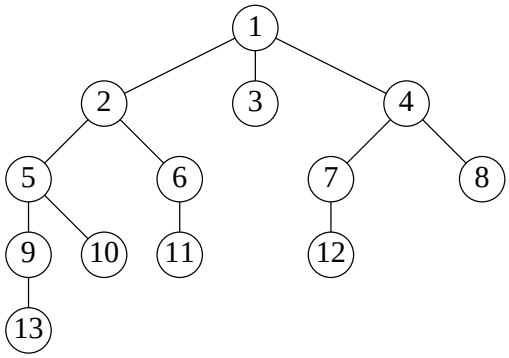


Figura 2.4: Esempio di albero con radice; la radice è il nodo 1

Ogni nodo y il cui unico discendente è y stesso è detto *foglia*; i rimanenti nodi sono detti *nodi interni*. La *profondità* di un nodo y appartenente ad un albero T con radice x è la lunghezza del cammino che collega x con y. La profondità dell'albero T è la massima profondità delle sue foglie.

Dato un grafo $G = (V, E)$, un *taglio* $C = (S, V \setminus S)$, ove $S \subseteq V, S \neq \emptyset$, è una partizione dei vertici V. Diciamo che un arco $e \in E$ attraversa il taglio C se uno degli estremi di e appartiene ad S e l'altro appartiene a $V \setminus S$ (vedi figura 2.5). Sarà utile talvolta identificare un taglio come l'insieme degli archi che lo attraversano, ma è importante ricordare che tecnicamente un taglio è una partizione dei vertici del grafo.

I normali operatori insiemistici possono essere estesi in modo naturale per operare su grafi. Se F è un sottografo di G, si pone

$$G \setminus F \stackrel{\text{def}}{=} (V(G), E(G) \setminus E(F))$$

Se F e G sono due grafi qualsiasi, si pone

$$G \cup F \stackrel{\text{def}}{=} (V(F) \cup V(G), E(F) \cup E(G))$$

Infine, se $E' \subseteq E$, si pone

$$G \setminus E' \stackrel{\text{def}}{=} (V(G), E(G) \setminus E')$$

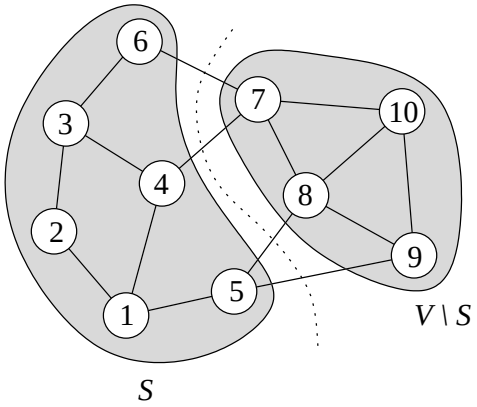


Figura 2.5: Rappresentazione di un grafo G con un taglio $C = (\{1,2,3,4,5,6\},\{7,8,9,10\})$. Gli archi che intersecano la linea tratteggiata sono quelli che attraversano il taglio.

e se $E' \subseteq V(G) \times V(G)$ è un insieme di archi qualsiasi

$$G \cup E' \stackrel{\text{def}}{=} (V(G), E(G) \cup E')$$

2.5.1 Foreste Ricoprenti

Dato un grafo non orientato $G = (V, E)$, una *foresta ricoprente* (*spanning forest*) $F = (V, E')$ è un sottografo aciclico di G avente esattamente le stesse componenti connesse di G (vedi figura 2.6). Se G è connesso, allora F sarà composta da un solo albero, detto *albero ricoprente* (*spanning tree*). Si noti che la foresta o l'albero ricoprenti non sono necessariamente unici.

Dato un grafo connesso $G = (V, E)$, è possibile determinare un albero ricoprente mediante gli algoritmi di ricerca in profondità o in ampiezza in tempo $O(|V| + |E|)$ (vedi ad esempio [9] per una descrizione dettagliata di tali algoritmi).

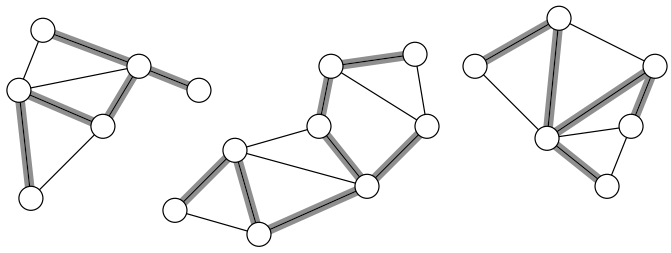


Figura 2.6: La figura illustra un grafo non connesso; gli archi evidenziati appartengono ad una delle foreste ricoprenti.

2.5.2 Minima Foresta Ricoprente

Si consideri un grafo $G = (V, E)$, e una funzione w che assegna a ciascun arco $e \in E$ un numero reale non negativo detto *peso*; G è detto *grafo pesato*. Dato un arco $e = \{u, v\}$, il peso di e verrà denotato indifferentemente come $w(e)$ oppure, evitando parentesi eccessive, $w\{u, v\}$.

Data una foresta ricoprente F di G, definiamo *peso di* F la quantità

$$w(F) = \sum_{e \in E(F)} w(e)$$

Siamo interessati a individuare, tra le foreste ricoprenti di G, quella avente peso minimo, che sarà indicata come *minima foresta ricoprente* (*Minimum Spanning Forest*). In generale, possono esserci foreste ricoprenti diverse aventi lo stesso peso, quindi la foresta di peso minimo può non essere unica.

È possibile determinare una minima foresta ricoprente di un grafo pesato utilizzando le regole seguenti [44]:

Regola del taglio Si aggiunge un arco alla volta alla foresta, finché questa ricopre interamente il grafo. Ad ogni passo, si individua un taglio del grafo tale che ogni arco che lo attraversa non compaia nella foresta determinata fino a quel momento; si rimuove l'arco di peso minimo che attraversa il taglio, e lo si inserisce nella foresta.

Regola del ciclo Si rimuove un arco alla volta dal grafo, finché rimane una foresta ricoprente. Ad ogni passo si individua un ciclo nel grafo residuo, e si elimina l'arco di peso massimo.

Queste proprietà vengono anche utilizzate per mantenere dinamicamente una minima foresta ricoprente di un grafo soggetto a cancellazione o inserimento di archi. Dato un grafo di partenza $G = (V, E)$ con funzione peso w, e una sua minima foresta ricoprente F, si supponga di voler determinare una nuova minima foresta ricoprente F' del grafo $G' = (V, E \cup \{u, v\})$.

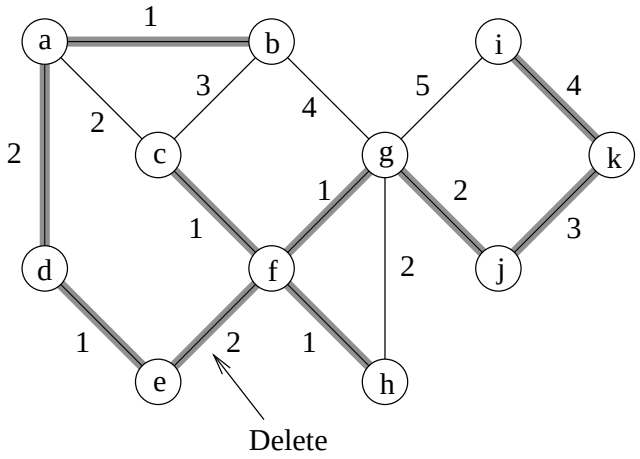
- Se $\{u, v\}$ connette due componenti connesse distinte di G, allora certamente $F \cup \{u, v\}$ è una minima foresta ricoprente di G'.

- Se i vertici u e v appartengono alla stessa componente connessa di G, allora $F \cup \{u, v\}$ è un grafo che contiene esattamente un ciclo, detto *ciclo indotto da* $\{u, v\}$ *su* F. Tale ciclo è facilmente individuabile: poiché u e v appartengono alla stessa componente connessa di G, esiste un cammino $u \overset{P}{\rightsquigarrow} v$ che collega u e v in F. Il ciclo indotto è dato da $C = P \cup \{u, v\}$. È possibile applicare la regola del ciclo su $F \cup \{u, v\}$: detto e l'arco più pesante di C, la minima foresta ricoprente di G' è data da $F \cup \{u, v\} \setminus \{e\}$.

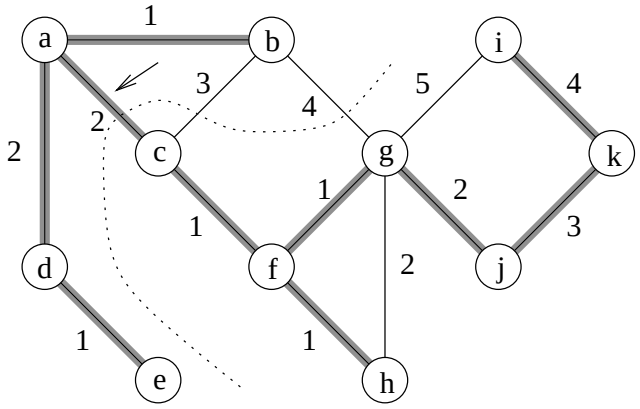
Si supponga invece di cancellare l'arco $\{x, y\}$ dal grafo $G = (V, E)$. Ancora una volta, si vuole determinare una nuova Foresta F' di $G' = (V, E \setminus \{x, y\})$.

- Se $\{x, y\} \in E(G) \setminus E(F)$, allora la minima foresta ricoprente resta immutata, e $F' = F$.

- Se $\{x, y\} \in E(F)$, sia T l'albero di F che contiene sia x che y. La rimozione dell'arco sconnette T in due sottoalberi T_x e T_y, contenenti rispettivamente i vertici x e y. Sia C l'insieme degli archi di G' aventi esattamente un estremo incidente a T_x e l'altro incidente a T_y. È immediato osservare che C è composto da archi che attraversano un taglio di G', nessuno dei quali compare nella minima foresta ricoprente. Applicando la regola del taglio, detto f l'arco di peso minimo in C, la nuova foresta F' è data da $F \setminus \{x, y\} \cup \{f\}$. L'arco f viene detto *arco di rimpiazzo*, o semplicemente *rimpiazzo*.

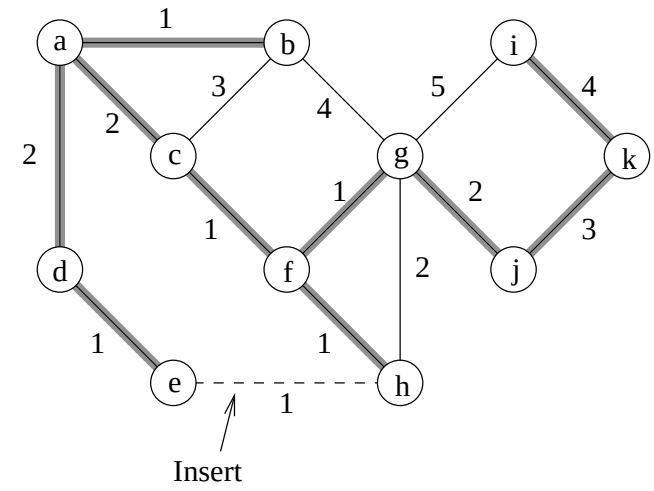


(a)

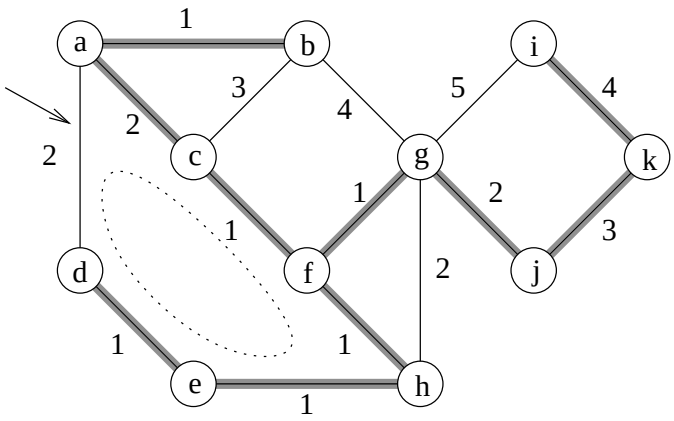


(b)

Figura 2.7: Effetto della cancellazione di un arco sulla minima foresta ricoprente. In 2.7(a) è riportato il grafo iniziale, con gli archi appartenenti alla minima foresta ricoprente F in evidenza. In 2.7(b) è riportato il grafo e la nuova foresta risultanti dalla cancellazione dell'arco $\{e, f\}$; l'arco più leggero che ricollega i sottoalberi di F risultanti dalla cancellazione di $\{e, f\}$, in questo caso l'arco $\{a, c\}$, viene inserito nella nuova foresta.



(a)



(b)

Figura 2.8: Effetto dell'inserimento di un arco sulla minima foresta ricoprente. In 2.8(a) è riportato il grafo iniziale, con gli archi appartenenti alla minima foresta ricoprente F in evidenza. In 2.8(b) l'effetto dell'inserimento del nuovo arco $\{e, h\}$ di peso 1 su F. Uno degli archi più pesanti sul ciclo indotto da $\{e, h\}$ in F è $\{a, d\}$, e poiché $w\{e, h\} < w\{a, d\}$, la nuova foresta ricoprente diventa $F \cup \{e, h\} \setminus \{a, d\}$.

Capitolo 3

Connettività dinamica

In questo capitolo viene descritto l'algoritmo deterministico di Holm, de Lichtenberg e Thorup [25] in grado di mantenere una foresta ricoprente di un grafo $G = (V, E)$, con $|V| = n$ e $|E| = m$, supportando inserimenti e rimozioni di archi in tempo ammortizzato $O(\log^2 n)$ per operazione, permettendo di rispondere a interrogazioni del tipo "I vertici u e v sono connessi?" in tempo $O(\log n / \log \log n)$ nel caso pessimo. Successivamente, viene brevemente illustrato l'algoritmo randomizzato di Henzinger e King [21] per la risoluzione del medesimo problema.

3.1 Lavori precedenti

Per quanto riguarda gli algoritmi deterministici, tutte le precedenti soluzioni al problema della connettività erano anche soluzioni al problema del mantenimento di una minima foresta ricoprente. Nel 1985 Frederickson [13] introdusse una struttura dati chiamata *topology tree* per risolvere il problema del mantenimento di una minima foresta ricoprente su grafi pesati dinamici in tempo $O(\sqrt{m})$ per aggiornamento nel caso pessimo, supportando interrogazioni sulla connettività in tempo $O(\log n / \log(\sqrt{m} / \log n)) = O(1)$. Nel 1992, Eppstein, Galil, Italiano e Nissenzweig [11] migliorarono il tempo per aggiornamento portandolo a $O(\sqrt{n})$ utilizzando la tecnica della *sparsificazione*. Infine, nel 1997 Henzinger e King [23] presentarono un algoritmo in grado di supportare

aggiornamenti in tempo ammortizzato $O(\sqrt[3]{n}\log n)$ per operazione, e rispondere a interrogazioni sulla connettività in tempo $O(\log n/\log\log n)$ nel caso pessimo.

Per ottenere migliori risultati sul problema della connettività, si è fatto ricorso ad algoritmi randomizzati. Nel 1995 Henzinger e King [21] svilupparono un algoritmo randomizzato per il mantenimento di una foresta ricoprente in tempo atteso ammortizzato $O(\log^3 n)$ per aggiornamento, supportando interrogazioni in tempo $O(\log n/\log\log n)$. Il tempo per l'aggiornamento è stato portato a $O(\log^2 n)$ nel 1996 da Henzinger e Thorup [24].

Nella tabella 3.1 sono riassunte le prestazioni degli algoritmi per la risoluzione del problema della connettività.

Autore	*Anno*	*Update time*	*Query time*
Frederickson [13] *D*	1985	$O(\sqrt{m})$	$O(1)$
Eppstein ed altri [11] *D*	1992	$O(\sqrt{n})$	$O(1)$
Henzinger e King [21] *R*	1995	$O(\log^3 n)$	$O(\log n/\log\log n)$
Henzinger e Thorup [24] *R*	1996	$O(\log^2 n)$	$O(\log n/\log\log n)$
Henzinger e King [23] *D*	1997	$O(\sqrt[3]{n}\log n)$	$O(\log n/\log\log n)$

Tabella 3.1: Cronologia dei risultati precedenti sul problema della connettività. Nella prima colonna, accanto all'autore, è riportata la lettera D se si tratta di un algoritmo deterministico, oppure R se si tratta di un algoritmo randomizzato.

In un recente lavoro [25], Holm, de Lichtenberg e Thorup hanno sviluppato un nuovo algoritmo deterministico in grado di risolvere il problema della connettività dinamica in tempo ammortizzato $O(\log^2 n)$ per aggiornamento, eguagliando quindi le prestazioni del miglior algoritmo randomizzato conosciuto e migliorando sensibilmente il miglior risultato deterministico. Dall'algoritmo per la connettività ne viene ricavato uno per il mantenimento di una minima foresta ricoprente su un gra-

fo dinamico in tempo $O(\log^4 n)$ per aggiornamento (si veda [25] per i dettagli).

Viene descritto in seguito l'algoritmo deterministico di Holm ed altri per la soluzione del problema della connettività, il quale, combinato con alcune tecniche proposte in [21] e con idee originali, costituisce una delle componenti degli algoritmi presentati nel capitolo 4.

3.2 Test di connettività deterministico

3.2.1 Descrizione ad alto livello

Si consideri un grafo non orientato $G = (V, E)$, con $|V| = n$. Dati due vertici u, v si vuole determinare nel modo più efficiente possibile se u e v sono connessi in G, ove G può essere modificato mediante inserimenti e rimozioni di archi. L'algoritmo deterministico descritto in seguito è in grado di rispondere a questo tipo di interrogazione mantenendo una foresta ricoprente F del grafo G, da cui due vertici sono connessi in G se e solo se sono connessi in F.

Internamente, l'algoritmo associa ad ogni arco $e \in E$ un numero intero $\ell(e)$, tale che $0 \leq \ell(e) \leq L = \lfloor \log_2 n \rfloor$; $\ell(e)$ denota il *livello* dell'arco e. Per ogni $i = 0, 1, \ldots, L$, F_i denota la sottoforesta di F composta dagli archi di livello maggiore o uguale ad i, cioè $E(F_i) = \{e \in E(F) \mid \ell(e) \geq i\}$; quindi $F = F_0 \supseteq F_1 \supseteq \cdots \supseteq F_L$. Durante l'esecuzione dell'algoritmo, vengono mantenute le seguenti invarianti:

(i) F è una foresta ricoprente di G avente peso massimo rispetto alla funzione peso ℓ; quindi il ciclo indotto su F da un arco $f \in E(G) \setminus E(F)$ è composto da archi di livello maggiore o uguale a $\ell(f)$.

(ii) Il massimo numero di vertici di un albero in F_i è $n/2^i$.

Inizialmente, tutti gli archi hanno livello 0, e quindi le invarianti sono soddisfatte; mano a mano che l'algoritmo procede, il livello di un arco può essere solo incrementato, in modo tale che non ci possano essere più di L incrementi di livello per ciascun arco. Intuitivamente, il livello

di un arco viene incrementato quando si scopre che i suoi estremi sono incidenti ad un albero di un livello più alto (e quindi con meno vertici, per l'invariante (ii)).

Diamo una descrizione ad alto livello delle procedure **Insert** e **Delete**, utilizzate rispettivamente per inserire e cancellare un arco dal grafo. La procedura **Insert**(e) aggiorna il grafo, ed eventualmente la foresta F, inserendo l'arco e. La funzione **Delete**(e) cancella l'arco e dal grafo. Se e appartiene ad un albero $T \subseteq F$, la rimozione di e sconnette T in due sottoalberi, T_1 e T_2, per cui si rende necessario cercare un arco f che riconnetta T_1 e T_2. Se un tale arco esiste, la nuova foresta ricoprente diventa $F \setminus \{e\} \cup \{f\}$, e la funzione ritorna f; altrimenti, la funzione ritorna **null** e la nuova foresta diventa $F \setminus \{e\}$.

Insert(e) Al nuovo arco è assegnato livello 0. Se gli estremi di e non sono connessi in $F = F_0$, e viene aggiunto ad F_0.

Delete(e) L'arco e viene rimosso da tutte le strutture dati in cui compare; inoltre, se $e \in E(F)$, si cerca un arco di livello massimo possibile che riconnetta F (tale arco viene indicato come *arco di rimpiazzo*). Dato che F era una massima foresta ricoprente rispetto ad ℓ, l'arco di rimpiazzo, se esiste, deve avere livello $\leq \ell(e)$; per trovare il rimpiazzo viene richiamata la funzione **Replace**$(e, \ell(e))$.

Replace$(\{v, w\}, i)$ Assumendo che non sia stato trovato un arco di rimpiazzo per $\{v, w\}$ a livello maggiore di i, cerca a livello $\leq i$.

Siano T_v e T_w gli alberi di F_i contenenti i vertici v e w rispettivamente, come risultano dopo la rimozione dell'arco $\{v, w\}$ da F. Definiamo *dimensione* di un albero T, che indicheremo con $s(T)$, il numero di vertici di T. Si assuma, senza perdita di generalità, che $s(T_v) \leq s(T_w)$; diremo in questo caso che l'albero T_v è più piccolo dell'albero T_w. Prima della rimozione di $\{v, w\}$, $T = T_v \cup \{v, w\} \cup T_w$ era un albero a livello i con al massimo il doppio dei vertici di T_v. Dall'invariante (ii) necessariamente T aveva $n/2^i$ vertici, così che T_v ne ha al più $n/2^{i+1}$. Dunque, preservando l'invariante, è possibile incrementare di uno il livello di tutti gli archi di T_v di livello i, in modo tale da rendere T_v un albero in F_{i+1}.

Fatto ciò, tutti gli archi di livello i incidenti a T_v che non appartengano già ad F sono esaminati uno alla volta fino a che viene trovato un rimpiazzo o tutti gli archi sono stati considerati. Sia f un arco esaminato durante la ricerca.

Se f collega T_v e T_w, cioè è incidente sia a T_v sia a T_w, l'arco f viene inserito nella foresta ricoprente come arco di rimpiazzo e la ricerca termina.

Se f non collega T_v e T_w, il livello di f viene incrementato; si osservi che in questo caso gli estremi di f sono entrambi in T_v.

Terminato l'esame di tutti gli archi a livello i incidenti a T_v, se $i = 0$ si conclude che non esiste un arco di rimpiazzo e si termina; altrimenti, si chiama **Replace**$(\{v, w\}, i - 1)$.

Algoritmo 3.1 Insert(e)

```
Let e = {u, v}
ℓ(e) := 0
Insert e in G
if u, v are not connected in F = F_0 then
  Insert e in F_0
end if
```

Algoritmo 3.2 Delete(e)

```
if e ∈ E(F) then
  Remove e from G
  for j = 0 to ℓ(e) do
    Remove e from F_j
  end for
  Exit and return Replace(e, ℓ(e))
else
  Remove e from G
  Exit and return null
end if
```

Algoritmo 3.3 Replace$(\{u, v\}, i)$

Require: There's no replacemente edge for $\{u, v\}$ on level $> i$
Ensure: Return the replacemente edge with maximum level, if one exists

```
Let T_v be the tree in F_i containing v
Let T_w be the tree in F_i containing w
if s(T_v) ≤ s(T_w) then
  T_small := T_v
else
  T_small := T_w
end if
Make T_small a tree in F_{i+1}
for all edges f ∈ E(G) \ E(F) on level i incident to T_small do
  if f connects T_v and T_w then
    for j = 0 to i do
      Insert f in F_j
    end for
    Exit and return f
  else
    ℓ(f) := ℓ(f) + 1
  end if
end for
if i = 0 then
  Exit and return null
else
  Exit and return Replace({u, v}, i − 1)
end if
```

In figura 3.1 è riportato un esempio di funzionamento della procedura **Delete**.

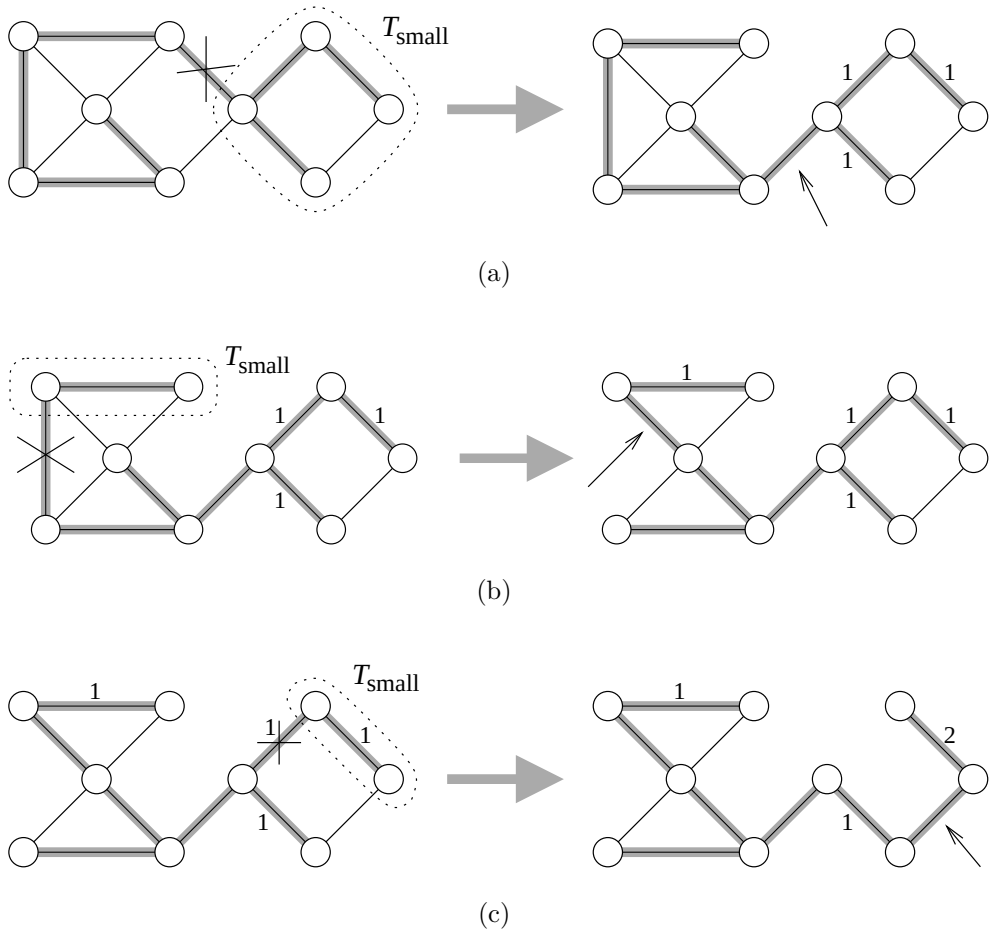


Figura 3.1: La figura illustra l'evoluzione della struttura dati per la connettività in seguito alla cancellazione di archi. Gli archi che appartengono alla foresta ricoprente F sono evidenziati in grassetto; accanto a ciascun arco viene riportato il livello assegnato, se diverso da zero.

3.2.2 Correttezza dell'algoritmo

Per dimostrare la correttezza dell'algoritmo si procederà verificando che le due invarianti descritte in precedenza sono mantenute nel corso dell'algo-

ritmo, cioè che esse rimangono valide dopo ogni chiamata delle procedure **Insert** e **Delete**. Ricordiamo tali invarianti:

(i) F è una foresta ricoprente di G avente peso massimo rispetto alla funzione peso ℓ; quindi il ciclo indotto su F da un arco $f \in E(G) \setminus E(F)$ è composto da archi di livello maggiore o uguale a $\ell(f)$.

(ii) Il massimo numero di vertici di un albero in F_i è $n/2^i$.

Lemma 3.1 *L'invariante (i) viene mantenuta.*

Dimostrazione. Supponiamo che l'invariante sia inizialmente valida. Essa è certamente valida anche dopo l'inserimento di un arco in G. Assumiamo che essa sia verificata prima della chiamata di **Delete**(e), e dimostriamo che è valida anche dopo l'esecuzione della procedura. Distinguiamo due casi:

1. $e \in E(G) \setminus E(F)$. L'esecuzione di **Delete**(e) comporta solamente la cancellazione di e e non viene modificata la foresta F; pertanto l'invariante si mantiene.

2. $e \in E(F)$. Dopo aver rimosso e, viene invocata la procedura **Replace**$(e, \ell(e))$ per cercare un arco che riconnetta F; l'inserimento del rimpiazzo non pregiudica di per sè l'invariante, dato che tra tutti gli archi che riconnettono T_v e T_w viene scelto sempre quello di livello massimo. Tuttavia, durante l'esecuzione della procedura, il livello degli archi presi in esame viene incrementato, e questo potrebbe invalidare l'invariante. Osserviamo però che incrementare il livello di un arco che appartiene già ad F non pregiudica l'invariante; inoltre, il livello di un arco $f \in E(G) \setminus E(F)$ viene incrementato da i a $i+1$ solo se *entrambi* i suoi estremi sono incidenti ad un albero $T \subseteq F_i$, ma per come è strutturata la procedura **Replace**, prima di incrementare il livello di f, tutti gli archi di T aventi livello i vengono spostati al livello $i+1$, da cui T risulta alla fine composto da archi di livello $\geq i+1$. Anche in questo caso l'invariante rimane valida.

□

Lemma 3.2 *L'invariante (ii) viene mantenuta.*

Dimostrazione. Poiché ad ogni nuovo arco viene assegnato livello 0, l'invariante rimane valida dopo l'esecuzione della procedura **Insert**(e). Durante la **Replace**, l'albero $T_v \subseteq F_i$ che viene aggiunto a F_{i+1} è l'albero più piccolo che risulta dalla cancellazione dell'arco $\{v, w\}$ da $T \subseteq F_i$. Come già puntualizzato nella descrizione della procedura, T_v ha al più $n/2^{i+1}$ vertici. Ciò preserva l'invariante (ii). □

3.2.3 Implementazione ed analisi

Per garantire una implementazione efficiente dell'algoritmo, è necessario disporre di strutture dati che:

- Mantengano efficientemente tutte le foreste F_i, insieme agli archi di livello i appartenenti all'insieme $E(G) \setminus E(F)$, per ogni $0 \leq i \leq \lfloor \log_2 n \rfloor$;
- Permettano di individuare l'albero T_v in F_i contenente un dato vertice v;
- Permettano di calcolare la dimensione (numero di vertici) di un albero T_v;
- Permettano di recuperare un arco a livello i incidente ad un albero T, se tale arco esiste;
- Supportino in maniera efficiente la possibilità di scollegare o ricollegare alberi in F_i, rispettivamente quando un arco viene cancellato e quando un arco di rimpiazzo viene trovato, un nuovo arco è inserito o il livello di un arco viene incrementato;

Tutte le precedenti operazioni possono essere eseguite in tempo $O(\log n)$ nel caso pessimo usando la struttura dati ET-tree descritta in [21]. Ogni

albero in F_i viene rappresentato mediante un ET-tree; ogni nodo dell'ET-tree contiene un intero che rappresenta la dimensione del segmento di Euler tour sottostante, un bit che indica se esiste un arco di F avente livello i in tale segmento e un bit che indica se c'è un arco nell'insieme $E(G) \setminus E(F)$ a livello i incidente ad un vertice del segmento. Naturalmente, anche con queste modifiche le operazioni descritte sopra sono supportate in tempo $O(\log n)$ nel caso pessimo (si veda l'appendice A per i dettagli relativi all'implementazione).

Verranno di seguito esaminati i costi computazionali delle varie procedure.

Inserimento di un nuovo arco L'inserimento di un arco nel grafo comporta un costo diretto pari a $O(\log n)$, e poiché il livello dell'arco può essere incrementato al più $\log n$ volte, ciascuna delle quali con un costo $O(\log n)$ (vedi la descrizione della procedura **Delete**), il costo totale ammortizzato caricato su ciascun arco è $O(\log n + \log^2 n) = O(\log^2 n)$.

Cancellazione di un arco La cancellazione di un arco non appartenente alla foresta ricoprente F richiede tempo $O(\log n)$. Quando un arco $e \in E(F)$ viene cancellato, occore innanzitutto eliminarlo da tutte le foreste F_j, $j \leq \ell(e)$, con un costo immediato di $O(\log^2 n)$; successivamente, si possono avere $O(\log n)$ chiamate ricorsive alla procedura **Replace**, ciascuna avente costo $O(\log n)$, più il costo relativo all'incremento del livello degli archi (che è già stato pagato durante l'inserimento). Infine, se viene trovato un arco di rimpiazzo si rende necessario ricollegare $O(\log n)$ foreste, in tempo totale $O(\log^2 n)$. Il costo ammortizzato relativo alla cancellazione è quindi $O(\log^2 n)$.

Test di connettività Si è visto che il costo ammortizzato relativo all'inserimento e alla cancellazione di un arco è pari a $O(\log^2 n)$. Gli ET-trees mediante i quali viene memorizzata la foresta $F = F_0$ permettono di controllare se due vertici u, v sono connessi in tempo $O(\log n)$, semplicemente controllando se appartengono allo stesso albero. Allo scopo di ridurre il tempo a $O(\log n / \log\log n)$, viene introdotto un ulteriore B-tree di grado $\Theta(\log n)$ sopra la ET-sequence di ciascun albero in F. L'albero

ha profondità $O(\log n / \log\log n)$, che è dunque il tempo necessario a rispondere a interrogazioni sulla connettività. Ogni modifica alla foresta F comporta al più un singolo inserimento e una singola cancellazione negli alberi di grado $\Theta(\log n)$, che possono essere portate a termine in tempo $O(\log^2 n / \log\log n)$.

Oltre a queste operazioni, è possibile ottenere la lista di tutti i vertici appartenenti ad una determinata componente connessa in tempo $O(C)$, essendo C il numero di vertici della componente connessa, sfruttando gli ET-trees di grado $\Theta(\log n)$ che memorizzano le componenti connesse del grafo G (si veda l'appendice A per i dettagli).

Il grafo viene rappresentato come segue: gli n vertici sono memorizzati in un vettore di dimensione n, in base ad un ordinamento totale arbitrario. Ad ogni vertice v è associato l'insieme degli archi del grafo incidenti ad esso, ordinati in base all'estremo diverso da v e memorizzati in un albero di ricerca bilanciato. Lo spazio richiesto è complessivamente $O(m+n)$, e mediante tale rappresentazione è possibile supportare inserimenti e rimozioni di archi in tempo $O(\log n)$ nel caso pessimo (poiché non vi possono essere più di n archi incidenti allo stesso vertice). Per la memorizzazione delle foreste di ET-trees sui livelli, è richiesto spazio $O(m + n\log n)$, in quanto ogni arco di $E(G) \setminus E(F)$ è incidente ad un solo ET-tree; lo spazio complessivamente richiesto dall'algoritmo è $O(m + n\log n)$.

Si può concludere con il seguente

Teorema 3.1 (Holm, de Lichtenberg e Thorup [25]) *Dato un grafo non orientato G con n vertici e m archi, esiste un algoritmo dinamico deterministico in grado di stabilire se due vertici dati sono connessi in tempo $O(\log n / \log\log n)$ nel caso pessimo, ed in grado di supportare inserimenti e cancellazioni di archi in tempo ammortizzato $O(\log^2 n)$ per operazione. Lo spazio richiesto dall'algoritmo è $O(m + n\log n)$.*

3.3 Test di connettività randomizzato

Per completezza, viene brevemente descritto l'algoritmo randomizzato per la connettività di Henzinger e King [21]. Esso viene presentato in due

fasi: dapprima viene descritto un algoritmo randomizzato *decrementale* per il mantenimento di una foresta ricoprente di un grafo G soggetto solamente a rimozioni di archi. Successivamente, tale algoritmo viene esteso per poter gestire anche inserimenti di archi.

L'algoritmo decrementale si basa sull'idea seguente: gli archi del grafo vengono partizionati in $O(\log n)$ livelli, in modo tale che archi che si trovano in componenti fortemente connesse del grafo (dove eventuali tagli sono densi) sono in livelli più bassi rispetto a quelli che si trovano in componenti debolmente connesse del grafo (dove eventuali tagli sono sparsi). Per ogni livello i viene mantenuta una foresta ricoprente del grafo composto da tutti gli archi di livello $\leq i$. Se viene cancellato un arco di livello i appartenente alla foresta ricoprente F, vengono selezionati in maniera casuale degli archi a livello i in modo tale che: (1) viene trovato un arco che riconnette F, oppure (2) il taglio indotto dalla cancellazione dell'arco è troppo sparso per il livello i. Nel caso (1) viene trovato rapidamente un arco di rimpiazzo, mentre nel caso (2) si provvede a ricopiare a livello $i+1$ tutti gli archi di livello i che attraversano il taglio, e si ripete la selezione casuale a livello $i+1$.

3.3.1 Algoritmo decrementale

Sia $G = (V, E)$ un grafo non orientato, con $|V| = n$ e $|E| = m$. Sia $l = \lfloor \log m - \log\log n \rfloor + 1$. Gli archi di G sono partizionato in l livelli $E_1, E_2, \ldots, E_l$ tali che $\cup_i E_i = E$, e per ogni $i \neq j$, $E_i \cap E_j = \emptyset$. Per ogni i viene mantenuta una foresta ricoprente F_i del grafo $G_i = (V, \cup_{j \leq i} E_j)$; quindi, per $i = 2, 3, \ldots, l$ risulta $F_{i-1} \subseteq F_i$ e $E(F_i) \setminus E(F_{i-1}) \subset E_i$. Conseguentemente F_l è una foresta ricoprente di G; un albero ricoprente T a livello i è un albero di F_i.

Il *peso* di T, denotato come $w(T)$ è il numero di archi incidenti a T che non fanno parte della foresta ricoprente F, ove archi aventi entrambi gli estremi incidenti a T sono contati due volte. La *dimensione* di T, come definita nel paragrafo precedente, denotata come $s(T)$, è il numero di vertici in T. Un albero T' è più piccolo di T'' se $s(T') \leq s(T'')$. Diremo che il livello i è inferiore al livello $i+1$.

Inizialmente, tutti gli archi del grafo sono memorizzato in E_1, e viene calcolata una foresta ricoprente F_1 di G.

Quando un arco e viene cancellato, esso viene rimosso dal grafo che lo contiene. Se $e \in E(F)$, sia i il livello tale che $e \in E_i$. Viene invocata la procedura **Rand-Replace**(e, i).

Rand-Replace(e, i) Sia T l'albero a livello i contenente e, e siano T_1 e T_2 i sottoalberi di T che risultano dalla rimozione di e, tali che $s(T_1) \leq s(T_2)$.

- Se $w(T_1) \leq \log^2 n$, si prosegue col caso 2.
- *Selezione*: Vengono scelti $c \log^2 n$ archi di $E_i \setminus E(F)$ incidenti a T_1, per una opportuna costante c. Un arco con entrambi gli estremi in T_1 è selezionato con probabilità $2/w(T_1)$, mentre un arco con un solo estremo in T_1 è selezionato con probabilità $1/w(T_1)$.
- *Caso 1 (Trovato un arco di rimpiazzo)*: se uno degli archi selezionati riconnette T_1 e T_2, tale arco viene inserito in tutte le foreste $F_j, j \geq i$.
- *Caso 2 (Non è stato trovato un rimpiazzo)*: Se nessuno degli archi selezionati riconnette T_1 e T_2, vengono esaminati tutti gli archi non in F incidenti a T_1, e si determina l'insieme $S = \{\text{Archi che riconnettono } T_1 \text{ e } T_2\}$.
  - Se $|S| > w(T_1)/(2c' \log n)$, uno degli elementi di S viene aggiunto a $F_j, j \geq i$.
  - Se $0 < |S| \leq w(T_1)/(2c' \log n)$, gli elementi di S vengono rimossi da E_i e reinseriti in E_{i+1}. Quindi uno di tali archi viene anche inserito in $F_j, j > i$.
  - Se $S = \emptyset$ e $i < l$, allora esegui **Rand-Replace**$(e, i + 1)$, altrimenti l'algoritmo termina.

Le costanti c e c' devono essere scelte in maniera opportuna, distinguendo se si tratta dell'algoritmo decrementale o di quello totalmente dinamico. Nel primo caso si pone $c = 8$ e $c' = 2$, nel secondo $c = 16$ e $c' = 4$.

Teorema 3.2 (Henzinger e King [21]) *Sia G un grafo con m_0 archi ed n vertici soggetto solamente a rimozioni di archi. Una foresta ricoprente di G può essere mantenuta in tempo atteso ammortizzato $O(\log^3 n)$ per cancellazione, se vi sono almeno $\Omega(m_0)$ cancellazioni. Inoltre, è possibile decidere se due vertici u e v sono connessi in tempo $O(\log n)$.*

3.3.2 Algoritmo dinamico

Vediamo ora come è possibile estendere l'algoritmo della sezione precedente per trattare anche inserimenti di archi. Quando un arco $\{u, v\}$ viene inserito in G, esso viene aggiunto ad E_l. Se u e v non erano precedentemente connessi in G, l'arco $\{u, v\}$ viene inserito anche in F.

Sia $l = \lceil 2 \log n \rceil$ il numero di livelli. La struttura dati deve ricostruita periodicamente; la *ricostruzione del livello i*, per $i \geq 1$, viene effettuata mediante una operazione **move-edges**(i), che sposta tutti gli archi di E_j per $j > i$ in E_i; in più, per ogni $j < i$, tutti gli archi di $E_j \cap E(F)$ sono inseriti in tutti gli $F_k, j < k \leq i$. Si noti che dopo una ricostruzione a livello i, $E_j, j > i$ non contiene alcun arco, e $F_j = F_i$ per ogni $j \geq i$, ossia le foreste ricoprenti a livello $j \geq i$ ricoprono interamente G.

Dopo ogni inserimento viene incrementato I, il numero di inserimenti modulo $2^{\lceil 2 \log n \rceil}$ da quando l'algoritmo è iniziato. Sia j il massimo intero tale che 2^j divide I. Dopo che un arco è stato inserito, viene effettuata una ricostruzione del livello $l - j - 1$. Rappresentando I in notazione binaria come $b_0 b_1 \ldots b_{l-1}$, ove b_0 è il bit più significativo, una ricostruzione del livello i ha luogo ogni volta che b_i passa da 0 a 1.

Teorema 3.3 (Henzinger e King [21]) *Sia G un grafo con m_0 archi ed n vertici, soggetto a rimozioni ed inserimenti di archi. Una foresta ricoprente F di G può essere mantenuta in tempo atteso ammortizzato $O(\log^3 n)$ per operazione, se vi sono almeno $\Omega(m_0)$ operazioni. Inoltre è possibile verificare se due vertici u e v sono connessi in G in tempo $O(\log n)$. Lo spazio richiesto è $O(m + n \log n)$.*

Si ricorda che recentemente Henzinger e Thorup hanno migliorato il tempo per aggiornamento del teorema 3.3 da $O(log^3 n)$ a $O(\log^2 n)$. Il lettore interessato può consultare [24]. Inoltre, mediante una opportuna

implementazione dell'algoritmo, è possibile ridurre il tempo richiesto per decidere se due vertici sono connessi da $O(\log n)$ a $O(\log n / \log \log n)$.

Capitolo 4

Algoritmi

In questo capitolo vengono illustrati gli algoritmi deterministici per il mantenimento di proprietà su grafi dinamici, basati sull'algoritmo per la connettività dinamica presentato in 3.2. Tutti gli algoritmi operano su grafi non orientati $G = (V, E)$, con V fisso (nel senso che non è possibile aggiungere o eliminare vertici), ed E modificabile mediante inserimenti e rimozioni di archi.

4.1 Minima Foresta Ricoprente con k pesi

Nel problema della minima foresta ricoprente con k pesi (k-Weight Minimum Spanning Forest) è richiesto di mantenere una minima foresta ricoprente di un grafo non orientato $G = (V, E)$ con funzione peso $w : E \to \{1, 2, \ldots, k\}$, per un dato intero $k \geq 1$ fissato.

Viene presentata una struttura dati in grado di mantenere una minima foresta ricoprente F di G, supportando le seguenti operazioni:

k-weight-Insert(e) per inserire un arco e in G, eventualmente aggiornando la foresta F; si assume di memorizzare preventivamente il peso $w(e)$ come attributo di e.

k-weight-Delete(e) per cancellare un arco e da G, aggiornando eventualmente la foresta F. Se viene cancellato un arco di F, e viene

trovato un rimpiazzo, la funzione restituisce il rimpiazzo, altrimenti viene restituito il valore speciale **null**.

4.1.1 L'algoritmo

Inizialmente, si determina una minima foresta ricoprente F di G. Per ogni $i = 1, 2, \ldots, k$, sia $E_i = \{e \in E(G) \mid w(e) = i\} \cup E(F)$, cioè E_i sia l'insieme costituito dagli archi di peso i e da quelli che fanno parte della foresta F; se in qualunque momento non esistono archi di peso i, sarà $E_i = E(F)$. Definiamo una sequenza di grafi $G_1, G_2, \ldots, G_k$ tali che $G_i = (V, E_i)$. Ciascuno di essi viene memorizzato in una struttura per la connettività $\mathcal{A}_i$, come descritta nel capitolo precedente; in questo modo per ciascun grafo G_i viene mantenuta una foresta ricoprente F_i. Chiameremo le foreste F_i *foreste ricoprenti locali*, mentre F indicherà la *foresta globale*, ossia la minima foresta ricoprente del grafo $G = (V, E)$ rispetto alla funzione peso w. Si tenga presente che l'algoritmo per la connettività ignora totalmente il peso degli archi, per cui le foreste locali non hanno alcun legame particolare con la foresta globale F. L'unica relazione che rimane valida è $F \subseteq G_i$ per ogni i.

Come visto nel capitolo precedente, ciascuna delle strutture dati $\mathcal{A}_i$ è in grado di supportare l'inserimento e la cancellazione di archi, aggiornando eventualmente la foresta ricoprente locale, in tempo ammortizzato $O(\log^2 n)$ per operazione.

La foresta globale F e le foreste locali F_i vengono memorizzate anche mediante Dynamic trees, per cui ogni modifica in $\mathcal{A}_i$ può comportare un costo aggiuntivo di $O(\log n)$ per l'aggiornamento del Dynamic tree associato; il costo ammortizzato per operazione nelle strutture $\mathcal{A}_i$ rimane comunque uguale a $O(\log^2 n)$, visto che ogni operazione di inserimento o rimozione di archi comporta al più $O(1)$ modifiche al Dynamic tree.

Le procedure **k-weight-Insert**(e) e **k-weight-Delete**(e) vengono implementate come segue:

k-weight-Insert(e) Sia $e = \{u, v\}$. Se u e v non sono connessi in F, allora si aggiunge e ed F e a tutti i grafi G_i, $i = 1, 2, \ldots, k$.

Se invece u e v sono già connessi in F, si cerca l'arco f di peso

massimo sul ciclo indotto da e in F, utilizzando il Dynamic tree che rappresenta F. Si possono verificare due casi:

- Se il peso di f è superiore a quello di e, la nuova foresta globale diventa $F \setminus \{f\} \cup \{e\}$, l'arco e viene inserito in tutti i grafi G_i, $i = 1, 2, \ldots, k$, mentre f viene rimosso da $G_j, j \neq w(f)$.
- Se il peso di f è minore o uguale al peso di e, quest'ultimo viene semplicemente inserito in $G_{w(e)}$, e la foresta globale F rimane immutata.

k-weight-Delete(e) Sia $e = \{u, v\}$. Si cancella e da tutti i grafi in cui compare, utilizzando la procedura di cancellazione di un arco su ciascuna delle strutture dati $\mathcal{A}_i$; questo fa sì che se $e \in E(F_i)$ per qualche i, la rimozione di e comporti l'aggiornamento della foresta locale F_i.

Se $e \in E(F)$, allora e viene cancellato anche da F, per cui si rende necessario trovare (se esiste) l'arco più leggero che riconnette gli alberi T_u e T_v contenenti gli estremi di e. Per fare ciò, si determina dapprima il minimo l tale che u e v siano connessi in G_l, e successivamente si cerca l'arco che riconnette T_u e T_v, mediante una ricerca binaria sul cammino $u \overset{P}{\rightsquigarrow} v$ in F_l nel modo seguente: si individua il vertice x che occupa la posizione centrale in P. Se u e x sono connessi in F, si ripete la ricerca nel cammino $x \overset{P'}{\rightsquigarrow} v$ in F_l, altrimenti la ricerca continua nel cammino $u \overset{P''}{\rightsquigarrow} x$. Tutto questo richiede al più $O(\log n)$ iterazioni, ciascuna di costo $O(\log n)$, utilizzando il Dynamic tree che rappresenta F_l; l'arco che attraversa il taglio indotto in F dalla rimozione di e può quindi essere individuato in tempo totale $O(\log^2 n)$, per poi essere inserito in F e in tutti i rimanenti grafi $G_j, j \neq l$.

4.1.2 Correttezza dell'algoritmo

Quando un arco $\{u, v\}$ viene inserito in G, la minima foresta ricoprente F viene sempre modificata in maniera corretta utilizzando il Dynamic

Algoritmo 4.1 k-weight-Insert(e)

```
Let e = {u, v}
if u, v are not connected in F then
  f-Insert(e)
else
  if e replaces f in F then
    f-Delete(f)
    Insert f in G_{w(f)}, updating F_{w(f)}
    f-Insert(e)
  else
    Insert e in G_{w(e)}, updating F_{w(e)}
  end if
end if
```

Algoritmo 4.2 k-weight-Delete(e)

```
Let e = {u, v}
if e ∈ E(F) then
  f-Delete(e)
  Let T_v be the tree in F containing v
  Let T_w be the tree in F containing w
  j := 1
  while j ≤ k do
    if u, v are connected in F_j then
      Let f be the edge in E(F_j) reconnecting T_v and T_w
      f-Insert(f)
      Exit and return f.
    else
      j := j + 1
    end if
    Exit and return null
  end while
else
  Remove e from G_{w(e)}, updating F_{w(e)}
  Exit and return null
end if
```

Algoritmo 4.3 f-Insert(e)

Insert e in F
for $i = 1$ to k **do**
  Insert e in G_i, updating F_i
end for

Algoritmo 4.4 f-Delete(e)

Remove e from F
for $i = 1$ to k **do**
  Remove e from G_i, updating F_i
end for

tree che la rappresenta. Se $\{u, v\}$ viene inserito in F, allora viene anche inserito in tutti i G_i, in modo da preservare l'invariante $F \subseteq G_i$ per ogni i. Se l'inserimento di $\{u, v\}$ rimpiazza l'arco f in F, allora f viene rimosso da tutti i grafi $G_j, j \neq w(f)$. Infine, se $\{u, v\}$ non viene inserito in F, allora entra a far parte solo di $G_{w\{u,v\}}$. In tutti i casi, dopo l'inserimento di $\{u, v\}$, vale sempre l'invariante $E_i = \{\text{archi di peso } i\} \cup E(F)$, per ogni $i = 1, 2, \ldots, k$.

Quando un arco $\{x, y\}$ viene cancellato, esso è rimosso da tutte le strutture in cui compare, aggiornando eventualmente le foreste ricoprenti locali. Se l'arco apparteneva ad F, ed esiste un rimpiazzo di peso minimo l, sicuramente x e y sono connessi in G_l, e la ricerca binaria sul percorso che connette x e y in F_l permette di individuare il rimpiazzo corretto. Le strutture dati vengono successivamente aggiornate per preservare l'invariante.

La figura 4.1 illustra con un esempio il funzionamento dell'algoritmo.

4.1.3 Analisi della complessità

Il tempo di esecuzione delle procedure **k-weight-Insert** e **k-weight-Delete** è dominato dal tempo necessario a inserire o cancellare un arco in tutte le strutture $\mathcal{A}_i$, $i = 1, 2, \ldots, k$, mediante le procedure **f-Insert** e **f-Delete**, che è pari a $O(k \log^2 n)$ per aggiornamento.

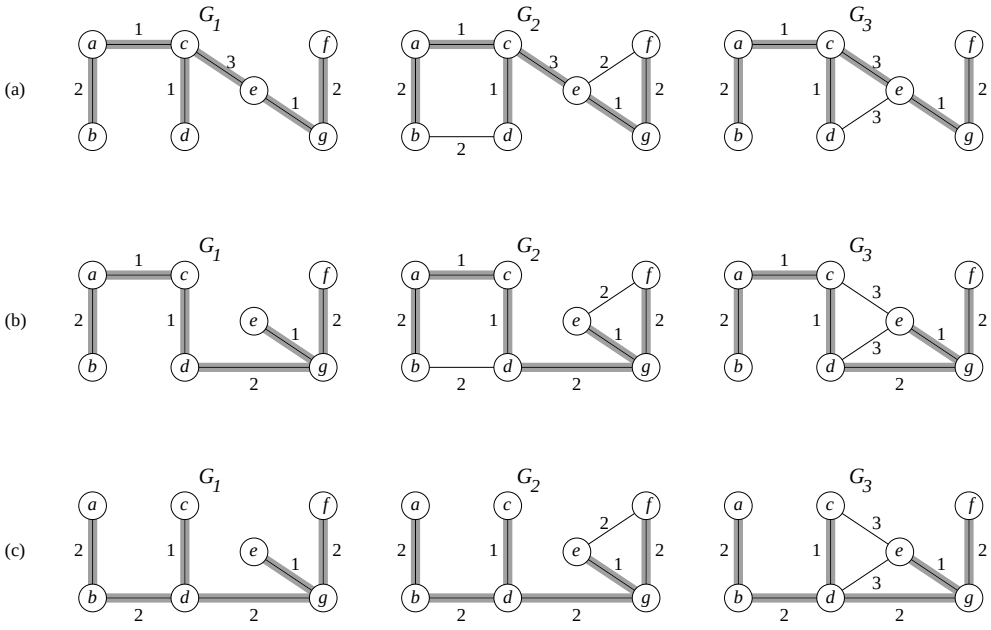


Figura 4.1: Esempio del funzionamento della struttura dati per il mantenimento di una minima foresta ricoprente di un grafo con 3 pesi distinti. In (a) è raffigurata la situazione iniziale; gli archi in grassetto individuano una minima foresta ricoprente (si noti che per ciascun grafo G_i l'algoritmo mantiene una foresta ricoprente locale che può **non** coincidere con la minima foresta ricoprente). In (b) la situazione dopo l'inserimento dell'arco $\{d, g\}$ di peso 2; si noti come cambia la minima foresta ricoprente. In (c) la situazione dopo la cancellazione dell'arco $\{a, c\}$.

Si ottiene così il seguente

Teorema 4.1 *Dato un grafo non orientato $G = (V, E)$ con n vertici, dotato di una funzione peso $w : E \to \{1, 2, \ldots, k\}$ con $k \geq 1$ intero fissato inizialmente, esiste un algoritmo deterministico in grado di mantenere una foresta ricoprente di peso minimo di G, supportando inserimenti e rimozioni di archi in tempo ammortizzato $O(k \log^2 n)$ per operazione.*

Si noti che per $k = 1$ il problema si riduce al mantenimento delle

componenti connesse di un grafo $G = (V, E)$; l'algoritmo proposto ha in tal caso la stessa complessità computazionale di quello visto in 3.2.

4.1.4 Estensioni

Con alcune semplici modifiche, l'algoritmo presentato può essere impiegato anche per mantenere una minima foresta ricoprente di un grafo G con funzione peso w non negativa, tale che in ogni istante gli archi presenti in G assumano al più k pesi arbitrari distinti, per un dato $k \geq 1$ fissato inizialmente.

Sia $G = (V, E)$ il grafo di partenza, i cui archi abbiano peso appartenente all'insieme $\{w_1, w_2, \ldots w_l\}$ per $l \leq k$. Sia F una minima foresta ricoprente di G. Per $i = 0, 1, \ldots, l$ sia $E_{w_i} = \{e \in E(G) \mid w(e) = w_i\} \cup E(F)$.

Sia $G_{w_i} = (V, E_{w_i})$, $i = 1, 2, \ldots, l$. Ogni grafo G_{w_i} viene memorizzato in un elemento di un vettore di dimensione k. Se $l < k$, $l - k$ elementi del vettore, che chiameremo *extra*, sono inizializzati a contenere grafi coincidenti con la minima foresta ricoprente F. Non appena viene introdotto un arco avente un nuovo peso, esso verrà memorizzato in uno degli extra; simmetricamente, quando tutti gli archi di peso w_j vengono cancellati da G, per qualche j, l'elemento del vettore che memorizza G_{w_j} diviene un extra. È opportuno osservare che ogni modifica della foresta F deve essere comunque riportata anche negli extra, per cui tutti i k grafi nel vettore devono essere aggiornati in seguito a ciascuna modifica della foresta globale F.

L'ordinamento dei G_{w_i} viene mantenuto in maniera dinamica rispetto ai pesi w_i, mediante un albero binario di ricerca bilanciato. Le chiavi memorizzate nell'albero sono i pesi distinti assunti dagli archi di G in un dato istante; nell'albero di ricerca, insieme a ciascun peso w_i viene memorizzato un puntatore al grafo G_{w_i} nel corrispondente elemento del vettore (si veda la figura 4.3). Le operazioni di ricerca mediante l'albero binario possono essere effettuate in tempo $O(\log k)$ nel caso pessimo. Inoltre è possibile inserire o cancellare nodi dall'albero, mantenendo il bilanciamento dello stesso, in tempo $O(\log k)$ nel caso pessimo.

Gli algoritmi presentati per il caso in cui il codominio della funzione

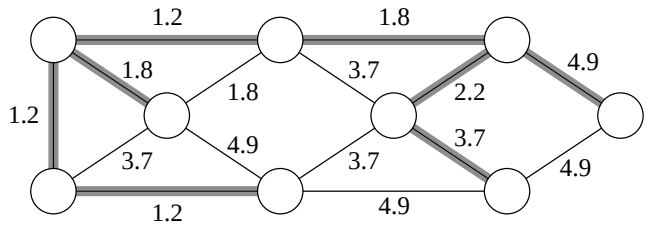


Figura 4.2: Esempio di grafo con 5 pesi distinti, appartenenti all'insieme $\{1.2, 1.8, 2.2, 3.7, 4.9\}$.

peso w è l'insieme $\{1, 2, \ldots, k\}$ si estendono con minime modifiche al caso più generale. Ogni volta che è richiesta la modifica di un grafo $G_{w(e)}$, l'elemento del vettore che lo contiene può essere individuato effettuando una ricerca sull'albero bilanciato, utilizzando il peso $w(e)$ come chiave. Nella procedura **k-weight-Delete**, invece di una ricerca sequenziale per individuare il minimo w_l tale che u e v siano connessi in G_{w_l}, si utilizza una ricerca binaria sull'albero bilanciato. La complessità della procedura **k-weight-Delete** risulta $O(\log k \log^2 n + k \log^2 n) = O(k \log^2 n)$. Si può quindi concludere con il seguente

Teorema 4.2 *Sia $G = (V, E)$ un grafo non orientato con n vertici, con funzione peso w non negativa. Supponiamo che in ogni istante gli archi eel grafo non possano assumere più di k pesi distinti, per un dato intero $k \geq 1$ fissato inizialmente. Allora esiste un algoritmo deterministico in grado di mantenere una foresta ricoprente di peso minimo di G, supportando inserimenti e rimozioni di archi in tempo ammortizzato $O(k \log^2 n)$ per operazione.*

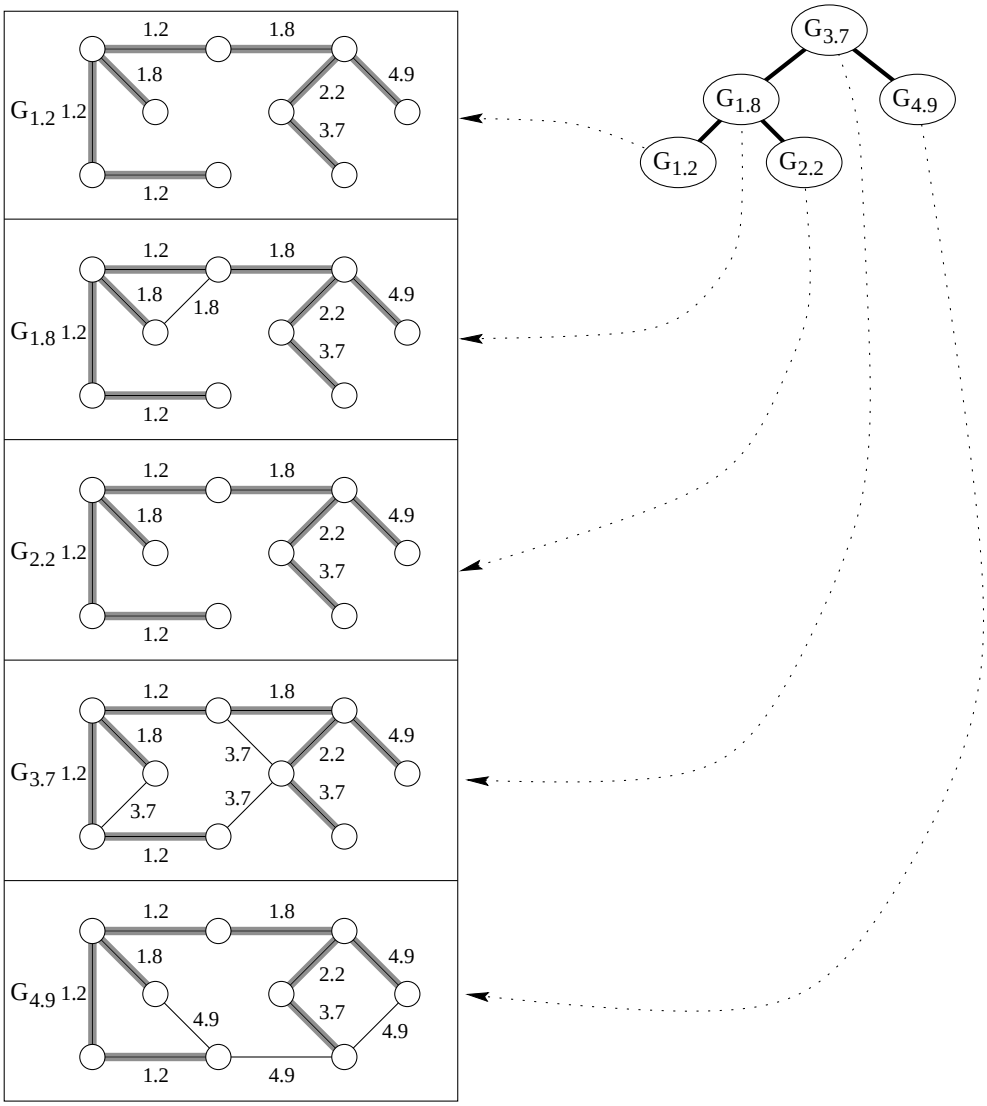


Figura 4.3: La struttura dati per il mantenimento di una minima foresta ricoprente con k pesi distinti. La figura si riferisce al grafo di figura 4.2, con 5 pesi. Si noti che in realtà le foreste ricoprenti locali che vengono mantenute sui grafi **non** coincidono necessariamente con la minima foresta ricoprente del grafo G. Gli archi evidenziati servono esclusivamente per rimarcare che in ogni grafo è contenuta la minima foresta ricoprente come sottografo.

4.2 1+ϵ-minima Foresta Ricoprente

Definizione 4.1 *Sia $G = (V, E)$ un grafo non orientato con funzione peso $w : E \to \mathbb{R}_{\geq 0}$. Sia F_m una minima foresta ricoprente di G rispetto alla funzione peso w. Una foresta ricoprente F di G è detta 1+ϵ-minima, per un dato numero reale $\epsilon > 0$, se*

$$w(F) \leq (1 + \epsilon)w(F_m)$$

Il problema della 1+ϵ-minima foresta ricoprente si può ricondurre a quello della minima foresta ricoprente con k pesi. D'ora in avanti supporremo che il peso di ogni arco di G sia compreso tra 1 e U, per un numero reale fissato $U > 1$. Sia $N = \lfloor \log U / \log(1 + \epsilon) \rfloor$, e per $i = 0, 1, \ldots, N$ sia

$$I_i = \left[(1 + \epsilon)^i, (1 + \epsilon)^{i+1}\right)$$

Risulta $\cup_{i=0}^{N} I_i = [1, (1 + \epsilon)U)$, e per ogni $i \neq j$, $I_i \cap I_j = \emptyset$. Definiamo la funzione h che mappa l'insieme $[1, (1 + \epsilon)U)$ nell'insieme $\{0, 1, \ldots, N\}$, come segue:

$$h(x) = j \text{ se e solo se } x \in I_j$$

o in modo equivalente

$$h(x) = \lfloor \log_{(1+\epsilon)} x \rfloor$$

Dimostreremo che gli archi che compongono una minima foresta ricoprente del grafo G rispetto alla funzione peso $h \circ w$ costituiscono una 1+ϵ-minima foresta ricoprente dello stesso grafo, con funzione peso w, ove si pone

$$\begin{aligned} (h \circ w)(F) &= \sum_{e \in E(F)} h(w(e)) \\ w(F) &= \sum_{e \in E(F)} w(e) \end{aligned}$$

4.2.1 Correttezza dell'algoritmo

Teorema 4.3 *Sia $G = (V, E)$ un grafo non orientato con funzione peso $w : E \to [1, U]$ per un dato $U > 1$. Sia $h(x) = \lfloor \log_{(1+\epsilon)} x \rfloor$; sia F una*

minima foresta ricoprente del grafo G rispetto alla funzione peso $h \circ w$. Allora F è una 1+ϵ-minima foresta ricoprente dello stesso grafo rispetto alla funzione peso w, ossia, data una minima foresta ricoprente F_m di G rispetto alla funzione peso w, si ha

$$w(F) \leq (1 + \epsilon)w(F_m)$$

Dimostrazione. Se $F = F_m$, la tesi segue immediatamente. Supponiamo quindi che $F \neq F_m$; dimostreremo che è possibile trasformare F_m in F, inserendo un arco di $E(F) \setminus E(F_m)$ e cancellandone uno di $E(F_m) \setminus E(F)$ alla volta. Per semplicità, d'ora in poi gli archi in $E(F) \setminus E(F_m)$ verranno indicati come *archi bianchi*, quelli di $E(F_m) \setminus E(F)$ come *archi neri*, e quelli in $E(F) \cap E(F_m)$ come *archi grigi* (in quanto sono sia neri che bianchi) (vedi figura 4.4).

L'idea nella trasformazione di F_m in F consiste nel cancellare di volta in volta un arco nero da F_m e sostituirlo con uno bianco, fino a quando non rimangono più archi neri. L'algoritmo 4.5 effettua la trasformazione di F_m in F.

Algoritmo 4.5 Trasformazione di F_m in F

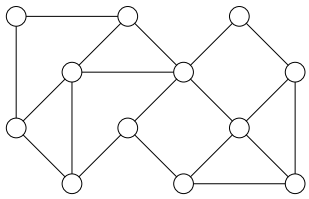
Require: F is a Minimum Spanning Forest w.r.t. $h \circ w$, F_m is a Minimum Spanning Forest w.r.t w.

Ensure: $F^* = F$

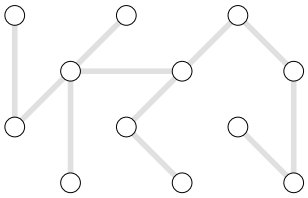
1: $F^* := F_m$
2: **for all** white edge $e \in E(F) \cup E(F_m)$ **do**
3: Let f be the heaviest black edge, w.r.t $h \circ w$, on the cycle induced by e in F^*
4: $F^* := F^* \setminus \{f\} \cup \{e\}$
5: **end for**

Osserviamo i fatti seguenti:

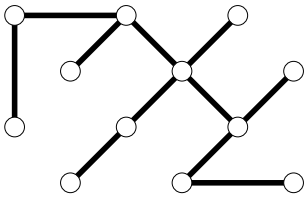
1. Nella linea 3 dell'algoritmo si deve selezionare l'arco nero più pesante sul ciclo C indotto dall'inserimento di e in F^* (un simile ciclo esiste perché durante tutta l'esecuzione dell'algoritmo F^* è una foresta ricoprente di G). Si osservi che C deve necessariamente contenere almeno un arco nero, in quanto se così non fosse, C sarebbe



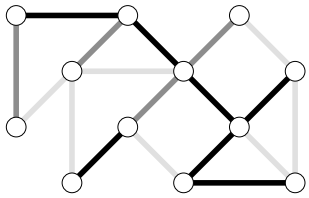
(a) G



(b) F



(c) F_m



(d) $F \cup F_m$

Figura 4.4: In 4.4(a) è rappresentato un grafo non orientato; in 4.4(b) è evidenziata una ipotetica minima foresta ricoprente F rispetto alla funzione peso $h \circ w$, mentre in 4.4(c) è evidenziata una minima foresta ricoprente F_m rispetto alla funzione peso w. In 4.4(d) è rappresentato il grafo $F \cup F_m$, e gli archi comuni ad F ed F_m sono rappresentati in grigio

composto interamente da archi bianchi o grigi, il che equivale a dire che tutti gli archi di C sono in $E(F)$, e questo è assurdo poiché F è una foresta e come tale aciclica.

Inoltre, l'arco nero f più pesante in C ha la proprietà che, per

ogni arco $x \in C$, $h(w(f)) \geq h(w(x))$. Se così non fosse, l'arco più pesante di C dovrebbe avere colore bianco o grigio, ma dato che tali archi sono in F, che è una minima foresta ricoprente di G rispetto alla funzione peso $h \circ w$, nessuno di essi può essere l'arco più pesante di un ciclo rispetto a $h \circ w$.

2. $w(e) - w(f) \leq \epsilon w(f)$, ove e, f sono gli archi considerati nella riga 4 dell'algoritmo. Infatti, essendo f l'arco più pesante di C rispetto a $h \circ w$, risulta

$$h(w(f)) \geq h(w(x)) \tag{4.1}$$

per ogni arco $x \in C$. D'altra parte, poiché f è un arco nero, non può essere l'arco più pesante di C rispetto a w. Dunque esiste un arco bianco $g \in C$, tale che

$$w(g) \geq w(y) \tag{4.2}$$

per ogni arco $y \in C$. Osserviamo che essendo la funzione h monotòna, risulta che $h(w(g)) \geq h(w(y))$, per ogni arco $y \in C$. Sostituendo g al posto di x nella (4.1), e sostituendo f al posto di y nella (4.2), si ha

$$h(w(g)) = h(w(f))$$

che implica immediatamente $w(g) - w(f) \leq \epsilon w(f)$, ed essendo $w(e) \leq w(g)$ (per la (4.2)), si conclude che

$$w(e) - w(f) \leq w(g) - w(f) \leq \epsilon w(f)$$

Quello che fa l'algoritmo è la costruzione implicita di una funzione biiettiva $\phi : E(F) \to E(F_m)$, definita come

$$\phi(e) = \begin{cases} e & \text{se } e \text{ è un arco grigio} \\ f & \text{se l'inserimento di } e \text{ in } F^* \text{ causa l'uscita di } f \end{cases}$$

La funzione ϕ ha la proprietà che $w(e) - w(\phi(e)) \leq \epsilon w(\phi(e))$. Quindi:

$$\begin{aligned} w(F) - w(F_m) &= \sum_{e \in E(F)} w(e) - \sum_{e \in E(F_m)} w(e) \\ &= \sum_{e \in E(F)} (w(e) - w(\phi(e)) \\ &\leq \epsilon \sum_{e \in E(F)} w(\phi(e)) = \epsilon w(F_m) \end{aligned}$$

che è la tesi del teorema. □

4.2.2 Analisi della complessità

In base al teorema 4.3, il problema del mantenimento di una $1+\epsilon$-minima foresta ricoprente può essere ricondotto a quello del mantenimento di una minima foresta ricoprente su un grafo con k pesi, per $k = 1+\log U/\log(1+\epsilon)$. In base al teorema 4.1, ogni operazione di inserimento o rimozione di archi può essere supportata in tempo ammortizzato $O((\log U/\log(1+\epsilon)+1)\log^2 n) = O(\log^2 n \log(U(1+\epsilon))/\log(1+\epsilon))$.

Si ottiene quindi il seguente

Teorema 4.4 *Sia $G = (V, E)$ un grafo non orientato con n vertici, con funzione peso $w : E \to [1, U]$, per un dato $U > 1$. Scelto un numero reale $\epsilon > 0$, esiste un algoritmo per mantenere una $1+\epsilon$-minima foresta ricoprente di G, in grado di supportare inserimenti e rimozioni di archi in tempo ammortizzato $O(\log^2 n \log(U(1+\epsilon))/\log(1+\epsilon))$ per operazione.*

4.3 Test di bipartitismo

Un grafo $G = (V, E)$ è *bipartito* se è possibile partizionare V in due sottoinsiemi disgiunti non vuoti V_1, V_2 tali che ogni arco di G abbia esattamente un estremo in V_1 e un estremo in V_2. In figura 4.5 è rappresentato un grafo bipartito.

Prima di procedere oltre col problema del bipartitismo, è utile introdurre alcune definizioni preliminari.

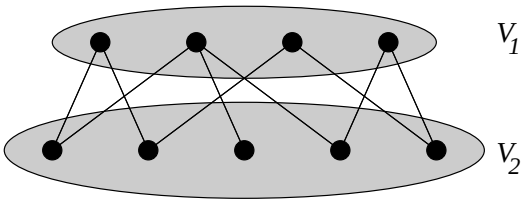
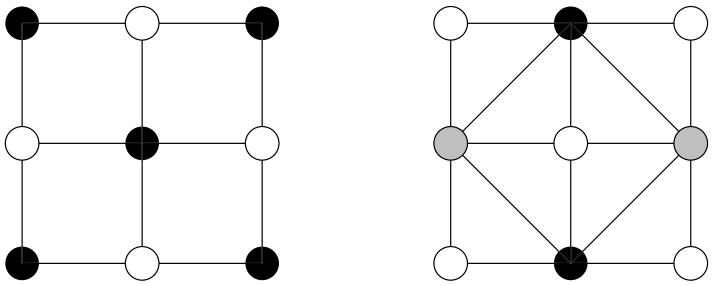


Figura 4.5: Esempio di grafo bipartito

Definizione 4.2 *Dato un grafo non orientato $G = (V, E)$ e un intero $k \geq 1$, una k-colorazione di G è una funzione $c : V \to \{1, 2, \ldots, k\}$ tale che $c(u) \neq c(v)$ se $\{u, v\} \in E$.*

In pratica, è come se si disponesse di k colori, e si volesse assegnare un colore a ciascun vertice in modo tale che vertici adiacenti non abbiano lo stesso colore. Diremo che un grafo G è k-colorabile, se esiste una k-colorazione per G, e per ogni $j < k$, non esiste alcuna j-colorazione di G. Il grafo in figura 4.6(a) è 2-colorabile, mentre quello in figura 4.6(b) è 3-colorabile.



(a) (b)

Figura 4.6: Un grafo 2-colorabile (a) e 3-colorabile (b)

Il problema della colorazione dei grafi è legato a quello del bipartitismo dal seguente

Teorema 4.5 *Sia $G = (V, E)$ un grafo non orientato. Le seguenti affermazioni sono equivalenti*

1. *G è bipartito;*
2. *G è 2-colorabile;*
3. *G non ha cicli di lunghezza dispari.*

Dimostrazione. $(1 \Leftrightarrow 2)$. Supponiamo che G sia bipartito; ciò significa che è possibile partizionare V in due insiemi non vuoti disgiunti V_1 e V_2 tali che ogni arco in E abbia esattamente un estremo in V_1 e un estremo in V_2. È immediato constatare che la funzione $c : V \to \{1, 2\}$ definita come

$$c(v) = \begin{cases} 1 & \text{se } v \in V_1 \\ 2 & \text{se } v \in V_2 \end{cases}$$

è una 2-colorazione del grafo. Viceversa, supponendo che esista una 2-colorazione c del grafo, definiamo

$$V_1 = \{v \in V \mid c(v) = 1\}$$
$$V_2 = \{v \in V \mid c(v) = 2\}$$

Gli insiemi V_1, V_2 formano una partizione di V e sono tali che ogni arco $e \in E$ abbia esattamente un estremo in V_1 e l'altro in V_2, quindi G è bipartito.

$(2 \Rightarrow 3)$. Supponiamo che G sia 2-colorabile, e supponiamo per assurdo che esista un ciclo di lunghezza dispari, $C = \langle v_0, v_1, \ldots, v_{2p}, v_0 \rangle$, per un certo $p \geq 1$ (vedi figura 4.7).

Possiamo supporre senza perdita di generalità che la 2-colorazione c sia tale che $c(v_0) = 1$. Risulta immediato constatare che, per ogni $i = 0, 1, \ldots, 2p$, deve essere

$$c(v_i) = \begin{cases} 1 & \text{se } i \text{ è pari} \\ 2 & \text{se } i \text{ è dispari} \end{cases}$$

Ma questo implica che $c(v_{2p}) = c(v_0)$, sebbene $\{v_{2p}, v_0\} \in E$, e questo è assurdo.

$(3 \Rightarrow 2)$. Supponiamo che G non abbia cicli di lunghezza dispari, e dimostriamo che G è 2-colorabile. Si osservi innanzitutto che ogni foresta

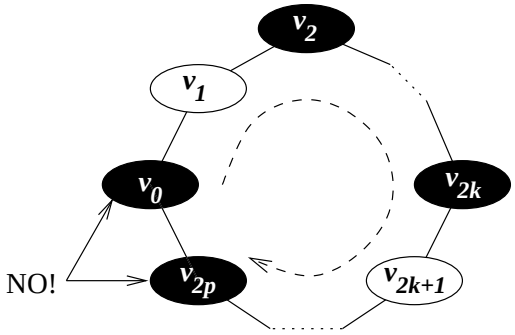


Figura 4.7: Un grafo contenente un ciclo di lunghezza dispari non può essere colorato con due colori

è 2-colorabile. Sia F una foresta ricoprente di G; sicuramente F è 2-colorabile mediante una certa 2-colorazione $c : V \to \{1, 2\}$; verifichiamo che c è una 2-colorazione anche per G. Infatti:

1. Se $\{x, y\} \in E(F)$, sicuramente $c(x) \neq c(y)$, perché F è 2-colorabile;

2. Se $\{x, y\} \in E(G) \setminus E(F)$, sicuramente esiste un cammino $x \overset{P}{\rightsquigarrow} y$ che collega x e y in F (vedi figura 4.8).

   Risulta che $P \cup \{x, y\}$ è un ciclo in G composto da un numero pari di archi, dato che si sta supponendo che in G non esistano cicli di lunghezza dispari. Ciò implica che il cammino P sia composto da un numero dispari di archi, da cui $c(x) \neq c(y)$. Dunque la colorazione c rimane valida anche con l'aggiunta dell'arco $\{x, y\}$. Poiché questo vale qualunque sia l'arco $\{x, y\} \in E(G) \setminus E(F)$, si ha la tesi.

□

4.3.1 L'algoritmo

Il problema del bipartitismo viene ricondotto al mantenimento di una minima foresta ricoprente con 2 pesi, sfruttando il fatto che un grafo è bipartito se e solo se non ha cicli di lunghezza dispari. Data una foresta

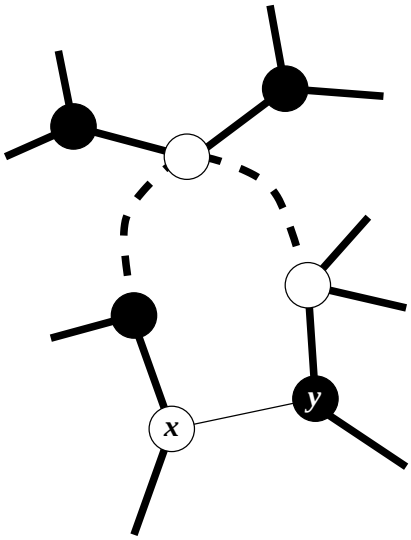


Figura 4.8: Un grafo senza cicli di lunghezza dispari è colorabile esattamente allo stesso modo in cui è possibile colorare una sua foresta ricoprente. Nella figura, gli archi della foresta ricoprente sono in grassetto; l'arco $\{x, y\}$ induce un ciclo di lunghezza dispari nella foresta F, quindi $c(x) \neq c(y)$.

ricoprente F di G, definiamo *archi pari* tutti gli archi di F e quelli che inducono su F un ciclo di lunghezza pari (cioè un ciclo con un numero pari di archi). I rimanenti archi di G vengono definiti *archi dispari*. Agli archi pari viene assegnato peso 0, mentre a quelli dispari viene assegnato peso 1 (vedi figura 4.9).

Dimostriamo il seguente

Lemma 4.1 *G è bipartito se e solo se non contiene archi dispari*

Dimostrazione. Dimostreremo che G è 2-colorabile se e solo se non contiene archi dispari; la tesi segue, poiché un grafo è 2-colorabile se e solo se è bipartito.

Sia c una 2-colorazione per il grafo G, e supponiamo per assurdo che esista un arco dispari $e = \{x, y\}$, ossia un arco che induce un ciclo di lunghezza dispari nella foresta ricoprente F. Consideriamo il cammino

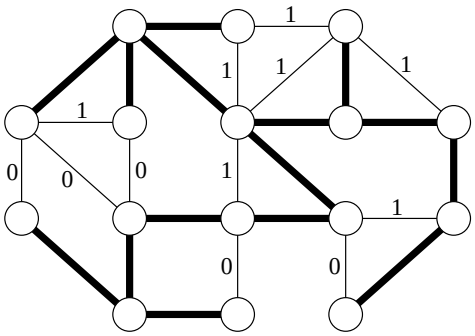


Figura 4.9: Esempio di assegnamento di pesi agli archi di un grafo per testare se è bipartito. Gli archi in grassetto appartengono ad una foresta ricoprente F, e ad essi si assegna peso 0. Agli archi rimanenti è assegnato peso 0 se il ciclo che inducono in F ha lunghezza pari, 1 altrimenti. Si osservi che il grafo non è bipartito, avendo archi dispari.

$x \overset{P}{\rightsquigarrow} y$ che collega x e y in F. Poiché per ipotesi G è 2-colorabile, tutti i nodi di P sono colorati con due colori. Per ipotesi, P è composto da un numero pari di archi, per cui $c(x) = c(y)$ sebbene $\{x, y\} \in E(G)$, e questo è assurdo.

Viceversa, supponiamo che G non contenga archi dispari e dimostriamo che esiste una 2-colorazione per G. La dimostrazione è identica a quella fatta nel teorema 4.5: data una qualsiasi foresta ricoprente F di G, osservando che F è 2-colorabile si dimostra immediatamente che anche G è 2-colorabile con la stessa colorazione di F. □

Il lemma suggerisce un algoritmo per testare se un grafo è bipartito, che consiste nel mantenimento di una foresta ricoprente del grafo insieme alle informazioni sulla parità degli archi. Vedremo ora come aggiornare efficientemente tali informazioni se il grafo è soggetto ad inserimenti e cancellazioni di archi.

Quando un arco $e \in E(F)$ viene cancellato, se esiste un rimpiazzo pari allora la parità di tutti gli archi di G rimane invariata. Supponiamo invece che e venga rimpiazzato da un arco dispari. In questo caso, detto C l'insieme degli archi che attraversano il taglio indotto dalla rimozione di e in F, la parità di tutti gli archi di C deve essere cambiata. Tuttavia,

se un arco pari viene rimpiazzato in F con uno dispari solo se necessario (cioè se non esistono rimpiazzi pari), la parità di un arco pari non viene mai cambiata. Questo può essere dimostrato formalmente nel seguente

Lemma 4.2 *Sia F una foresta ricoprente di G. Sia $e \in E(F)$ un arco pari, e sia C l'insieme degli archi che attraversano il taglio indotto dalla cancellazione di e in F. Se e è rimpiazzato in F da un arco dispari, allora cambia la parità di tutti e soli gli archi in C. Se e è rimpiazzato in F da un altro arco pari, la parità di tutti gli archi di G rimane immutata.*

Dimostrazione. Supponiamo che $e \in E(F)$ debba essere rimpiazzato da un arco dispari f. Consideriamo un altro arco g che attraversa il taglio indotto dalla cancellazione di e, e supponiamo che g sia pari. Dobbiamo dimostrare che la parità di g, dopo la rimozione di e, deve essere cambiata.

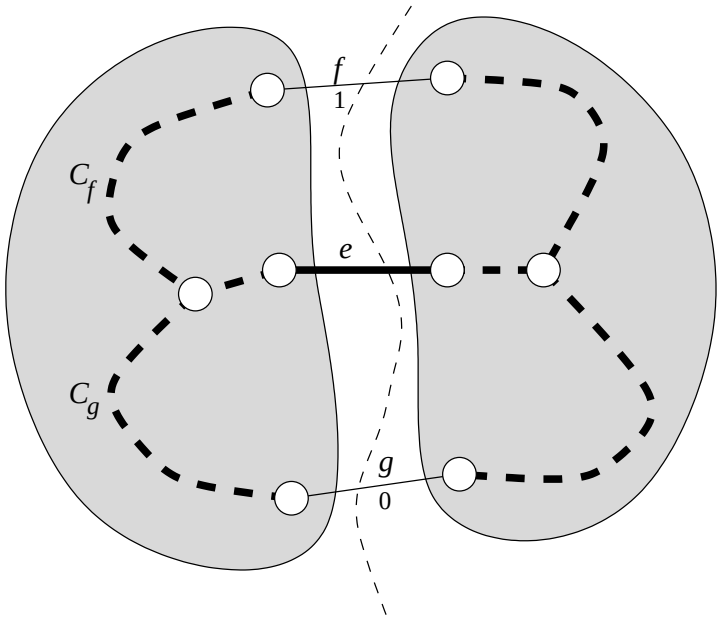


Figura 4.10: L'arco e deve essere rimpiazzato da un arco dispari f; si dimostra che la parità di ogni altro arco che attraversa il taglio indotto dalla rimozione di e deve essere cambiata.

Sia C_f il ciclo indotto su F da f, e sia C_g il ciclo indotto su F da g. Per ipotesi si ha che $|C_f|$ (il numero di archi in C_f) è dispari, mentre $|C_g|$ è pari. Supponiamo ora di rimuovere e da F, e di inserire al suo posto f; la nuova foresta ricoprente diventa $F' = F \setminus \{e\} \cup \{f\}$. Il ciclo indotto da g su F' è $C'_g = (C_f \cup C_g) \setminus (C_f \cap C_g)$. Siamo interessati a conoscere il numero di archi di cui è composto C'_g. Osserviamo che risulta:

$$\begin{aligned} |(C_f \cup C_g) \setminus (C_f \cap C_g)| &= |C_f \cup C_g| - |C_f \cap C_g| \\ &= |C_f| + |C_g| - |C_f \cap C_g| - |C_f \cap C_g| \\ &= |C_f| + |C_g| - 2|C_f \cap C_g| \end{aligned}$$

Poiché per ipotesi $|C_f|$ è dispari e $|C_g|$ è pari, $|C_f| + |C_g|$ è sicuramente dispari, da cui $|C_f| + |C_g| - 2|C_f \cap C_g|$ è dispari. Questo significa che la parità di g deve essere cambiata. Naturalmente lo stesso ragionamento vale supponendo che g sia pari.

Seguendo lo stesso procedimento si può dimostrare che se e viene rimpiazzato in F da un arco pari, la parità degli archi di G rimane invariata.

Infine, è immediato constatare che la parità di tutti gli archi di G che non attraversano il taglio C rimane in ogni caso immutata, dato che il ciclo da essi indotto nella foresta ricoprente non viene modificato dalla cancellazione di e e dall'inserimento dell'eventuale rimpiazzo. □

L'algoritmo per mantenere l'informazione riguardante il bipartitismo di un grafo G può quindi essere espresso nel modo seguente: inizialmente si calcola una foresta ricoprente F di G. A tutti gli archi di F, e ai rimanenti archi pari si assegna peso 0; agli archi dispari si assegna peso 1. Si noti che è possibile determinare la parità di un arco in tempo $O(\log n)$ memorizzando F mediante una foresta di Dynamic trees. Il grafo è bipartito se e solo se non ci sono archi dispari.

Bipartite-Insert(e) Se e collega due sottografi disgiunti di G, allora ad e viene assegnato peso 0 e viene inserito in F. Altrimenti, si determina la parità di e usando il Dynamic tree che rappresenta F, si assegna al nuovo arco il peso corretto, e si inserisce e nel grafo G. Si noti che in questo ultimo caso la foresta F non viene modificata, in quanto tutti i suoi archi hanno peso 0.

Bipartite-Delete(e) Se e non appartiene ad F, viene semplicemente rimosso. Altrimenti, si cerca un rimpiazzo di peso 0. Se tale rimpiazzo esiste, viene inserito in F e si termina; altrimenti, si cerca un rimpiazzo di peso 1, lo si cancella dal grafo e si cerca il rimpiazzo successivo, procedendo in questo modo fino a che l'ultimo rimpiazzo dispari è stato eliminato. Quindi, tutti gli archi cancellati vengono reinseriti nel grafo con peso 0.

Bipartite-Query Ritorna **vero** se e solo se il grafo è bipartito, cioè se e solo se non ci sono archi dispari.

Algoritmo 4.6 Bipartite-Insert(e)

Let $e = \{u, v\}$
Let T_v be the tree containing v
Let T_w be the tree containing w
if $T_v \neq T_w$ **then**
  $w(e) := 0$
else
  $w(e) :=$ "parity" of e
end if
k-weight-Insert$(e, w(e))$

Algoritmo 4.7 Bipartite-Delete(e)

Let $e = \{u, v\}$
$S := \emptyset$
$f :=$ **k-weight-Delete**(e)
while $f \neq$ **null** and $w(f) = 1$ **do**
  $S := S \cup \{f\}$
  $f :=$ **k-weight-Delete**(f)
end while
for all $f \in S$ **do**
  $w(f) := 0$
  k-weight-Insert(f)
end for

Oltre a queste operazioni, è possibile implementare anche la funzione seguente:

2-Color-Query(v_1, c_1, v_2, c_2) Ritorna **vero** se e solo se esiste una 2-colorazione c del grafo tale che $c(v_1) = c_1$ e $c(v_2) = c_2$. Chiaramente, se esiste qualche arco dispari, allora il grafo non è bipartito e quindi neanche 2-colorabile. Se invece il grafo è bipartito, sia F la foresta ricoprente come mantenuta dall'algoritmo. Si possono verificare due casi:

- *u e v non sono connessi in F.* In tal caso, essi possono essere colorati arbitrariamente, quindi il risultato è **vero**.
- *u e v sono connessi in F.* Si determina la lunghezza del cammino che collega u e v in F. Se il cammino è composto da un numero pari di archi, e $c_1 = c_2$, ritorna **vero**. Se il cammino è composto da un numero dispari di archi e $c_1 \neq c_2$, ritorna **vero**. In tutti gli altri casi, ritorna **falso**.

È utile ricordare che il problema generale di decidere se un grafo è k-colorabile, per $k > 2$, è NP-completo.

4.3.2 Correttezza dell'algoritmo

Il problema, posto in questo modo, equivale a mantenere una minima foresta ricoprente di G, ove gli archi possano avere solamente peso 0 oppure 1. Questo garantisce che in seguito alla cancellazione di un arco di peso 0 appartenente alla foresta ricoprente, si cerchi un rimpiazzo di peso 0, se esiste, in modo da non dover modificare inutilmente le parità di altri archi. Se non esiste alcun rimpiazzo pari, ma ne esiste uno dispari, la parità di tutti gli archi che attraversano il taglio indotto dall'arco cancellato deve essere modificata. Tali archi sono necessariamente dispari, e possono essere individuati cancellandoli uno alla volta e cercando ulteriori rimpiazzi.

4.3.3 Analisi della complessità

Un arco pari non può mai diventare dispari. Di conseguenza il peso di un arco può cambiare al massimo una sola volta, e un arco può essere reinserito nella struttura dati al massimo una volta con peso 1 e una volta con peso 0. Dato che il problema è riconducibile al mantenimento di una minima foresta ricoprente su un grafo con 2 pesi (2-weight Minimum Spanning Forest), le operazioni **Bipartite-Insert** e **Bipartite-Delete** possono essere completate in tempo ammortizzato $O(\log^2 n)$. Mantenendo un contatore con il numero di archi dispari presenti in ogni momento nel grafo, è possibile determinare se questo è bipartito in tempo $O(1)$; è infine possibile implementare la procedura **2-Color-Query** in tempo $O(\log n)$, in quanto l'unica operazione complessa che viene eseguita è la determinazione della lunghezza del cammino tra due nodi, operazione che può essere implementate in tempo $O(\log n)$ mediante la foresta di Dynamic trees che memorizza F.

Si ottiene il seguente:

Teorema 4.6 *Dato un grafo non orientato $G = (V, E)$ con n vertici, esiste un algoritmo deterministico in grado di stabilire se il grafo è bipartito in tempo $O(1)$ nel caso pessimo, supportando inserimenti e rimozioni di archi in tempo ammortizzato $O(\log^2 n)$ per operazione.*

4.4 k-connettività

Dato un grafo non orientato $G = (V, E)$, ricordiamo che due vertici distinti $u, v \in V$ sono *k-archi connessi* (*k-edge connected*), per $k \geq 2$, se sono connessi e la rimozione di $k-1$ archi qualsiasi lascia u e v connessi. La definizione viene estesa al grafo G, dicendo che G è k-archi connesso se ogni coppia di vertici distinti u, v è k-archi connessa.

Come si può constatare dalla tabella 3.1, il problema generale di decidere se un grafo è k-archi connesso per un dato k fissato è di non facile soluzione. L'algoritmo per la connettività descritto in 3.2 può essere direttamente impiegato per risolvere una versione semplificata del problema della connettività, cioè il problema del *testimone* alla $k+1$-archi

	Autore	*Update time*
$k = 2$	Holm ed altri [26]	$O(\log^4 n)$
$k = 3$	Eppstein ed altri [11]	$O(n^{2/3})$
$k = 4$	Eppstein ed altri [11]	$O(n\alpha(n))$
$k \geq 2$	Eppstein ed altri [11]	$O(n \log n)$

Tabella 4.1: Risultati relativi al problema della k-archi connettività

connettività: stabilire se due nodi qualsiasi u e v di un grafo $G = (V, E)$ rimangono connessi in seguito alla rimozione degli archi $e_1, e_2, \ldots, e_k$, supportando nel contempo la possibilità di modificare G con inserimenti e rimozioni di archi. Per fare questo è sufficiente

- cancellare uno alla volta gli archi, in tempo totale $O(k \log^2 n)$;
- testare se u e v sono connessi in tempo $O(\log n / \log \log n)$;
- reinserire gli archi cancellati nel grafo, in tempo totale $O(k \log^2 n)$.

Ne deriva il seguente:

Teorema 4.7 *Dato un grafo non orientato $G = (V, E)$ con n vertici, esiste un algoritmo deterministico in grado di stabilire se la rimozione di k archi dati $e_1, e_2, \ldots, e_k$ lascia connessi i vertici u e v in tempo ammortizzato $O(k \log^2 n)$, supportando inserimenti e rimozioni di archi in tempo ammortizzato $O(\log^2 n)$ per operazione.*

Questo risultato, già interessante di per sé, trova una importante applicazione nella possibilità di testare efficientemente se due archi arbitrari di un grafo dinamico G sono equivalenti rispetto ai cicli, come vedremo in dettaglio nel paragrafo successivo.

4.4.1 Equivalenza rispetto ai cicli

Dato un grafo non orientato $G = (V, E)$, due archi $e_1, e_2 \in E$ sono *equivalenti rispetto ai cicli* se ogni ciclo contenente e_1 contiene anche e_2 e viceversa.

La proprietà di equivalenza rispetto ai cicli è legata alla k-archi connettività dal seguente

Lemma 4.3 *Due archi $e_1 = \{x, y\}, e_2 = \{u, v\}$ sono equivalenti rispetto ai cicli se e solo se x, y sono sconnessi in $G \setminus \{e_1, e_2\}$ e u, v sono sconnessi in $G \setminus \{e_1, e_2\}$.*

Dimostrazione. ($\Rightarrow$) Supponiamo che e_1, e_2 siano equivalenti rispetto ai cicli. Supponiamo per assurdo che x, y siano connessi in $G' = G \setminus \{e_1, e_2\}$. Ciò significa che esiste un cammino $P_{\{x,y\}}$ in G' che collega i vertici x, y, e tale che e_1, e_2 non compaiano in $P_{\{x,y\}}$ (vedi figura 4.11). Allora, $P_{\{x,y\}} \cup \{x, y\}$ è un ciclo che contiene $e_1 = \{x, y\}$ ma non e_2, e ciò contraddice l'ipotesi che e_1, e_2 fossero equivalenti rispetto ai cicli. Lo stesso ragionamento vale considerando per assurdo un percorso $P_{\{u,v\}}$ in G' che colleghi u, v, e ciò conclude la prima parte della dimostrazione.

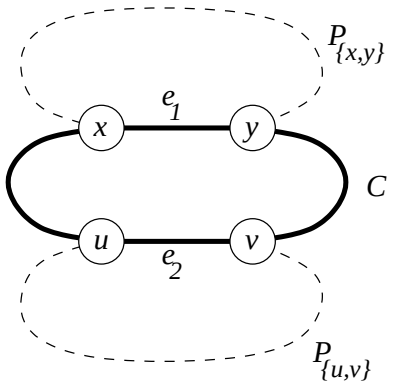


Figura 4.11: La figura evidenzia che se x, y sono connessi in $G \setminus \{e_1, e_2\}$, allora e_1, e_2 non sono equivalenti rispetto ai cicli

($\Leftarrow$) Supponiamo ora che x, y e u, v siano sconnessi in $G' = G \setminus \{e_1, e_2\}$. Supponiamo per assurdo che e_1 ed e_2 non siano equivalenti rispetto ai

cicli; dunque, esiste un ciclo C che contiene ad esempio e_1 ma non e_2. Ciò significa che $C \setminus \{e_2\} = C$, e quindi $C \setminus \{e_1, e_2\}$ è un cammino che collega x e y, contro l'assunto che x e y fossero sconnessi in $G \setminus \{e_1, e_2\}$. Analogo discorso vale per u, v, e la dimostrazione si conclude. □

È importante osservare che per verificare se $e_1 = \{x, y\}$ ed $e_2 = \{u, v\}$ sono equivalenti rispetto ai cicli **non** è sufficiente verificare solo se x e y sono sconnessi in $G \setminus \{e_1, e_2\}$: in figura 4.12 è riportato un esempio.

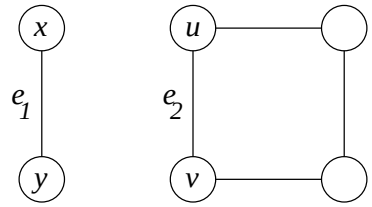


Figura 4.12: Gli archi $e_1 = \{x, y\}$ ed $e_2 = \{u, v\}$ non sono equivalenti rispetto ai cicli, sebbene x e y non siano connessi in $G \setminus \{e_1, e_2\}$. Questo dimostra che è indispensabile controllare anche se u, v sono sconnessi in $G \setminus \{e_1, e_2\}$.

Dal lemma deriva immediatamente un algoritmo per testare se, dato un grafo non orientato $G = (V, E)$, due archi $e_1, e_2 \in E$ sono equivalenti rispetto ai cicli. È infatti sufficiente rimuovere gli archi $e_1 = \{x, y\}, e_2 = \{u, v\}$ dal grafo; se a seguito di ciò x e y sono connessi, oppure u, v sono connessi, si conclude che e_1, e_2 *non* sono equivalenti rispetto ai cicli; altrimenti, e_1, e_2 sono equivalenti rispetto ai cicli. Effettuato il test, gli archi possono essere reinseriti nel grafo. Tutto ciò si può portare a termine in tempo totale ammortizzato $O(\log^2 n)$, da cui si ha immediatamente il seguente

Teorema 4.8 *Dato un grafo non orientato $G = (V, E)$ con n vertici, esiste un algoritmo deterministico per verificare se due archi $e_1, e_2 \in E$ sono equivalenti rispetto ai cicli in tempo ammortizzato $O(\log^2 n)$, supportando inserimenti e rimozioni di archi in tempo ammortizzato $O(\log^2 n)$ per operazione.*

4.5 Decomposizione Ricoprente Massimale

Si consideri un grafo non orientato $G = (V, E)$. Una *decomposizione ricoprente massimale* di ordine k di G, per $k \geq 1$, è una sequenza di k foreste $F_1, F_2, \ldots, F_k$ tali che per $i \neq j$, $E(F_i) \cap E(F_j) = \emptyset$ (cioè, gli archi di una foresta non possono appartenere anche ad un'altra), e tali che F_i sia una foresta ricoprente del grafo $G_i = G \setminus \cup_{j<i} F_j$. Quindi, F_1 è una foresta ricoprente di G, F_2 è una foresta ricoprente di $G \setminus F_1$, F_3 è una foresta ricoprente di $G \setminus (F_1 \cup F_2)$, e così via. La figura 4.13 illustra la costruzione della decomposizione ricoprente massimale di ordine 2 di un grafo.

La decomposizione ricoprente massimale è utile perché il grafo $G^* = \cup_{i=1}^{k} F_i$ ha $O(kn)$ archi, ed ha esattamente le stesse componenti k-archi connesse del grafo G [39]; ciò significa che G^* costituisce un *certificato* per la proprietà di k-archi connettività [11].

Viene presentto un algoritmo in grado di mantenere esplicitamente la sequenza di grafi G_i e di foreste F_i, $i = 1, 2, \ldots, k$, con riferimento ad un grafo $G = (V, E)$ soggetto alle seguenti operazioni:

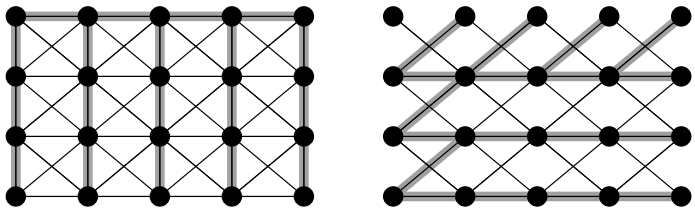
MFD-Insert(e) Inserisce l'arco e in G. I grafi G_i, e le relative foreste ricoprenti, vengono eventualmente aggiornati.

MFD-Delete(e) Cancella l'arco e da G e da tutte le strutture in cui compare. Anche in questo caso, i grafi G_i e le relative foreste ricoprenti vengono eventualmente aggiornate.

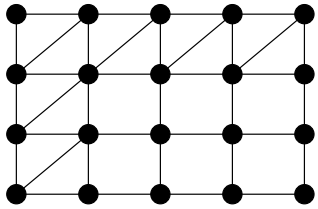
4.5.1 L'algoritmo

La struttura dati presentata in 3.2 è in grado di mantenere una foresta ricoprente di un grafo G, soggetto a inserimenti e rimozioni di archi, in tempo ammortizzato $O(\log^2 n)$ per operazione.

Per mantenere una decomposizione massimale di ordine k, con k fissato inizialmente, è sufficiente memorizzare ogni grafo G_i, $i = 1, 2, \ldots, k$ in una struttura dati $\mathcal{A}_i$ per la connettività; ciò garantisce che per ciascun grafo venga mantenuta automaticamente una foresta ricoprente locale F_i.



(a) G_1 (b) G_2



(c) $F_1 \cup F_2$

Figura 4.13: Costruzione della decomposizione ricoprente massimale di ordine 2 di un grafo. In 4.13(a) è rappresentato il grafo $G_1 = G$ con una foresta ricoprente F_1 (archi evidenziati); in 4.13(b) è rappresentato il grafo $G_2 = G \setminus F_1$ con una foresta ricoprente F_2. Infine, il grafo in 4.13(c) è ottenuto dall'unione delle foreste $F_1 \cup F_2$. Si tratta di un grafo con $O(2n) = O(n)$ vertici, dove $n = |V(G)|$, e con le stesse componenti 2-archi connesse di G.

Il problema consiste nel garantire che in ogni istante sia $G_i = G \setminus \cup_{j<i} F_j$, per $i = 1, 2, \ldots, k$.

Le procedure seguenti modificano la sequenza di grafi in maniera corretta, implementando le procedure **MFD-Insert** e **MFD-Delete**.

Algoritmo 4.8 MFD-Insert(e)

```
i := 1
while i ≤ k do
  Insert e in G_i, updating F_i
  if e ∈ E(F_i) then
    Stop
  else
    i := i + 1
  end if
end while
```

Algoritmo 4.9 MFD-Delete(e)

```
i := 1
while i ≤ k and e ≠ null do
  if e ∈ E(F_i) then
    Remove e from G_i, updating F_i; let f be the replacement edge (or
    null)
    e := f
  else
    Remove e from G_i, updating F_i
  end if
  i := i + 1
end while
```

4.5.2 Correttezza dell'algoritmo

L'algoritmo per la connettività garantisce che per ogni $1 \leq i \leq k$, F_i sia una foresta ricoprente del grafo G_i. È quindi sufficiente dimostrare che le operazioni di inserimento e cancellazione di archi preservano la proprietà $G_i = G \setminus \cup_{j<i} F_j$.

Quando un arco e viene inserito, viene inserito in tutti i grafi, a partire da G_1; per ogni grafo, si determina la nuova foresta ricoprente. Se questa non contiene e, esso viene inserito nel grafo successivo. Se invece

e appartiene alla foresta ricoprente di G_i, per qualche i, sicuramente non dovrà comparire in nessuno dei grafi G_j, $j > i$, quindi l'algoritmo termina.

Quando un arco e viene cancellato, viene rimosso da tutti i grafi in cui compare, a partire da G_1; per ogni i, se $e \in E(F_i)$ sicuramente e non comparirà in nessuno dei grafi successivi. Se la rimozione di e lascia F_i sconnessa, il procedimento si interrompe; se invece l'arco e viene sostituito da f in F_i, f deve essere cancellato dai grafi $G_j, j > i$.

4.5.3 Analisi della complessità

È facile constatare che le procedure **MFD-Insert** e **MFD-Delete** effettuano $O(1)$ modifiche su ciascuna struttura dati $\mathcal{A}_i$, ciascuna operazione avente costo ammortizzato $O(\log^2 n)$. Si ha pertanto il seguente

Teorema 4.9 *Dato un grafo non orientato $G = (V, E)$ con n vertici, e un intero $k \geq 1$ fissato inizialmente, esiste un algoritmo deterministico in grado di mantenere una sequenza di foreste $F_1, F_2, \ldots, F_k$ tali che F_i sia una foresta ricoprente del grafo $G_i = G \setminus \cup_{j<i} F_j$, supportando inserimenti e rimozioni di archi in G in tempo ammortizzato $O(k \log^2 n)$ per operazione.*

Capitolo 5

Conclusioni

5.1 Risultati conseguiti

In questa tesi sono stati presentati algoritmi deterministici per il mantenimento delle seguenti proprietà su grafi dinamici $G = (V, E)$ con n vertici:

- Mantenere una minima foresta ricoprente di G, i cui archi possano assumere al più k pesi distinti per un $k \geq 1$ fissato, supportando inserimenti e rimozioni di archi in tempo ammortizzato $O(k \log^2 n)$ per operazione;
- Mantenere una $1+\epsilon$-minima foresta ricoprente di G con funzione peso $w : E \to [1, U]$ in tempo ammortizzato $O(\log^2 n \log(U(1 + \epsilon))/\log(1 + \epsilon))$ per operazione;
- Determinare se G è bipartito in tempo $O(1)$ nel caso pessimo, supportando modifiche in tempo ammortizzato $O(\log^2 n)$ per operazione;
- Determinare se la rimozione di k archi dati rende i vertici u e v sconnessi in tempo ammortizzato $O(k \log^2 n)$, supportando aggiornamenti in tempo ammortizzato $O(\log^2 n)$ per operazione;

- Mantenere una decomposizione ricoprente massimale di ordine k di G in tempo ammortizzato $O(k \log^2 n)$ per aggiornamento.

Tutti questi algoritmi rappresentano risultati originali ottenuti combinando in modo nuovo tecniche algoritmiche proposte di recente da Holm ed altri [25] con alcune idee di Henzinger e King [21]. In tal modo si sono potuti ottenere i primi algoritmi *deterministici* aventi complessità polilogaritmica in grado di mantenere le proprietà sopra elencate su grafi dinamici, eguagliando le prestazioni delle migliori soluzioni randomizzate esistenti; lo scopo della tesi può dunque considerarsi raggiunto.

5.2 Sviluppi futuri

Vi sono tuttavia diversi spunti per ulteriori approfondimenti. Innanzitutto, sarebbe molto utile realizzare una implementazione pratica dell'algoritmo deterministico per la connettività dinamica di Holm ed altri [25] e di quello randomizzato di Henzinger e King nella versione migliorata di Henzinger e Thorup [21, 24], allo scopo di condurre analisi sperimentali sulle prestazioni effettive (risultati sperimentali relativi all'algoritmo randomizzato per la connettività nella versione originale sono descritti in [3]). Disponendo di una implementazione pratica dell'algoritmo deterministico per la connettività, le seguenti domande potrebbero trovare una risposte: è possibile semplificare l'implementazione per ottenere delle prestazioni migliori in pratica? Inoltre, è ragionevole aspettarsi che l'algoritmo deterministico si comporti *sempre* meglio in pratica rispetto a quello randomizzato, come suggerisce la teoria, oppure questo avviene solo in casi eccezionali?

Dal punto di vista teorico, sarebbe interessante verificare la possibilità di rendere l'algoritmo di Holm ed altri di tipo "vertex-dynamic", consentendo cioè anche la rimozione e l'inserimento di nuovi vertici isolati nel grafo *senza* peggiorarne la complessità. Un ulteriore spunto potrebbe riguardare la specializzazione di tale algoritmo per operare su grafi piani o planari; attualmente, l'algoritmo deterministico di Eppstein ed altri [12] è in grado di mantenere una minima foresta ricoprente (e quindi le componenti connesse) di un grafo piano in tempo $O(\log n)$ per operazione. È

possibile trarre vantaggio dalla topologia dei grafi piani per migliorare di un fattore (almeno) $\log n$ la complessità dell'algoritmo di Holm ed altri?

Infine, è possibile estendere l'approccio presentato in questa tesi per il mantenimento di altre proprietà su grafi dinamici?

Ringraziamenti

Il primo ringraziamento va ai miei familiari per il sostegno e l'incoraggiamento costante durante questi lunghi anni di studio.

Un sentito ringraziamento va al prof. Giuseppe F. Italiano per la disponibilità dimostrata nei miei confronti e per i suggerimenti forniti durante la stesura della tesi, senza i quali questo lavoro non sarebbe mai stato completato. Ringrazio la dott.ssa Stefania Serafin per avermi suggerito il prof. Italiano come relatore; senza il suo consiglio non avrei forse mai avuto occasione di approfondire lo studio degli algoritmi su grafi dinamici, argomento di crescente interesse sia teorico che pratico.

Ringrazio il prof. Antonino Salibra per avere accettato il ruolo di controrelatore, dedicando parte del suo tempo alla lettura di questa tesi.

Un ringraziamento va anche ai colleghi Davide Buzzi e Massimo Brollo, per le stimolanti conversazioni e lo scambio di chiarimenti durante la lunga fase preliminare di raccolta e studio del materiale bibliografico, e a tutti coloro che mi sono stati vicino contribuendo, direttamente e non, alla riuscita di questo lavoro.

Appendice A

Strutture dati avanzate: ET-trees e Dynamic trees

A.1 ET-tree

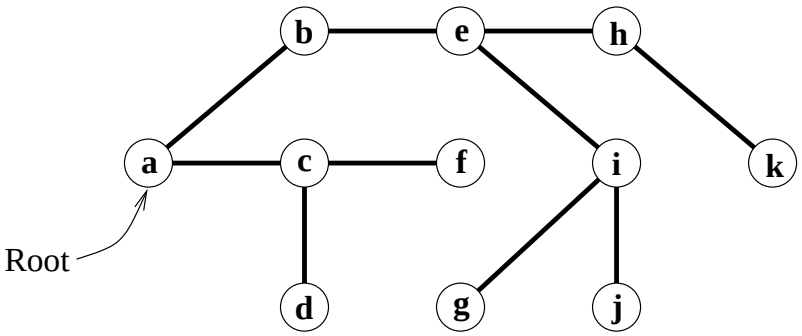
Negli algoritmi presentati nel capitolo 3 è necessario disporre di una struttura dati in grado di rappresentare gli alberi di una foresta ricoprente F di un grafo G, memorizzando gli archi dell'insieme $E(G) \setminus E(F)$ incidenti a ciascun albero e supportando la possibilità di suddividere ogni albero cancellando un arco (*split*), o di unire due alberi distinti collegandoli con un nuovo arco (*join*). Queste, ed altre, operazioni possono essere realizzate con la struttura dati *Euler Tour*, introdotta nel 1985 da Tarjan e Vishkin [45].

Sia T un albero con n vertici. È possibile codificare T in una sequenza di $2n-1$ simboli generata nel modo seguente: se v è un qualsiasi vertice di T, si pone v come radice di T e si esegue la procedura $\mathbf{ET}(v)$. Tale procedura $\mathbf{ET}(v)$ effettua una visita in profondità dell'albero T a partire dalla foglia v; la sequenza di simboli stampati, detta *ET-sequence*, rappresenta in modo univoco l'albero T, e verrà denotata come $ET(T)$. Si noti che la ET-sequence di un albero T non è necessariamente unica, in quanto dipende dal vertice scelto come radice dell'albero e dall'ordine in cui la visita stessa viene effettuata.

Ogni arco di T è attraversato esattamente due volte, ed ogni vertice di

Algoritmo A.1 ET(v)

```
print v
for all child c of v do
  ET(c)
  print v
end for
```



(a)

a	b	e	h	k	h	e	i	j	i	g	i	e	b	a	c	f	c	d	c	a

(b)

Figura A.1: In A.1(a) è rappresentato un albero libero; in A.1(b) è rappresentata una ET-sequence che rappresenta l'albero, considerando il vertice a come radice.

T di grado d è presente in $ET(T)$ d volte, tranne la radice che è presente $d + 1$ volte. Ogni occorrenza del vertice u in $ET(T)$ verrà denotata con $o(u)$; date due occorrenze $o(u), o(v)$, diremo che $o(u)$ *precede* $o(v)$ in $ET(T)$ se $ET(T) = [\alpha o(u)\beta o(v)\gamma]$, e scriveremo in tal caso $o(u) < o(v)$.

Vediamo ora come cambiano le ET-sequence a seguito della cancellazione o dell'inserimento di un arco negli alberi rappresentati:

Per cancellare un arco $\{a, b\}$ da T Siano T_a e T_b gli alberi che risultano dalla cancellazione di $\{a, b\}$, ove $a \in V(T_a)$ e $b \in V(T_b)$. Siano $o(a_1), o(a_2), o(b_1), o(b_2)$ le occorrenze di a e b incontrate nei due attraversamenti dell'arco $\{a, b\}$ in $ET(T)$, ove $o(a_1) < o(a_2)$, e $o(b_1) < o(b_2)$. Possiamo supporre senza perdita di generalità che $o(a_1) < o(b_1)$ (se così non fosse i ruoli di a e b possono essere invertiti), quindi $ET(T) = [\alpha o(a_1) o(b_1) \beta o(b_2) o(a_2) \gamma]$. Allora risulta $ET(T_a) = [\alpha o(a_1) \gamma]$ e $ET(T_b) = [o(b_1) \beta o(b_2)]$ (vedi figura A.2).

Per cambiare la radice di T da r a s Sia $o(s)$ una qualsiasi occorrenza di s in $ET(T)$. Dunque $ET(T) = [o(r) \alpha o(s) \beta o(r)]$. Allora $ET(T') = [o(s) \beta o(r) \alpha o(s)]$ (vedi figura A.3).

Per unire due alberi T_1 e T_2 mediante l'arco e Sia $e = \{a, b\}$, con $a \in V(T_1)$ e $b \in V(T_2)$. Si cambia innanzitutto la radice di T_b, facendola diventare b. Se $ET(T_a) = [\alpha o(a) \beta]$ e $ET(T_b) = [\gamma]$, allora $ET(T_a \cup \{a, b\} \cup T_b) = [\alpha o(a) \gamma o(a) \beta]$ (vedi figura A.2).

Se la sequenza $ET(T)$ è memorizzata in un B-tree di grado b e altezza $O(\log n / \log b)$, allora è possibile inserire un intervallo, o eliminare un intervallo in tempo $O(b \log n / \log b)$ mantenendo il bilanciamento dell'albero, e determinare se due occorrenze di due nodi sono nello stesso albero, o se una occorrenza precede un'altra nella sequenza $ET(T)$ in tempo $O(\log n / \log b)$; si veda ad esempio [9] per una descrizione dettagliata delle operazioni sui B-trees. La radice del B-tree contiene un identificatore univoco dell'albero, e ciascun nodo terminale (foglia) del B-tree contiene puntatori al nodo che lo segue e a quello che lo precede nell'ordinamento della sequenza $ET(T)$. Chiameremo *ET-trees* i B-trees che memorizzano le ET-sequence. D'ora in avanti, col termine *nodi* faremo riferimento ai vertici dell'ET-tree, e col termine *vertici* indicheremo i vertici dell'albero che la ET-sequence rappresenta.

La struttura dati ET-tree può essere facilmente estesa per memorizzare gli archi non appartenenti alla foresta ricoprente che sono incidenti ad un albero T. Infatti, per ogni vertice $v \in V(T)$, una delle occorrenze di v in $ET(T)$ è designata come *occorrenza attiva*. Insieme all'occorrenza

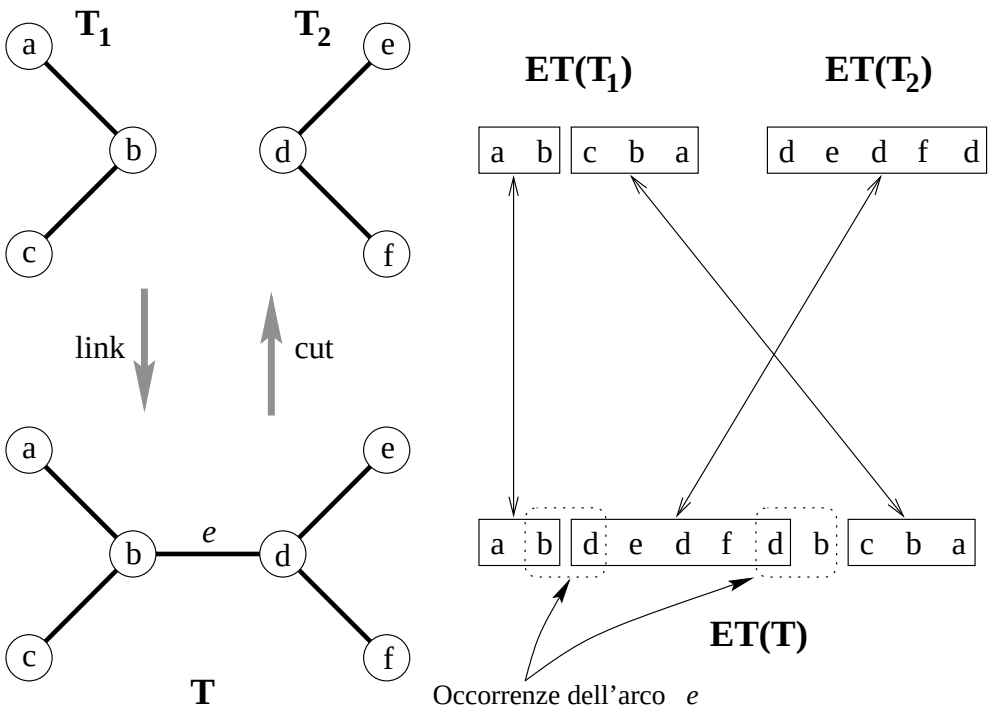


Figura A.2: La figura illustra l'effetto dell'inserimento di un nuovo arco e tra gli alberi T_1 e T_2. Per convenienza, la radice di T_2 è già posta al nodo d, altrimenti come prima operazione occorre reimpostarne la radice. La figura illustra anche l'effetto della cancellazione dell'arco e in T, se vista dal basso verso l'alto.

attiva di v in $ET(T)$ viene memorizzata la lista di tutti gli archi incidenti a v che non appartengono alla foresta ricoprente. Nei nodi interni dell'ET-tree viene mantenuto un bit che indica se una delle occorrenze del sottoalbero che fa capo a quel nodo è attiva, e un bit che indica se una delle liste di archi memorizzate in una occorrenza attiva del sottoalbero è non vuota.

La struttura dati ET-tree consente di implementare le seguenti operazioni:

tree(vertex x**)** Ritorna l'identificatore dell'ET-tree contenente il ver-

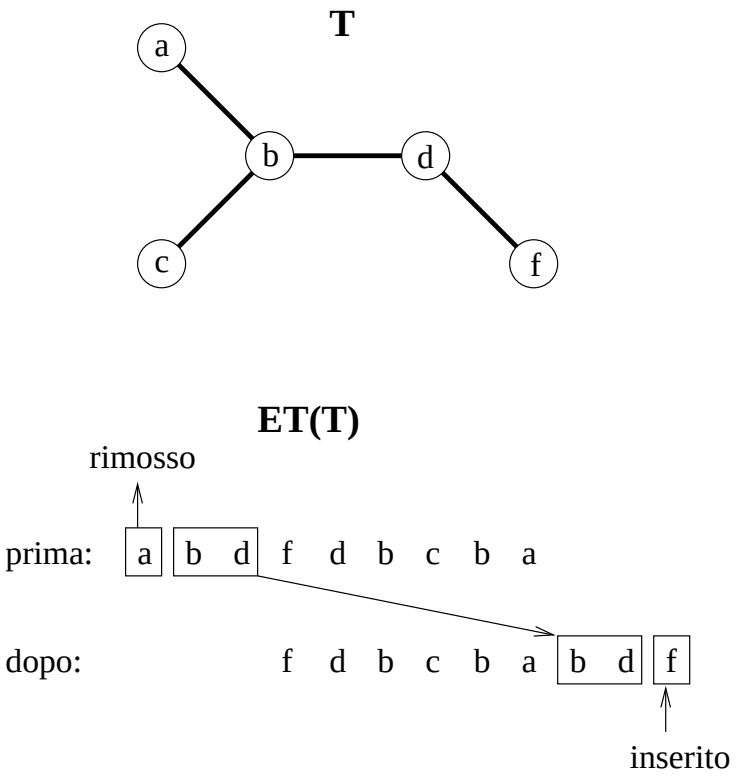


Figura A.3: La figura illustra l'effetto del cambiamento della radice di T da a a f.

tice x. È sufficiente risalire dall'occorrenza attiva di x fino alla radice dell'ET-tree che la contiene, restituendo l'identificatore ad essa associato.

insert-tree(edge e**)** Se $e = \{u, v\}$, unisce gli alberi T_u e T_v contenenti rispettivamente u e v, mediante il nuovo arco e. Per effettuare l'operazione, si determinano le occorrenze attive degli estremi di e in T_u e T_v e si segue la procedura indicata precedentemente per unire due ET-tree.

delete-tree(edge e**)** Se $e = \{u, v\}$, si determinano le occorrenze di u e v associate all'attraversamento di e nell'albero che lo contiene,

quindi si segue la procedura descritta sopra per suddividere un ET-tree cancellando un arco. Dopo la suddivisione, se necessario, si aggiornano le occorrenze attive dei vertici u e v negli alberi risultanti.

nontree-edge(tree T) Ritorna un arco incidente all'albero T che non appartenga già alla foresta ricoprente. Partendo dalla radice, si segue un qualsiasi percorso composto da nodi dell'ET-tree che risultino marcati per indicare che una delle liste di archi contenute nel sottoalbero è non vuota. Raggiunta l'occorrenza attiva di un vertice di T, viene restituito il primo arco della lista associata; tale arco viene anche rimosso dalle liste in cui compare.

insert-nontree(edge e) Inserisce un nuovo arco, non appartenente alla foresta ricoprente. Dopo aver individuato le occorrenze attive degli estremi di e, quest'ultimo viene aggiunto alla lista degli archi incidenti ad esse.

active-occurrences(tree T) Elenca tutti i nodi corrispondenti ad occorrenze attive memorizzate nell'albero T; è sufficiente individuare la foglia più a sinistra di T, e seguire i puntatori alle foglie successive, restituendo via via le occorrenze marcate come attive.

Lemma A.1 *Sia G un grafo non orientato con n vertici ed m archi. Rappresentanto una foresta ricoprente di G mediante ET-trees memorizzati mediante alberi binari di ricerca bilanciati, l'operazione* ***active-occurrences*** *può essere completata in tempo $O(\log n + n) = O(n)$ nel caso pessimo; le rimanenti operazioni possono essere completate in tempo $O(\log n)$. Lo spazio richiesto dalla struttura dati è $O(m + n)$.*

A.2 Dynamic tree

La struttura dati *Dynamic Tree*, introdotta da Sleator e Tarjan [43], è in grado di risolvere il seguente problema: data una foresta composta da

alberi con radice, si vogliono mantenere in maniera efficiente determinate proprietà su di essi, consentendo nel contempo di modificare la foresta mediante le operazioni:

link(v, w) Se v è la radice di un albero, e w è un vertice di un albero disgiunto, si collegano i due alberi contenenti v e w con l'aggiunta del nuovo arco (v, w).

cut(v) Se il vertice v non è una radice, l'albero contenente v viene suddiviso in due parti cancellando l'arco che collega v con il proprio padre.

evert(v) Rende il vertice v la nuova radice dell'albero che lo contiene.

In particolare, con la struttura proposta in [43] viene risolto il problema di mantenere una foresta di alberi dotati di radice, i cui archi hanno associato un numero reale detto *costo*, sotto una arbitraria sequenza delle seguenti operazioni:

parent(vertex v**)** Ritorna il padre del vertice v. Se v è una radice, ritorna il valore speciale **null**.

root(vertex v**)** Ritorna la radice dell'albero contenente v.

cost(vertex v**)** Ritorna il costo dell'arco $(v, \textbf{parent}(v))$. Si assume che v non sia un vertice radice.

mincost(vertex v**)** Ritorna il vertice w più vicino alla radice $\textbf{root}(v)$ tale che l'arco $(w, \textbf{parent}(w))$ abbia costo minimo tra tutti gli archi del cammino $v \rightsquigarrow \textbf{root}(v)$. Si assume che v non sia una radice.

update(vertex v**,real** x**)** Modifica il costo di ogni arco sul cammino $v \rightsquigarrow \textbf{root}(v)$ aggiungendo la quantità x al costo di ciascun arco.

link(vertex v, w**,real** x**)** Combina gli alberi contenenti v e w aggiungendo un nuovo arco (v, w) di costo x, rendendo w il padre di v. L'operazione assume che v e w siano in alberi differenti, e che v sia un nodo radice.

cut(vertex v**)** Suddivide l'albero contenente il vertice v in due alberi, mediante la cancellazione dell'arco $(v, \textbf{parent}(v))$; viene restituito il costo dell'arco cancellato. L'operazione assume che v non sia una radice.

evert(vertex v**)** Modifica l'albero contenente il vertice dv, rendendo v la radice (questa operazione può essere vista come l'inversione della direzione di ogni arco sul percorso da v alla radice originale).

Le operazioni **parent, root, cost, mincost** estraggono informazioni dalla foresta senza modificarla. L'operazione **update** altera il costo degli archi ma non modifica la struttura degli alberi, mentre le rimanenti operazioni **link, cut, evert** alterano la struttura della foresta. La figura A.4 illustra l'effetto dell'esecuzione delle varie operazioni su una foresta di Dynamic trees.

Teorema A.1 (Sleator e Tarjan [43]) *Ciascuna delle operazioni precedenti può essere supportata in tempo* $O(\log n)$ *nel caso pessimo, essendo* n *il numero di vertici nella foresta di Dynamic trees. Lo spazio richiesto è* $O(n)$*.*

L'idea di base consiste nel partizionare ciascun albero in cammini disgiunti, che vengono memorizzati in alberi binari di ricerca bilanciati; per i dettagli si veda [43].

L'operazione **mincost** può essere opportunamente ridefinita per mentenere altre proprietà sui cammini degli alberi. In particolare, è possibile determinare la lunghezza (numero di archi) del cammino da v a $\textbf{root}(v)$, oppure determinare l'arco più pesante (anziché più leggero) di un cammino da v a $\textbf{root}(v)$, sempre in tempo $O(\log n)$ nel caso pessimo. È quindi possibile risolvere diversi problemi su grafi dinamici utilizzando versioni modificate dei Dynamic trees [10]. Vale ad esempio il seguente

Lemma A.2 *Sia* $G = (V, E)$ *un grafo non orientato, con* $|V| = n$*, dotato di una funzione peso* $w : E \to \mathbb{R}_{\geq 0}$*. Esiste un algoritmo deterministico in grado di mantenere una foresta ricoprente di peso minimo* F *del grafo* G*, supportando inserimenti di archi in tempo* $O(\log n)$ *per operazione.*

Dimostrazione. Dal paragrafo 2.5.2 è noto che, data una minima foresta ricoprente F del grafo $G = (V, E)$, per determinarne una del grafo $G' = (V, E \cup \{e\})$ è necessario conoscere l'arco più pesante sul ciclo indotto da e in F. Se la foresta F viene rappresentata come una foresta di Dynamic trees, il problema viene risolto implementando l'operazione seguente:

maxcost(vertex v) Restituisce il vertice w più vicino alla radice **root**(v) tale che l'arco $(w, \textbf{parent}(w))$ abbia costo *massimo* tra tutti gli archi del cammino $v \rightsquigarrow \textbf{root}(v)$. Si assume che v non sia una radice.

Tale operazione può essere completata in tempo $O(\log n)$ nel caso pessimo. A questo punto l'algoritmo per mantenere una minima foresta ricoprente su un grafo soggetto a inserimenti di archi segue immediatamente: dopo aver calcolato una foresta iniziale F, viene utilizzata la procedura A.2 per inserire nuovi archi nel grafo e per modificare se necessario la minima foresta ricoprente

Algoritmo A.2 Insert-Edge(e)

```
Let e = {u, v}
if root(u) ≠ root(v) then
  evert(v)
  link(u, v)
else
  evert(v)
  x := maxcost(u)
  if cost(x) > w(e) then
    cut(x)
    evert(v)
    link(u, v)
  end if
end if
```

È immediato constatare che la procedura ha complessità $O(\log n)$ nel caso pessimo, in quanto comporta l'esecuzione di $O(1)$ operazioni su una foresta di Dynamic trees. □

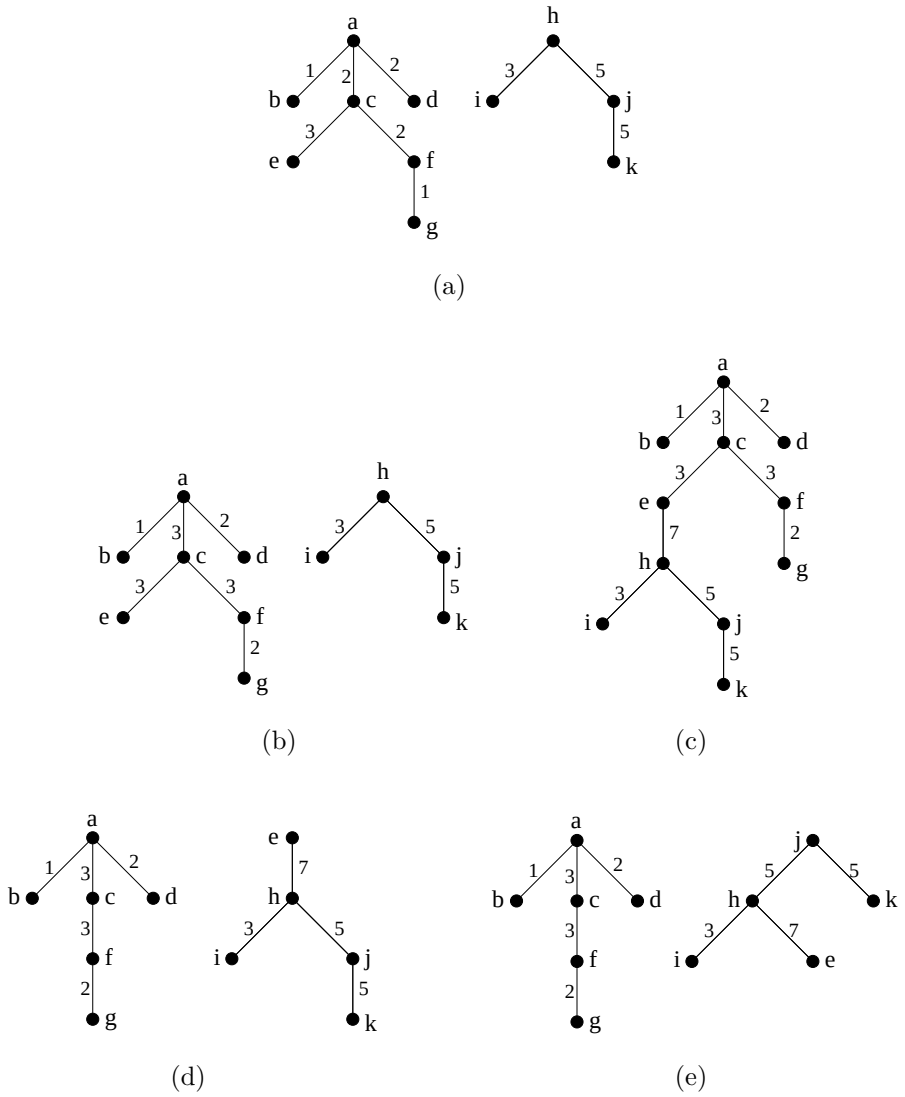


Figura A.4: Operazioni sui Dynamic trees. In A.4(a) sono rappresentati due alberi iniziali; **parent**(c) ritorna a, **root**(k) ritorna h, **cost**(e) ritorna 3, **mincost**(f) ritorna c. A.4(b): la foresta risultante dopo l'operazione **update**$(g, 1)$. A.4(c): l'albero risultante dopo l'operazione **link**$(h, e, 7)$. A.4(d) la foresta risultante dopo l'operazione **cut**(e) (tale operazione ritorna come valore il numero 3). A.4(e): la foresta risultante dopo l'operazione **evert**(j).

Appendice B

Glossario

Algoritmo La parola deriva dal nome del matematico arabo del IX secolo AL-Khārezmī, e indica qualunque procedimento sistematico di calcolo che a partire da un insieme di dati forniti in ingresso produce un insieme di valori in uscita.

Algoritmo dinamico Algoritmo in grado di operare su strutture dati soggette a modifiche, senza ricominciare di volta in volta da zero.

Algoritmo randomizzato Qualsiasi algoritmo che durante la sua esecuzione effettui delle scelte casuali.

Componenti connesse Classi di equivalenza in cui i vertici di un grafo non orientato $G = (V, E)$ sono partizionati dalla relazione di equivalenza $x\mathcal{R}y \equiv$ "x è connesso a y".

Derandomizzazione Tecnica algoritmica mediante la quale è possibile ottenere un algoritmo deterministico, a partire da uno randomizzato, avente la stessa complessità.

Dynamic tree Struttura dati mediante la quale è possibile mantenere determinate proprietà su una foresta di alberi dotati di radice, aventi n vertici in totale, supportando la possibilità di modificare la struttura della foresta con inserimenti e rimozioni di archi in tempo $O(\log n)$ per operazione nel caso pessimo.

ET-sequence Sequenza di $2n + 1$ simboli mediante la quale è possibile rappresentare un qualsiasi albero libero di n vertici.

ET-tree Struttura dati in grado di memorizzare efficientemente una foresta ricoprente F di un grafo G, insieme agli archi dell'insieme $E(G) \setminus E(F)$, supportando la possibilità di modificare F mediante inserimenti e rimozioni di archi.

Foresta Ricoprente Dato un grafo non orientato $G = (V, E)$, una foresta ricoprente F di G è un albero libero avente esattamente le stesse componenti connesse di G.

Foresta Ricoprente di peso minimo Dato un grafo non orientato $G = (V, E)$ e una funzione peso $w : E \to \mathbb{R}_{\geq 0}$, una foresta ricoprente di peso minimo F di G è una foresta ricoprente avente il minimo peso complessivo $w(F) = \sum_{e \in E(F)} w(e)$.

Grafo Un grafo non orientato è una coppia $G = (V, E)$, con V un insieme finito di vertici, e E un insieme di coppie non ordinate di vertici appartenenti all'insieme V.

Proprietà su grafi Una proprietà $\mathcal{P}$ su grafi è una funzione che soddisfa una delle condizioni seguenti: per ogni grafo G, per ogni $u, v \in V(G)$, mappa ogni tripla (G, u, v) nell'insieme $\{\text{vero}, \text{falso}\}$; oppure per ogni grafo G, mappa G in $\{\text{vero}, \text{falso}\}$; oppure mappa ogni grafo G in un suo sottografo G'.

Bibliografia

[1] A. V. Aho, J. E. Hopcroft, e J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, Mass., 1974.

[2] A. V. Aho, R. Sethi, e J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, Mass., 1986.

[3] D. Alberts, G. Cattaneo, e G. F. Italiano, *An empirical study of dynamic graph algorithms*, ACM Journal of Experimental Algorithmics, 2 (1997). `http://www.jea.acm.org/1997/AlbertsDynamic/`.

[4] G. Amato, G. Cattaneo, e G. F. Italiano, *Experimental analysis of dynamic minimum spanning tree algorithms*, in Proc. 8th ACM-SIAM Annual Symp. on Discrete Algorithms (SODA 97), 1997.

[5] I. Bárány e Z. Füredi, *Computing the volume is difficult*, Discrete Comput. Geometry, 2 (1987), pp. 319–326.

[6] H. Brönnimann, *Derandomization of geometric algorithms*, PhD Dissertation, Princeton University, May 1995.

[7] B. Chazelle, *A faster deterministic algorithm for minimum spanning trees*, in Proceedings 38th Annual Symposium on Foundations of Computer Science (FOCS 97), 1997, pp. 22–31.

[8] B. Chazelle e J. Matoušek, *On linear-time deterministic algorithms for optimization problems in fixed dimension*, in Proc. 4th ACM-SIAM Sympos. Discrete Algorithms, 1993, pp. 281–290.

[9] T. H. Cormen, C. E. Leiserson, e R. R. Rivest, *Introduction to Algorithms*, The MIT Press, 1990.

[10] D. Eppstein, Z. Galil, e G. F. Italiano, *Dynamic graph algorithms*, in CRC Handbook of Algorithms and Theory of Computation, CRC Press, 1997, ch. 22.

[11] D. Eppstein, Z. Galil, G. F. Italiano, e A. Nissenzweig, *Sparsification - A technique for speeding up dynamic graph algorithms*, Journal of ACM, 44 (1997), pp. 669–696.

[12] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, e M. Yung, *Maintenance of a minimum spanning forest in a dynamic plane graph*, Journal of Algorithms, 13 (1992), pp. 33–54.

[13] G. N. Frederickson, *Data structures for on-line updating of minimum spanning trees, with applications*, SIAM Journal on Computing, 14 (1985), pp. 781–798.

[14] D. Frigioni, M. Ioffreda, U. Nanni, e G. Pasqualone, *Experimental analysis of dynamic algorithms for the single source shortest path problem*, in Proc. of the Workshop on Alg. Engineering (WAE'97), G. F. Italiano e S. Orlando, eds., Venice, Italy, Sept. 1997, pp. 54–63.

[15] D. Frigioni e G. F. Italiano, *Dynamically switching vertices in planar graphs*, in Proc. 5th Annual European Symposium on Algorithms (ESA 97), Lecture Notes in Computer Science, Graz, Austria, Sept. 1997, Springer-Verlag, Berlin, pp. 186–199.

[16] D. Frigioni, D. Marchetti-Spaccamela, e U. Nanni, *Fully dynamic output bounded single source shortest path problem*, in Proc. ACM-SIAM Symp. on Discrete Algorithms, 1996, pp. 212–221.

[17] B. Gärtner, *A subexponential algorithm for abstract optimization problems*, in Proc. 33rd Annual IEEE Sympos. Found. Comput. Sci., 1992, pp. 464–472.

[18] Gopalakrishnan e Stinson, *Derandomization*, in Charles J. Colbourn and Jeffrey H. Dinitz (Eds.), The CRC Handbook of Combinatorial Designs, CRC Press, 1996.

[19] R. Gupta e M. L. Soffa, *Region scheduling*, in Proc. 2nd International Conference on Supercomputing, 1987, pp. 141–148.

[20] M. R. Henzinger, *Fully dynamic cycle-equivalence in graphs*, in Proceedings of the 35th Annual Symposium on Foundations of Computer Science, S. Goldwasser, ed., Los Alamitos, CA, USA, Nov. 1994, IEEE Computer Society Press, pp. 744–757.

[21] M. R. Henzinger e V. King, *Randomized dynamic graph algorithms with polylogarithmic time per operation*, in Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing, Las Vegas, Nevada, 29 May–1 June 1995, pp. 519–527.

[22] ——, *Fully dynamic 2-edge connectivity algorithm in polylogarithmic time per operation*, Technical note 1997-014, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, July 1997.

[23] ——, *Maintaining minimum spanning trees in dynamic graphs*, in Automata, Languages and Programming, 24th International Colloquium, P. Degano, R. Gorrieri, e A. Marchetti-Spaccamela, eds., vol. 1256 of Lecture Notes in Computer Science, Bologna, Italy, 7–11 July 1997, Springer-Verlag, pp. 594–604.

[24] M. R. Henzinger e M. Thorup, *Improved sampling with applications to dynamic graph algorithms*, in Automata, Languages and Programming, 23rd International Colloquium, F. M. auf der Heide e B. Monien, eds., vol. 1099 of Lecture Notes in Computer Science, Paderborn, Germany, 8–12 July 1996, Springer-Verlag, pp. 290–299.

[25] J. Holm, K. de Lichtenberg, e M. Thorup, *Poly-logarithmic deterministic fully-dynamic graph algorithms I: Connectivity and minimum spanning tree*, Rapporto Tecnico DIKU-TR-97/17, Department of Computer Science, University of Copenhagen, Sept. 1997.

[26] ——, *Poly-logarithmic deterministic fully-dynamic graph algorithms II: 2-edge and biconnectivity*, Rapporto Tecnico DIKU-TR-97/26, Department of Computer Science, University of Copenhagen, Nov. 1997.

[27] R. Johnson, D. Pearson, e K. Pingali, *Finding regions fast: Single entry single exit and control regions in linear time*, in Proc. Sigplan'94 PLDI, 1994.

[28] R. Johnson e K. Pingali, *Dependence-based program analysis*, in Proc. Sigplan'93 PLDI, 1993, pp. 78–89. Published as ACM SIGPLAN Notices 28(6).

[29] G. Kalai, *A subexponential randomized simplex algorithm*, in Proc. 24th Annual ACM Sympos. Theory Comput., 1992, pp. 475–482.

[30] D. R. Karger, P. N. Klein, e R. E. Tarjan, *A randomized linear-time algorithm to find minimum spanning trees*, Journal of the ACM, 42 (1995), pp. 321–328.

[31] M. Kaufmann, J. F. Sibeyn, e T. Suel, *Derandomizing algorithms for routing and sorting on meshes*, in Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms, D. D. Sleator, ed., Arlington, VA, Jan. 1994, ACM Press, pp. 669–679.

[32] S. B. Lippman, *C++ - Corso di programmazione*, Addison-Wesley, 1993.

[33] L. Lovász e M. Simonovits, *On the randomized complexity of volume and diameter*, in Proc. 33rd Annual IEEE Sympos. Found. Comput. Sci., 1992, pp. 482–492.

[34] S. Mahajan e H. Ramesh, *Derandomizing semidefinite programming based approximation algorithms*, Research Report MPI-I-96-1-013, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, May 1996.

[35] J. Matoušek, *Derandomization in computational geometry*, Journal of Algorithms, 20 (1996), pp. 545–580.

[36] K. Melhorn e S. Näher, *LEDA, a platform for combinatorial and geometric computing*, Communications of the ACM, 38 (1996), pp. 267–305.

[37] R. Motwani e R. Prabhakar, *Randomized Algorithms*, Cambridge University Press, 1995.

[38] K. Mulmuley, *Computational Geometry: An Introduction Through Randomized Algorithms*, Prentice Hall, New York, 1993.

[39] N. Nagamochi e T. Ibaraki, *Linear time algorithms for finding a sparse k-connected spanning subgraph of a k-connected graph*, Algorithmica, 7 (1992), pp. 583–596.

[40] S. K. Park e K. W. Miller, *Random number generators: Good ones are hard to find*, Communications of the ACM, 31 (1988), pp. 1192–1201.

[41] G. Ramalingam, *Bounded incremental computation*, vol. 1089 of Lecture Notes in Computer Science, Springer Verlag, 1996.

[42] G. Ramalingam e T. Reps, *An incremental algorithm for a generalization of the shortest path problem*, J. of Algorithms, 21 (1996), pp. 267–305.

[43] D. D. Sleator e R. E. Tarjan, *A data structure for dynamic trees*, J. Comput. System Sci., 26 (1983), pp. 362–390.

[44] R. E. Tarjan, *Data Structures and Network Algorithms*, SIAM, 1983.

[45] R. E. Tarjan e U. Vishkin, *An efficient parallel biconnectivity algorithm*, SIAM J. Comput., 18 (1989), pp. 1–11.

Indice analitico