Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

ALGORITMI PER L'ARBORESCENZA MINIMA: TEORIA ED IMPLEMENTAZIONI

ALGORITHMS FOR THE MINIMUM ARBORESCENCE: THEORY AND IMPLEMENTATIONS

MARTA CIARLEGLIO

Relatrice: *Prof.ssa Maria Cecilia Verri*

Anno Accademico 2024-2025

INDICE

ELENCO DELLE FIGURE

A Nonna Clara

1

INTRODUZIONE

La teoria dei grafi costituisce uno degli strumenti matematici più versatili e potenti per la modellazione di fenomeni complessi. Un grafo fornisce infatti un linguaggio naturale per rappresentare entità (i nodi) e relazioni tra esse (gli archi), trovando applicazione in campi eterogenei quali l'informatica, la biologia, l'ingegneria dei trasporti e le scienze sociali. La capacità dei grafi di descrivere in modo astratto strutture relazionali rende possibile affrontare problemi di natura diversa all'interno di un quadro teorico unificato.

Dal punto di vista strutturale, una prima distinzione fondamentale riguarda la direzionalità delle relazioni rappresentate. Nei *grafi non orientati*, gli archi esprimono connessioni simmetriche: la relazione tra due nodi è bidirezionale e non presenta un verso privilegiato. Nei *grafi orientati*, al contrario, ciascun arco è dotato di una direzione, che definisce un vincolo di ordine tra nodo di partenza e nodo di arrivo. Questa distinzione, pur apparendo elementare, ha conseguenze profonde sia sulle proprietà topologiche delle strutture ottenute, sia sulla complessità computazionale dei problemi che le riguardano.

In particolare, i grafi orientati sono lo strumento naturale per modellare processi in cui il flusso dell'informazione, del controllo o delle risorse non è simmetrico. Esempi classici includono le reti di dipendenze nei programmi informatici, i sistemi di trasporto con vincoli di direzione, le reti neurali e le relazioni gerarchiche in organizzazioni sociali o biologiche. La direzionalità introduce quindi una nozione di causalità o precedenza, che arricchisce il modello ma al tempo stesso ne aumenta la complessità analitica.

All'interno della teoria dei grafi, un concetto cardine è quello di *albero*. Un albero è un grafo connesso e privo di cicli, che rappresenta la forma più semplice di struttura gerarchica in un contesto non orientato. La sua importanza risiede nella capacità di garantire una connessione minima tra i nodi, evitando ridondanze: tra due qualsiasi nodi esiste infatti un unico

cammino semplice. Questa proprietà fa sì che gli alberi siano utilizzati come architetture di riferimento in numerosi ambiti applicativi, dalla rappresentazione di strutture gerarchiche in informatica alla modellazione di processi di ramificazione in biologia.

Quando a ciascun arco di un grafo non orientato viene associato un peso, diventa naturale ricercare la sottostruttura connessa che colleghi tutti i nodi minimizzando la somma dei pesi. Tale struttura prende il nome di *albero ricoprente minimo* (Minimum Spanning Tree, MST). Gli algoritmi per il calcolo dell'MST, tra cui quelli classici di *Kruskal* e *Prim*, costituiscono un risultato fondamentale dell'informatica teorica e hanno trovato ampia applicazione in problemi pratici quali la progettazione di reti di comunicazione e di trasporti, la pianificazione di infrastrutture e l'ottimizzazione di processi logistici.

L'MST rappresenta quindi una soluzione ottimale a problemi di connettività in grafi non orientati, garantendo al contempo semplicità strutturale ed efficienza nei costi. Nel caso dei grafi orientati, la nozione di albero deve essere adattata per tener conto della direzionalità degli archi. Un grafo orientato introduce infatti una distinzione sostanziale: non è sufficiente che la struttura sia connessa e aciclica, ma occorre stabilire anche una direzione univoca per la propagazione dei cammini. È in questo contesto che si parla di *arborescenza*.

Un'arborescenza è un grafo orientato aciclico in cui tutti gli archi sono orientati a partire da un nodo speciale, detto *radice*, verso tutti gli altri nodi. Ciò significa che per ogni vertice esiste esattamente un cammino orientato che collega ad esso la radice, e che la radice stessa non ha archi entranti. La scelta della radice è un aspetto cruciale: mentre negli alberi non orientati la struttura ricoprente è unica a prescindere da un nodo privilegiato, in un grafo orientato la struttura dell'arborescenza dipende dal vertice selezionato come radice. Per questo motivo, non è corretto parlare di "albero" in senso stretto, ma occorre utilizzare la nozione più generale di arborescenza.

Analogamente al caso degli alberi ricoprenti minimi nei grafi non orientati, si definisce *arborescenza minima* (Minimum Spanning Arborescence, MSA) una sottostruttura orientata che collega tutti i nodi del grafo partendo da una radice fissata e che minimizza la somma dei pesi degli archi. Questo problema, noto anche come *problema dell'arborescenza di Edmonds*, rappresenta la naturale estensione del concetto di MST ai grafi orientati.

Le arborescenze minime trovano applicazioni in contesti in cui la direzionalità dei legami è intrinseca al problema, come nelle reti di comunicazione orientate, nei modelli di dipendenza e nei sistemi di

flusso di informazioni. Rispetto agli alberi ricoprenti minimi, il loro studio richiede tecniche algoritmiche specifiche che tengano conto della struttura orientata del grafo e della presenza di cicli orientati.

La presente tesi si concentra sullo studio approfondito degli algoritmi per la costruzione di arborescenze minime, con particolare attenzione agli aspetti teorici, implementativi e sperimentali. Il fulcro del lavoro è costituito dall'analisi dettagliata degli algoritmi descritti nell'articolo *Efficiently Computing Directed Minimum Spanning Trees* [2] di Bother et al., esaminandone la logica operativa, le strutture dati utilizzate e le strategie di ottimizzazione adottate per garantire efficienza computazionale. Accanto alla trattazione teorica, viene condotta una valutazione sperimentale dei diversi algoritmi su una vasta gamma di reti, sia reali sia generate artificialmente, al fine di confrontarne le prestazioni e evidenziare punti di forza e limitazioni.

L'obiettivo principale della tesi è dunque duplice: da un lato fornire una comprensione chiara e completa dei meccanismi algoritmici per la costruzione di arborescenze minime; dall'altro esplorare le applicazioni concrete di questi algoritmi, come la gestione sincrona della memoria in linguaggi con *garbage collection* e la stima della gerarchia in reti complesse orientate. In questo modo, il lavoro mette in evidenza non solo la validità teorica degli algoritmi trattati, ma anche la loro rilevanza pratica, dimostrando come le arborescenze minime possano rappresentare uno strumento versatile e potente in contesti diversi.

2

ALGORITMI PER L'ARBORESCENZA MINIMA

Il problema che verrà di seguito affrontato riguarda la determinazione di un'arborescenza radicata di costo minimo, dato un grafo orientato e pesato. Più precisamente, fissata una radice, gli algoritmi presentati individuano un sottoinsieme degli archi che forma un albero orientato che ha origine nella suddetta radice, tale che da questo vertice esista un cammino orientato verso ciascun altro nodo, e che la somma dei pesi degli archi selezionati sia minima. Come accennato in precedenza, questo risultato rappresenta l'analogo, nel contesto dei grafi orientati, del ben noto problema dell'albero ricoprente minimo nei grafi non orientati.

2.1 L'ALGORITMO DI EDMONDS

L'algoritmo di Edmonds [6], anche detto algoritmo di Chu–Liu–Edmonds, costituisce uno dei risultati più rilevanti nello studio delle strutture ottimali in grafi orientati. Esso è stato sviluppato in maniera indipendente da Chu e Liu (1965) [3] e da Jack Edmonds (1967) [6]. In particolare, Edmonds non solo formalizzò il problema, ma fornì anche una dimostrazione della correttezza del metodo [6], caratterizzata da un livello di complessità notevole.

L'importanza dell'algoritmo di Edmonds risiede non soltanto nel contributo teorico, ma anche nelle sue numerose applicazioni pratiche. La possibilità di determinare arborescenze minime trova infatti impiego in vari contesti, come la progettazione di reti di comunicazione e di trasporto, l'analisi di sistemi complessi e lo studio di strutture gerarchiche. Inoltre, il lavoro di Edmonds ha aperto la strada a una lunga serie di sviluppi successivi, che hanno portato al miglioramento delle prestazioni computazionali e alla nascita di varianti e generalizzazioni dell'algoritmo stesso, alcune delle quali verranno analizzate nelle sezioni successive.

2.1.1 *Descrizione dell'algoritmo*

L'algoritmo di Edmonds affronta il suddetto problema della costruzione di un'arborescenza minima attraverso un procedimento iterativo basato sulla selezione degli archi e sulla ricerca e contrazione di cicli. L'idea principale è quella di garantire che, per ogni nodo diverso dalla radice, sia scelto esattamente un arco entrante di peso minimo, riducendo progressivamente la complessità della struttura del grafo fino a ottenere una soluzione che rappresenti effettivamente un'arborescenza.

Dato un grafo $G = (V, E)$ orientato e pesato e un nodo $r \in V$, l'algoritmo di Edmonds restituirà dunque l'albero ricoprente orientato con radice r di costo minimo $T = (V, E^*)$, dove E^* rappresenta un sottoinsieme di archi del grafo originario.

Il funzionamento può essere descritto nei seguenti passaggi fondamentali:

1. **Scelta degli archi entranti minimi.** Per ogni vertice $v \neq r$, dove r è la radice prefissata, si seleziona l'arco entrante di peso minimo $\pi(v)$. Se il grafo derivante da questa selezione non contiene cicli, l'insieme di tali archi costituisce direttamente un'arborescenza minima.

2. **Individuazione dei cicli.** Se gli archi selezionati generano uno o più cicli, l'insieme ottenuto non può costituire un'arborescenza. In questo caso, ciascun ciclo viene individuato e considerato come una componente fortemente connessa.

3. **Contrazione dei cicli.** Ogni ciclo viene contratto in un singolo "super-nodo". In questa fase, i pesi degli archi che entrano nel ciclo vengono opportunamente aggiornati, in modo da preservare la possibilità di ottenere, al termine del procedimento, una soluzione di peso minimo.

4. **Iterazione.** L'algoritmo viene ripetuto sul grafo contratto: si selezionano nuovamente gli archi entranti minimi, si individuano i cicli e li si contrae, fino a quando non si ottiene un grafo privo di cicli.

5. **Ricostruzione della soluzione.** Una volta raggiunto un grafo aciclico, gli archi selezionati costituiscono un'arborescenza minima. Infine, si procede a espandere progressivamente i super-nodi, ricostruendo l'arborescenza nel grafo originale.

2.1.2 *Operazioni principali*

La **contrazione dei cicli** è un passaggio cruciale dell'algoritmo, poiché consente di ridurre la complessità del grafo preservando le proprietà necessarie per ottenere un'arborescenza minima. Ogni ciclo individuato viene considerato come un singolo super-nodo, e tutti gli archi entranti nel ciclo vengono "ricalibrati" in termini di peso. In particolare, per un arco (u, v) con v appartenente al ciclo C e u esterno ad esso, il peso aggiornato dell'arco verso il super-nodo corrisponde alla differenza tra il peso originale dell'arco e il peso dell'arco entrante minimo di v all'interno del ciclo. Tra i diversi archi così ricalibrati uscenti da u ed entranti nel super-nodo, che formano a questo punto dei multi-archi, viene mantenuto solo quello con peso minimo. Questa trasformazione garantisce che, nella fase successiva di selezione degli archi minimi nel grafo contratto, la scelta ottimale verso il super-nodo corrisponda esattamente alla scelta ottimale tra gli archi che entrano nel ciclo nel grafo originale. Durante la contrazione, gli archi interni al ciclo vengono temporaneamente esclusi dalla considerazione, ma la loro struttura e i loro archi minimi vengono memorizzati per poter essere reintegrati correttamente nella soluzione finale.

Per illustrare tale fase, consideriamo un grafo orientato e pesato contenente un ciclo, come in Figura 1, e consideriamo A come radice. L'operazione di contrazione consiste nel sostituire l'intero ciclo con un unico nodo, detto "super-nodo", mantenendo solo gli archi in ingresso e in uscita dal ciclo stesso. In questo modo la struttura del grafo si semplifica, rendendo possibile proseguire l'elaborazione senza perdere informazioni sulla connettività. Notiamo che, in questo caso, l'unico arco uscente dal ciclo è l'arco (C, A), che è entrante nella radice, e dunque non deve essere considerato.

Figura 1: Contrazione di un ciclo: a sinistra il grafo originale con il ciclo evidenziato in rosso; a destra lo stesso grafo dopo la contrazione, in cui il ciclo è sostituito dal super-nodo X ed il peso dell'arco entrante nel ciclo è stato opportunamente aggiornato.

La **ricostruzione dell'arborescenza minima** avviene in modo retrogrado: ogni super-nodo ottenuto dalle contrazioni deve essere progressivamente "espanso" nel grafo originale. Per ciascun super-nodo si reinserisce l'arco esterno che lo collega al resto dell'arborescenza, ossia l'arco entrante selezionato durante la fase di contrazione. All'interno del ciclo corrispondente, si considerano poi gli archi entranti minimi precedentemente scelti per ciascun vertice. Questi archi, presi nel loro insieme, formano un sottografo ciclico: per ripristinare la struttura aciclica è dunque necessario eliminare esattamente uno di essi, ovvero quello che chiuderebbe il ciclo in conflitto con l'arco esterno già fissato. Gli altri archi possono invece essere mantenuti, garantendo che ogni vertice del ciclo resti correttamente connesso. In questo modo, la ricostruzione conserva la proprietà di minimalità: la somma dei pesi degli archi reinseriti e di quello esterno coincide con il peso complessivo dell'arborescenza minima radicata nel grafo originale.

Per chiarire il funzionamento della fase di ricostruzione, si consideri il super-nodo X ottenuto in seguito alla contrazione precedentemente illustrata. Nella fase di espansione, tale super-nodo viene sostituito dai vertici originari, a cui viene reinserito l'arco esterno che collegava X al resto del grafo (in questo caso $A \to B$). All'interno del ciclo vengono quindi ripristinati gli archi entranti minimi scelti per ciascun vertice, che nel loro insieme formano un sottografo ciclico. Per garantire l'assenza di cicli, viene eliminato l'arco $D \to B$, cioè quello che chiude il ciclo, così da ottenere l'arborescenza minima completa e consistente con il grafo originale.

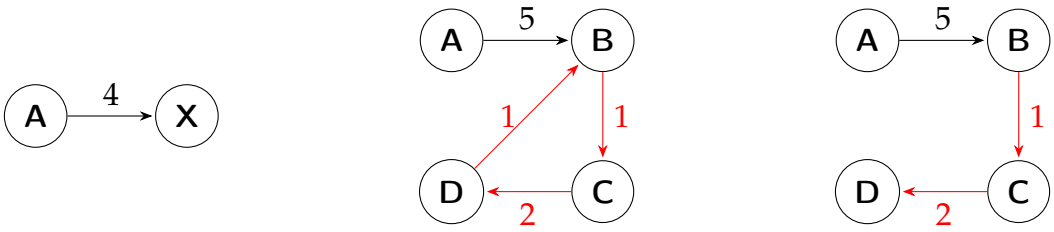

Figura 2: Espansione di un super-nodo: a sinistra il grafo contratto con X; al centro il super-nodo espanso con il ciclo ricostruito; a destra la struttura finale dell'arborescenza minima, ottenuta eliminando l'arco $D \to B$.

Questa doppia operazione di contrazione e successiva espansione garantisce sia la correttezza sia l'ottimalità della soluzione, trasformando un problema di selezione locale di archi in un procedimento ricorsivo controllato che preserva la struttura globale minima, e producendo, al

termine, un sottoinsieme di archi che soddisfa le condizioni richieste: connessione di tutti i nodi a partire dalla radice e minimizzazione del peso complessivo.

2.1.3 *Complessità*

La complessità temporale dell'algoritmo, nella sua forma originale, è $O(n \cdot m)$ [6], dove n è il numero di nodi e m il numero di archi. Tale risultato deriva da una combinazione di fattori legati alla selezione degli archi minimi, all'individuazione dei cicli e alla loro contrazione. In particolare:

1. **Selezione degli archi entranti minimi:** per ogni vertice $v \neq r$ bisogna determinare l'arco entrante di peso minimo. Questa operazione richiede, nel caso più generale, una scansione di tutti gli archi, quindi un costo complessivo di $O(m)$.

2. **Individuazione dei cicli:** una volta scelti gli archi minimi entranti, è necessario verificare se la selezione induce dei cicli. Tale identificazione può essere effettuata mediante una visita in profondità o con algoritmi per il calcolo delle componenti fortemente connesse, entrambi eseguibili in $O(n + m)$.

3. **Contrazione dei cicli:** ogni ciclo individuato viene sostituito da un singolo super-nodo. Gli archi entranti nel ciclo vengono ricalibrati aggiornando i pesi. Anche questa operazione, nel caso peggiore, può richiedere l'esame di tutti gli archi, dunque $O(m)$.

4. **Iterazione sul grafo contratto:** l'algoritmo viene ripetuto sul grafo contratto fino a quando non si ottiene un grafo privo di cicli. Ad ogni iterazione il numero di nodi diminuisce (poiché ogni ciclo diventa un singolo nodo), e quindi il numero massimo di livelli ricorsivi è $O(n)$.

Combinando questi fattori, la complessità totale dell'algoritmo nella sua implementazione classica è:

$$O(n \cdot m).$$

Essa è polinomiale ed è considerata "accettabile" per grafi di dimensioni moderate. Tuttavia, su grafi molto grandi e densi, questo costo diventa significativo.

Proprio per questo motivo, dopo Edmonds, sono stati sviluppati miglioramenti sostanziali: prima Tarjan (1977) [14], con un algoritmo avente

complessità $O(m \log n)$, e poi Gabow et al. (1986) [8], con un'ulteriore ottimizzazione che porta la complessità a $O(n \log n + m)$, entrambi ottenuti tramite strutture dati avanzate.

2.2 LA VERSIONE DI TARJAN

L'algoritmo di Tarjan [14] rappresenta un'evoluzione dell'approccio classico di Edmonds per la ricerca di un'arborescenza minima radicata in un grafo orientato e pesato. L'idea fondamentale consiste nel combinare la selezione degli archi entranti di peso minimo con strutture dati avanzate, in grado di gestire cicli e contrazioni in modo più efficiente. In particolare, Tarjan sfrutta l'utilizzo di algoritmi union-find con *path compression* e di heap, o altre implementazioni efficienti di code con priorità, per aggiornare rapidamente i pesi degli archi durante la contrazione dei cicli, riducendo così il numero di operazioni necessarie rispetto alla versione classica.

L'algoritmo mantiene lo schema generale di Edmonds, con alcune differenze chiave che ne migliorano l'efficienza. Il funzionamento può essere sintetizzato nei seguenti passaggi fondamentali:

1. **Scelta dell'arco minimo.** Per ogni vertice $v \neq r$, dove r è la radice prefissata, si seleziona l'arco entrante di peso minimo.

2. **Rilevamento dei cicli.** I cicli generati dagli archi selezionati vengono individuati e gestiti tramite strutture union-find, che consentono di rilevare rapidamente se l'aggiunta di un arco creerebbe un ciclo.

3. **Contrazione dei cicli.** Ogni ciclo viene contratto in un super-nodo e i pesi degli archi esterni vengono aggiornati mediante heap o liste di priorità, evitando di riesaminare tutti gli archi a ogni iterazione.

4. **Iterazione.** Il procedimento viene ripetuto sul grafo contratto fino a ottenere un grafo privo di cicli.

5. **Ricostruzione della soluzione.** Infine, i super-nodi vengono espansi retroattivamente, ricostruendo l'arborescenza minima nel grafo originale grazie alle informazioni memorizzate durante le contrazioni.

Grazie all'impiego di strutture dati efficienti come union-find e heap, l'algoritmo di Tarjan migliora la complessità rispetto alla versione base di Edmonds. In particolare, in grafi sparsi la complessità è $O(m \log n)$, dove n è il numero di vertici e m il numero di archi. Tale riduzione

rispetto a $O(n \cdot m)$ deriva proprio dal miglioramento mostrato nella fase di contrazione: ad ogni iterazione, l'algoritmo originale esegue al più $O(m)$ operazioni e, poiché il numero massimo di iterazioni non supera $O(n)$, la complessità totale risulta $O(n \cdot m)$; l'implementazione di Tarjan, invece, ottimizza le operazioni per ridurre il fattore moltiplicativo tramite heap e union-find [9, 13].

L'algoritmo richiede queste strutture dati per tre compiti principali [2]: individuare l'arco entrante più economico per ogni vertice, riconoscere i cicli tra gli archi scelti e tenere traccia delle contrazioni dei cicli. I due ultimi compiti possono essere gestiti efficientemente tramite strutture union-find. Per rilevare i cicli, una union-find mantiene le componenti debolmente connesse formate dagli archi selezionati. Poiché ogni vertice può avere al massimo un arco entrante selezionato, un nuovo arco chiude un ciclo orientato solo se collega due vertici già appartenenti alla stessa componente connessa. Un'altra union-find viene invece utilizzata per gestire le contrazioni dei cicli e per mappare i vertici originali ai super-nodi creati. Gli estremi degli archi non vengono aggiornati ad ogni contrazione: ogni volta che l'algoritmo deve gestire un arco, viene effettuata una semplice ricerca nella union-find.

Per gestire gli archi entranti in modo efficiente, infine, è necessario mantenere un insieme per ciascun vertice che supporti quattro operazioni: aggiungere un arco, estrarre l'arco di peso minimo, aggiornare i pesi di tutti gli archi di una costante e unire due insiemi di archi [2]. Se tutte queste operazioni possono essere eseguite in tempo logaritmico, l'intero algoritmo ha complessità $O(m \log n)$, in quanto ogni arco può essere considerato al massimo una volta. Strutture come gli heap unibili (*mergeable heaps*) supportano le operazioni principali e, tramite propagazione *lazy*, permettono anche l'aggiornamento dei pesi. Se invece le operazioni più costose vengono eseguite in tempo lineare, ad esempio utilizzando una matrice di adiacenza, la complessità diventa $O(n^2)$, che risulta più conveniente nei grafi molto densi [2].

L'algoritmo di Tarjan presenta diversi vantaggi: consente un significativo miglioramento dell'efficienza, in particolare nei grafi sparsi, e le strutture dati avanzate permettono di gestire cicli e contrazioni in modo elegante e sistematico. Tuttavia, la maggiore complessità di implementazione rappresenta uno svantaggio rispetto all'algoritmo base di Edmonds, richiedendo attenzione nella gestione della memoria e degli aggiornamenti. Inoltre, in grafi molto densi il guadagno in termini di tempo di esecuzione può risultare meno evidente, riducendo l'impatto pratico delle ottimizzazioni introdotte [2].

2.3 L'ALGORITMO DI GABOW ET AL. (GGST VERSION)

L'algoritmo di Gabow et al. [8] rappresenta un ulteriore sviluppo degli approcci per la determinazione di un'arborescenza minima radicata in un grafo orientato e pesato. L'idea fondamentale consiste nell'utilizzare strutture dati avanzate per mantenere informazioni sui cicli e sui vertici contratti, riducendo il numero di scansioni degli archi necessarie e migliorando la gestione dei cicli rispetto all'algoritmo di Edmonds. In particolare, Gabow et al. introducono una strategia di contrazione dei cicli che consente di gestire simultaneamente più cicli, combinata con una gestione efficiente degli archi entranti minimi per ciascun vertice.

Il funzionamento dell'algoritmo può essere sintetizzato, come per i suoi predecessori, nei seguenti passaggi:

1. **Selezione degli archi minimi.** Per ogni vertice $v \neq r$ si determina l'arco entrante di peso minimo.

2. **Rilevamento dei cicli multipli.** I cicli generati dagli archi scelti vengono individuati simultaneamente tramite strutture dati che consentono di rilevare rapidamente le componenti cicliche.

3. **Contrazione dei cicli.** Tutti i cicli individuati vengono contratti in super-nodi in modo efficiente, aggiornando i pesi degli archi esterni senza riesaminare tutti gli archi.

4. **Iterazione.** Il procedimento viene ripetuto sul grafo contratto fino a ottenere un grafo aciclico.

5. **Ricostruzione della soluzione.** I super-nodi vengono infine espansi retroattivamente per ricostruire l'arborescenza minima nel grafo originale.

L'algoritmo di Gabow et al. rappresenta un raffinamento della versione proposta da Tarjan [14], con l'obiettivo di ridurre ulteriormente il tempo di esecuzione. La complessità raggiunta è $O(n \log n + m)$, risultato che si ottiene sfruttando in maniera più efficiente l'ordine con cui i vertici vengono processati [2]. In particolare, gli autori introducono il concetto di *growth path*, ossia un cammino di vertici costruito scegliendo sempre l'arco entrante più economico del vertice di testa. Questo approccio consente di evitare ricerche globali, poiché la gestione degli archi è localizzata lungo il cammino stesso.

Per semplificare l'analisi e rendere l'algoritmo indipendente dalla radice, vengono inoltre introdotti archi fittizi di peso zero dal nodo radice a tutti gli altri vertici: tali archi non incidono sul peso finale ma permettono di trattare in modo uniforme tutti i casi, senza dover considerare la radice un caso speciale.

La gestione degli archi esterni al cammino è organizzata tramite *exit lists*, liste ordinate per posizione del vertice di destinazione lungo il *growth path*. Di ciascuna lista viene mantenuto attivo solo il primo arco, mentre gli altri restano passivi, riducendo i costi di aggiornamento. Gli archi attivi sono memorizzati in una struttura dati dedicata, l'*active forest*, che permette di aggiornare ed estrarre archi in tempo logaritmico. Gli archi passivi, allo stesso modo, vengono mantenuti in apposite liste (*passive lists*) per gestire correttamente i multi-archi e per poterli eliminare o consolidare in caso di contrazioni. Infine, come nella versione precedente, le contrazioni dei cicli vengono gestite tramite strutture union-find [9, 13], che tracciano la corrispondenza tra i vertici originali e quelli contratti.

La combinazione di questi accorgimenti permette di raggiungere la complessità dichiarata di $O(n \log n + m)$, significativamente migliore rispetto a Edmonds e competitiva anche rispetto alla versione di Tarjan. In particolare, notiamo che ogni arco del grafo viene considerato al massimo una volta, contribuendo con un costo totale $O(m)$. Le operazioni sui vertici, come la gestione del percorso di crescita, la fusione dei prefissi in caso di contrazioni e l'estrazione del minimo dagli insiemi di archi, sono effettuate mediante union-find e heap unibili (*mergeable heaps*), che garantiscono un costo $O(\log n)$ per vertice. Poiché ci sono n vertici, queste operazioni aggiungono un contributo di $O(n \log n)$, portando così alla complessità totale $O(m + n \log n)$. Rispetto all'algoritmo di Tarjan, in cui il termine logaritmico compare per ogni arco considerato ($O(m \log n)$), la versione di Gabow sfrutta il *growth path* e l'ordine di elaborazione dei vertici per ridurre il numero di operazioni logaritmiche necessarie.

Il principale punto di forza dell'algoritmo di Gabow et al. è quindi, senza dubbio, l'efficienza: mentre per grafi sparsi e densi eguagli la versione di Tarjan, la riduzione a $O(n \log n + m)$ lo rende particolarmente adatto per grafi di grandi dimensioni e a densità intermedia, nei quali le versioni precedenti mostrano prestazioni inferiori. D'altro canto, la complessità della descrizione e dell'implementazione costituisce una debolezza evidente. L'introduzione di nuove strutture come l'*active forest* e la gestione articolata delle *exit lists* richiedono una progettazione attenta, rendendo l'algoritmo meno immediato da comprendere e implementare rispetto a Edmonds e Tarjan. Inoltre, l'aggiunta di archi fittizi, sebbene

utile per uniformare i casi, introduce un ulteriore livello di astrazione che può rendere meno intuitivo il funzionamento. Per grafi molto piccoli o molto densi, il vantaggio asintotico può essere marginale, e in tali scenari soluzioni più semplici come Tarjan possono risultare preferibili.

Per chiarezza, si riporta qui lo pseudocodice dell'algoritmo di Gabow et al. [2].

Algorithm 1 Algoritmo per l'arborescenza minima, Gabow et al. [2]

```
initialize growth path with an arbitrary vertex;
insert its incoming edges into exit lists;
while not all vertices are on the growth path do
    query min. incoming edge (u, v) of the path head from active forest;
    remember (u, v) for reconstruction;
    if find(u) is not on growth path then
        insert u's incoming edges into exit lists;
    else
        delete prefix of path up to last occurrence of find(u);
        update incoming edge costs for all vertices on prefix;
        delete outgoing edges of prefix from exit lists;
        merge prefix in DSU and Active Forest;
        limit edges into the cycle to at most 1 per origin;
    end if
    insert find(u) at the front of the path;
end while
```

2.4 BASI TEORICHE SULLE STRUTTURE DATI UTILIZZATE

Gli ultimi due algoritmi esaminati, come già spiegato, si basano fortemente su strutture dati avanzate per garantire efficienza sia nella gestione degli archi sia nel tracciamento dei cicli e delle contrazioni. In particolare, le strutture union-find [9, 13] permettono di rappresentare insiemi disgiunti di vertici e di eseguire operazioni di *find* e *union* in tempo quasi costante, grazie alle ottimizzazioni note come *union by rank* e *path compression* [13]. Queste tecniche consentono di identificare rapidamente a quale componente appartiene un vertice e di aggiornare la struttura quando i cicli vengono contratti in super-nodi.

La tecnica di *union by rank* consiste nel collegare l'albero con rango minore a quello con rango maggiore durante l'unione di due insiemi, in

modo da mantenere basse le altezze degli alberi e ridurre il costo delle successive operazioni di ricerca. La *path compression*, invece, ottimizza le operazioni di *find* aggiornando i puntatori di tutti i nodi lungo il cammino verso la radice, collegandoli direttamente ad essa; in questo modo, i successivi accessi ai vertici dello stesso insieme diventano molto più rapidi [13].

Parallelamente, gli heap unibili (*mergeable heaps*) svolgono un ruolo cruciale nel mantenimento degli archi entranti minimi per ciascun vertice. Essi supportano operazioni come l'inserimento, l'estrazione del minimo e la fusione di due heap, che sono fondamentali per la selezione efficiente degli archi durante le iterazioni degli algoritmi. Alcune varianti avanzate, come gli *hollow heaps* o i *skew heaps* con propagazione "pigra" (*lazy propagation*), permettono di aggiornare i pesi degli archi in maniera ritardata, riducendo ulteriormente i costi computazionali.

In particolare, gli *hollow heaps* [10] sono una variante dei classici heap, progettata per supportare efficientemente le già citate operazioni, riducendo il numero di operazioni per l'aggiornamento degli archi. In questa struttura, quando un nodo viene rimosso o il suo peso aggiornato, esso diventa *hollow* e i suoi figli vengono spostati senza ricostruire l'intero heap, migliorando così le prestazioni sulle operazioni di fusione.

I *skew heaps* [12], invece, sono heap binari auto-bilancianti che permettono di effettuare la fusione di due heap in modo semplice ed efficiente, senza mantenere informazioni aggiuntive sui ranghi dei nodi. La loro caratteristica principale è che durante la fusione si scambiano ricorsivamente i figli dei nodi, distribuendo uniformemente la profondità degli alberi e garantendo un tempo medio logaritmico per le operazioni di estrazione del minimo e fusione.

La combinazione di queste strutture dati consente agli algoritmi di scalare efficientemente con il numero di vertici e di archi del grafo, mantenendo tempi di esecuzione ottimali anche in grafi di grandi dimensioni o con topologie complesse. La scelta accurata della struttura dati più adatta in funzione delle caratteristiche del grafo risulta quindi determinante per le prestazioni complessive degli algoritmi.

2.5 ARBORESCENZE DINAMICHE OTTIMALI

Gli algoritmi finora analizzati affrontano il problema della ricerca di un'arborescenza minima in contesti statici, ovvero assumendo che la struttura del grafo non subisca variazioni nel tempo. Tuttavia, in molte applicazioni pratiche i grafi sono soggetti a modifiche dinamiche, come

l'aggiunta o la rimozione di archi. In questi casi, ricalcolare da zero l'arborescenza ottimale a ogni variazione può risultare estremamente costoso.

Un contributo significativo in questa direzione è rappresentato dal lavoro di Espada et al. [7], che propone un approccio per mantenere efficientemente l'arborescenza ottimale al variare del grafo. L'idea centrale è sfruttare i risultati già calcolati e aggiornare soltanto le parti della soluzione direttamente influenzate dalle modifiche, riducendo così il costo computazionale complessivo. Gli autori analizzano diversi scenari di applicazione, come l'ottimizzazione di reti e l'inferenza filogenetica, e presentano risultati sperimentali su grafi sia reali sia sintetici.

Questo filone di ricerca mostra come gli algoritmi classici possano essere estesi e adattati a contesti dinamici, ampliando la portata e la rilevanza pratica delle arborescenze ottimali in domini applicativi di larga scala. In pratica, quando viene inserito o rimosso un arco, l'algoritmo proposto verifica se tale cambiamento influisce sulla validità o sul peso dell'arborescenza corrente. Se l'arco è irrilevante, la struttura rimane invariata; se invece introduce un'alternativa più economica o rompe un equilibrio esistente, vengono applicate operazioni di aggiornamento mirate. Queste operazioni consistono in una serie di sostituzioni locali di archi, effettuate attraverso meccanismi di confronto dei pesi e, se necessario, di ricontrazione dei cicli. In questo modo, l'arborescenza viene mantenuta "minima" ad ogni passo, con un costo computazionale molto inferiore rispetto a un ricalcolo globale.

3

IMPLEMENTAZIONI E VALUTAZIONI SPERIMENTALI

In questo capitolo ci si propone di analizzare le implementazioni pratiche degli algoritmi per la determinazione dell'arborescenza minima presentati nel capitolo precedente. L'obiettivo non è proporre nuove implementazioni, ma fornire una valutazione critica delle versioni esistenti, con particolare attenzione a quelle descritte da Böther et al. [2].

Studiare le implementazioni concrete è fondamentale per comprendere come le scelte teoriche influiscano sulle prestazioni reali degli algoritmi. In particolare, dettagli come la gestione dei cicli, la rappresentazione dei grafi, l'utilizzo di strutture dati avanzate come union-find o heap unibili, e le ottimizzazioni pratiche legate al *growth path* nell'algoritmo di Gabow [8], possono avere un impatto significativo sul tempo di esecuzione e sull'efficienza in memoria.

Il capitolo è strutturato in modo da fornire una panoramica generale delle implementazioni dei tre algoritmi principali (Edmonds [6], Tarjan [13] e Gabow [8]), seguita da un'analisi dettagliata delle scelte implementative effettuate nell'articolo di Böther et al. [2], con un confronto dei punti di forza e dei limiti riscontrati. Infine, verranno discusse le metriche sperimentali utilizzate per la valutazione delle implementazioni e i risultati principali riportati dagli autori.

3.1 PANORAMICA DELLE IMPLEMENTAZIONI

Prima di entrare nei dettagli delle implementazioni analizzate da Böther et al. [2], è utile fornire una panoramica generale delle principali strategie impiegate per realizzare gli algoritmi di Edmonds, Tarjan e Gabow.

Per l'algoritmo di Edmonds [6], le implementazioni si concentrano sulla selezione degli archi entranti minimi per ciascun vertice e sulla gestione dei cicli mediante contrazioni. In molte versioni pratiche, il grafo è rappresentato mediante liste di adiacenza, mentre le contrazioni

vengono gestite tramite strutture dati relativamente semplici, in grado di aggiornare i pesi degli archi e ricostruire l'arborescenza finale.

Nell'implementazione di Tarjan [13], le strutture dati assumono un ruolo centrale: union-find con *path compression* e *union by rank* vengono utilizzate per riconoscere e gestire rapidamente i cicli, mentre heap unibili permettono di mantenere gli archi entranti minimi aggiornati in modo efficiente. Queste scelte consentono di ridurre il costo computazionale rispetto alla versione base di Edmonds, sfruttando la complessità logaritmica delle operazioni sugli insiemi di archi.

L'algoritmo di Gabow [8], infine, introduce ulteriori ottimizzazioni legate all'ordine di elaborazione dei vertici tramite il concetto di *growth path*. Le implementazioni pratiche mantengono strutture aggiuntive, come *exit lists* e *active forest*, per gestire in modo efficiente la selezione degli archi e le contrazioni. Inoltre, vengono introdotti archi fittizi di peso nullo uscenti dalla radice ed entranti in ogni altro nodo del grafo: tali archi semplificano la logica dell'algoritmo senza influenzare il peso finale dell'arborescenza, e rendono inoltre il grafo connesso.

In generale, le prestazioni effettive delle implementazioni dipendono in modo significativo da alcuni dettagli tecnici. Tra questi, rivestono un ruolo centrale l'efficienza con cui vengono individuati gli archi entranti minimi, la gestione delle contrazioni tramite strutture dati avanzate, e la capacità delle strutture di supporto (come heap unibili o altri tipi di code con priorità) di mantenere aggiornati i pesi e le priorità degli archi. Anche la rappresentazione del grafo, ad esempio mediante liste di adiacenza o matrici, influisce sensibilmente sul comportamento pratico degli algoritmi, in particolare quando si passa da grafi sparsi a grafi densi. Questi aspetti, ancor più delle complessità asintotiche, determinano le prestazioni osservabili in applicazioni reali.

3.2 ANALISI DELLE IMPLEMENTAZIONI DI BÖTHER ET AL.

Sulla base di queste considerazioni generali, è interessante analizzare nel dettaglio le implementazioni pratiche sviluppate nella letteratura scientifica. In particolare, Böther et al. [2] propongono e confrontano versioni efficienti degli algoritmi di Tarjan e Gabow, con l'obiettivo di valutarne il comportamento sperimentale su differenti tipologie di grafi. Nella presente sezione verranno descritte le principali scelte implementative riportate dagli autori, evidenziando le strutture dati adottate, le ottimizzazioni introdotte e le differenze rispetto alla formulazione teorica presentata nel capitolo precedente.

3.2.1 *Implementazioni di Tarjan*

Un primo gruppo di implementazioni realizzate da Böther et al. [2] riguarda l'algoritmo di Tarjan. Tutte le versioni condividono la stessa logica di base, inclusa la fase di ricostruzione, e differiscono esclusivamente nella struttura dati utilizzata per gestire gli insiemi di archi entranti. La variante denominata *Matrix solver* mantiene una matrice di adiacenza e realizza le operazioni in tempo lineare. Le versioni *Hollow* e *Treap solver* utilizzano rispettivamente implementazioni di *hollow heaps* [10] e *treaps*[1] [1], entrambe capaci di supportare la *lazy propagation* per l'aggiornamento dei pesi.

Nel caso degli *hollow heaps* non è necessaria l'operazione classica di *decrease-key*, poiché non richiesta dall'algoritmo; ciò permette di implementare in modo efficiente l'operazione di fusione (*merge*), riducendo la complessità di alcune operazioni di gestione. Per comprendere il funzionamento di tali strutture, consideriamo il seguente esempio in cui un nodo con valore 2 diventa *hollow*. Il nodo non viene rimosso immediatamente dalla struttura, ma rimane come "segnaposto" fino a una futura operazione di fusione o ricostruzione. Questo approccio evita di dover aggiornare in tempo reale l'intera struttura, riducendo il costo degli aggiornamenti.

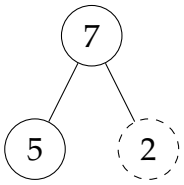

Figura 3: Esempio di *hollow heap*: il nodo con valore 2 è stato marcato come *hollow* dopo un aggiornamento, in attesa di essere eliminato o "riciclato" durante una fusione.

Nel caso dei *treaps*, l'efficienza deriva invece dalla possibilità di effettuare operazioni di fusione in maniera rapida, mantenendo la struttura bilanciata grazie alla gestione casuale delle priorità. Inoltre, come per gli *hollow heaps*, viene sfruttata la *lazy propagation* per l'aggiornamento dei pesi, evitando costosi ricalcoli immediati. In questo caso, come mostrato nell'esempio, ogni nodo è caratterizzato da una coppia formata da una

1 Un treap è una struttura dati che combina un albero binario di ricerca (basato su chiavi) e un heap binario (basato su priorità casuali), garantendo in media operazioni efficienti di ricerca, inserimento e cancellazione.

chiave e da una priorità casuale. La struttura mantiene contemporaneamente due proprietà: la disposizione delle chiavi segue quella di un albero binario di ricerca, mentre le priorità rispettano la proprietà di heap. L'equilibrio dell'albero è quindi garantito dalla componente casuale delle priorità, che rende efficienti le operazioni di fusione e aggiornamento.

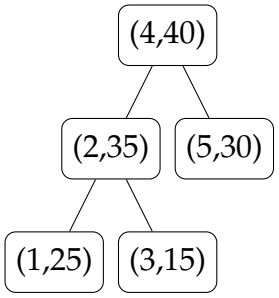

Figura 4: Esempio di *treap*: le chiavi mantengono la proprietà di albero binario di ricerca, mentre le priorità rispettano la proprietà di heap.

Altre due varianti, denominate *set* e *pq solver*, sfruttano invece le strutture dati disponibili nella libreria standard del C++: rispettivamente `std::set`, implementata tipicamente come albero rosso-nero[2] [4], e `std::priority_queue`, basata su heap binario [13]. Queste strutture non supportano efficientemente l'operazione di fusione, per cui viene adottata la tecnica nota come *smaller-to-larger*: in caso di fusione, si itera sugli elementi dell'insieme più piccolo e li si inserisce in quello più grande. Tale tecnica può essere illustrata considerando due insiemi: uno più piccolo, ad esempio $\{1, 3\}$, e uno più grande, $\{2, 4, 5, 6\}$. In caso di fusione, gli elementi verranno gestiti come mostrato in Figura 5.

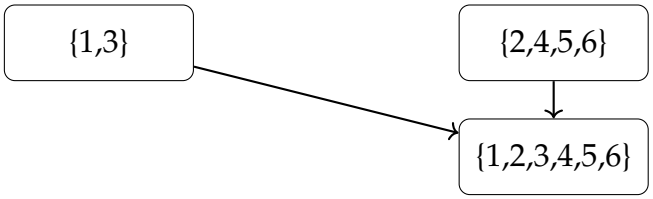

Figura 5: Esempio di fusione con tecnica *smaller-to-larger*: gli elementi dell'insieme più piccolo $\{1, 3\}$ vengono inseriti in quello più grande $\{2, 4, 5, 6\}$, riducendo il numero complessivo di spostamenti.

2 Un albero rosso-nero è una struttura dati di tipo albero binario di ricerca auto-bilanciato. Garantisce che le operazioni di inserimento, cancellazione e ricerca vengano eseguite in tempo $O(\log n)$, mantenendo l'albero sempre bilanciato mediante regole di colorazione dei nodi e rotazioni.

Un aspetto importante dell'analisi riguarda proprio quest'ultima tecnica: ad ogni passaggio la struttura in cui l'elemento viene spostato avrà dimensione almeno doppia rispetto a quella in cui l'elemento risiedeva fino a quel momento; di conseguenza, ciascun elemento può subire al più $O(\log n)$ spostamenti. Tuttavia, ogni spostamento non è un'operazione a costo costante: sia negli alberi bilanciati, sia negli heap binari, l'inserimento di un nuovo elemento richiede tempo $O(\log n)$. Ne deriva quindi che il costo complessivo associato alla gestione di ciascun arco è pari a $O(\log^2 n)$, e moltiplicando tale costo per tutti gli m archi si ottiene una complessità totale pari a $O(m \log^2 n)$.

In questo approccio, gli aggiornamenti dei pesi non richiedono *lazy propagation*, ma vengono gestiti tramite un offset mantenuto per ciascun insieme, applicato solo quando un elemento entra o esce dalla struttura.

3.2.2 *Implementazioni di Gabow et al.*

La versione di Gabow et al. implementata da Böther et al. [2] presenta tre ottimizzazioni principali rispetto alla descrizione teorica. La prima consiste nell'evitare l'inserimento di archi fittizi: ogni volta che il *growth path* raggiunge la radice, viene semplicemente avviato un nuovo percorso. Ad esempio, immaginiamo un grafo in cui il percorso di crescita parte dal nodo X, attraversa alcuni nodi e arriva di nuovo alla radice R. Nella formulazione teorica, verrebbe introdotto un arco fittizio da R a un nuovo nodo per continuare il percorso. Nell'implementazione di Böther et al., invece, non viene inserito alcun arco artificiale: si interrompe il percorso e se ne avvia uno nuovo da R. In questo modo si semplifica la gestione della struttura dati e si riducono le operazioni inutili.

La seconda ottimizzazione riguarda l'uso di array dinamici al posto delle liste concatenate: le *exit lists*, le *passive lists* e il *growth path* sono modificate solo in testa, consentendo di utilizzare gli array salvando gli elementi in ordine inverso. Questo semplifica anche la gestione delle *passive lists*, eliminando la necessità di riferimenti incrociati per ciascun arco. Per illustrare il vantaggio, si considerino le *exit list* di un nodo contenenti gli archi uscenti ordinati per costo: [e1, e2, e3]. Poiché l'algoritmo modifica sempre e solo la testa della lista (qui e1), è possibile rappresentare la lista in un array "rovesciato" [e3, e2, e1]. In questo modo, l'inserimento e la rimozione in testa diventano semplici operazioni di push o pop su array dinamici, evitando i riferimenti incrociati tipici delle liste concatenate.

La terza ottimizzazione riduce ulteriormente il bisogno di riferimenti

incrociati tra le strutture, semplificando i meccanismi di cancellazione degli archi durante le contrazioni: gli archi vengono rimossi solo durante la contrazione di un ciclo. In particolare, gli archi in uscita vengono eliminati cancellando intere *exit lists* e sincronizzando le cancellazioni con le *passive lists*, mentre i multi-archi in ingresso sono gestiti cancellando le *passive lists* complete e riflettendo le modifiche sulle *exit lists*. Böther et al. [2] propongono di semplificare ulteriormente questo processo evitando completamente la rimozione dalle *passive lists*. Gli archi non validi che rimangono in queste liste diventano cappi temporanei, che vengono identificati e ignorati durante la contrazione.

Infine, la consolidazione dei multi-archi viene gestita confrontando, per ciascun arco passivo che entra nel ciclo, i primi due archi nella *exit list* del suo nodo d'origine e eliminando quello più costoso. In questo modo, il più economico tra gli archi rimanenti resta sempre in testa alla lista. Questa strategia semplifica la gestione degli archi passivi rispetto alla versione originale proposta da Gabow et al. [8], che richiedeva riferimenti diretti agli elementi della *exit list* per ciascun arco passivo. È importante notare che questa strategia funziona correttamente quando gli archi verso lo stesso ciclo sono contigui nella *exit list*, condizione garantita dall'ordinamento della lista per costo crescente e dalla gestione delle contrazioni: in questo modo, confrontando i primi due archi della lista si può sempre identificare e mantenere quello più economico, senza rischiare di trascurare archi validi verso lo stesso ciclo.

Per chiarire il funzionamento di questa strategia, consideriamo l'esempio illustrato in Figura 6. Supponiamo che dal nodo X escano tre archi, di cui due verso lo stesso ciclo C_1, con costi crescenti:

$$(X \to C_1, \text{costo } 2)$$
$$(X \to C_1, \text{costo } 5)$$
$$(X \to Y, \text{costo } 7)$$

Secondo la logica di Böther et al., si confrontano i primi due archi della *exit list* del nodo X, eliminando quello più costoso. In questo caso, l'arco con costo 5 viene rimosso, lasciando in testa alla *exit list* l'arco più economico uscente da X ed entrante nel ciclo, cioè l'arco di costo 2. In questo modo, si mantiene sempre in testa l'arco valido più economico, evitando la gestione complicata dei riferimenti incrociati tra *exit list* e *passive list* e semplificando la gestione dei multi-archi durante le contrazioni.

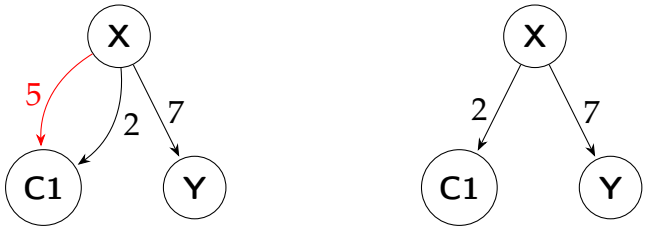

Figura 6: Esempio di gestione dei multi-archi da X verso il ciclo C_1. L'arco con costo maggiore tra i primi due archi della *exit list* viene rimosso.

3.2.3 *Confronto tra le implementazioni di Tarjan e Gabow*

Le implementazioni analizzate mostrano differenze significative sia dal punto di vista delle strutture dati utilizzate sia delle strategie di ottimizzazione. L'implementazione di Tarjan utilizza diverse varianti per la gestione degli insiemi di archi entranti, tra cui matrici di adiacenza, *hollow heaps*, *treaps*, `std::set` e `std::priority_queue` [2]. Queste strutture influiscono direttamente sulla complessità operativa e sulle prestazioni osservate nei test su grafi sparsi e densi. In particolare, l'approccio basato su heap unibili con *lazy propagation* permette aggiornamenti efficienti dei pesi, mentre le varianti con strutture standard richiedono tecniche come la *smaller-to-larger* [2] per garantire una fusione efficiente degli insiemi.

L'implementazione di Gabow et al., al contrario, ottimizza principalmente la gestione dei *growth path* e delle liste di archi in uscita e passivi, evitando l'inserimento di archi fittizi e semplificando le operazioni di cancellazione e fusione dei multi-archi [8, 2]. L'uso di array dinamici e la semplificazione della sincronizzazione tra *exit list* e *passive list* riduce il costo delle operazioni e rende la versione più adatta a grafi di dimensioni medio-grandi con frequenti contrazioni di cicli.

In termini di complessità, entrambe le implementazioni mantengono le proprietà asintotiche dei rispettivi algoritmi: Tarjan con complessità $O(m \log n)$ o $O(m \log^2 n)$ a seconda della struttura dati, e Gabow con complessità $O(n \log n + m)$. Tuttavia, le differenze pratiche emergono chiaramente quando si considerano i dettagli implementativi, come la gestione dei multi-archi, l'aggiornamento dei pesi e l'efficienza delle operazioni di fusione.

Per rendere il confronto più immediato, la Tabella 1 riassume i principali parametri delle due implementazioni, inclusi le strutture dati utilizzate, le ottimizzazioni adottate, la complessità teorica e le osservazioni pratiche relative alle prestazioni.

Dal confronto emerge che la scelta dell'implementazione dipende for-

temente dalle caratteristiche del grafo e dagli obiettivi dell'utente. Le versioni di Tarjan risultano vantaggiose in contesti in cui la struttura dati avanzata permette aggiornamenti rapidi e il numero di archi è elevato ma sparso. Al contrario, l'implementazione di Gabow è più semplice da gestire, riduce il numero di operazioni costose e risulta più performante in grafi più grandi o quando il pattern di crescita genera frequenti contrazioni di cicli. La Tabella 1 evidenzia inoltre come le strutture dati abbiano un impatto diretto sulle prestazioni osservate, confermando che non basta considerare la complessità asintotica, ma anche i dettagli implementativi e la gestione pratica dei dati.

Tabella 1: Confronto tra le implementazioni di Tarjan e Gabow secondo Böther et al. [2]

Parametro	**Tarjan**	**Gabow**
Strutture dati principali	Matrice, Hollow heaps, Treaps, `std::set`, `std::priority_queue`	Array dinamici, exit list, passive list, growth path
Gestione pesi	Lazy propagation (Hollow/Treap), offset per set (set/PQ)	Aggiornamento durante la contrazione, senza lazy propagation
Ottimizzazioni principali	Smaller-to-larger merge, gestione multi-archi	Nessun arco fittizio, array dinamici, semplificazione cancellazioni e multi-archi
Complessità teorica	$O(m \log n)$ o $O(m \log^2 n)$	$O(n \log n + m)$
Scenario ideale	Grafi sparsi o con aggiornamenti frequenti dei pesi	Grafi medio-grandi, ottimizzazione per contrazioni frequenti

3.3 SPERIMENTAZIONE

Per valutare le implementazioni degli algoritmi di arborescenza minima presentate, Böther et al. [2] hanno condotto un'analisi sperimentale dettagliata utilizzando diverse soluzioni. Gli esperimenti sono stati eseguiti su un server con due processori Intel Xeon™ Gold 6144 a 8 core ciascuno e

192GB di memoria RAM, utilizzando sistemi openSUSE Leap 15.3. Tutti i codici erano implementati in C++ e adattati ad un'interfaccia comune, compilati con GCC 10.3.0. Ogni esecuzione era limitata a un timeout massimo di 30 minuti.

Sono stati considerati sei insiemi di reti, per un totale di 656 istanze. I dataset includevano reti reali e generate artificialmente: reti orientate provenienti dal progetto KONECT[3], selezioni di reti sparse da Network Repository[4], grafi geometrici casuali (GIRGs, 200 reti)[5], reti appositamente create per essere difficili da risolvere per i solver (antilemon, 5 reti)[6], casi di programmazione competitiva (fastestspeedrun, 47 reti)[7], e reti test per il problema del Directed MST su Library Checker (yosupo, 10 reti)[8]. Per i grafi privi di pesi, sono stati assegnati pesi interi casuali uniformi; per quelli senza radice specificata, si è aggiunto un nodo radice con archi di peso infinito verso tutti gli altri vertici.

I risultati, come riportato in Figura 7, hanno mostrato come, sulle istanze *untied* (cioè quelle in cui un solver era chiaramente più veloce degli altri), i solver basati su heap binario (*pq*) dominassero con 229 successi, seguiti da GGST con 177, *Felerius* (ottimizzato per gestire le istanze di yosupo) con 76 e *Matrix* (basato su matrici di adiacenza) con 50.

Complessivamente, questi quattro solver hanno risolto più del 98% delle istanze *untied*. È emerso un chiaro legame tra il tipo di istanza e il solver più adatto: il solver basato su matrice eccelle sui grafi densi, i solver basati su heap (*pq* e *Felerius*) sui grafi sparsi, mentre l'algoritmo GGST su casi intermedi. Inoltre, per i dataset sparsi reali, *Felerius* è particolarmente efficiente su istanze con basso grado medio. L'analisi della scalabilità con i grafi GIRG ha confermato queste osservazioni: i solver basati su

3 La Koblenz Network Collection (KONECT) è una raccolta di reti reali provenienti da ambiti diversi, come social network, reti di citazioni e reti biologiche.

4 Il Network Repository è un archivio online di grafi e dataset usati comunemente come benchmark per algoritmi di grafi.

5 I Geometric Inhomogeneous Random Graphs (GIRGs) sono un modello generativo di grafi casuali che generalizza i grafi iperbolici e produce strutture con distribuzioni di gradi realistiche.

6 Gli antilemon sono grafi sintetici introdotti da Böther et al. come casi particolarmente ostici, progettati per forzare numerose contrazioni durante l'esecuzione degli algoritmi di arborescenza minima.

7 Il dataset fastestspeedrun deriva da task di programmazione competitiva proposti durante l'ICPC Northwestern Europe Regional Contest 2018.

8 Library Checker è una piattaforma di online judge che fornisce insiemi di test standardizzati per problemi algoritmici; yosupo è lo pseudonimo del suo creatore, sotto cui vengono spesso identificati i test.

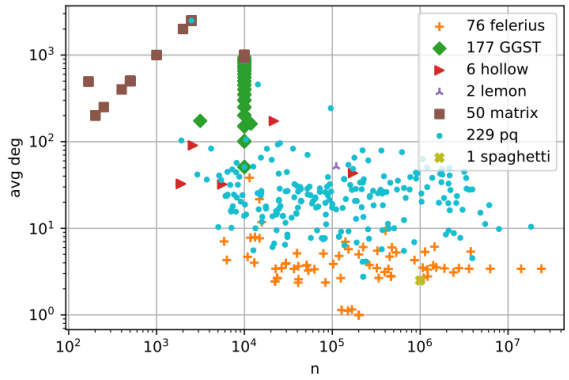

Figura 7: Risultati dei solver. Nella legenda è incluso, per ogni solver, il numero di successi. [2]

matrice non sono influenzati dal numero di archi e diventano vantaggiosi su grafi molto densi, mentre gli altri solver mostrano una scalabilità quasi lineare rispetto al numero di archi. Alcuni solver, come *Treap* e *Hollow*, scalano peggio con l'aumentare della densità, principalmente perché utilizzano strutture dati basate su puntatori per gestire gli archi. Al contrario, *Felerius* e GGST sfruttano indici o strutture ottimizzate che permettono operazioni più efficienti. La Figura 8 mostra l'andamento dei tempi di esecuzione dei diversi solver, evidenziando quanto appena discusso.

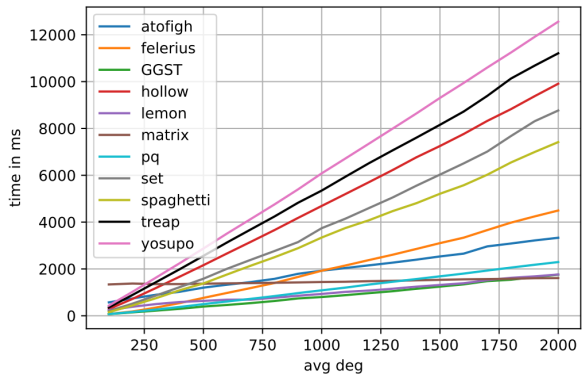

Figura 8: Tempo medio di esecuzione dei solver su 10 grafi GIRG con 10^4 vertici, al variare della densità. [2]

Infine, Böther et al. [2] hanno osservato che la fase di ricostruzione dell'arborescenza minima richiede solo una frazione del tempo totale,

indipendentemente dal solver o dal dataset, mentre l'inizializzazione, che comprende la costruzione delle strutture dati interne e l'inserimento degli archi negli heap, rappresenta uno dei principali colli di bottiglia teorici, con complessità $O(m \log n)$. Alcune implementazioni, come *yosupo*, mostrano tempi di deallocazione della memoria elevati a causa dell'uso di `std::shared_ptr` [9] al posto della gestione manuale della memoria. La Figura 9 mostra il tempo totale di esecuzione per ciascun algoritmo sui diversi dataset, con le barre suddivise per evidenziare la frazione di tempo impiegata in ciascuna sotto-routine e i timeout conteggiati come intervallo completo di 30 minuti.

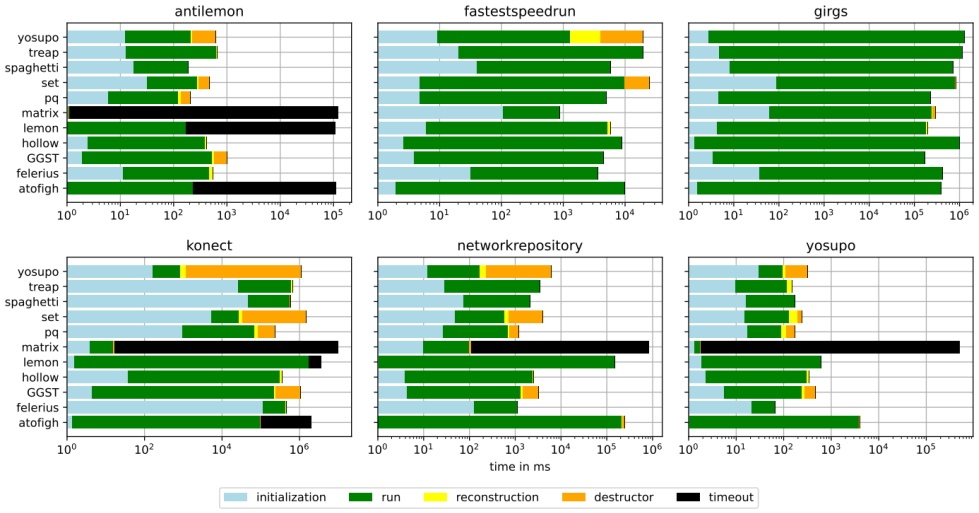

Figura 9: Tempo totale di esecuzione per dataset e per algoritmo, suddiviso tra le diverse fasi. In caso di timeout, sono conteggiati 30 minuti indipendentemente dalle operazioni svolte. [2]

3.4 CONSIDERAZIONI FINALI

In sintesi, le valutazioni sperimentali confermano che le prestazioni dei diversi solver dipendono fortemente dalla struttura dei grafi e dalle scelte implementative. Solver come *pq* e *Felerius* risultano particolarmente efficienti sui grafi sparsi, mentre le implementazioni basate su matrici mostrano vantaggi solo su grafi densi, dove le operazioni di accesso diretto compensano la maggiore complessità teorica.

9 `std::shared_ptr` è una smart pointer della libreria standard di C++ che implementa la gestione automatica della memoria tramite un meccanismo di reference counting.

Le differenze nella gestione dei pesi, nella memoria e nelle strutture dati hanno un impatto significativo sul tempo di esecuzione reale, spesso più rilevante della complessità teorica. In particolare, l'uso di heap ottimizzati, array dinamici e strategie di propagazione dei pesi consente di ridurre il numero di operazioni critiche, migliorando le prestazioni su scenari specifici.

L'analisi dei dataset sperimentali ha inoltre evidenziato come il comportamento degli algoritmi sia fortemente influenzato da caratteristiche come densità del grafo, presenza di contrazioni frequenti e struttura dei cicli, confermando che le scelte implementative hanno un peso determinante nella pratica.

La Tabella 3 riassume i risultati sperimentali delle implementazioni degli algoritmi di Tarjan e Gabow come riportati da Böther et al. [2]. Essa fornisce un quadro chiaro dei punti di forza e delle limitazioni di ciascun solver, consentendo di identificare scenari in cui un'implementazione risulta preferibile rispetto a un'altra e di comprendere l'impatto delle scelte implementative sui tempi di calcolo effettivi.

Infine, questi risultati forniscono indicazioni utili per possibili sviluppi futuri. La comprensione dei fattori che influenzano le prestazioni apre la strada a ottimizzazioni mirate, estensioni per grafi dinamici, e integrazioni con librerie e tool per grafi reali.

Tabella 3: Sintesi delle prestazioni dei principali solver su diversi tipi di reti secondo Böther et al. [2]

Solver	**Tipo di istanze migliori**	**Note implementative / comportamento**
pq (binary heap)	Grafi sparsi, basso numero di contrazioni	Ottima cache locality, semplice logica, merge non necessario; eccezionale prestazione nonostante la complessità teorica
Felerius	Grafi sparsi con basso grado medio	Ottimizzato per casi con pochi archi, struttura dati efficiente, vince su yosupo e Network Repository
GGST	Grafi intermedi	Buona scalabilità, ottimizzato per casi generici, molte vittorie su GIRGs
Matrix-based Tarjan	Grafi molto densi	Complessità $O(n^2)$; molto efficiente su grafi completamente connessi, ma non gestisce grandi grafi sparsi
Treap / Hollow heap	Grafi sparsi	Supportano lazy propagation, gestione efficiente dei pesi; prestazioni in linea con complessità teorica $O(n \log n)$ per sparsi
Set / std::priority_queue	Grafi piccoli / sparsi	Merge eseguito con tecnica “smaller into larger”, costi $O(m \log^2 n)$; buona scalabilità su grafi sparsi

4

APPLICAZIONI PRATICHE E SVILUPPI FUTURI

Gli algoritmi per la costruzione di arborescenze minime non rappresentano soltanto un tema di interesse accademico, ma trovano applicazioni dirette in contesti concreti, spesso molto diversi tra loro. La capacità di ridurre una rete complessa a una struttura ad albero, che conserva proprietà di minimalità e aciclicità, si rivela infatti utile in domini in cui è necessario semplificare, organizzare o ottimizzare la gestione di informazioni e risorse.

In questo capitolo vengono analizzati due ambiti applicativi particolarmente significativi, ciascuno dei quali è stato oggetto di studi recenti:

- **Gestione automatica della memoria**: il lavoro di Lahaie-Bertrand et al. [11] propone l'*Arborescent Garbage Collection*, un approccio innovativo alla deallocazione automatica di memoria (*garbage collection*). Tradizionalmente, il recupero di memoria in presenza di cicli è un problema critico per i linguaggi di programmazione con *reference counting* come, ad esempio, Python. La soluzione classica si affida a processi asincroni che però introducono incertezza e ritardi nella deallocazione. L'approccio arborescente, ispirato ad algoritmi di raggiungibilità su grafi dinamici, prevede la costruzione e l'aggiornamento dinamico di una foresta di riferimenti che consente di individuare immediatamente gli oggetti non più accessibili, anche in presenza di cicli, garantendo così un comportamento deterministico e prevedibile.

- **Analisi di reti complesse orientate**: lo studio di Coscia [5] affronta il problema della determinazione di gerarchie in reti complesse. Molti sistemi reali, dalle organizzazioni sociali alle reti biologiche, mostrano una tendenza intrinseca a strutturarsi in livelli gerarchici. L'autore propone una stima della gerarchia, rappresentata da un

punteggio numerico, basata sull'arborescenza: una rete può considerarsi tanto più gerarchica quanto più simile ad essa è la sua arborescenza ideale. Questo approccio viene confrontato con altri metodi noti di rilevazione gerarchica (ad esempio *agony*[1], *flow hierarchy*[2] e *global reaching centrality*[3]), mostrando una maggiore capacità di distinguere tra reti con e senza una reale struttura gerarchica, oltre a fornire uno schema esplicito della gerarchia sottostante.

I due contributi, pur operando in contesti molto differenti (uno nell'ingegneria del software, l'altro nell'analisi dei sistemi complessi) condividono la stessa intuizione di fondo: l'arborescenza, in quanto struttura ordinata, radicata e priva di cicli, rappresenta un modello potente per interpretare e gestire reti orientate. Questa osservazione apre la strada a sviluppi futuri in cui l'uso delle arborescenze non si limita alla teoria dei grafi, ma diventa uno strumento trasversale applicabile a discipline diverse, dalla programmazione alla scienza dei dati.

Va osservato che, pur essendo il fulcro di questa tesi lo studio di algoritmi per l'arborescenza minima, gli esempi applicativi presentati non si basano sulla minimizzazione in quanto trattano di grafi non pesati. Ciò nonostante, in entrambi i casi la struttura dell'arborescenza riveste un ruolo centrale: le arborescenze consentono di rappresentare in modo chiaro e computazionalmente efficace relazioni gerarchiche e vincoli di raggiungibilità. In altre parole, gli algoritmi per l'arborescenza minima analizzati finora forniscono la base teorica e gli strumenti algoritmici da cui derivano approcci pratici, anche quando l'obiettivo non è la minimizzazione, ma l'organizzazione efficiente di strutture dati complesse.

4.1 ARBORESCENZE E GARBAGE COLLECTION

Un ambito applicativo particolarmente interessante per le arborescenze è quello della *garbage collection*, ossia il recupero automatico della memoria

1 *Agony* è un metodo per stimare la gerarchia in un grafo orientato, basato sulla minimizzazione del numero di archi che puntano da nodi di livello inferiore a nodi di livello superiore (archi "contro-flusso").

2 *Flow hierarchy* misura la gerarchicità di un grafo orientato calcolando la frazione di archi che non partecipano a cicli; più alta è questa frazione, più la rete è considerata gerarchica.

3 *Global reaching centrality* (GRC) quantifica la gerarchia di un grafo identificando il nodo che può raggiungere la maggior parte degli altri nodi tramite archi uscenti; quanto più questo nodo è distinto dal resto, tanto più la rete appare gerarchica.

allocata da un programma ma non più utilizzata. Come descritto da Lahaie-Bertrand et al. [11], le tecniche tradizionali di *reference tracking* incontrano una difficoltà significativa nella gestione immediata dei cicli di riferimenti: in questi casi, infatti, gli oggetti non più raggiungibili possono continuare a occupare memoria finché un processo asincrono non provvede al loro rilascio. Questa caratteristica introduce comportamenti imprevedibili, rendendo tali approcci inadatti a contesti in cui sia richiesta una gestione deterministica delle risorse, come nei database a grafo o nei file system con *hard link*.

Per risolvere questo problema, gli autori propongono l'*Arborescent Garbage Collector* (Arborescent GC) [11], un algoritmo ispirato a studi precedenti sui grafi dinamici. L'idea centrale è quella di inscrivere una foresta di arborescenze all'interno del grafo dei riferimenti del programma, così da trattare la verifica della raggiungibilità come un problema di *edge-deletion reachability*. Ogni volta che un riferimento viene eliminato, la struttura viene aggiornata in modo efficiente per stabilire se l'oggetto corrispondente rimane ancora raggiungibile dalle radici del programma, come raffigurato nell'esempio in Figura 10.

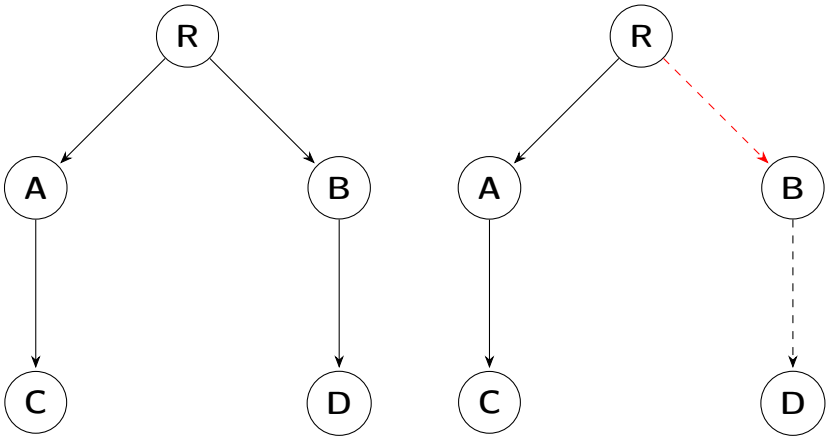

Figura 10: Eliminazione di un riferimento: a sinistra, l'arborescenza iniziale; a destra, viene eliminato l'arco (R, B), e dunque il nodo B non è più raggiungibile dalla radice; in questo particolare esempio, anche il nodo D non è più raggiungibile, e la struttura viene quindi aggiornata di conseguenza.

La novità principale consiste nell'introduzione di una nozione "debole" di rango, che riduce il costo degli aggiornamenti rispetto alle implementazioni sincrone precedenti. In questo contesto, a ciascun nodo è attribuito non un valore preciso, ma un limite superiore sulla distanza

dalla radice. Ciò significa che il rango debole fornisce solo un'indicazione approssimativa della posizione gerarchica del nodo, sufficiente per guidare le euristiche di manutenzione della foresta arborescente. In particolare, la perdita di precisione consente di evitare aggiornamenti globali a ogni modifica locale: quando un arco viene rimosso, i nodi interessati non richiedono un ricalcolo immediato dei propri ranghi, ma restano marcati come "potenzialmente da ricontrollare". La verifica effettiva della raggiungibilità avviene quindi soltanto se necessario, riducendo drasticamente il numero di operazioni eseguite.

Grazie a questa ottimizzazione, l'algoritmo è in grado di recuperare memoria in modo sincrono, senza ricorrere a processi concorrenti. Pur comportando un rallentamento rispetto ai più diffusi algoritmi di tipo *Mark-and-Sweep*[4], l'*Arborescent GC* garantisce prestazioni competitive e, soprattutto, un comportamento affidabile e deterministico. Per questo motivo si configura come una soluzione adatta in scenari in cui la prevedibilità e l'immediatezza del rilascio di memoria sono requisiti fondamentali.

4.2 STIMA DELLE GERARCHIE IN RETI COMPLESSE ORIENTATE

Le reti complesse rappresentano un modello analitico fondamentale per lo studio di fenomeni che emergono dall'interazione di molteplici componenti in sistemi reali, sia naturali che artificiali. Tra le proprietà più rilevanti di tali reti vi è la tendenza a organizzarsi in strutture gerarchiche, suddivise in livelli nei quali i nodi di livello superiore esercitano un'influenza su quelli inferiori. La letteratura distingue diverse tipologie di gerarchia, tra cui quella d'ordine (basata su un punteggio associato ai nodi), quella annidata (costruita tramite strutture contenute in altre più ampie) e quella di flusso, che descrive relazioni direzionali di tipo *top-down*. Quest'ultima, in particolare, è la più intuitiva e si ritrova in numerosi contesti organizzativi, dalle strutture aziendali ai sistemi biologici.

L'articolo di Coscia [5] propone una nuova metodologia per stimare il grado di gerarchicità di una rete complessa orientata, basata sul concetto di arborescenza. A partire da un grafo orientato, l'autore suggerisce di ridurlo a un'arborescenza: tanto maggiore è la quantità di archi (rispetto

4 Gli algoritmi di tipo *Mark-and-Sweep* sono tecniche di garbage collection che operano in due fasi: nella fase di *mark* vengono identificati tutti gli oggetti raggiungibili a partire dalle radici, mentre nella fase di *sweep* vengono deallocati tutti gli oggetti non raggiunti, liberando la memoria.

al totale) che sopravvive all'operazione, tanto più la rete si avvicina a una gerarchia perfetta. Ne deriva così un punteggio di gerarchicità, interpretato come numero di modifiche tra il grafo originale e l'arborescenza corrispondente. La Figura 11 mostra un esempio di tale concetto.

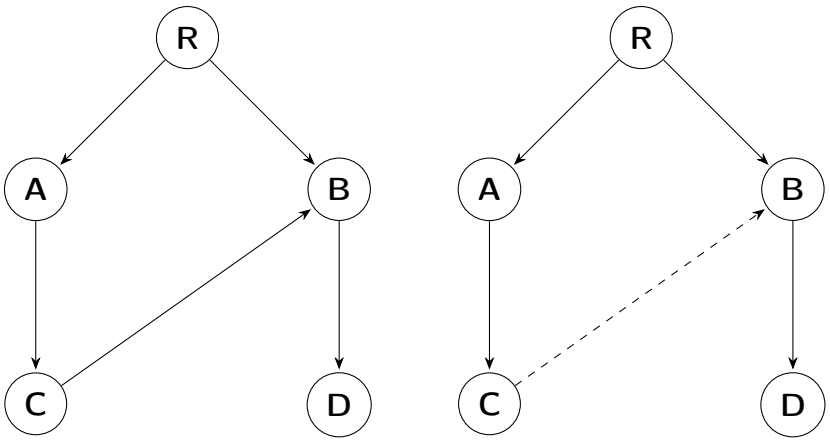

Figura 11: Riduzione di un grafo ad arborescenza: a sinistra, il grafo iniziale contenente un ciclo; a destra, l'arborescenza derivante dal grafo a seguito dell'eliminazione dell'arco (C, B) (tratteggiato). In questo caso, solo un arco è stato "sacrificato", il grafo originale ha dunque grado di gerarchicità alto.

Il metodo è stato confrontato con tre approcci consolidati nello studio delle gerarchie: *agony*, *flow hierarchy* e *global reaching centrality*. I risultati mostrano come l'approccio arborescente sia più severo, tendendo cioè a restituire valori più bassi, ma anche più robusto, in grado di distinguere meglio reti prive di struttura da quelle con una forte componente gerarchica. Inoltre, la natura grafica del metodo permette una rappresentazione visiva della gerarchia, evidenziando non solo i livelli dei nodi, ma anche le connessioni tra predecessori e discendenti, cosa che lo differenzia dai precedenti approcci. Nonostante i punti di forza, l'autore segnala anche alcune limitazioni, tra cui la severità del punteggio e la perdita di precisione dovuta alla fase di collasso delle componenti fortemente connesse.

Questi aspetti rappresentano direzioni interessanti per sviluppi futuri, volti ad ampliare la capacità descrittiva del punteggio di arborescenza e a renderlo uno strumento ancora più efficace per l'analisi delle reti complesse.

4.3 SVILUPPI FUTURI

Le arborescenze si rivelano uno strumento potente non solo nella teoria dei grafi ma anche nelle applicazioni pratiche, come la stima della gerarchia nelle reti orientate e la gestione sincrona dei cicli in *garbage collection* appena trattate. Questi esempi mostrano come i principi studiati nei capitoli precedenti possano guidare scelte algoritmiche e strutturali che influenzano efficienza, precisione e interpretabilità dei risultati.

Guardando avanti, vi sono numerose possibilità di approfondimento. Nel contesto delle reti complesse, il calcolo del punteggio di arborescenza potrebbe essere raffinato per preservare più archi e identificare con maggiore precisione nodi radice e livelli gerarchici. In ambito di *garbage collection*, ulteriori ottimizzazioni potrebbero rendere più efficiente la gestione sincrona dei cicli e permettere l'applicazione a sistemi con requisiti di memoria più complessi.

Più in generale, l'esplorazione di nuovi domini applicativi offre l'opportunità di sviluppare algoritmi più robusti e adattabili. Una prospettiva di ricerca promettente riguarda scenari applicativi che potrebbero emergere nel campo della biologia computazionale: pensiamo, ad esempio, alla rappresentazione dei processi di segnalazione cellulare o alla ricostruzione di alberi filogenetici sotto vincoli di direzionalità. In questi casi, le arborescenze minime offrirebbero una struttura naturale per combinare efficienza e interpretabilità.

Tutti questi scenari suggeriscono come i concetti teorici possano evolvere in direzioni concrete, aprendo nuove prospettive per l'analisi e l'ottimizzazione dei sistemi basati su arborescenze.

5

CONCLUSIONI

Il presente lavoro di tesi si è concentrato sullo studio degli algoritmi per la costruzione di arborescenze minime, con l'obiettivo di analizzarne i fondamenti teorici, le tecniche risolutive e le potenziali applicazioni. Il problema dell'arborescenza minima si colloca all'interno dell'ottimizzazione combinatoria su grafi orientati e rappresenta una generalizzazione naturale del problema dell'albero di copertura minimo per grafi non orientati. La sua rilevanza teorica risiede nella capacità di descrivere strutture che, a partire da una data radice, collegano tutti i nodi di un grafo, garantendo al contempo un costo complessivo minimo sugli archi selezionati.

Nella prima parte del lavoro sono stati esposti i principi fondamentali della teoria delle arborescenze, introducendone i principali concetti e mostrando il parallelismo con il problema del MST. Particolare attenzione è stata successivamente dedicata all'algoritmo di Edmonds e alle sue due varianti principali, che si basano su procedure di contrazione ed espansione dei cicli che consentono di ridurre progressivamente il problema a sottografi più semplici. Questa impostazione ha permesso di mettere in luce sia la solidità del quadro teorico sia la sua trasposizione in procedure algoritmiche efficienti.

Il capitolo centrale del lavoro ha dedicato ampio spazio alle implementazioni pratiche e alla loro valutazione sperimentale, come presentate da Böther et al. [2]. Sono state descritte le varianti implementative basate su diverse scelte di strutture dati, tra cui implementazioni matriciali, *hollow heaps* e *treaps*, ed è stato illustrato come tali scelte influenzino direttamente le operazioni fondamentali dell'algoritmo (inserimento, estrazione del minimo, fusione di insiemi e aggiornamento dei pesi). La sezione presenta inoltre la versione di Tarjan adattata con differenti heap unibili e la variante di Gabow con ottimizzazioni per il *growth path*, confrontandone l'efficienza pratica. L'analisi sperimentale riportata sintetizza i risultati su un ampio insieme di istanze reali e sintetiche, evidenziando tendenze

significative: soluzioni basate su matrici risultano preferibili per grafi molto densi, mentre implementazioni con diversi tipi di heap (quando ben ottimizzate) sono più efficaci su grafi sparsi; l'effetto delle ottimizzazioni implementative (gestione manuale della memoria, tecniche di *lazy propagation* o *smaller-to-larger merge*) si è rivelato spesso determinante nella pratica, e talvolta anche più influente dei limiti asintotici teorici. Infine, il capitolo accenna anche agli aspetti sperimentali secondari (tempi di inizializzazione, costi della fase di ricostruzione e gestione dei timeout), offrendo così una mappatura completa del comportamento empirico delle diverse soluzioni.

Infine, nella parte applicativa sono stati considerati due ambiti di particolare interesse: la gestione automatica della memoria mediante tecniche di *garbage collection* e la valutazione di strutture gerarchiche in reti complesse orientate. Tali casi studio, tratti dalla letteratura recente, hanno mostrato come il concetto di arborescenza possa fungere da strumento interpretativo e operativo in contesti tra loro profondamente differenti, ponendo in evidenza la versatilità e la rilevanza del problema trattato.

Dal punto di vista scientifico, il lavoro presentato contribuisce a consolidare e ad approfondire la comprensione degli algoritmi per l'arborescenza minima, mettendone in luce sia gli aspetti teorici fondamentali, sia le implicazioni pratiche in contesti applicativi concreti. L'analisi sistematica delle principali varianti algoritmiche, accompagnata da un confronto empirico, permette di delineare con chiarezza i *trade-off* tra efficienza teorica e prestazioni reali.

Un ulteriore elemento di rilevanza è rappresentato dall'individuazione del ruolo che le arborescenze possono svolgere al di fuori dell'ambito strettamente algoritmico. L'analisi dei casi di studio considerati, dalla gestione deterministica della memoria mediante *garbage collection* alla valutazione della gerarchicità nelle reti complesse orientate, dimostra come tali strutture forniscano strumenti versatili, capaci di collegare teoria e pratica in settori anche molto distanti tra loro. In questo senso, la tesi non si limita a discutere un problema classico dell'informatica teorica, ma ne evidenzia anche il potenziale contributo alla risoluzione di problematiche attuali e trasversali, rafforzandone la rilevanza tanto dal punto di vista accademico quanto da quello applicativo.

Le prospettive di ricerca che emergono dal tema presentato sono molteplici. Dal punto di vista teorico, un primo sviluppo naturale consiste nell'esplorare varianti degli algoritmi per l'arborescenza minima in grado di affrontare scenari ancora più generali, ad esempio grafi dinamici soggetti a modifiche frequenti, o contesti in cui siano presenti vincoli

addizionali sui pesi e sulle connessioni. In parallelo, l'analisi di strategie di approssimazione e di algoritmi distribuiti potrebbe fornire soluzioni più adatte a sistemi di grandi dimensioni o ad architetture parallele.

Sul piano applicativo, un ulteriore ambito promettente riguarda l'estensione delle arborescenze minime a domini ancora poco esplorati. Oltre alla gestione della memoria e all'analisi delle gerarchie, esse potrebbero rivelarsi strumenti efficaci per la modellazione di flussi informativi in reti di comunicazione, per l'ottimizzazione di infrastrutture logistiche e di trasporto, o per lo studio di sistemi biologici complessi in cui le relazioni asimmetriche tra componenti giocano un ruolo determinante.

Infine, un obiettivo di più ampio respiro consiste nel favorire l'integrazione tra teoria dei grafi e strumenti applicativi, così da colmare il divario tra formulazioni matematiche astratte e implementazioni pratiche. In questa prospettiva, lo studio delle arborescenze minime non rappresenta soltanto un capitolo specifico dell'informatica teorica, ma un paradigma attraverso cui sviluppare metodologie generali per l'analisi, l'ottimizzazione e il controllo di sistemi complessi.

APPENDICE

Grafo Un *grafo* è una coppia $G = (V, E)$, dove V è un insieme non vuoto di vertici ed E è un insieme di coppie di vertici che rappresentano relazioni tra essi.

Nodo (o vertice) Un *nodo* (o *vertice*) è un elemento dell'insieme V di un grafo e rappresenta una singola entità o punto di connessione all'interno della struttura.

Arco (o ramo) Un *arco* (o *ramo*) è una connessione tra due vertici di un grafo, rappresentata da una coppia di vertici (u, v) che può essere ordinata, nel caso di grafo orientato, o non ordinata, nel caso di grafo non orientato.

Grafo orientato Un *grafo orientato* è un grafo in cui ogni arco è una coppia ordinata di vertici (u, v), e quindi rappresenta una relazione asimmetrica dal vertice u al vertice v.

Cammino Un *cammino* è una sequenza finita di vertici $(v_1, v_2, \ldots, v_k)$ tale che per ogni coppia consecutiva (v_i, v_{i+1}) esiste un arco nel grafo che collega v_i a v_{i+1}.

Ciclo Un *ciclo* è un cammino chiuso in cui il primo e l'ultimo vertice coincidono e tutti gli altri vertici sono distinti tra loro.

Cappio Un *cappio* è un arco che collega un vertice a sé stesso, ossia un arco della forma (v, v) con $v \in V$.

Multi-archi (o archi paralleli) I *multi-archi* (o *archi paralleli*) sono due o più archi distinti che collegano la stessa coppia di vertici in un grafo.

Grafo pesato Un *grafo pesato* è un grafo in cui a ciascun arco è associato un valore numerico, detto *peso*, che rappresenta un costo, una distanza o un'altra misura quantitativa.

Componente connessa Una *componente connessa* è un sottoinsieme massimale di vertici di un grafo non orientato in cui ogni coppia di vertici è collegata da almeno un cammino.

Grafo completo Un *grafo completo* è un grafo non orientato in cui ogni coppia di vertici distinti è collegata da uno e un solo arco.

Grafi sparsi e densi Un grafo è detto *sparso* se il numero di archi è molto inferiore al numero massimo possibile, mentre è detto *denso* se possiede un numero di archi prossimo a quello massimo consentito dal numero di vertici.

Sottografo Un *sottografo* è un grafo $G' = (V', E')$ tale che $V' \subseteq V$ ed $E' \subseteq E$, ossia è formato da un sottoinsieme dei vertici e degli archi di un grafo $G = (V, E)$.

Albero Un *albero* è un grafo non orientato, connesso e privo di cicli, in cui tra ogni coppia di vertici esiste un unico cammino.

Albero binario Un *albero binario* è un albero in cui ogni nodo che non è una *foglia* ha esattamente due figli, detti rispettivamente *figlio sinistro* e *figlio destro*.

Albero orientato Un *albero orientato* è un albero in cui ogni arco possiede una direzione, generalmente dalla radice verso i vertici figli (*albero con radice*), formando una struttura gerarchica orientata.

Radice Una *radice* è il vertice di riferimento in un albero con radice, dal quale tutti gli altri vertici sono raggiungibili seguendo la direzione degli archi.

Rango di un albero — Nei sistemi di insiemi disgiunti (*Disjoint Set Union*), il *rango* è un valore intero associato alla radice di ciascun albero e rappresenta una stima della sua altezza.

Visita in profondità — La *visita in profondità* (*Depth-First Search, DFS*) è un algoritmo di esplorazione che visita ricorsivamente i vertici di un grafo, proseguendo lungo ciascun cammino fino a raggiungere un vertice senza adiacenti non visitati prima di retrocedere.

Albero bilanciato — Un *albero bilanciato* è un albero in cui l'altezza dei sottoalberi dei nodi differisce al massimo di una quantità costante, garantendo una distribuzione equilibrata dei vertici.

Minimum Spanning Tree — Un *minimum spanning tree* (MST) è un sottoinsieme di archi che collega tutti i vertici di un grafo connesso e pesato, formando un albero di peso totale minimo.

Foresta — Una *foresta* è un grafo aciclico non necessariamente connesso, costituito da un insieme di alberi disgiunti.

Lista di adiacenza — Una *lista di adiacenza* è una rappresentazione di un grafo in cui a ogni vertice è associata una lista contenente tutti i vertici adiacenti, ossia collegati da un arco uscente o entrante.

Matrice di adiacenza — Una *matrice di adiacenza* è una matrice quadrata A di dimensione $|V| \times |V|$ in cui l'elemento A_{ij} indica la presenza, e eventualmente il peso, di un arco tra i vertici v_i e v_j.

Array dinamico

Un *array dinamico* è una struttura dati che consente di memorizzare una sequenza di elementi in memoria contigua, adattando automaticamente la propria capacità quando il numero di elementi supera lo spazio disponibile.

Lista di priorità

Una *lista di priorità* è una struttura dati in cui a ogni elemento è associato un valore di priorità, e le operazioni di accesso o estrazione riguardano l'elemento con priorità più alta (o più bassa).

Heap

Un *heap* è una struttura dati ad albero binario che soddisfa la proprietà di heap, secondo cui ogni nodo ha un valore non maggiore (o non minore) dei propri figli, consentendo l'accesso efficiente all'elemento di priorità estrema.

Heap unibile

Un *heap unibile* è una variante dell'heap che, oltre alle operazioni standard di inserimento ed estrazione del minimo (o massimo), consente di unire efficientemente due heap distinti in un unico heap valido.

Hollow heap

Un *hollow heap* è una struttura dati basata su alberi che estende il concetto di heap unibile, riducendo il costo delle operazioni tramite nodi "vuoti" (*hollow*) usati per ritardare la ristrutturazione e migliorare l'efficienza ammortizzata.

Skew heap

Uno *skew heap* è un tipo di heap unibile che non mantiene vincoli strutturali fissi, ma riequilibra dinamicamente la struttura scambiando i sottoalberi durante le operazioni di unione per garantire buone prestazioni ammortizzate.

Treap

Un *treap* è una struttura dati che combina un albero binario di ricerca e un heap, assegnando a ogni nodo una chiave e una priorità casuale, in modo che l'ordinamento per chiavi rispetti la proprietà di ricerca e quello per priorità la proprietà di heap.

Heap binario

Un *heap binario* è un heap implementato come albero binario quasi completo, in cui ogni nodo soddisfa la proprietà di heap rispetto ai propri figli e l'albero è rappresentabile in modo efficiente tramite un array.

Operazioni union e find

Le operazioni *union* e *find* sono primitive delle strutture dati per insiemi disgiunti: *find* determina il rappresentante dell'insieme contenente un elemento, mentre *union* unisce due insiemi distinti in un unico insieme.

Algoritmi union-find

Gli *algoritmi union-find* gestiscono collezioni di insiemi disgiunti attraverso le operazioni *union* e *find*, utilizzando tecniche come la *path compression* e la *union by rank* per garantire un'efficienza quasi costante nel tempo ammortizzato.

Struttura union-find

Una *struttura union-find* è una struttura dati progettata per rappresentare e gestire insiemi disgiunti, supportando in modo efficiente le operazioni di unione di due insiemi e di ricerca del rappresentante di un elemento.

Lazy propagation

La *lazy propagation* è una tecnica di ottimizzazione che rimanda l'aggiornamento di alcune informazioni in una struttura dati, applicandolo solo quando strettamente necessario per ridurre il costo computazionale complessivo.

BIBLIOGRAFIA

[1] Cecilia R. Aragon e Raimund Seidel. «Randomized Search Trees». In: *Algorithmica* 16.4-5 (1996), pp. 464–497.

[2] Maximilian Böther, Otto Kißig e Christopher Weyand. «Efficiently computing directed minimum spanning trees». In: *2023 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2023, pp. 86–95.

[3] Yoeng-Jin Chu. «On the shortest arborescence of a directed graph». In: *Scientia Sinica* 14 (1965), pp. 1396–1400.

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

[5] Michele Coscia. «Using arborescences to estimate hierarchicalness in directed complex networks». In: *PloS one* 13.1 (2018), e0190825.

[6] Jack Edmonds. «Optimum branchings». In: *Journal of Research of the national Bureau of Standards B* 71.4 (1967), pp. 233–240.

[7] Joaquim Espada, Alexandre P Francisco, Tatiana Rocher, Luís MS Russo e Cátia Vaz. «On Finding Optimal (Dynamic) Arborescences». In: *Algorithms* 16.12 (2023), p. 559.

[8] Harold N Gabow, Zvi Galil, Thomas Spencer e Robert E Tarjan. «Efficient algorithms for finding minimum spanning trees in undirected and directed graphs». In: *Combinatorica* 6.2 (1986), pp. 109–122.

[9] Bernard A Galler e Michael J Fisher. «An improved equivalence algorithm». In: *Communications of the ACM* 7.5 (1964), pp. 301–303.

[10] Thomas Dueholm Hansen, Haim Kaplan, Robert E Tarjan e Uri Zwick. «Hollow heaps». In: *ACM Transactions on Algorithms (TALG)* 13.3 (2017), pp. 1–27.

[11] Frédéric Lahaie-Bertrand, Léonard Oest O'Leary, Olivier Melançon, Marc Feeley e Stefan Monnier. «Arborescent Garbage Collection: A Dynamic Graph Approach to Immediate Cycle Collection». In: *Proceedings of the 2025 ACM SIGPLAN International Symposium on Memory Management*. 2025, pp. 14–26.

[12] Daniel Dominic Sleator e Robert Endre Tarjan. «Self-adjusting binary search trees». In: *Journal of the ACM (JACM)* 32.3 (1985), pp. 652–686.

[13] Robert Endre Tarjan. «Efficiency of a good but not linear set union algorithm». In: *Journal of the ACM (JACM)* 22.2 (1975), pp. 215–225.

[14] Robert Endre Tarjan. «Finding optimum branchings». In: *Networks* 7.1 (1977), pp. 25–35.